Succinct Adaptive Garbled RAM

Ran Canetti* Yilei Chen† Justin Holmgren‡ Mariana Raykova§

November 4, 2015

Abstract

We show how to garble a large persistent database and then garble, one by one, a sequence of adaptively and adversarially chosen RAM programs that query and modify the database in arbitrary ways. Still, it is guaranteed that the garbled database and programs reveal only the outputs of the programs when run in sequence on the database. The runtime, space requirements and description size of the garbled programs are proportional only to those of the plaintext programs and the security parameter. We assume indistinguishability obfuscation for circuits and poly-to-one collision-resistant hash functions. The latter can be constructed based on standard algebraic assumptions such as the hardness of discrete log or factoring. In contrast, all previous garbling schemes with persistent data were shown secure only in the static setting where all the programs are known in advance.

As an immediate application, our scheme is the first to provide a way to outsource large databases to untrusted servers, and later query and update the database over time in a private and verifiable way, with complexity and description size proportional to those of the unprotected queries.

Our scheme extends the non-adaptive RAM garbling scheme of Canetti and Holmgren [ITCS 2016]. We also define and use a new primitive, called adaptive accumulators, which is an adaptive alternative to the positional accumulators of Koppula et al [STOC 2015] and somewhere statistical binding hashing of Hubáček and Wichs [ITCS 2015]. This primitive might well be useful elsewhere.

*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.

†Boston University, `chenyl@bu.edu`. Supported by NSF grants CNS-1012798, CNS-1012910, CNS1413920 and AF-1218461. Research conducted while at SRI International funded by NSF grant CNS-1421102.

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

§Yale University, SRI, `mariana@columbia.edu`. Supported by NSF grant CNS-1421102 and DARPA SafeWare.

Contents

1 Introduction

Database delegation. Consider an owner of a large database that wishes to delegate the database to an untrusted remote server, and then update and query the database in arbitrary ways over time. We wish to hide both the database and the queries/updates from the server, and continue to do so even when the results of queries are exposed, say via using the results elsewhere. Furthermore, the correctness of responses to queries, taking into account all past updates, should be verifiable. Does there exist a delegation scheme that meets these security requirements? Can this be done with complexity and communication comparable to those of the unprotected mechanism, both for server and data owner? If so, then under what hardness assumptions?

This task is a practically motivated generalization of the basic delegation-of-computation task. It is also a natural generalization of the tasks of encrypted and verifiable databases. These are all well studied tasks with a variety of solutions that obtain a variety of partial verifiability and secrecy guarantees, under a variety of assumptions, both for general computations and database queries and for specific ones.

Still, the above general question has remained unresolved so far. Indeed, it poses strong requirements: First, since queries and updates may come over time, the solution must be able to guarantee security even in the face of adaptively chosen queries and updates. Furthermore, since the size and the complexity of database queries are typically much smaller than the size of the database, a solution must adhere to stringent efficiency and succinctness requirements. Out of the many partial solutions to this question let us mention only the recent work of Kalai and Paneth [KP15], which addresses the same problem, and provides a general solution assuming LWE. However, their solution guarantees only verifiability and no secrecy.

We provide the first general solution to this problem, guaranteeing both verifiability and strong secrecy, as well as asymptotically optimal efficiency and succinctness (up to polynomial factors in the security parameter.) We assume indistinguishability obfuscation for circuits and collision resistant hash functions that are at most poly-to-one. We also construct the latter from standard algebraic assumptions such as the hardness of discrete log, factoring, or finding shortest independent vectors of lattices.

The key element in our solution is an adaptively secure garbling scheme for RAM computations with persistent data. We thus provide some background on garbling schemes.

Program garbling. The concept of program garbling [Yao86, Rog91] is central in cryptography with a variety of applications. The goal of a program garbling scheme is to "encode" the functionality of a given program and input in such a way that one can evaluate the program on the input without learning anything beyond the output of the evaluation. A closely related notion is that of *randomized encodings* of functions [IK00].

The original Yao construction, set in the context of secure two-party computation, uses Boolean circuits for programs representation. This means that the size of the garbled program is proportional to both the runtime and space of the plaintext program. It can also be used securely for the evaluation of only a single garbled input. Many improvements have been made since, e.g., in optimizing the size of the garbled circuit [KS08, KMR14, ZRE15], in garbling arithmetic circuits [AIK11], in providing improved security guarantees against both malicious evaluator and garbler [LP11, MR13], in garbling Turing machines (TMs) [GKP+13] and in developing *reusable* garbling [GKP+13, GHRW14].

Garbled RAM and persistent memory. Most relevant to our setting is the notion of a garbled RAM (GRAM) [LO13], which allows the evaluator to do work that is only proportional to the random-access-machine (RAM) complexity of the plaintext program. Beyond the general complexity advantages of RAM computation over circuit or even TM computation, the notion of of GRAM naturally opens the door to the the concept of garbling with *persistent memory.* Here one garbles a sizable memory (database), and then garbles a sequence of programs, where the programs are meant to run on the same memory, in sequence. Persistence implies that any modifications made by some machine should be visible by all subsequent machines. Still, the complexity of the garbled programs should be comparable to that of the plaintext ones. This should hold even when this complexity is sub-polynomial in the size of the memory.

A central challenge in garbled RAM constructions, that does not exist in garbling of circuits or Turing machines, is the need to hide the access pattern to the random access memory. This is typically done by incorporating an oblivious RAM (ORAM) mechanism [GO96] in the garbing scheme. The fact that ORAM schemes inherently make the computation randomized adds significant complication.

The GRAM scheme of Lu and Ostrovsky [LO13] needs one way functions (OWFs) with a strong circular security property. The need for such a property was removed by Gentry et al. [GHL+14], who present two GRAM constructions: In the first construction, which is based on identity-based encryption (IBE), the size of the garbled program is proportional to the running time of the plaintext program; the second construction uses only standard OWFs but incurs overhead for the garbled program of $O(n^\epsilon)$, where n is the size of the memory used by the plaintext program. The additional complexity overhead is removed by Garg et al. [GLOS15] assuming only standard OWFs.

The constructions of garbled RAM mentioned above are inherently one-time, whereas in the work by Gentry et al. [GHRW14], the garbled RAM program is reusable. They construct reusable GRAM without persistent memory from indistinguishability obfuscation for circuits, and reusable GRAM with persistent memory from obfuscation with a stronger property called strong differing inputs obfuscation.

Succinct GRAM. In the above schemes the description size of the garbled machines is proportional to the *runtime* of the plaintext machines. Several works make progress towards reducing the description size of the garbled machines. Bitansky et al. [BGL+15] and Canetti et al. [CHJV14] construct GRAMs where the size of the garbled program is proportional only to the space complexity of the plaintext RAM program. Koppula et al. [KLW15] construct, using similar assumptions, a beautiful fully succinct garbling scheme for Turing machines. These constructions are based on indistinguishability obfuscation for circuits and injective one way functions.

Building on the techniques of [KLW15] and using similar assumptions, Canetti and Holmgren [CH16] and independently Chen et al. [CCC+15] present a fully succinct GRAM where the size of the garbled program is proportional only to the size of the plaintext RAM program. The main contribution here is in showing how to encapsulate and hide the randomness necessary for the oblivious RAM mechanism. Chen et al. also demonstrate how to preserve the Parallel RAM (PRAM) complexity of the plaintext program. Canetti and Holmgren show how to apply their scheme in the setting of persistent memory as described above.

Adaptive Security for Garbled Programs. The schemes mentioned so far only address the static setting where all inputs are chosen by the adversary in advance before it sees any garbled program. In contrast, adaptive security considers the case where new challenges may adversarially depend on the public information released so far. In the context of one-time garbling, this means that the the input may depend on the garbed program. This setting is considered in Goldwasser et al. [GKR08] and Bellare et al. [BHR12]. The latter work presents transformations from statically-secure one-time garbling schemes to adaptively-secure one-time garbling schemes that either incur overhead for the input garbling that is proportional to the size of the function circuit or that are instantiated in the random oracle model (ROM). Ananth et al. [ABSV15, AS15] and Waters [Wat15] construct adaptively secure functional encryption and program garbling for Turing machines in the plain model. In particular, both input and program in [AS15] are succinct. However, none of them is able to provide persistent memory.

Overall, while RAM garbling with persistent memory is a natural potential solution to the problem of outsourcing databases presented earlier, none of the existing solutions appears adequate. Indeed, an *adaptive* and *succinct* garbling scheme with *persistent memory* seems to be needed.

1.1 This work

We construct an adaptively secure succinct garbling scheme for RAM programs with persistent memory. That is, the scheme allows its user to garble an initial memory, and then garble RAM programs that arrive one by one in sequence. The machines can read from and update the memory, and have local output. It is guaranteed that:
(1) Running the garbled programs one after the other in sequence on the garbled memory results in the same sequence of outputs as the result of running the plaintext machines one by one in sequence on the plaintext memory.
(2) The view of any adversary that generates a database and programs and obtains their garbled versions is

simulatable by a machine that sees only the outputs of the plaintext programs when run in sequence on the plaintext database. This holds even when the adversary chooses new plaintext programs adaptively, based on the garbled memory and programs seen so far.
(3) The scheme is both efficient and succinct: The time to garble the memory is proportional to the plaintext memory, and the memory is garbled only once in the beginning. Up to polynomial factors in the security parameter, the garbling time and size of the garbled program are proportional only to the size of the plaintext RAM program. The runtime and space of the garbled machine are comparable to those of the plaintext machine.

Given such a scheme, constructing a database delegation scheme as specified above is straightforward: The database owner lets the server store a garbled version of the database. To delegate a query, garble the program that executes the query. To obtain verifiability use the technique from [CHJV15, BGL+15]: Each program will contain a signing key and will sign all its outputs. The verification key will be publicized by the client. To hide the query results from the server, encrypt the program's output to the querying party, say using symmetric encryption. We provide a more complete definition and construction within.

1.2 Overview of the construction

Our starting point is the statically-secure garbling scheme of Canetti and Holmgren [CH16]. We briefly sketch their construction, and then explain where the issues with adaptivity come up and how we solve them.

Statically-secure garbling scheme for RAMs - an overview. The Canetti-Holmgren construction consists of three main steps. They first build a *fixed-transcript garbling scheme*, i.e. a garbling scheme which guarantees indistinguishability of the garbled machines and inputs as long as the entire transcripts of the communication with the external memory, as well as the local states kept between activations, are the same in the two computations. In other words, if the computation of machine M_1 on input x_1 has the same transcript as that of M_2 on input x_2, then $\tilde{M_1}, \tilde{x_1} \approx \tilde{M_2}, \tilde{x_2}$. This step closely follows the scheme of Koppula, Lewko and Waters [KLW15] for garbling of Turing machines. The garbled program is essentially an obfuscated CPU-step circuit, which takes the previous state and a memory symbol as input and outputs the next state, the symbols to write into memory, and the next location to read from. The main challenge here is to guarantee the authenticity and freshness of the values read from the memory. This is done using a number of mechanisms, namely splittable signatures, iterators and positional accumulators.

The second step is to obtain a *fixed-address garbling scheme*, namely a scheme that guarantees indistinguishability of the garbled machines as long as only the sequence of *addresses* of memory accesses is the same in the two computations. This is achieved by encrypting the state and memory content in an obfuscation-friendly way. The third step is to use an obfuscation-friendly ORAM in order to hide the program's memory access pattern. (Specifically, they use the ORAM of Chung and Pass [CP13].)

The challenge of adaptive security. We outline three issues which prevent this construction from being adaptively secure and explain how we deal with them.

The first (and main) issue has to do with the *positional accumulator,* which is an obfuscation-friendly variant of a Merkle-hash-tree built on top of the memory. That is, the contents of the memory is hashed down until a short root (called the accumulator value ac) is obtained. Then this value is signed together with the current step by the CPU and is kept (in memory) for subsequent verification of database accesses. Using the accumulator, the evaluator is later able to efficiently convince the CPU that the contents of a certain memory location L is v. We call this operation "opening" accumulator value ac to contents v at location L. Intuitively, the main security property is that it should be infeasible to open an accumulator value to two different contents values at the same location.

However, to be useful with indistinguishability obfuscation, the accumulator needs an additional property, called *enforceability.* In [KLW15], this property allows to generate, given memory location L^* and symbol v^*, a "rigged" public key for the accumulator along with a "rigged" accumulator value ac^*. The rigged public key and accumulator look indistinguishable from honestly generated public key and accumulator value, and also have the property that there does not *exist* a way to open ac^* to value other than v^* at location L^*.

To get an idea of why enforceability is needed, consider two programs C_0 and C_1, such that $C_0(L^*, v^*) = C_1(L^*, v^*)$, but whose functionality may differ elsewhere, and let $C'_i(L, v)$ be the program "if L, v are consistent with ac^* then run C_i, else output $\perp$". Let $\mathsf{i}\mathcal{O}$ be an indistinguishability obfuscator, i.e. it is guaranteed that $\mathsf{i}\mathcal{O}(A) \approx \mathsf{i}\mathcal{O}(B)$ whenever equal sized programs A, B have the same functionality everywhere. Positional accumulators allow arguing that $\mathsf{i}\mathcal{O}(C'_0) \approx \mathsf{i}\mathcal{O}(C'_1)$ in spite of the fact that the programs C'_0 and C'_1 have different functionality. This is done as follows: using the enforceability property it is possible to argue that, when C'_0 and C'_1 use the rigged public key for the accumulator, the two programs have exactly the same functionality, and so indistinguishability holds. Due to the indistinguishability of rigged public accumulator keys from honest ones, indistinguishability holds even for the case of non-rigged accumulator keys.

However, the fact that the special values v^*, L^*, and ac^* are encoded in the rigged public key forces these values to be known before the adversary sees the public key. This suffices for the case of static garbling, since the special values depend only on the underlying computation, and this computation is fixed in advance and does not depend on adversary's view. However, in the adaptive setting, this is not the case. This is so since the adversary can choose new computations — and thus new special values v^*, L^* — depending on its view so far, which includes the public key of the accumulator.

A naive solution to this problem would be to generate a fresh accumulator instance for every execution. But this is not effective in the context of persistent memory, since it requires recomputing a new accumulator root (corresponding to the new parameters) before every execution and thus doing work proportional to the entire memory size at every execution.

A more viable potential solution is to replace the accumulator of [KLW15] with the *somewhere statistically binding (SSB) hash* of Hubáček and Wichs [HW15], assuming fully homomorphic encryption, or alternatively from DDH or the ϕ-hiding assumption [OPWW15]. Similar to the enforcing mechanism in the accumulator of [KLW15], the SSB hash can also be set up with a hidden statistical binding location, with an additional feature that *only the special location* L^* needs to be known at the time of generation of the rigged public key. The guarantee is that, with the rigged public key, and for any accumulator value ac, there exists at most a single value v such that ac can be opened to value v at location L^*.

The fact that only the location needs to be fixed in advance is a significant, since it allows the proof of security to go through even in the case of an adaptive adversary — as long as the program uses only a polynomial number of potential memory locations. Indeed, in this case the reduction to the security of the SSB hash can guess the (adaptively chosen) special location L^* ahead of time and be correct with polynomial probability.

Adaptive Accumulators. We propose an alternative solution to SSB hashing. Our solution works regardless of the size of the potential address space, and obtains better parameters. Specifically, we define and construct *adaptive accumulators,* which are an adaptive alternative to SSB hashing and positional accumulators. In our adaptive accumulators there are no "rigged" public keys. Instead, correctness of an opening of a hash value at some location is verified using a *verification key* which can be generated later. In addition to the usual computational binding guarantees, it should be possible to generate, given a special accumulator value ac^*, value v^* and location L^*, a "rigged" verification key vk^* that looks indistinguishable from an honestly generated one, and such that vk^* does not verify an opening of ac^* at location L^* to any value other than v^*. Furthermore, it is possible to generate multiple verification keys, that are all rigged to enforce the same accumulator value ac^* to different values v^* at different locations L^*, where all are indistinguishable from honest verification keys.

We then use adaptive accumulators as follows: There is a single set of public parameters that is posted together with the garbled database and is used throughout the lifetime of the system. Now, each new garbled machine is given a different, independently generated verification key. This allows us, at the proof of security, to use a different rigged verification key for each machine. Since the key is determined only when a machine is being garbled (and its computation and output values are already fixed), we can use a rigged verification key that enforces the correct values, and obtain the same tight security reduction as in the static setting.

Adaptively puncturable hash functions. We build adaptive accumulators from a new primitive called *adaptively puncturable (AP)* hash function ensembles. In this primitive a standard collision resistant hash function $h(x)$ is augmented with three algorithms Verify, GenVK, $\mathsf{GenBindingVK}$. GenVK generates a verification key vk, which can be later used in $\mathsf{Verify}(\mathsf{vk}, x, y)$ to check that $h(x) = y$. $\mathsf{GenBindingVK}(x^*)$ produces a binding key vk^* such that $\mathsf{Verify}(\mathsf{vk}^*, x, y = h(x^*))$ accepts only if $x = x^*$. Finally, we require that real and binding verification keys should be indistinguishable even for the adversary which chooses x^* adaptively after seeing h.

The construction of adaptive accumulators from AP hash functions proceeds as follows. The public key is an AP hash function h, and the initial accumulator value ac_0 is the root of a Merkle tree on the initial data store (which can be thought of as empty, or the all-0 string) using h. We maintain the invariant that at every moment the root value ac is the result of hashing down the store. In order to write a new symbol v to a position L the evaluator recomputes all hashes on the path from the root to L. The "opening information" for v at L is all hashes of siblings on the path from the root to L.

The verification key is a sequence of $d = \log |S|$ (honest) verification keys for h - one for each level of the tree. The "rigged" verification key for accumulator value ac^* and value v^* at location L consists of a sequence of d rigged keys for the AP hash - where each key forces the opening of a single value along the path from the root to leaf L^*. Security of the adaptive accumulator follows from the security of the AP hash via standard reduction.

Constructing AP hash. We construct adaptively puncturable hash function ensembles from indistinguishability obfuscation for circuits, plus collision-resistant hash functions with the property that any image has at most polynomially many preimages. (This implies that the CRHF shrinks at most logarithmically many bits). We say that a hash function is *c-bounded* if the number of preimages for any image is no more than c. To be able to "compose" functions in various ways we will also need that the hash functions have domain $\{0,1\}^\lambda$ and range $\{0,1\}^{\lambda'}$ for some $\lambda' < \lambda$. For simplicity we focus on the setting where $\lambda = \lambda' + 1$. We construct 4-bounded CRHFs assuming hardness of discrete log and 64-bounded CRHFs assuming hardness of factoring.

The construction of AP hash proceeds in two steps.

1. First we construct a c-bounded AP hash function ensemble from any c-bounded hash function ensemble $\{h_k\}$. This is done as follows: The public key is the description of the hash function h_k. A verification key vk is $\mathsf{i}\mathcal{O}(V)$, where V is the program that on input x, y outputs 1 if $h_k(x) = y$. A "rigged" verification key vk^* that is binding for input x^* is $\mathsf{i}\mathcal{O}(V_{x^*})$ where V_{x^*} is the program that on input (x, y) does the following:

   - if $y = f_h(x^*)$, it accepts if and only if $x = x^*$;
   - otherwise it accepts if and only if $y = h_k(x)$.

   Since h_k is c-bounded, the functionality of V and V_{x^*} differ only on polynomially many inputs. Therefore, the real and "rigged" verification keys are indistinguishable following the $\mathsf{di}\mathcal{O}$-$\mathsf{i}\mathcal{O}$ equivalence for circuits with polynomially many differing inputs [BCP14].

2. Next we construct AP hash functions which are, say, length halving (and are thus not polynomially bounded) from bounded AP hashing. This is done in the natural way by extending the hash function's domain using Merkle-Damgård, and then obfuscating the resulting function. We show that if the underlying poly-bounded hash is adaptively puncturable, then so is the composed one.

From Adaptive Accumulators to Adaptively Secure Garbling. We return to the challenges encountered when trying to use the [CH16] construction in our adaptive setting. Now that we have adaptive accumulators, we are able to complete the first step in the [CH16] construction in the natural way, and prove its security in the adaptive setting. Here we generate fresh instances of an iterator and splittable signature scheme for each new garbled machine. This does not cause any problems since these primitives do not access the long-lived shared memory.

Towards obtaining fixed-address garbling. Recall that in this step the programs encrypt each memory cell with a "long lived key" that remains unchanged for all programs. Specifically, [CH16] replaces writing a symbol s to location addr at timestep t with writing $(t, F_k(\mathsf{addr}, t) \oplus s)$ instead, for some puncturable PRF F_k. The initial memory is encrypted as if it were written at time $t = 0$.

This approach is problematic in the adaptive setting. The proof of fixed-access garbling involves puncturing F_k and changing its value to random at points which depend on the computation, so a straightforward adaptation of the proof (for $F(i, \mathsf{addr}, t)$, where i is execution number) for the static case does not work. Indeed, the points at which the memory needs to be punctured may depend on the garbled memory itself.

Our first observation is that there is no need to use addr as a PRF input; instead we can use $F_k(i, t)$, where i is the execution number. This is because at a single step the program only writes to one address (for the initial memory, we will now think of each address a as having been written at a distinct time, e.g. $-a$), so there is no danger of reusing the pseudorandom padding. Next, note that each program M_i can only use PRF values $F(i, \ldots)$ for Write and $F(0, \ldots), \ldots, F(i, \ldots)$ for Read.

Thus it is possible to puncture F at (i^*, t^*) as follows:

- We hardwire a punctured key into programs $\tilde{M}_1, \ldots, \tilde{M}_{i-1}$, without hardwiring the PRF value; this works since these programs never use $F(i^*, \ldots)$. Note that i^* and t^* do not depend on the computation (we do puncturing for every (i^*, t^*) one by one) and therefore these programs with a punctured key inside can be generated before i-th computation is known.

- We hardwire a punctured key together with a challenge value into programs $\tilde{M}_i, \ldots, \tilde{M}_n$; this is possible since the challenge value becomes known upon receiving M_i from the adversary.

Note that we could also prove security if we left addr in the input of a PRF and still use it in our adaptive setting. For this one would need to use a special puncturable PRF (the GGM construction suffices) which allows one to generate subkeys K_i for computing only $F(i, \mathsf{addr}, t)$ for fixed i and arbitrary addr, t.

Issues with the full garbling step. Recall that the full garbling in [CH16] is achieved by applying an ORAM on top of the fixed-access garbling. The randomness for the real ORAM accesses and the simulated accesses is sampled using a PRF. This leads to a situation where a PRF key is first used inside a program M_i for some execution i and later needs to be punctured.

We get around this issue by noticing that the Chung-Pass ORAM has a special property which allows us to guess which points to puncture with only polynomial security loss. This property, which we call *strong localized randomness*, is sketched as follows. Let R be the randomness used by the ORAM. Let $\vec{A_i} = \vec{a_{i1}}, \ldots, \vec{a_{im}}$ be a set of locations accessed by the ORAM during emulation of access i. The strong localized randomness property guarantees that there exists a set of intervals $I_{11}, \ldots, I_{Tm}$, $I_{ij} \subset [1, |R|]$, such that:

1. Each $\vec{a_{ij}}$ depends only on $R_{I_{ij}}$, i.e., the part of the randomness R indexed with I_{ij}; furthermore, $\vec{a_{ij}}$ is efficiently computable from I_{ij};

2. All I_{ij} are mutually disjoint;

3. All I_{ij} are efficiently computable given the sequence of memory operations.

To see that the Chung-Pass ORAM has strong localized randomness, observe that in its non-recursive form, each virtual access of addr touches two paths: one is the path used for the eviction, which is purely random, and the other is determined by the randomness chosen in the previous virtual access of addr. Therefore, the set of accessed locations is determined by two randomness intervals. When the ORAM is applied recursively, the sequence of accesses is determined by $O(\log S)$ intervals. Since the number of intervals in the range $[1, \ldots, |R|]$ is only polynomial in the security parameter, the reduction can guess the interval (and therefore the points to puncture at) with only polynomial security loss.

2 Preliminaries

2.1 Function families

A function ensemble $\mathcal{F}$ has a key generation function $g : S_\lambda \to K_\lambda$; on seeds s of length λ, g produces a key k for a function f_k with domain D_λ and range R_λ:

$$\mathcal{F} = \{f_k : D_\lambda \to R_\lambda, k = g(s), s \in \{0,1\}^\lambda\}_{\lambda \in \mathbb{N}}$$

2.2 Collision resistant hash function

A hash function ensemble $\mathcal{H} = \{h_k : D_\lambda \to R_\lambda\}_{\lambda \in \mathbb{N}}$ is collision resistant if for all p.p.t. adversary $\mathcal{A}$, there is a negligible function $\text{negl}(\cdot)$ such that

$$\Pr_{\mathcal{A},k}[\mathcal{A}(1^\lambda, h_k) \to x_1, x_2 \in D_\lambda : h_k(x_1) = h_k(x_2) \wedge x_1 \neq x_2] < \text{negl}(\lambda)$$

2.3 Obfuscation

For a circuit family $\mathcal{F} = \{f : D_\lambda \to R_\lambda\}_{f \in \mathcal{F}_\lambda}$, a probabilistic algorithm Obf is an obfuscator, if

1. The circuit $\mathsf{Obf}(f)$ has the exact same functionality as f;
2. There is a polynomial $B(\cdot)$ such that $|\mathsf{Obf}(f)| \leq B(|f|)$.

The security properties are defined as follows:

Definition 2.1 (Indistinguishability Obfuscation [BGI+12, GGH+13])**.** Obf is an Indistinguishability Obfuscator (iO) for $\mathcal{F}$ if for any p.p.t. distinguisher $\mathcal{D}$, there is a negligible function $\text{negl}(\cdot)$ such that for all circuits f_0 and f_1 that have identical functionalities, and are of the same size, it holds that

$$|\Pr[\mathcal{D}(\mathsf{Obf}(\lambda, f_0)) = 1] - \Pr[\mathcal{D}(\mathsf{Obf}(\lambda, f_1)) = 1]| \leq \text{negl}(\lambda)$$

Definition 2.2 (Differing-inputs Obfuscation [BGI+12, ABG+13, BCP14])**.** A function family $\mathcal{F}$ associated with a p.p.t. sampler Sam is a differing-inputs function family if for all p.p.t. adversary $\mathcal{A}$, there exists a negligible function $\text{negl}(\cdot)$ such that:

$$\Pr[f_0(x) \neq f_1(x) : \ (f_0, f_1, \mathsf{aux}) \leftarrow \mathsf{Sam}(1^\lambda), \ x \leftarrow A(1^\lambda, f_0, f_1, \mathsf{aux}) \leq \text{negl}(\lambda)$$

Obf is a Differing-inputs Obfuscator for a differing-inputs function family $\mathcal{F}$ if for any p.p.t. distinguisher $\mathcal{D}$, there exists a negligible function $\text{negl}(\cdot)$ such that for $(f_0, f_1, \mathsf{aux}) \leftarrow \mathsf{Sam}(1^\lambda)$, we have that

$$\Pr[\mathcal{D}(\mathsf{di}\mathcal{O}(\lambda, f_0), \mathsf{aux}) = 1] - \Pr[\mathcal{D}(\mathsf{di}\mathcal{O}(\lambda, f_1), \mathsf{aux}) = 1] \leq \text{negl}(\lambda)$$

Boyle, Chung and Pass show that an indistinguishability obfuscator is also a differing-input obfuscator for functions with only polynomially many differing-inputs [BCP14].

Lemma 2.1 ([BCP14])**.** For every polynomial $p(\cdot)$, for all differing-input sampler $\mathsf{Sam}'(1^\lambda)$ that outputs functions with differing-inputs less than $p(\lambda)$:

$$\Pr[(f_0, f_1, \mathsf{aux}) \leftarrow \mathsf{Sam}'(1^\lambda) : \ |\{x \in D_\lambda | f_0(x) \neq f_1(x)\}| < p(\lambda)] = 1$$

an indistinguishability obfuscators is also a differing-input obfuscator for Sam'.

2.4 Puncturable pseudorandom functions

Definition 2.3 (Puncturable PRF [KPTZ13, BW13, BGI14, SW14])**.** Let $\ell(\lambda)$ and $m(\lambda)$ be the input and output lengths. A family of puncturable pseudorandom functions $\mathcal{F}$ is given by a triple of efficient functions (Gen, Eval, $\mathsf{Puncture}$), where $\mathsf{Gen}(1^\lambda)$ generates the key F, such that F maps from $\{0,1\}^{\ell(\lambda)}$ to $\{0,1\}^{m(\lambda)}$; $\mathsf{Eval}(F, x)$ takes a PRF F, an input x, outputs $F(x)$; $\mathsf{Puncture}(F, x^*)$ takes a key and an input x^*, outputs a punctured key $F\{x^*\}$.

It satisfies the following conditions:

- Functionality preserved over unpunctured points: Let $F\{x^*\} = \mathsf{Puncture}(F, x^*)$, then for all $x \neq x^*$, $\mathsf{Eval}(F, x) = \mathsf{Eval}(F\{x^*\}, x)$.
- Pseudorandom on the punctured points: For every p.p.t distinguisher D who chooses an input x^*, the following two distributions are indistinguishable: $(x^*, F\{x^*\}, F(x^*))$ and $(x^*, F\{x^*\}, r^*)$, where r^* is uniform in $\{0,1\}^{m(\lambda)}$.

Theorem 2.2 ([GGM86, KPTZ13, BW13, BGI14, SW14])**.** If one-way function exists, then for all length parameters $\ell(\lambda)$, $m(\lambda)$, there is a puncturable PRF family that maps from $\ell(\lambda)$ bits to $m(\lambda)$ bits.

2.5 The RAM Model

2.5.1 RAM Machines

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

- Σ is a finite set, which is the possible contents of a memory cell. 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 $Q \times (\Sigma \cup \{\epsilon\}) \to (Q \times O_\Sigma) \cup Y$. Here O_Σ denotes the set of memory operations with Σ 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 ℓ_Σ is the length of a binary encoding of Σ, and similarly for ℓ_Q and ℓ_Y.

2.5.2 Memory Configurations

A memory configuration on alphabet Σ is a function $s : \mathbb{N} \to \Sigma \cup \{\epsilon\}$. Let $\|s\|_0$ denote $|\{a : s(a) \neq \epsilon\}|$ and, in an 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 $x = x_1 \dots x_n \in \Sigma^*$ 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.

2.5.3 Execution

We now define what it means to execute a RAM machine $M = (\Sigma, Q, Y, C)$ on an initial memory configuration $s_0 \in \Sigma^{\mathbb{N}}$ 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 $M(s_0) = \perp$. When $M(s_0) \neq \perp$, it is convenient to define the following functions:

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

- Define the *addresses accessed* by M on s_0 as $(a_0, \ldots, a_{t-1})$, and denote this $\mathsf{Addr}(M, s_0)$.
- Define the *resultant memory configuration* of M on s_0 as s_t, and denote it $\mathsf{NextMem}(M, s_0)$.

2.5.4 Probabilistic RAM Machines

We will also consider RAM machines with randomized transition functions. We define a probabilistic RAM machine as a tuple (Σ, Q, Y, C). As in deterministic RAM machines, Σ is the alphabet of symbols that can be stored in memory, Q is the set of local states, and Y is the set of possible machine outputs.

The transition function C now maps

$$C : Q \times (\Sigma \cup \{\epsilon\}) \times \{0,1\} \to (Q \times O_\Sigma) \cup Y$$

For any function $f : \mathbb{N} \to \{0,1\}$, and any probabilistic RAM machine $M = (\Sigma, Q, Y, C)$, we define a deterministic RAM machine $M^f = (\Sigma, Q', Y, C')$, where

$$Q' = \mathbb{N} \times Q.$$

and

$$C'((t,q),\sigma) = ((t+1,q'),\mathsf{op})$$

where $(q', \mathsf{op}) \leftarrow C(q, \sigma, f(t))$

2.5.5 RAM Machine Concatenation

For RAM machines $M_1, \ldots, M_t$, we let $M_1; \ldots; M_t$ denote the RAM machine which sequentially executes M_1 through M_t on the same initial memory s_0, and then outputs whatever M_t outputs.

2.6 Garbled RAM

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

- **Key Generation:** $\mathsf{KeyGen}(1^\lambda, S)$ takes the security parameter λ in unary and a space bound S, and outputs a secret key SK.
- **Memory Garbling:** $\mathsf{GbMem}(SK, s)$ takes as input a secret key SK and a memory configuration s, and then outputs a memory configuration $\tilde{s}$.
- **Machine Garbling:** $\mathsf{GbPrg}(SK, M_i, T_i, i)$ takes as input a secret key SK, a RAM machine M_i, a running time bound T_i, and a sequence number i, and outputs a RAM machine $\tilde{M}_i$.

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

Correctness. A garbling scheme is said to be correct if for all p.p.t. adversaries $\mathcal{A}$ and every $t = \mathrm{poly}(\lambda)$

$$\Pr\left[\tilde{M}_t(\tilde{s}_{t-1}) = M_t(s_{t-1}) \;\middle|\; \begin{array}{l} (s_0, S) \leftarrow \mathcal{A}(1^\lambda) \\ SK \leftarrow \mathsf{KeyGen}(1^\lambda, S) \\ \tilde{s}_0 \leftarrow \mathsf{GbMem}(SK, s_0) \\ \text{for } i = 1, \ldots, t \\ \quad M_i, T_i \leftarrow \mathcal{A}(\tilde{s}_0, \tilde{M}_1, \ldots \tilde{M}_{i-1}) \\ \quad \tilde{M}_i \leftarrow \mathsf{GbPrg}(SK, M_i, T_i, i) \\ \quad s_i = \mathsf{NextMem}(M_i, s_{i-1}) \\ \quad \tilde{s}_i = \mathsf{NextMem}(\tilde{M}_i, \tilde{s}_{i-1}) \end{array}\right] \geq 1 - \mathrm{negl}(\lambda),$$

where

- $\sum T_i \leq \mathrm{poly}(\lambda)$, $|s_0| \leq S \leq \mathrm{poly}(\lambda)$;

- $\mathsf{Space}(M_i, s_{i-1}) \leq S$ and $\mathsf{Time}(M_i, s_{i-1}) \leq T_i$ for each i.

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

1. KeyGen, GbPrg, and GbMem are probabilistic polynomial-time algorithms. Furthermore, GbMem runs in time linear in $\|s_0\|$. We require *succinctness* for the garbled programs, which means that the size of a garbled program $\tilde{M}$ is linear in the description length of the plaintext program M. The bounds T_i and S are encoded in binary, so the time to garble does not significantly depend on either of these quantities.

2. With $\tilde{M}_i$ and $\tilde{s}_i$ defined as above, it holds that $\mathsf{Time}(\tilde{M}_i, \tilde{s}_{i-1}) = \tilde{O}(\mathsf{Time}(M_i, s_{i-1}))$ and $\mathsf{Space}(\tilde{M}_i, \tilde{s}_{i-1}) = \tilde{O}(S)$ (hiding polylogarithmic factors in S).

Security. We define the security property of GRAM as follows.

Definition 2.4. Let $\mathcal{GRAM} = (\mathsf{Setup}, \mathsf{GbMem}, \mathsf{GbPrg})$ be a garbling scheme. We define the following two experiments, where each M_i is a program with time and space complexity T_i and S that is evaluated with memory s_{i-1} and $y_i = M_i(s_{i-1})$, $s_i = \mathsf{NextMem}(M_i, s_{i-1})$, and $T_i = \mathsf{Time}(M_i, s_{i-1})$.

Experiment $\mathrm{REAL}_{\mathcal{A}}(1^\lambda)$	**Experiment** $\mathrm{IDEAL}_{\mathcal{A}}(1^\lambda)$
$(s_0, S) \leftarrow \mathcal{A}(1^\lambda)$	$(s_0, S) \leftarrow \mathcal{A}(1^\lambda)$
$SK \leftarrow \mathsf{Setup}(1^\lambda, S), \tilde{s}_0 \leftarrow \mathsf{GbMem}(SK, s_0)$	$\tilde{s} \leftarrow \mathsf{Sim}(1^\lambda, \ell)$
$(M_1, 1^{T_1}) \leftarrow \mathcal{A}(\tilde{s}_0)$	$(M_1, 1^{T_1}) \leftarrow \mathcal{A}(\tilde{s}_0)$
$\tilde{M}_1 \leftarrow \mathsf{GbPrg}(SK, M_1, T_1, 1)$	$\tilde{M}_1 \leftarrow \mathsf{Sim}(y_1, T_1)$
for $i = 1$ to $\ell = \mathsf{poly}(\lambda)$	for $i = 1$ to $\ell = \mathsf{poly}(\lambda)$
$\quad (M_{i+1}, 1^{T_{i+1}}) \leftarrow \mathcal{A}(\tilde{M}_i)$	$\quad (M_{i+1}, 1^{T_{i+1}}) \leftarrow \mathcal{A}(\tilde{M}_i)$
$\quad \tilde{M}_{i+1} \leftarrow \mathsf{GbPrg}(SK, M_{i+1}, T_{i+1}, i+1)$	$\quad \tilde{M}_{i+1} \leftarrow \mathsf{Sim}(y_{i+1}, t_{i+1})$
Output : $b \leftarrow \mathcal{A}(\tilde{M}_{n+1})$	**Output** : $b' \leftarrow \mathcal{A}(\tilde{M}_{n+1})$

The garbling scheme $\mathcal{GRAM}$ is *ϵ-adaptively secure* if

$$\left|\mathsf{Pr}[1 \leftarrow \mathrm{REAL}_{\mathcal{A}}(1^\lambda)] - \mathsf{Pr}[1 \leftarrow \mathrm{IDEAL}_{\mathcal{A}}(1^\lambda)]\right| < \epsilon.$$

2.7 Splittable Signatures

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

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

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

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

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

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

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

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

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

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

A splittable signature scheme must satisfy the following properties.

Correctness

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

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

and

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

Correctness requires that with probability 1 over the above sampling:

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

Verification Key Indistinguishability

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

Verification Key Indistinguishability requires that the following indistinguishabilities hold:

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

Splitting Indistinguishability

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

Splitting indistinguishability requires that

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

2.8 Cryptographic Iterators

Roughly speaking, a cryptographic iterator is a family of collision-resistant hash functions which is i$\mathcal{O}$-friendly when used to authenticate a chain of values. In particular, we think of using a hash function H to hash a chain of values $m_k, \ldots, m_1$ as $H(m_k \| H(m_{k-1} \| \cdots H(m_1 \| 0^\lambda)))$, which we shall denote as $H^k(m_k, \ldots, m_1)$. A cryptographic iterator provides two indistinguishable ways of sampling the hash function H. In addition to "honest" sampling, one can also sample H so that for a 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 $\mathsf{Setup\text{-}Itr}$ to Setup and renaming $\mathsf{Setup\text{-}Itr\text{-}Enforce}$ to $\mathsf{SetupEnforce}$. Formally, a cryptographic iterator for the message space $\mathcal{M} = \{0,1\}^n$ consists of the following probabilistic polynomial-time algorithms. Setup and $\mathsf{SetupEnforce}$ are randomized algorithms, but $\mathsf{Iterate}$ is deterministic, corresponding to our above discussion of a hash function.

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

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

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

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

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

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

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

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

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

A cryptographic iterator must satisfy the following properties.

Indistinguishability of Setup

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

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

Enforcing

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

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

3 c-Bounded Collision-Resistant Hash Functions

We say that a hash function ensemble $\mathcal{H} = \{\mathcal{H}_\lambda\}_{\lambda \in \mathbb{N}}$ with $\mathcal{H} = \{h_k : D_\lambda \to R_\lambda\}_{k \in \mathcal{K}_\lambda}$ is c-bounded if

$$\Pr_{h \leftarrow \mathcal{H}_\lambda}[\forall y \in R_\lambda, \#\{x : h(x) = y\} \leq c] \geq 1 - \mathrm{negl}(\lambda)$$

That is, with high probability, every element in the codomain of h has at most c pre-images. In this paper we focus on the case where $D_\lambda = \{0,1\}^{\lambda+1}$, $R_\lambda = \{0,1\}^\lambda$, and hope to get c as a polynomial $\mathrm{poly}(\lambda)$. For both of the constructions presented in this section, the bounds are constant.

Our constructions use claw-free pairs of permutations (π_0, π_1) on a domain $\mathcal{D}$. Our starting point is the construction of [Dam88], in which for some fixed y_0, the hash $h(x)$ is defined as $(\pi_{x_0} \circ \cdots \circ \pi_{x_n})(y_0)$. Unfortunately, this construction does not give any non-trivial bound.

However, we observe that the same techniques allow us to take an injective function $\iota : \{0,1\}^n \to \mathcal{D}$ and turn it into a 2^k-bounded collision-resistant function mapping $\{0,1\}^{n+k} \to \mathcal{D}$. As long there is such an injection ι for large enough n (within $\log(\lambda)$ of the bit-length m of elements of $\mathcal{D}$), then we obtain a $\text{poly}(\lambda)$-bounded collision-resistant hash function.

Theorem 3.1. If for a random λ-bit prime p, it is hard to solve the discrete log problem in $\mathbb{Z}_p^*$, then there exists a 4-bounded CRHF ensemble $\mathcal{H} = \{\mathcal{H}_\lambda\}_{\lambda \in \mathbb{N}}$ where $\mathcal{H}_\lambda$ consists of functions mapping $\{0,1\}^{\lambda+1} \to \{0,1\}^\lambda$.

Proof. Let g and h be randomly chosen generators of $\mathbb{Z}_p^*$. Then the permutations $\pi_0(x) = g^x$ and $\pi_1(x) = g^x h$ are a claw-free pair of permutations. It is easy to see there is an injection $\iota_{in} : \{0,1\}^{\lambda-1} \to \mathbb{Z}_p^*$ and an injection $\iota_{out} : \mathbb{Z}_p^* \to \{0,1\}^\lambda$. Define a hash function

$$f : \{0,1\}^{\lambda-1} \times \{0,1\} \times \{0,1\} \to \{0,1\}^\lambda$$

$$a, b, c \mapsto \iota_{out}(\pi_c(\pi_b(\iota_{in}(a))))$$

Clearly given $x \neq x'$ such that $f(x) = f(x')$, one can find a claw (and therefore find $\log_g h$), so f is collision-resistant. Also for any given image, there is at most one corresponding pre-image per choice of b, c, so f is 4-bounded. $\square$

Theorem 3.2. If for random λ-bit primes p and q, with $p \equiv 3 \pmod 8$ and $q \equiv 7 \pmod 8$, it is hard to factor $N = pq$, then there exists a 64-bounded CRHF ensemble.

Proof. First, we construct injections $\iota_0 : \{0,1\}^{2\lambda-4} \to [N/6]$ and $\iota_1 : [N/6] \to \mathbb{Z}_N^* \cap [N/2]$, using the fact that for sufficiently large p and q, for any integer $x \in [N/6]$, at least one of $3x, 3x+1$, and $3x+2$ is relatively prime to N. Let $\iota_{in} : \{0,1\}^{2\lambda-4} \to \mathbb{Z}_N^* \cap [N/2]$ denote $\iota_1 \circ \iota_0$. Let ι_{out} denote an injection from $\mathbb{Z}_n^* \to \{0,1\}^{2\lambda}$.

Next, following [GMR88], we define the claw-free pair of permutations $\pi_0(x) = x^2 \pmod N$ and $\pi_1(x) = 4x^2 \pmod N$, where the domain of π_0 and π_1 is the set of quadratic residues mod N.

Now we define the hash function

$$f : \{0,1\}^{2\lambda-4} \times \{0,1\}^5 \to \{0,1\}^{2\lambda}$$

$$f(x, y) = (\iota_{out} \circ \pi_{y_5} \circ \cdots \circ \pi_{y_1})(\iota_{in}(x)^2 \bmod N)$$

This is 64-bounded because for any given image, there is at most one pre-image under $\iota_{out} \circ \pi_{y_5} \circ \cdots \circ \pi_{y_1}$. This accounts for a factor of 32. The remaining factor of 2 comes from the fact that every quadratic residue has four square roots, two of which are in $[N/2]$ (the image of ι_{in}). The collision resistance of $x \mapsto \iota_{in}(x)^2$ follows from the fact that the two square roots are nontrivially related, i.e., neither is the negative of the other. $\square$

Notation. For a function $h : \{0,1\}^{\lambda+1} \to \{0,1\}^\lambda$, we let h^0 denote the identity function and for $k > 0$ inductively define

$$h^k : \{0,1\}^{\lambda+k} \to \{0,1\}^\lambda$$
$$h^k(x) = h(x_1 \| h^{k-1}(x_2 \| \cdots \| x_{\lambda+k}))$$

4 Adaptively Puncturable Hash Functions

We say that an ensemble $\mathcal{H}$ is *adaptively puncturable* if there are algorithms Verify, GenVK, and $\mathsf{ForceGenVK}$ such that:

Correctness

For all x,y,

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}, x, y) = 1 \iff y = h(x) \middle| \begin{array}{l} h \leftarrow \mathcal{H}_\lambda \\ \mathsf{vk} \leftarrow \mathsf{GenVK}(1^\lambda, h) \end{array}\right] = 1$$

Forced Verification

For all x, x^*,

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}, x, h(x^*)) = 1 \iff x = x^* \middle| \begin{array}{l} h \leftarrow \mathcal{H}_\lambda \\ \mathsf{vk} \leftarrow \mathsf{ForceGenVK}(1^\lambda, h, x^*) \end{array}\right] = 1$$

Indistinguishability

For all p.p.t. $\mathcal{A}_1$, $\mathcal{A}_2$

$$\Pr\left[\mathcal{A}_2(s, \mathsf{vk}_b) = b \middle| \begin{array}{l} h \leftarrow \mathcal{H}_\lambda \\ x^*, s \leftarrow \mathcal{A}_1(1^\lambda, h) \\ \mathsf{vk}_0 \leftarrow \mathsf{GenVK}(1^\lambda, h) \\ \mathsf{vk}_1 \leftarrow \mathsf{ForceGenVK}(1^\lambda, h, x^*) \\ b \leftarrow \{0,1\} \end{array}\right] \leq \frac{1}{2} + \mathrm{negl}(\lambda)$$

Theorem 4.1. If there is a poly(λ)-bounded CRHF ensemble mapping $\{0,1\}^{\lambda+1} \to \{0,1\}^\lambda$ and if $\mathsf{i}\mathcal{O}$ exists, then there is an adaptively puncturable hash function ensemble mapping $\{0,1\}^{2\lambda}$ to $\{0,1\}^\lambda$.

Let $\mathcal{H} = \{\mathcal{H}_\lambda\}$ be a poly(λ)-bounded CRHF ensemble, where $\mathcal{H}_\lambda$ is a family of functions mapping $\{0,1\}^{\lambda+1} \to \{0,1\}^\lambda$. We define an adaptively puncturable hash function ensemble $\mathcal{F} = \{\mathcal{F}_\lambda\}$, where $\mathcal{F}_\lambda$ is a family of functions mapping $\{0,1\}^{2\lambda} \to \{0,1\}^\lambda$.

Setup

The key space for $\mathcal{F}_\lambda$ is the same as the key space for $\mathcal{H}_\lambda$.

Evaluation

For a key $h \in \mathcal{H}_\lambda$ and a string $x \in \{0,1\}^{2\lambda}$, we define

$$f_h(x) = h^\lambda(x)$$

Verification

$\mathsf{GenVK}(1^\lambda, f_h)$ outputs an $\mathsf{i}\mathcal{O}$-obfuscation of a circuit which directly computes

$$x, y \mapsto \begin{cases} 1 & \text{if } f_h(x) = y \\ 0 & \text{otherwise} \end{cases}$$

$\mathsf{ForceGenVK}(1^\lambda, f_h, x^*)$ outputs an $\mathsf{i}\mathcal{O}$-obfuscation of a circuit which directly computes

$$x, y \mapsto \begin{cases} 1 & \text{if } y \neq f_h(x^*) \wedge y = f_h(x) \\ 1 & \text{if } (x, y) = (x^*, f_h(x^*)) \\ 0 & \text{otherwise} \end{cases}$$

$\mathsf{Verify}(\mathsf{vk}, x, y)$ simply evaluates and outputs $\mathsf{vk}(x, y)$.

Claim 4.1.1. No p.p.t. adversary which adaptively chooses x^* after seeing h can distinguish between $\mathsf{GenVK}(1^\lambda, h)$ and $\mathsf{ForceGenVK}(1^\lambda, h, x^*)$.

Proof. We present $\lambda+1$ hybrid games $H_0, \ldots, H_\lambda$. In each game h is sampled from $\mathcal{H}_\lambda$, but the circuit given by the challenger to the adversary depends on the game and on x^*. In hybrid H_i, the challenger computes $y^* = h^\lambda(x^*)$ and $y_{\lambda-i} = h^{\lambda-i}(x^*_{i+1}\|\cdots\|x^*_{2\lambda})$. The challenger then sends $\mathsf{i}\mathcal{O}(C_i)$ to the adversary, where C_i has y^*, $y_{\lambda-i}$, and $x^*_1, \ldots, x^*_i$ hard-coded and is defined as

$$C_i(x,y) = \begin{cases} 1 & \text{if } y = y^* \wedge x_1 = x^*_1 \wedge \cdots \wedge x_i = x^*_i \wedge h^{\lambda-i}(x_{i+1}\|\cdots\|x_{2\lambda}) = y_{\lambda-i} \\ 1 & \text{if } y \neq y^* \wedge y = h^\lambda(x) \\ 0 & \text{otherwise} \end{cases}$$

The challenger sends $\mathsf{i}\mathcal{O}(C_i)$ to the adversary.

It is easy to see that C_0 is functionally equivalent to the circuit produced by GenVK, and C_λ is functionally equivalent to the circuit produced by $\mathsf{ForceGenVK}$. So we only need to show that $H_i \approx H_{i+1}$ for $0 \leq i < \lambda$. We give a sequence of indistinguishable changes to the challenger, by which we transform the circuit given to the adversary from C_i to C_{i+1}.

1. We first modify C so that if $y \neq y^*$, it does the same as before. If $y = y^*$, it computes $y' = h^{\lambda-i-1}(x_{i+2}\|\cdots\|x_{2\lambda})$ and outputs 1 if:
   - $h(x_{i+1}\|y') = y_{\lambda-i}$
   - For all $1 \leq j \leq i$, $x_i = x^*_i$.

   . This change preserves functionality and hence is indistinguishable by $\mathsf{i}\mathcal{O}$.

2. Now we change C so that instead of directly checking whether $h(x_{i+1}\|y') = y_{\lambda-i}$, it uses a hard-coded helper circuit $\tilde{V} = \mathsf{i}\mathcal{O}(V)$, where

$$V : \{0,1\} \times \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}$$
$$V(a,b,c) = \begin{cases} 1 & \text{if } c = h(a\|b) \\ 0 & \text{otherwise} \end{cases}$$

   This is functionally equivalent and hence indistinguishable by $\mathsf{i}\mathcal{O}$.

3. Now we change V. We first compute $y_{\lambda-i-1} = h^{\lambda-i-1}(x^*_{i+2}\|\cdots\|x^*_{2\lambda})$ and $y_{\lambda-i} = h(x^*_{i+1}\|y_{\lambda-i-1})$, and define

$$V(a,b,c) = \begin{cases} 1 & \text{if } c \neq y_{\lambda-i} \wedge c = h(a\|b) \\ 1 & \text{if } (a,b,c) = (x^*_{i+1}, y_{\lambda-i-1}, y_{\lambda-i}) \\ 0 & \text{otherwise} \end{cases},$$

   with $y_{\lambda-i}$, $y_{\lambda-i-1}$, and x^*_{i+1} hard-coded. The old and new $\tilde{V}$'s are indistinguishable because:
   - By the collision-resistance of h, it is difficult to find an input on which they differ.
   - By the property of poly(λ)-bounded, they differ on only polynomially many points.
   - $\mathsf{i}\mathcal{O}$ is equivalent to $\mathsf{di}\mathcal{O}$ for circuits which differ on polynomially many points.

4. C is now functionally equivalent to C_{i+1} and hence is indistinguishable by $\mathsf{i}\mathcal{O}$.

$\square$

5 Adaptively Secure Positional Accumulators

Formally, an adaptive positional accumulator consists of the following polynomial-time algorithms. $\mathsf{SetupAcc}$, $\mathsf{SetupVerify}$, and $\mathsf{SetupEnforceVerify}$ are randomized, while Update and Verify are deterministic.

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

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

$\mathsf{Update}(\mathsf{PP}, \mathsf{store}, \mathsf{op}) \to \mathsf{store}', \mathsf{ac}', v, \pi$

The update algorithm takes as input the public parameters PP, a data store store, and a memory operation op. Update then outputs a new store store', a memory value v, a succinct accumulator ac', and a succinct proof π.

In our garbling scheme, the evaluator runs Update to process the memory operations made by garbled programs. The proof π is verified with respect to an accumulator ac which is a binding image to the memory configuration s represented by store. π proves (in a computationally sound way) that v is the result of executing op on s, and ac' is a commitment of the resulting memory configuration.

$\mathsf{Verify}(\mathsf{vk}, \mathsf{ac}, \mathsf{op}, \mathsf{ac}', v, \pi) \to \{0, 1\}$

The local update algorithm takes as inputs a verification key vk, an initial accumulator value ac, a memory operation op, a resulting accumulator ac', a memory value v, and a proof π. Verify then outputs 0 or 1. Intuitively, Verify checks the following statement:

> *π is a proof that the operation op, when applied to the memory configuration corresponding to ac, yields a value v and results in a memory configuration corresponding to ac'.*

Verify is run by a garbled program to authenticate the memory values that the evaluator gives it.

$\mathsf{SetupVerify}(\mathsf{PP}) \to \mathsf{vk}$

$\mathsf{SetupVerify}$ generates a regular verification key for checking Update's proofs. This is the verification key that is used in the "real world" garbled programs.

$\mathsf{SetupEnforceVerify}(\mathsf{PP}, (\mathsf{op}_1, \ldots, \mathsf{op}_k)) \to \mathsf{vk}$

$\mathsf{SetupEnforceVerify}$ takes a sequence of memory operations and a particular address, and generates a verification key which is perfectly sound when verifying the action of op_k in the sequence $(\mathsf{op}_1, \ldots, \mathsf{op}_k)$. This type of verification key is used in the hybrid garbled programs in our security proof.

An adaptive positional accumulator must satisfy the following properties.

Correctness

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

Let v_i^* denote the result of the i^{th} memory operation when $(\mathsf{op}_0, \ldots, \mathsf{op}_{k-1})$ are sequentially executed on an initially empty memory.

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

$$\Pr\left[v_j = v_j^* \wedge b_j = 1 \;\middle|\; \begin{array}{l} \mathsf{PP}, \mathsf{ac}_0, \mathsf{store}_0 \leftarrow \mathsf{SetupAcc}(1^\lambda, S) \\ \mathsf{vk} \leftarrow \mathsf{SetupVerify}(\mathsf{PP}) \\ \text{For } i = 0, \ldots, k: \\ \quad \mathsf{store}_{i+1}, \mathsf{ac}_{i+1}, v_i, \pi_i \leftarrow \mathsf{Update}(\mathsf{PP}, \mathsf{store}_i, \mathsf{op}_i) \\ \quad b_i \leftarrow \mathsf{Verify}(\mathsf{vk}, \mathsf{ac}_i, \mathsf{op}_i, \mathsf{ac}_{i+1}, v_i, \pi_i) \end{array} \right] = 1$$

Enforcing

Enforcing requires that for all space bounds S, all sequences of operations $\mathsf{op}_0, \ldots, \mathsf{op}_{k-1}$, all accumulators $\hat{\mathsf{ac}}$, all values $\hat{v}$, and all proofs $\hat{\pi}$, we have

$$\Pr\left[b = 1 \implies (\hat{v}, \hat{\mathsf{ac}}) = (v_{k-1}, \mathsf{ac}_k) \;\middle|\; \begin{array}{l} \mathsf{PP}, \mathsf{ac}_0, \mathsf{store}_0 \leftarrow \mathsf{SetupAcc}(1^\lambda, S) \\ \mathsf{vk} \leftarrow \mathsf{SetupEnforceVerify}(\mathsf{PP}, (\mathsf{op}_0, \ldots, \mathsf{op}_{k-1})) \\ \text{For } i = 0, \ldots, k-1 \\ \quad \mathsf{store}_{i+1}, \mathsf{ac}_{i+1}, v_i, \pi_i \leftarrow \mathsf{Update}(\mathsf{PP}, \mathsf{store}_i, \mathsf{op}_i) \\ b \leftarrow \mathsf{Verify}(\mathsf{PP}, \mathsf{ac}_{k-1}, \mathsf{op}_{k-1}, \hat{\mathsf{ac}}, \hat{v}, \hat{\pi}) \end{array} \right] = 1$$

Indistinguishability of Enforcing Verify

Now we require that the output of $\mathsf{SetupVerify}(\mathsf{PP})$ is indistinguishable from the output of $\mathsf{SetupEnforceVerify}(\mathsf{PP}, (\mathsf{op}_1, \ldots$ even when $(\mathsf{op}_1, \ldots, \mathsf{op}_k)$ are chosen adaptively as a function of PP.

More formally, for all p.p.t. $\mathcal{A}_1$ and $\mathcal{A}_2$,

$$\Pr\left[\mathcal{A}_2(s, \mathsf{vk}_b) = b \;\middle|\; \begin{array}{l} \mathsf{PP}, \mathsf{ac}_0, \mathsf{store}_0 \leftarrow \mathsf{SetupAcc}(1^\lambda, S) \\ (\mathsf{op}_0, \ldots, \mathsf{op}_{k-1}), s \leftarrow \mathcal{A}_1(1^\lambda, \mathsf{PP}) \\ \mathsf{vk}_0 \leftarrow \mathsf{SetupVerify}(\mathsf{PP}) \\ \mathsf{vk}_1 \leftarrow \mathsf{SetupEnforceVerify}(\mathsf{PP}, (\mathsf{op}_0, \ldots, \mathsf{op}_{k-1})) \\ b \leftarrow \{0,1\} \end{array} \right] \leq \frac{1}{2} + \operatorname{negl}(\lambda)$$

Efficiency

In addition to all the algorithms being polynomial-time, we require that:

- The size of an accumulator is $\operatorname{poly}(\lambda)$.
- The size of proofs is $\operatorname{poly}(\lambda, \log S)$.
- The size of a store is $O(S)$

Theorem 5.1. If there is an adaptively puncturable hash function ensemble $\mathcal{H} = \{\mathcal{H}_\lambda\}_{\lambda \in \mathbb{N}}$ with $\mathcal{H}_\lambda = \{H_k : \{0,1\}^{2\lambda} \to \{0,1\}^\lambda\}_{k \in \mathcal{K}_\lambda}$, then there exists an adaptive positional accumulator.

Proof. We construct an adaptive positional accumulator in which stores are low-depth binary trees, each node of which contains a λ-bit value. The accumulator corresponding to a given store is the value held by the root node. The public parameters for the accumulator consist of an adaptively puncturable hash $h : \{0,1\}^{2\lambda} \to \{0,1\}^\lambda$, and we preserve the invariant that the value in any internal node is equal to the hash h applied to its children's values. It will be convenient for us to assume the existence of a $\perp$, which is represented as a λ-bit string not in the image of h. Without loss of generality, h can be chosen to have such a value.

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

Setup samples $h \leftarrow \mathcal{H}_\lambda$, and sets $\mathsf{PP} = h$, $\mathsf{ac}_0 = h(\perp \| \perp)$, and store_0 to be a root node with value $h(\perp \| \perp)$.

$\mathsf{Update}(h, \mathsf{store}, \mathsf{op}) \to \mathsf{store}', \mathsf{ac}', v, \pi$

Suppose op is $\mathsf{ReadWrite}(\mathsf{addr} \mapsto v')$. There is a unique leaf node in store which is indexed by a prefix of addr. Let v be the value of that leaf, and let π be the values of all siblings on the path from the root to that leaf.

Update adds a leaf node indexed by the entirety of addr to store if no such node already exists, and sets the value of the leaf to v'. Then Update updates the value of ancestor of that leaf to preserve the invariant.

$\mathsf{SetupVerify}(h) \to \mathsf{vk}$

For $i = 1, \dots, \log S$, $\mathsf{SetupVerify}$ samples

$$vk_i \leftarrow \mathsf{GenVK}(1^\lambda, h)$$

and sets $\mathsf{vk} = (\mathsf{vk}_1, \dots, \mathsf{vk}_{\log S})$.

$\mathsf{Verify}((\mathsf{vk}_1, \dots, \mathsf{vk}_{\log S}), \mathsf{ac}, \mathsf{op}, \mathsf{ac}', v, (w_1, \dots, w_d)) \to \{0, 1\}$

Define $z_d := v$. Let $b_1 \cdots b_{d'}$ denote the bit representation of the address on which op acts. For $0 \le i < d$, Verify computes

$$z_i = \begin{cases} h(w_{i+1} \| z_{i+1}) & \text{if } b_{i+1} = 1 \\ h(z_{i+1} \| w_{i+1}) & \text{otherwise} \end{cases}$$

For all i such that $b_i = 1$, Verify checks that $\mathsf{vk}_i(w_{i+1} \| z_{i+1}, z_i) = 1$. For all i such that $b_i = 0$, Verify checks that $\mathsf{vk}_i(z_{i+1} \| w_{i+1}, z_i) = 1$. If all these checks pass, then Verify outputs 1; otherwise, Verify outputs 0.

$\mathsf{SetupEnforceVerify}(h, (\mathsf{op}_1, \dots, \mathsf{op}_k)) \to \mathsf{vk}$

Computes the store_{k-1} which would result from processing $\mathsf{op}_1, \dots, \mathsf{op}_{k-1}$. Suppose op_k accesses address $\mathsf{addr}_k \in \{0, 1\}^{\log S}$. Then there is a unique leaf node in store_{k-1} which is indexed by a prefix of addr_k; write this prefix as $b_1 \cdots b_d$.

For each $i \in \{1, \dots, d\}$, define z_i as the value of the node indexed by $b_1 \cdots b_i$, and let w_i denote the value of that node's sibling. If $b_i = 0$, sample

$$\mathsf{vk}_i \leftarrow \mathsf{ForceGenVK}(1^\lambda, h, z_i \| w_i).$$

Otherwise, sample

$$\mathsf{vk}_i \leftarrow \mathsf{ForceGenVK}(1^\lambda, h, w_i \| z_i).$$

For $i \in \{d + 1, \dots, \log S\}$, just sample $\mathsf{vk}_i \leftarrow \mathsf{GenVK}(1^\lambda, h)$.

Finally we define the total verification key to be $(\mathsf{vk}_1, \dots, \mathsf{vk}_{\log S})$.

All the requisite properties of this construction are easy to check. $\square$

6 Fixed-Transcript Garbling

We define fixed-transcript security via the following game.

1. The challenger samples $SK \leftarrow \mathsf{Setup}(1^\lambda, S)$ and $b \leftarrow \{0, 1\}$.
2. The adversary sends a memory configuration s to the challenger. The challenger sends back $\mathsf{GbMem}(SK, s)$.
3. The adversary repeatedly sends pairs of RAM programs (M_i^0, M_i^1) to the challenger, along with a time bound 1^{T_i}, and the challenger sends back $\tilde{M}_i^b \leftarrow \mathsf{GbPrg}(SK, M_i^b, T_i, i)$. Each pair (M_i^0, M_i^1) is chosen adaptively after seeing $\tilde{M}_{i-1}^b$.
4. The adversary outputs a guess b'.

Let $((M_1^0, M_1^1), \dots, (M_\ell^0, M_\ell^1))$ denote the sequence of pairs of machines output by the adversary. The adversary is said to win if $b' = b$ and:

- Sequentially executing $M_1^0, \dots, M_\ell^0$ on initial memory configuration s yields the same transcript as executing $M_1^1, \dots, M_\ell^1$.
- Each M_i^b runs in time at most T_i and space at most S.

Definition 6.1. A garbling scheme is *fixed-transcript secure* if for all p.p.t. algorithms $\mathcal{A}$, there is a negligible function negl so that $\mathcal{A}$'s probability of winning the game is at most $\frac{1}{2} + \text{negl}(\lambda)$.

Theorem 6.1. Assuming the existence of indistinguishability obfuscation and an adaptive positional accumulator, there is a fixed-transcript secure garbling scheme.

Proof. We give a construction and prove its security. As in [CH16], our construction follows that of [KLW15]. In addition to our adaptive positional accumulator, we use [KLW15]'s splittable signatures and cryptographic iterators, defined in Section 2.7 and 2.8.

$\mathsf{Setup}(1^\lambda, S)$ samples $\mathsf{Acc.PP} \leftarrow \mathsf{Acc.Setup}(1^\lambda, S)$ and samples a PPRF F.

$\mathsf{GbMem}(SK, s) \to \tilde{s}$ computes an accumulator ac_s corresponding to s, generates $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; F(0,0))$ and computes $\sigma_s \leftarrow \mathsf{Spl.Sign}(\mathsf{sk}, (\bot, \bot, \mathsf{ac}_s, \mathsf{ReadWrite}(0 \mapsto 0)))$. $\tilde{s}$ is then defined as a memory configuration which contains both $(\mathsf{ac}_s, \sigma_s)$ and store_0.

$\mathsf{GbPrg}(SK, M_i, T_i, i) \to \tilde{M}_i$ first transforms M_i so that its initial state is $\bot$. Note this can be done without loss of generality by hard-coding the "real" initial state in the transition function. GbPrg then computes $\tilde{C}_i \leftarrow \mathsf{i}\mathcal{O}(C_i)$, where C_i is described in Algorithm 1. Finally, we define $\tilde{M}_i$ not by its transition function, but by pseudocode, as the RAM machine which:

1. Reads $(\mathsf{ac}_0, \sigma_0)$ from memory (recall these were inserted under the names $(\mathsf{ac}_s, \sigma_s)$). Define $\mathsf{op}_0 = \mathsf{ReadWrite}(0 \mapsto 0)$, $q_0 = \bot$, and $\mathsf{itr}_0 = \bot$.
2. For $i = 0, 1, 2, \ldots$:
   (a) Compute $\mathsf{store}_{i+1}, \mathsf{ac}_{i+1}, v_i, \pi_i \leftarrow \mathsf{Acc.Update}(\mathsf{Acc.PP}, \mathsf{store}_i, \mathsf{op}_i)$.
   (b) Compute $\mathsf{out}_i \leftarrow \tilde{C}_i(i, q_i, \mathsf{itr}_i, \mathsf{ac}_i, \mathsf{op}_i, \sigma_i, v_i, \mathsf{ac}_{i+1}, \pi_i)$.
   (c) If out_i parses as (y, σ), then write $(\mathsf{ac}_{i+1}, \sigma)$ to memory, output y, and terminate.
   (d) Otherwise, out_i must parse as $(q_{i+1}, \mathsf{itr}_{i+1}, \mathsf{ac}_{i+1}, \mathsf{op}_{i+1}), \sigma_{i+1}$.

We note that $\tilde{M}_i$ can be compiled from $\tilde{C}_i$ and $\mathsf{Acc.PP}$. This means that later, when we prove security, it will suffice to analyze a game in which the adversary receives $\tilde{C}_i$ instead of $\tilde{M}_i$. This also justifies our relatively informal description of $\tilde{M}_i$.

Input: Time t, state q, iterator itr, accumulator ac, operation op, signature σ, memory value v, new accumulator ac', proof π

Data: Puncturable PRF F, RAM machine M_i with transition function δ_i, Accumulator verification key $\mathsf{vk}_{\mathsf{Acc}}$, index i, iterator public parameters $\mathsf{Itr.PP}$, time bound T_i

$(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; F(i, t))$;
if $t > T_i$ *or* $\mathsf{Spl.Verify}(\mathsf{vk}, (q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}), \sigma) = 0$ *or* $\mathsf{Acc.Verify}(\mathsf{vk}_{\mathsf{Acc}}, \mathsf{ac}, \mathsf{op}, \mathsf{ac}', v, \pi) = 0$ **then return** $\bot$;
$\mathsf{out} \leftarrow \delta_i(q, v)$;
if $\mathsf{out} \in Y$ **then**
  $(\mathsf{sk}', \mathsf{vk}') \leftarrow \mathsf{Spl.Setup}(1^\lambda; F(i+1, 0))$;
  return $\mathsf{out}, \mathsf{Sign}(\mathsf{sk}', (\bot, \bot, \mathsf{ac}', \mathsf{ReadWrite}(0 \mapsto 0))$
else
  Parse out as (q', op');
  $\mathsf{itr}' \leftarrow \mathsf{Itr.Iterate}(\mathsf{Itr.PP}, (q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}))$;
  $(\mathsf{sk}', \mathsf{vk}') \leftarrow \mathsf{Spl.Setup}(1^\lambda; F(i, t+1))$;
  return $(q', \mathsf{itr}', \mathsf{ac}', \mathsf{op}'), \mathsf{Sign}(\mathsf{sk}', (q', \mathsf{itr}', \mathsf{ac}', \mathsf{op}'))$

Algorithm 1: Transition function for M_i, with memory verified by a signed accumulator.

Correctness and efficiency are easy to verify.

We show that the challenger in the real security game is indistinguishable from one for which the adversary's view is independent of b. We present a sequence of hybrid games $H_0, \ldots, H_{\ell+1}$, and show that all p.p.t. algorithms $\mathcal{A}$ have negligibly different advantages in adjacent games. In each hybrid, the memory query is answered as in the real game.

Hybrid H_0 In hybrid H_0, we add a "B"-track for execution to C_i. Instead of just checking that $((q, \mathsf{op}, \mathsf{ac}, \mathsf{itr}), \sigma)$ is accepted under vk_t^A, we also allow it to be accepted under vk_t^B, which is derived from a different puncturable PRF F_B. In this second case, we proceed as before except that we compute with δ_i^0 instead of δ_i^b, and we sign the eventual outputs using sk_{t+1}^B instead of sk_{t+1}^A.

The indistinguishability of this change follows by $O(t)$ applications of the indistinguishability of punctured keys, together with the security of $i\mathcal{O}$. In particular, we can add any functionality we want (by IO) under an always-rejecting $\mathsf{vk}_{i,j,\varnothing}^B$ verification key, and then indistinguishably replace $vk_{i,j,\varnothing}^B$ with vk^B. We start by modifying the last time step, and work backwards because under $\mathsf{vk}_{i,j}^B$, we use the signing key $\mathsf{sk}_{i,j+1}^B$. By working backwards, we avoid the issue that $\mathsf{vk}_{i,j,\varnothing}^B$ is *not* indistinguishable from $\mathsf{vk}_{i,j}^B$ if also given $\mathsf{sk}_{i,j}^B$.

Hybrids H_i In hybrid H_i for $1 \le i \le \ell+1$, the first $i-1$ program queries are answered differently. For $1 \le j \le i-1$, the circuits C_j have hard-coded the transition function for M_j^0 instead of M_j^b. The challenger computes $s_j = \mathsf{NextMem}(M_j^0(s_{j-1}))$, and hard-codes the corresponding accumulator ac_{s_j} into the circuit C_j. The resulting circuit is illustrated in Algorithm 3.

It remains to show that $H_i \approx H_{i+1}$. This is shown using the techniques of [KLW15]. The main difference is that in our setting the positional accumulator needs to be adaptively secure.

1. We hard-code $\mathsf{vk}_{i,0}^A$ and $\mathsf{vk}_{i,0}^B$, and puncture F_A and F_B at $\{(i, 0)\}$. This change preserves functionality and is hence indistinguishable by $i\mathcal{O}$.

2. We replace $\mathsf{vk}_{i,0}^A$ and $\mathsf{vk}_{i,0}^B$ by keys punctured on the sets $\mathcal{M}\backslash\{(q_{i,0}, \mathsf{itr}_{i,0}, \mathsf{ac}_{i,0}, \mathsf{op}_{i,0})\}$ and $\{(q_{i,0}, \mathsf{itr}_{i,0}, \mathsf{ac}_{i,0}, \mathsf{op}_{i,0})\}$ respectively. These changes are indistinguishable by the (selective) indistinguishability of punctured keys.

3. The verification key for the accumulator $\mathsf{vk}_{\mathsf{Acc}}$ is generated by $\mathsf{SetupEnforceVerify}(\mathsf{PP}, (\mathsf{op}_{i,0}))$ instead of by $\mathsf{SetupVerify}(\mathsf{PP})$, so that if $\mathsf{Acc.Verify}(\mathsf{vk}_{\mathsf{Acc}}, \mathsf{ac}_{i,0}, \mathsf{op}_{i,0}, \mathsf{ac}', v, \pi) = 1$, then $ac' = \mathsf{ac}_{i,1}$ and $v = v_{i,0}$. This is indistinguishable by the positional accumulator's indistinguishability of enforcing setup. We note this holds even though $\mathsf{op}_{i,0}$ and $\mathsf{ac}_{i,0}$ may be chosen adversarially after observing the positional accumulator's public parameters.

4. At time 0, we use δ_i^0 instead of δ_i^b (on both tracks A and B). By the hypothesis that M_i^0 and M_i^1 have the same transcripts, we know that $\delta_i^0(q_{i,0}, v_{i,0}) = \delta_i^1(q_{i,0}, v_{i,0})$. Because in steps 2 and 3 we have already made our verification keys perfectly binding, this change is indistinguishable by $i\mathcal{O}$.

5. The verification key for the accumulator $\mathsf{vk}_{\mathsf{Acc}}$ is generated normally as $\mathsf{SetupVerify}(\mathsf{PP})$ instead of by $\mathsf{SetupEnforceVerify}(\mathsf{PP}, (\mathsf{op}_{i,0}))$. This is again indistinguishable by the positional accumulator's adaptively enforcing setup.

6. We modify C_i so that at time 0, instead of deciding to sign with $\mathsf{sk}_{i,1}^A$ or $\mathsf{sk}_{i,1}^B$ based on which branch we are in, we decide by looking at $(q, \mathsf{itr}, \mathsf{ac}, \mathsf{op})$. Namely, we use $\mathsf{sk}_{i,1}^A$ if and only if $(q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}) = (q_{i,0}, \mathsf{itr}_{i,0}, \mathsf{ac}_{i,0}, \mathsf{op}_{i,0})$. This is functionally equivalent because of how we have punctured the verification keys $\mathsf{vk}_{i,0}^A$ and $\mathsf{vk}_{i,0}^B$, and hence is indistinguishable by $i\mathcal{O}$. Note the 'A' branch and 'B' branch are now identical.

7. We generate $\mathsf{Itr.PP}$ using $\mathsf{SetupEnforce}$ so that $\mathsf{itr}' = \mathsf{itr}_{i,1}$ if and only if $(q, \mathsf{itr}, \mathsf{ac}, \mathsf{op})$ is equal to $(q_{i,0}, \mathsf{itr}_{i,0}, \mathsf{ac}_{i,0}, \mathsf{op}_{i,0})$. This change is indistinguishable by the iterator's (selective) setup indistinguishability.

8. Instead of choosing whether to use $\mathsf{sk}^A_{i,1}$ or $\mathsf{sk}^B_{i,1}$ based on the value of $(q, \mathsf{itr}, \mathsf{ac}, \mathsf{op})$, we choose based on the value of $(q', \mathsf{itr}', \mathsf{ac}', \mathsf{op}')$. This is functionally equivalent because itr' is equal to $\mathsf{itr}_{i,1}$ (and in fact $(q', \mathsf{ac}', \mathsf{op}')$ is equal to $(q_{i,1}, \mathsf{ac}_{i,1}, \mathsf{op}_{i,1})$) if and only if $(q, \mathsf{itr}, \mathsf{ac}, \mathsf{op})$ is equal to $(q_{i,0}, \mathsf{itr}_{i,0}, \mathsf{ac}_{i,0}, \mathsf{op}_{i,0})$, and therefore this change is indistinguishable by the security of $\mathsf{i}\mathcal{O}$.

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

10. Instead of checking whether the signature σ on $(q, \mathsf{ac}, \mathsf{itr})$ verifies under one of vk^A_0 (which is punctured at $\mathcal{M} \setminus \{(q_{i,0}, \mathsf{itr}_{i,0}, \mathsf{ac}_{i,0}, \mathsf{op}_{i,0})\}$) and vk^B_0 (which is punctured at $\{q_{i,0}, \mathsf{itr}_{i,0}, \mathsf{ac}_{i,0}, \mathsf{op}_{i,0}\}$), we only check that it verifies under the *unpunctured* $\mathsf{vk}^A_{i,0}$. This is indistinguishable by the splittable signature's splitting indistinguishability property.

11. We unpuncture F_A and F_B at $(i, 0)$ and un-hardcode $\mathsf{vk}^A_{i,0}$ and $\mathsf{vk}^B_{i,0}$. This is functionally equivalent and hence indistinguishable by $\mathsf{i}\mathcal{O}$.

12. We repeat steps 1 through 11 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 δ^0_i (M^0_i's transition function) to δ^1_i (M^1_i's transition function), starting at the beginning of the computation.

Input: Time t, state q, iterator itr, accumulator ac, operation op, signature σ, memory value v, new accumulator ac', proof π

Data: Puncturable PRFs F_A and F_B, RAM machine M_i with transition function δ_i, Accumulator verification key $\mathsf{vk}_{\mathsf{Acc}}$, index i, iterator public parameters $\mathsf{Itr.PP}$, time bound T_i

$(\mathsf{sk}_A, \mathsf{vk}_A) \leftarrow \mathsf{Spl.Setup}(1^\lambda; F_A(i, t))$;
$(\mathsf{sk}_B, \mathsf{vk}_B) \leftarrow \mathsf{Spl.Setup}(1^\lambda; F_B(i, t))$;
if $t > T_i$ *or* $\mathsf{Acc.Verify}(\mathsf{vk}_{\mathsf{Acc}}, \mathsf{ac}, \mathsf{op}, \mathsf{ac}', v, \pi) = 0$ **then return** $\perp$;
if $\mathsf{Spl.Verify}(\mathsf{vk}_A, (q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}), \sigma) = 1$ **then** track:='A';
else if $\mathsf{Spl.Verify}(\mathsf{vk}_B, (q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}), \sigma) = 1$ **then** track:='B';
else return $\perp$;
$\mathsf{out} \leftarrow \delta_i(q, v)$;
if $\mathsf{out} \in Y$ **then**
    $(\mathsf{sk}', \mathsf{vk}') \leftarrow \mathsf{Spl.Setup}(1^\lambda; F_{track}(i+1, 0))$;
    return $\mathsf{out}, \mathsf{Sign}(\mathsf{sk}', (\perp, \perp, \mathsf{ac}', \mathsf{ReadWrite}(0 \mapsto 0))$
else
    Parse out as (q', op');
    $\mathsf{itr}' \leftarrow \mathsf{Itr.Iterate}(\mathsf{Itr.PP}, (q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}))$;
    $(\mathsf{sk}', \mathsf{vk}') \leftarrow \mathsf{Spl.Setup}(1^\lambda; F_{track}(i, t+1))$;
    return $(q', \mathsf{itr}', \mathsf{ac}', \mathsf{op}'), \mathsf{Sign}(\mathsf{sk}', (q', \mathsf{itr}', \mathsf{ac}', \mathsf{op}'))$

Algorithm 2: Transition function for hybrid M_i, with memory verified by an accumulator.

□

7 Fixed-Access Garbling

Fixed-access security is defined in the same way as fixed-transcript security, but the left and right machines produced by $\mathcal{A}$ do not need to have the same transcripts for $\mathcal{A}$ to win - they may not have the same intermediate states, but only need to perform the same memory operations.

Definition 7.1 (Fixed-access security)**.** We define fixed-access security via the following game.

Input: Time t, state q, iterator itr, accumulator ac, operation op, signature σ, memory value v, new accumulator ac', proof π

Data: Puncturable PRFs F_A and F_B, RAM machine M_j^0 with transition function δ_j^0, Accumulator verification key $\mathsf{vk}_{\mathsf{Acc}}$, index i, iterator public parameters $\mathsf{Itr.PP}$, accumulator ac_{s_j}, time bound T_i.

$(\mathsf{sk}_A, \mathsf{vk}_A) \leftarrow \mathsf{Spl.Setup}(1^\lambda; F_A(i,t))$;
$(\mathsf{sk}_B, \mathsf{vk}_B) \leftarrow \mathsf{Spl.Setup}(1^\lambda; F_B(i,t))$;
if $t > T_i$ *or* $\mathsf{Acc.Verify}(\mathsf{vk}_{\mathsf{Acc}}, \mathsf{ac}, \mathsf{op}, \mathsf{ac}', v, \pi) = 0$ **then return** $\perp$;
if $\mathsf{Spl.Verify}(\mathsf{vk}_A, (q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}), \sigma) = 1$ **then** track:=‘A’;
else if $\mathsf{Spl.Verify}(\mathsf{vk}_B, (q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}), \sigma) = 1$ **then** track:=‘B’;
else return $\perp$;
$\mathsf{out} \leftarrow \delta_j^0(q, v)$;
if $\mathsf{out} \in Y$ **then**
  $(\mathsf{sk}', \mathsf{vk}') \leftarrow \mathsf{Spl.Setup}(1^\lambda; F_{track}(i+1, 0))$;
  return $\mathsf{out}, \mathsf{Sign}(\mathsf{sk}', (\perp, \perp, \mathsf{ac}_{s_j}, \mathsf{ReadWrite}(0 \mapsto 0))$
else
  Parse out as (q', op');
  $\mathsf{itr}' \leftarrow \mathsf{Itr.Iterate}(\mathsf{Itr.PP}, (q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}))$;
  $(\mathsf{sk}', \mathsf{vk}') \leftarrow \mathsf{Spl.Setup}(1^\lambda; F_{track}(i, t+1))$;
  return $(q', \mathsf{itr}', \mathsf{ac}', \mathsf{op}'), \mathsf{Sign}(\mathsf{sk}', (q', \mathsf{itr}', \mathsf{ac}', \mathsf{op}'))$

Algorithm 3: Response to j^{th} program query, with hard-coded final accumulator value.

1. The challenger samples $SK \leftarrow \mathsf{Setup}(1^\lambda, S)$ and $b \leftarrow \{0,1\}$.
2. The adversary sends a memory configuration s to the challenger. The challenger sends back $\mathsf{GbMem}(SK, s)$.
3. The adversary repeatedly sends pairs of RAM programs (M_i^0, M_i^1) to the challenger, together with a time bound 1^{T_i}, and the challenger sends back $\tilde{M}_i^b \leftarrow \mathsf{GbPrg}(SK, M_i^b, T_i, i)$. Each pair (M_i^0, M_i^1) is chosen adaptively after seeing $\tilde{M}_{i-1}^b$.
4. The adversary outputs a guess b'.

Let $((M_1^0, M_1^1), \ldots, (M_\ell^0, M_\ell^1))$ denote the sequence of pairs of machines output by the adversary. The adversary is said to win if $b' = b$ and:

- Sequentially executing $M_1^0, \ldots, M_\ell^0$ on initial memory configuration s yields the same transcript as executing $M_1^1, \ldots, M_\ell^1$, except that the local states can be different.
- Each M_i^b runs in time at most T_i and space at most S.

A garbling scheme is said to have fixed-access security if all p.p.t. adversaries $\mathcal{A}$ win in the game above with probability less than $1/2 + \mathrm{negl}(\lambda)$.

To achieve fixed-access security, we adapt the exact same technique from [CH16]: xoring the state with a pseudorandom function applied on the local time t. The PRF keys used in different machines are sampled independently.

Theorem 7.1. If there is a fixed-transcript garbling scheme, then there is a fixed-access garbling scheme.

Proof. Suppose $(\mathsf{Setup}', \mathsf{GbMem}', \mathsf{GbPrg}')$ is a fixed-transcript garbling scheme. We define and prove the security of a fixed-access garbling scheme $(\mathsf{Setup}, \mathsf{GbMem}, \mathsf{GbPrg})$.

$\mathsf{Setup}(1^\lambda, S)$ samples $SK' \leftarrow \mathsf{Setup}'(1^\lambda, S)$, sets it as SK.

$\mathsf{GbMem}(SK, s)$ outputs $\tilde{s}' \leftarrow \mathsf{GbMem}'(SK', s)$.

$\mathsf{GbPrg}(SK, M_i, T_i, i)$ samples a PPRF F_i, and outputs $\tilde{M}_i' \leftarrow \mathsf{GbPrg}'(SK', M_i', T_i, i)$, where M_i' is defined as in Algorithm 4. If M_i's initial state is q_0, the initial state of M_i' is $(0, q_0 \oplus F_i(0))$.

Input: State (t, c_q), memory symbol σ
Data: RAM machine M_i, puncturable PRF F_i
$q \leftarrow c_q \oplus F_i(t)$;
$\mathsf{out} \leftarrow M_i(q, \sigma)$;
if $\mathsf{out} \in Y$ **then return** out;
Parse out as (q', op);
return $((t+1, q' \oplus F_i(t+1)), \mathsf{op})$;

Algorithm 4: M_i', the modified version of M_i which encrypts its state.

We introduce hybrid games H_ℓ through H_0, starting with the real security game, and ending with one in which the adversary's view is independent of b. In hybrid H_i, the j^{th} query (M_i^0, M_i^1) is answered with $\tilde{M}_i^b$ if $j \leq i$ and $\tilde{M}_i^0$ otherwise. It remains to show that hybrid H_i is indistinguishable from H_{i+1}.

To show this, we introduce intermediate hybrids $\{H_{i,j}\}_{j=0,\ldots,T_i}$, each of which differs from H_i only in the answer to the i^{th} query. In $H_{i,j}$, the answer to the i^{th} machine query is answered by $\mathsf{GbPrg}'(SK', M_{i,j}', T_i, i)$, where the machine $M_{i,j}'$ is defined in Algorithm 5. Informally, $M_{i,j}'$ executes M_i^b for the first $T_i - j$ steps, and executes the next j steps with machine M_i^0.

Input: State (t, c_q), memory symbol σ
Data: RAM machines M_i^0, M_i^1, punctured PRF $F_i' = F_i\{T_i - j\}$, hard-coded state q^*, hard-coded ciphertext c^*, bit b
if $t = T_i - j$ **then** $q \leftarrow q^*$;
else $q \leftarrow c_q \oplus F_i'(t)$;
if $t < T_i - j$ **then** $M_i \leftarrow M_i^b$;
else $M_i \leftarrow M_i^0$;
$\mathsf{out} \leftarrow M_i(q, \sigma)$;
if $\mathsf{out} \in Y$ **then return** out;
Parse out as (q', op);
if $t = T_i - j - 1$ **then return** $((t+1, c^*), \mathsf{op})$;
else return $((t+1, q' \oplus F_i'(t+1)), \mathsf{op})$;

Algorithm 5: $M_{i,j}'$ executes M_i^b for $t_i - j$ steps, and then executes M_i^0.

Claim 7.1.1. $H_i \approx H_{i,0}$ and $H_{i-1} \approx H_{i,T_i}$.

Proof. This follows from the underlying fixed-transcript garbling. □

Claim 7.1.2. For every $j \in \{0, \ldots, T_i - 1\}$, $H_{i,j} \approx H_{i,j+1}$

Proof. We introduce another intermediate hybrid $H_{i,j,0}$, in which $c^* = q^1_{i,T_i-j} \oplus F_i(T_i - j)$. The indistinguishability of $H_{i,j}$ and $H_{i,j,0}$ follows from the pseudorandomness of the (selectively) puncturable PRF F_i on $T_i - j$. The indistinguishability of $H_{i,j,0}$ and $H_{i,j+1}$ follows from the underlying fixed-transcript garbling. So we have shown that $H_{i,j} \approx H_{i,j,0} \approx H_{i,j+1}$. □

The proof completes by combining the claims above. □

8 Fixed-Address Garbling

Fixed-address security is defined in the same way as fixed-access security, but the left and right machines produced by $\mathcal{A}$ do not need to make the same memory operations for $\mathcal{A}$ to win - their memory operations only need to access the same addresses. Additionally, the adversary $\mathcal{A}$ now provides not only a single memory configuration s_0, but two memory configurations s_0^0 and s_0^1. The challenger returns $\mathsf{GbMem}(SK, s_0^b)$. In keeping with the spirit of fixed-address garbling, we require s_0^0 and s_0^1 to have the same set of addresses storing non-ϵ values.

Definition 8.1 (Fixed-address security)**.** We define fixed-address security via the following game.

1. The challenger samples $SK \leftarrow \mathsf{Setup}(1^\lambda, S)$ and $b \leftarrow \{0,1\}$.
2. The adversary sends the initial memory configurations s_0^0, s_0^1 to the challenger. The challenger sends back $\tilde{s}_0^b \leftarrow \mathsf{GbMem}(SK, s_0^b)$.
3. The adversary repeatedly sends pairs of RAM programs (M_i^0, M_i^1) to the challenger, together with a time bound 1^{T_i}, and the challenger sends back $\tilde{M}_i^b \leftarrow \mathsf{GbPrg}(SK, M_i^b, T_i, i)$. Each pair (M_i^0, M_i^1) is chosen adaptively after seeing $\tilde{M}_{i-1}^b$.
4. The adversary outputs a guess b'.

Let $((s_0^0, s_0^1), (M_1^0, M_1^1), \ldots, (M_\ell^0, M_\ell^1))$ denote the sequence of pairs of memory configurations and machines output by the adversary. The adversary is said to win if $b' = b$ and:

- $\{a : s_0^0(a) \neq \epsilon\} = \{a : s_0^1(a) \neq \epsilon\}$.
- The sequence of addresses accessed and the outputs during the sequential execution of $M_1^0, \ldots, M_\ell^0$ on initial memory configuration s_0^0 is the same as from executing $M_1^1, \ldots, M_\ell^1$ on s_0^1.
- Each M_i^b runs in time at most T_i and space at most S.
- $|M_i^0| = |M_i^1|$, $i = 1, \ldots, \ell$.

A garbling scheme is said to have fixed-address security if all p.p.t. adversaries $\mathcal{A}$ win in the game above with probability less than $1/2 + \mathrm{negl}(\lambda)$.

Our construction of fixed-address garbling is almost the same with the two-track solution in [CH16], with a slight modification at the way to "encrypt" the memory configuration. In [CH16], the memory configurations are xored with different puncturable PRF values in the two tracks, where the PRFs are applied on the time t and address a. In this work, the PRFs are applied on the execution index i and time t, not on the address a. This is enough for our purpose, because in each execution index i and step t, the machine only writes on a single address (for the initial memory configuration, the index is assigned as 0, and different timestamps will be assigned on different addresses). By this modification, we are able to prove adaptive security based on selective secure puncturable PRF, and adaptively secure fixed-access garbling.

We note that, even if the address a is included in the domain of PRF, as in [CH16], the construction is still adaptively secure if the underlying PRF is based on GGM's tree construction. Here we choose to present the simplified version which suffices for our purpose.

Construction 8.1. Given a fixed-access garbling scheme $(\mathsf{Setup}', \mathsf{GbMem}', \mathsf{GbPrg}')$, we define a fixed-address garbling scheme $(\mathsf{Setup}, \mathsf{GbMem}, \mathsf{GbPrg})$:

$\mathsf{Setup}(1^\lambda)$ samples $SK' \leftarrow \mathsf{Setup}'(1^\lambda)$ and puncturable PRFs F_A and F_B.

$\mathsf{GbMem}(SK, s)$ outputs $\mathsf{GbMem}'(SK', s_0')$, where

$$s_0'(a) = \begin{cases} (0, -a, F_A(0, -a) \oplus s_0(a), F_B(0, -a) \oplus s_0(a)) & \text{if } s_0(a) \neq \epsilon \\ \epsilon & \text{otherwise} \end{cases}$$

$\mathsf{GbPrg}(SK, M_i, T_i, i)$ outputs $\mathsf{GbPrg}'(SK', M_i', T_i, i)$, where M_i' is defined as in Algorithm 6. If the initial state of M_i was q_0, the initial state of M_i' is $(0, q_0, q_0)$.

Input: State (t_q, q_A, q_B), memory symbol $(i_{in}, t_{in}, c_A, c_B)$
Data: RAM machine M_i, puncturable PRFs F_A, F_B
$\mathsf{out} \leftarrow M_i(q_A, F_A(i_{in}, t_{in}) \oplus c_A)$;
if $\mathsf{out} \in Y$ **then return** out;
Parse out as $(q', \mathsf{ReadWrite}(\mathsf{addr}' \mapsto v'))$;
$\mathsf{op}' := \mathsf{ReadWrite}(\mathsf{addr}' \mapsto (i, t_q, F_A(i, t_q) \oplus v', F_B(i, t_q) \oplus v')$;
return $(t_q + 1, q', q'), \mathsf{op}'$;

Algorithm 6: M_i': Modified version of M_i which encrypts its memory twice in parallel.

Theorem 8.2. If $(\mathsf{Setup}', \mathsf{GbMem}', \mathsf{GbPrg}')$ is a fixed-access garbling scheme, then Construction 8.1 is a fixed-address garbling scheme.

Proof. We give a sequence of hybrid games, starting with the real game H^b, and ending with one in which the adversary's view is independent of b. We show that the adversary's advantage differs negligibly in each pair of adjacent games. This will imply that in the real security game, all adversaries have advantage at most $1/2 + \mathrm{negl}(\lambda)$.

The hybrid structure follows closely from [CH16]. Purely for ease of informal exposition, we think of the machines $M_1^b, \ldots, M_\ell^b$ as being concatenated into one RAM machine $M = M^b$ with running time at most T. Recall that in our construction, if M^b would write v_t^b to address a at time t, then $\tilde{M}$ writes $(F_A(t) \oplus v_t^b, F_B(t))$ to a. Our hybrids make the following changes to the way in which the challenger generates $\tilde{M}$ and $\tilde{s}_0$:

1. $\tilde{s}_0$ is now defined as

$$\tilde{s}_0(a) = \begin{cases} (F_A(-a) \oplus s_0^b(a), F_B(-a) \oplus s_0^0(a)) & \text{if } s_0^b(a) \neq \epsilon \\ \epsilon & \text{otherwise} \end{cases}$$

   This is indistinguishable by the puncturable PRF security of F_B, because the contents on "B"-track are not decrypted at all in the real garbled program.

2. Let $v_1^b, \ldots, v_T^b$ denote the values that M^b would write when executed on s^b. For $i = 1, \ldots, T$, we have a hybrid in which:

   - On timesteps $t < i$, $\tilde{M}$ writes $(F_A(t) \oplus v_t^b, F_B(t) \oplus v_t^0)$.
   - On subsequent timesteps, $\tilde{M}$ writes $(F_A(t) \oplus v_t^b, F_B(t))$.

   Here, the addresses which $\tilde{M}$ accesses are determined by the implicit internal execution of M^b.

   These hybrids are indistinguishable by puncturable PRF security together with fixed-access security: one can freely puncture F_B at i and hard-code its value because no other point in the computation uses $F_B(i)$.

3. Now that M^b and M^0 are both being implicitly executed in parallel, we determine where $\tilde{M}$ writes by following M^0. This is indistinguishable by fixed-access security because M^0 and M^b access the same addresses.

4. Symmetrically to step 1, we define another sequence of hybrids for $i = T, \ldots, 1$, in which:

   - On timesteps $t < i$, $\tilde{M}$ writes $(F_A(t) \oplus v_t^b, F_B(t) \oplus v_t^0)$.
   - On subsequent timesteps, $\tilde{M}$ writes $(F_A(t), F_B(t) \oplus v_t^0)$.

5. Finally, we remove M^b from $\tilde{M}$ altogether. As it is no longer used, this change is indistinguishable by security of the fixed-access garbling scheme. Thus, in this hybrid the adversary's view is independent of b.

Formally we define the hybrids with full exposition.

Hybrids $H_{i,z,b,0}$ In $H_{i,z,b,0}$, where the subscripts represent the execution index $i \in \{1, 2, ..., \ell\}$, timestep $z \in \{0, 1, ..., T_i\}$, initial memory configuration on track-"A" and "B" being the encryption of s_0^b and s_0^0.

1. The challenger samples $SK' \leftarrow \mathsf{Setup}'(1^\lambda, S)$ and $b \leftarrow \{0, 1\}$.
2. The adversary sends the initial memory configurations s_0^0, s_0^1 to the challenger. The challenger sends back $\tilde{s}'_{0,0,b,0} \leftarrow \mathsf{GbMem}'(SK', s'_{0,0,b,0})$, where $s'_{0,0,b,0}(a)$ is constructed as:

$$s'_{0,0,b,0}(a) = \begin{cases} (0, -a, F_A(0, -a) \oplus s_0^b(a), F_B(0, -a) \oplus s_0^0(a)) & \text{if } s_0^b(a) \neq \epsilon \\ \epsilon & \text{otherwise} \end{cases}$$

3. The adversary sends pairs of RAM programs $(M_1^0, M_1^1), \ldots, (M_\ell^0, M_\ell^1)$ to the challenger, each pair chosen adaptively after seeing the garbling of previous programs. In the RAM machine $M'_{i,z,b,0}$ defined by algorithm 7, for the first z steps, the resulting memory configurations of M_i^b evaluated on s_{i-1}^b are written on track A, those of M_i^0 evaluated on s_{i-1}^0 are written on track B; for the next $T_i - z$ steps, the resulting memory configuration of M_i^b on s_{i-1}^b is written on both tracks.

   The response of the challenger is the fixed-access garbling of machine $M'_{j,z,b,0}$, set up in different ways depending on the relation of j and i:

   (a) For $j \in \{1, \ldots, i-1\}$, the challenger sends back $\tilde{M}'_{j,T_1,b,0} \leftarrow \mathsf{GbPrg}'(SK, M'_{j,T_1,b,0}, j)$;

   (b) For $j = i$, the challenger sends back $\tilde{M}'_{i,z,b,0} \leftarrow \mathsf{GbPrg}'(SK, M'_{i,z,b,0}, i)$;

   (c) For $j \in \{i+1, \ldots, \ell\}$, the challenger sends back $\tilde{M}'_{j,0,b,0} \leftarrow \mathsf{GbPrg}'(SK, M'_{j,0,b,0}, j)$.
4. The adversary outputs a guess b'.

Input: State (t_q, q_A, q_B), memory symbol $(i_{in}, t_{in}, c_A, c_B)$
Data: i, z, RAMs M_i^b, M_i^0, PPRFs F_A, F_B

$\mathsf{out}_A \leftarrow M_i^b(q_A, F_A(i_{in}, t_{in}) \oplus c_A)$;
if $\mathsf{out}_A \in Y$ **then return** out_A;
Parse out_A as $(q'_A, \mathsf{ReadWrite}(\mathsf{addr}' \mapsto v'_A))$;
if $t_q < z$ **then**
  $\mathsf{out}_B \leftarrow M_i^0(q_B, F_B(i_{in}, t_{in}) \oplus c_B)$;
  Parse out_B as $(q'_B, \mathsf{ReadWrite}(\mathsf{addr}' \mapsto v'_B))$;
  $\mathsf{op}' := \mathsf{ReadWrite}(\mathsf{addr}' \mapsto (i, t_q, F_A(i, t_q) \oplus v'_A, F_B(i, t_q) \oplus v'_B)$;
else
  $\mathsf{op}' := \mathsf{ReadWrite}(\mathsf{addr}' \mapsto (i, t_q, F_A(i, t_q) \oplus v'_A, F_B(i, t_q) \oplus v'_A)$;
return $(t_q + 1, q'_A, q'_B), \mathsf{op}'$;

Algorithm 7: $M'_{i,z,b,0}$

Lemma 8.3. $H^b \approx H_{1,0,b,0}$.

Proof. The initial memory configurations in H^b and $H_{0,0,b,0}$ differ in the "B"-track. Because $M_1'^b$ and $M'_{1,0,b,0}$ don't "decrypt" the contents in the "B"-track, $s^b(a) \oplus F_B(0,-a)$ and $s^0(a) \oplus F_B(0,-a)$ are indistinguishable by the pseudorandomness of F_B on $(0,-a)$, $a \in \{a : s^0(a) \neq \epsilon\}$.

The RAM machine $M_1'^b$ and $M'_{1,0,b,0}$ have the exact same functionality and are accessing the same memory configuration, so the garbling of them are indistinguishable following the fixed-access security. □

Lemma 8.4. For $i \in \{2, ..., \ell\}$, $H_{i-1,T_{i-1},b,0} \approx H_{i,0,b,0}$.

Proof. This follows directly from the underlying fixed-access security. □

Lemma 8.5. For $i \in \{1, 2, ..., \ell\}$, $z \in \{0, 1, ..., T_i - 1\}$, $H_{i,z,b,0} \approx H_{i,z+1,b,0}$.

Proof. For each i and z, we introduce one more intermediate hybrid $H_{i,z,b,0,0}$, where the adversary receives

$$\tilde{s}'_{0,0,b,0},\ \tilde{M}'_{1,T_1,b,0},\ ...,\ \tilde{M}'_{i-1,T_{i-1},b,0},\ \tilde{M}'_{i,z,b,0,0},\ \tilde{M}'_{i+1,0,b,0},\ ...,\ \tilde{M}'_{\ell,0,b,0}.$$

The RAM machine $M'_{i,z,b,0,0}$ is defined by algorithm 8. The hard-coded ciphertext c^* in $H_{i,z,b,0,0}$ is $F_B(i,z) \oplus v'_B$. The difference of $H_{i,z,b,0}$ and $H_{i,z,b,0,0}$ are

1. The i^{th} RAM machine $M'_{i,z,b,0}$ versus $M'_{i,z,b,0,0}$.
2. In the other RAM machines, F_B is also punctured on (i,z). Note that this won't change the functionality of $M'_{1,T_1,b,0}$, ..., $M'_{i-1,T_{i-1},b,0}$, $M'_{i+1,0,b,0}$, ..., $M'_{\ell,0,b,0}$, since the first $i-1$ machines won't read or write with index i, and the last $\ell - i$ ones won't read "B"-track or write with index i.

Input: State (t_q, q_A, q_B), memory symbol $(i_{in}, t_{in}, c_A, c_B)$
Data: i, z, RAMs M_i^b, M_i^0, PPRFs F_A, $F'_B = F_B\{i,z\}$, ciphertext c^*.
$\mathsf{out}_A \leftarrow M_i^b(q_A, F_A(i_{in}, t_{in}) \oplus c_A)$;
if $\mathsf{out}_A \in Y$ **then return** out_A;
Parse out_A as $(q'_A, \mathsf{ReadWrite}(\mathsf{addr}' \mapsto v'_A))$;
if $t_q < z$ **then**
  $\mathsf{out}_B \leftarrow M_i^0(q_B, F'_B(i_{in}, t_{in}) \oplus c_B)$;
  Parse out_B as $(q'_B, \mathsf{ReadWrite}(\mathsf{addr}' \mapsto v'_B))$;
  $\mathsf{op}' := \mathsf{ReadWrite}(\mathsf{addr}' \mapsto (i, t_q, F_A(i,t_q) \oplus v'_A, F'_B(i,t_q) \oplus v'_B)$;
else if $t_q = z$ **then**
  $\mathsf{out}_B \leftarrow M_i^0(q_B, F'_B(i_{in}, t_{in}) \oplus c_B)$;
  Parse out_B as $(q'_B, \mathsf{ReadWrite}(\mathsf{addr}' \mapsto v'_B))$;
  $\mathsf{op}' := \mathsf{ReadWrite}(\mathsf{addr}' \mapsto (i, t_q, F_A(i,t_q) \oplus v'_A, c^*)$;
else
  $\mathsf{op}' := \mathsf{ReadWrite}(\mathsf{addr}' \mapsto (i, t_q, F_A(i,t_q) \oplus v'_A, F'_B(i,t_q) \oplus v'_A)$;
return $(t_q + 1, q'_A, q'_B), \mathsf{op}'$;

Algorithm 8: $M'_{i,z,b,0,0}$.

Note that $H_{i,z,b,0,0} \approx H_{i,z+1,b,0}$ by the underlying fixed-access garbling. If we define $c^* = F_B(i,z) \oplus v'_A$, the RAM machines has the same functionality with those in hybrid $H_{i,z,b,0}$. By the pseudorandomness of the punctured PRF F_B on (i,z), $H_{i,z,b,0,0} \approx H_{i,z,b,0}$. This shows that $H_{i,z,b,0} \approx H_{i,z,b,0,0} \approx H_{i,z+1,b,0}$. □

Combining Lemma 8.3, 8.4 and 8.5, we obtain that $H^b \approx H_{1,0,b,0} \approx ... \approx H_{\ell,0,b,0} \approx H_{\ell,T_\ell,b,0}$.

The rest of the proof can be done symmetrically: First, instead of returning out_A, return out_B, by the underlying fixed-access garbling. Then switch the computation on "A"-track from running M^b on s^b into running M^0 on s^0, and prove the indistinguishability of them analogously via the puncturability F_A and the underlying fixed-access security. Finally b is not in the view of the adversary.

□

9 Full Garbling

In order to construct a fully secure garbling scheme, we will need to make use of an oblivious RAM (ORAM) [GO96] to hide the addresses accessed by the machine.

9.1 Oblivious RAMs

An ORAM is a probabilistic scheme for memory storage and access that provides obliviousness for access patterns with sublinear access complexity. It is convenient for us to model an ORAM scheme as follows. We define a deterministic algorithm OProg so that for a security parameter 1^λ, a memory operation op, and a space bound S, $\mathsf{OProg}(1^\lambda, \mathsf{op}, S)$ outputs a probabilistic RAM machine M_{op}. More generally, for a RAM machine M, we can define $\mathsf{OProg}(1^\lambda, M, S)$ as one which executes $\mathsf{OProg}(1^\lambda, \mathsf{op}, S)$ for every operation op output by M.

We also define OMem, a procedure for making a memory configuration oblivious, in terms of OProg, as follows: Given a memory configuration s with n non-empty addresses $a_1, \ldots, a_n$, all less than or equal to a space bound S, $\mathsf{OMem}(1^\lambda, s, S)$ iteratively samples

$$s'_0 \leftarrow \epsilon^{\mathbb{N}}$$

and

$$s'_i = \mathsf{NextMem}(\mathsf{OProg}(1^\lambda, \mathsf{ReadWrite}(a_i \mapsto s(a_i)), S), s'_{i-1})$$

and outputs s'_n.

Correctness An ORAM is said to be correct if for all memory operations $\mathsf{op}_1, \ldots, \mathsf{op}_\ell$ accessing addresses less than or equal to S, it holds with high probability that

$$(M_{\mathsf{op}_1}; \ldots; M_{\mathsf{op}_\ell})(\epsilon^{\mathbb{N}}) = (\mathsf{op}_1; \ldots; \mathsf{op}_\ell)(\epsilon^{\mathbb{N}})$$

That is, when one sequentially executes $M_{\mathsf{op}_1}, \ldots, M_{\mathsf{op}_\ell}$ on the initially empty memory, M_{op_ℓ} outputs the same result as op_n when executing $\mathsf{op}_1, \ldots, \mathsf{op}_\ell$ from the initially empty memory.

Efficiency An ORAM is said to have multiplicative space overhead $\zeta : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ if for all memory operations op accessing an address less than or equal to a space bound S, and for all memory configurations s, it holds with probability 1 that

$$\mathsf{Space}(M_{\mathsf{op}}, s) \leq \zeta(S, \lambda) \cdot S$$

An ORAM is said to have multiplicative time overhead $\eta : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ if for all memory operations op and all memory configurations s, it holds with probability 1 that

$$\mathsf{Time}(M_{\mathsf{op}}, s) = \eta(S, \lambda)$$

Security (Strong Localized Randomness). We now define a notion of strong localized randomness[1] for an ORAM, which is satisfied by the ORAM construction of [CP13].

Informally, we consider obliviously executing operations $\mathsf{op}_1, \ldots, \mathsf{op}_t$ on a memory of size S, i.e. executing machines $M_{\mathsf{op}_1}; \ldots; M_{\mathsf{op}_t}$ using a random tape $R \in \{0,1\}^{\mathbb{N}}$. This yields a sequence of addresses $\vec{A} = \vec{a}_1 \| \cdots \| \vec{a}_t$. There should be a natural way to decompose each $\vec{a}_i$ (in the Chung-Pass ORAM, we consider each recursive level of the construction) such that we can write $\vec{a}_i = \vec{a}_{i,1} \| \cdots \| \vec{a}_{i,m}$. Our notion of strong localized randomness requires that (after having fixed $\mathsf{op}_1, \ldots, \mathsf{op}_t$), each $\vec{a}_{i,j}$ depends on some small substring of R, which does not influence any other $\vec{a}_{i',j'}$. In other words:

- There is some $\alpha_{i,j}, \beta_{i,j} \in \mathbb{N}$ such that $0 < \beta_{i,j} - \alpha_{i,j} \leq \mathrm{poly}(\log S)$ and such that $\vec{a}_{i,j}$ is a function of $R_{\alpha_{i,j}}, \ldots, R_{\beta_{i,j}}$.
- The collection of intervals $[\alpha_{i,j}, \beta_{i,j})$ for $i \in \{1, \ldots, t\}$, $j \in \{1, \ldots, m\}$ is pairwise disjoint.

[1]This notion is similar but stronger to the "localized randomness" defined in [CH16]

Formally, we say that an ORAM with multiplicative time overhead η has strong localized randomness if:

- For all λ and S, there exists m and $\tau_1 < \tau_2 < \cdots < \tau_m$ with $\tau_1 = 1$ and $\tau_m = \eta(S, \lambda) + 1$, and there exist circuits $C_1, \ldots, C_m$, such that for all memory operations $\mathsf{op}_1, \ldots, \mathsf{op}_t$, there exist pairwise disjoint intervals $I_1, \ldots, I_m \subset \mathbb{N}$ such that:
  - If we write
$$\vec{A}_1 \| \cdots \| \vec{A}_t \leftarrow \mathsf{addr}(M_{\mathsf{op}_1}^{R_1}; \ldots; M_{\mathsf{op}_t}^{R_t}, \epsilon^{\mathbb{N}})$$
where $R = R_1 \| \cdots \| R_t$ denotes the randomness used by the oblivious accesses and each $\vec{A}_i$ denotes the addresses accessed by $M_{\mathsf{op}_i}^{R_i}$, then $(\vec{A}_t)_{[\tau_j, \tau_{j+1})} = C_j(R_{I_j})$ with high probability over R. Here R_{I_j} denotes the contiguous substring of R indexed by the interval $I_j \subset [|R|]$.
  - With high probability over the choice of $R_{\mathbb{N} \setminus I_j}$, $\vec{A}_1, \ldots, \vec{A}_{t-1}$ does not depend on R_{I_j} as a function.
- τ_j and the circuits C_j are computable in polynomial time given 1^λ, S, and j.
- I_j is computable in polynomial time given 1^λ, S, $\mathsf{op}_1, \ldots, \mathsf{op}_t$, and j.

Definition 9.1. An S-ORAM is an ORAM where correctness, efficiency, and security need hold only if the space bound is at most S.

Claim 9.0.1. For any c, there is a c-ORAM with multiplicative space overhead of 1 and multiplicative time overhead of c.

Proof. This is just the brute-force ORAM which accesses the entire memory for each underlying memory operation. Since this ORAM is deterministic, the localized randomness property is trivial. ☐

Claim 9.0.2. There is an ORAM with polylogarithmic time and space overhead and localized randomness.

Proof. Suppose we have an S-ORAM with multiplicative space overhead ζ and time overhead η. We show how to build a $2S$-ORAM with multiplicative space overhead $\zeta'(N) = \zeta(N) + \text{poly}(\log N, \lambda)$, and multiplicative time overhead $\eta'(N) = \eta(N) + \text{poly}(\log N, \lambda)$. Our base case will be the brute-force ORAM.

Next we construct an ORAM with strong localized randomness.

Construction 9.1 (Chung-Pass)**.** Given a memory operation $\mathsf{op} = \mathsf{ReadWrite}(a \mapsto v')$ with alphabet Σ, we construct a RAM machine M_{op}, which we describe with pseudocode:

M_{op}'s view of memory has two parts:

- A memory $\mathcal{M}$ with $\zeta(S) \cdot S$ addresses, which we think of as a smaller ORAM.
- A complete binary tree $\mathcal{T}$ of depth $\log S$. Each node in $\mathcal{T}$ is a bucket with capacity sufficient to hold λ tuples of the form $(\mathsf{addr}, \mathsf{pos}, \sigma_0, \sigma_1) \in [S] \times [S] \times \Sigma \times \Sigma$.

M_{op} does the following:

1. Samples a random position pos', and executes $\mathsf{pos} \leftarrow \mathsf{OProg}(1^\lambda, \mathsf{ReadWrite}(\lfloor \frac{a}{2} \rfloor \mapsto \mathsf{pos}'), S)$ on $\mathcal{M}$.
2. On the path from the root of $\mathcal{T}$ to the pos^{th} leaf, M_{op} searches for a tuple $(\mathsf{addr}, \mathsf{pos}, \sigma_0, \sigma_1)$ such that $\mathsf{addr} = \lfloor \frac{a}{2} \rfloor$. If such a tuple is found, M_{op} records σ_0 and σ_1, and then overwrites the tuple with $\perp$.
3. Adds a tuple $(\lfloor \frac{a}{2} \rfloor, \mathsf{pos}', \sigma_0', \sigma_1')$ to the root bucket, where
$$\sigma_b' = \begin{cases} v' & \text{if } b \equiv a \pmod 2 \\ \sigma_b & \text{otherwise} \end{cases}$$
4. Traverses a path from the root to a random leaf of T, moving every tuple $(\mathsf{addr}, \mathsf{pos}, \sigma_0, \sigma_1)$ to the deepest node on the path which is a prefix of pos.

5. Returns $\sigma_{a \bmod 2}$.

Correctness and efficiency of Construction 9.1 are easy to see, assuming the following lemma, which is proved in [CP13].

Lemma 9.2. For all memory operations $\mathsf{op}_1, \ldots, \mathsf{op}_t$, with high probability, Construction 9.1 will not exceed the capacity of any of its buckets.

Claim 9.2.1. Construction 9.1 has strong localized randomness.

Proof. We show how each of the chunks of addresses accessed by Construction 9.1 are functions of prior contiguous chunks of randomnesses.

- In step 1, we access the addresses that the S-ORAM would access; hence we get localized randomness for free.
- In step 2, we access all nodes on the path to pos, where pos was retrieved from the S-ORAM in step 1. pos was chosen at random and written to the S-ORAM on the last time that M accessed a (or a' with $\lfloor \frac{a'}{2} \rfloor = \lfloor \frac{a}{2} \rfloor$).
- In step 4, we simply choose a fresh random path and access all of the nodes on that path.

□

□

Remark 1. A more usual definition of obliviousness requires that if two machines M_0 and M_1 have the same running time, then the addresses accessed by $\mathsf{OProg}(M_0)$ and $\mathsf{OProg}(M_1)$ will be statistically close. Although it is not immediately clear, our definition of strong localized randomness in fact implies this definition.

9.2 Full Garbling Construction

Theorem 9.3. If there is an efficient fixed-address garbling scheme, then there is an efficient full garbling scheme.

Proof. Suppose we are given a fixed-address garbling scheme $(\mathsf{Setup}', \mathsf{GbMem}', \mathsf{GbPrg}')$ and an oblivious RAM OProg with space overhead ζ and time overhead η. We construct a full garbling scheme $(\mathsf{Setup}, \mathsf{GbMem}, \mathsf{GbPrg})$.

$\mathsf{Setup}(1^\lambda, T, S)$ samples $SK' \leftarrow \mathsf{Setup}'(1^\lambda, \eta(S, \lambda) \cdot T, \zeta(S, \lambda) \cdot S)$ and samples a PPRF $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^{\ell_R}$, where ℓ_R is the length of randomness needed to obliviously execute one memory operation. We will sometimes think of the domain of F as $[2^{2\lambda}]$.

$\mathsf{GbMem}(SK, s_0)$ outputs $\mathsf{GbMem}'(SK', \mathsf{OMem}(1^\lambda, s_0, S))$.

$\mathsf{GbPrg}(SK, M_i, i)$ outputs $\mathsf{GbPrg}'(SK', \mathsf{OProg}(1^\lambda, M_i, S)^{F(i, \cdot)}, i)$.

Simulator To show security of this construction, we define the following simulator.

1. The adversary provides S, and an initial memory configuration s_0. Say that s_0 has n non-ϵ addresses. The simulator samples $SK' \leftarrow \mathsf{Setup}'(1^\lambda, \zeta(S, \lambda) \cdot S)$ and sends $\mathsf{GbMem}'(SK', \mathsf{OMem}(1^\lambda, 0^n, S))$ to the adversary.
2. When the adversary makes a query $M_i, 1^{T_i}$, the simulator computes $s_i = \mathsf{NextMem}(M_i, s_{i-1})$, $t_i = \mathsf{Time}(M_i, s_{i-1})$, and $y_i = M_i(s_{i-1})$, and outputs $\mathsf{GbPrg}'(SK', D_i, \eta(S, \lambda) \cdot T_i, i)$, where D_i is a "dummy program". As described in Algorithm 9, D_i independently samples addresses to access for t_i steps, and then outputs y_i.

```
Data: Underlying running time t_i, output value y_i, PPRF G_i, circuits C_1, ..., C_m guaranteed by
      localized randomness
for t = 1, ..., t_i do
    for k = 1, ..., m do
        r_k ← G_i(t, k);
        Access addresses given by C_k(r_k)
return y_i.
```

Algorithm 9: Pseudocode for a dummy RAM machine which simulates pseudorandom addresses to access using the circuits $C_1, \ldots, C_m$ given in the definition of localized randomness, and then outputs y_i.

The rest of this section is devoted to proving that this simulator is indistinguishable from the real challenger.

Hybrid $H_{i,j}$ We show indistinguishability by giving a sequence of "hybrid" challengers $H_{i,j}$ for $i = 1, \ldots, \ell$ and $j = 1, \ldots, T$, and show that they are all indistinguishable. In hybrid $H_{i,j}$, the challenger:

- Answers the memory query s_0 with $\mathsf{GbMem}'(SK', \mathsf{OMem}(1^\lambda, s_0, S))$ as in the real game.
- For $k < i$, the k^{th} query $M_k, 1^{T_k}$ is answered with $\mathsf{GbPrg}'(SK', \mathsf{OProg}(1^\lambda, M_k, S), \eta(S, \lambda) \cdot T_k, k)$, just as in the real game.
- The i^{th} query $M_i, 1^{T_i}$ is answered with $\mathsf{GbPrg}(SK', N_{i,j}, \eta(S, \lambda) \cdot T_i, i)$, where N_i is a RAM machine which acts like M_i for the first j underlying steps, and acts like D_i for the rest of the steps. $N_{i,j}$ is described more precisely in Algorithm 10.
- For $k > i$, the k^{th} query $M_k, 1^{T_k}$ is answered with $\mathsf{GbPrg}'(SK', D_k, \eta(S, \lambda) \cdot T_k, k)$ for a dummy program D_k, just as in the simulator.

```
Data: RAM machine M_i, Underlying running time t_i, output value y_i, PPRFs F and G_i
op := ReadWrite(0 ↦ 0);
for t = 1, ..., j do
    Execute OProg(op)^{F(i,t)}, yielding a result v;
    Run one step of M_i with memory input v, yielding a new value for op;
for t = j + 1, ..., t_i do
    for k = 1, ..., m do
        r_k := G_i(t, k);
        Access addresses given by C_k(r_k)
return y_i.
```

Algorithm 10: Pseudocode for a RAM machine $N_{i,j}$ which starts acting like a dummy machine after j steps.

$H_{\ell+1,0}$ is identical to the real world, so it remains to show the following three claims:

Claim 9.3.1. $H_{i,T} \approx H_{i+1,0}$.

Proof. This follows directly from fixed-address security, because the semi-dummy machine $N_{i,0}$ accesses the same addresses and has the same output as the dummy machine D_i. $\square$

Claim 9.3.2. $H_{1,0}$ is indistinguishable from the simulator.

Proof. This follows from fixed-address security, and from the fact that the set of non-empty addresses in $\mathsf{OMem}(1^\lambda, s, S)$ is simulatable given $\|s\|_0$. $\square$

Our main claim is the following:

Claim 9.3.3. $H_{i,j} \approx H_{i,j+1}$.

Proof. Recall the definition of an ORAM with strong localized randomness. The addresses accessed in the oblivious exeution of $\mathsf{op}_{i,j}$ consist of m different chunks $\vec{a}_1, \ldots, \vec{a}_m$. Each $\vec{a}_k$ depends on some contiguous substring of the random tape R, indexed by an interval I_k, via a circuit C_k. The interval I_k depends on the underlying operations being executed.

We present $m+1$ hybrids $H_{i,j,m}$ through $H_{i,j,0}$. In hybrid $H_{i,j,k}$, the addresses $\vec{a}_1, \ldots, \vec{a}_k$ are generated honestly, and addresses $\vec{a}_{k+1}, \ldots, \vec{a}_m$ are simulated as $C_{k+1}(r_{k+1}), \ldots, C_m(r_m)$ for pseudorandomly chosen $r_{k+1}, \ldots, r_m$.

We prove that no adversary $\mathcal{A}$ can distinguish between $H_{i,j,k}$ and $H_{i,j,k-1}$ if $\mathcal{A}$ first commits to $2\log(T \cdot \ell_R)$ bits about what it is going to do. Specifically, we suppose that $\mathcal{A}$ initially sends I_k (which depends on the machines $M_1, \ldots, M_i$). In this case, we can show the indistinguishability of $H_{i,j,k}$ and $H_{i,j,k-1}$ by making a sequence of indistinguishable changes.

1. The puncturable PRF F sampled during Setup is punctured at I_k, and has the values $F(I_k)$ hard-coded in all machines. This is indistinguishable because of fixed-address security of the underlying garbling scheme.

2. The machine M_i' has $\vec{a}_k$, the addresses accessed in the k^{th} chunk of M_i's j^{th} operation, hard-coded. This also is indistinguishable because of fixed-address security.

3. The hard-coded values $F(I_k)$ are replaced by truly random values r_k, and $\vec{a}_k$ is replaced by $C_k(r_k)$. This is indistinguishable by the security of the punctured PRF F.

4. r_k is replaced by $F(I_k)$ and F is unpunctured. By localized randomness properties – namely, no other $\vec{A}_i$ depends on R_{I_k} and $I_1, \ldots, I_m$ are pairwise disjoint – this doesn't affect the addresses accessed by $M_1', \ldots, M_i'$. So this is indistinguishable by fixed-address security.

5. $C_k(r_k)$ is replaced by $C_k(G_i(j,k))$. This is indistinguishable by the puncturable PRF security of G_i.

It suffices to analyze this semi-selective game because (by a usual complexity leveraging technique) if no adversary has advantage ϵ in this game, then no adversary has advantage $\epsilon' = \epsilon/(T\ell_R)^2$ in distinguishing $H_{i,j,k}$ from $H_{i,j,k-1}$. Since T, ℓ_R, and S are polynomial in the security parameter, if ϵ is negligible then ϵ' is as well.

$\square$

$\square$

10 Database delegation

We define security for the task of delegating a database to an untrusted server. Here we have a database owner that wishes to keep the database on a remote server. Over time, the owner wishes to update the database and query it. Furthermore, the owner wishes to enable other parties to do so as well, perhaps under some restrictions. Informally, the security requirements from the scheme are:

Verifiability: The data owner should be able to verify the correctness of the answers to its queries, relative to the up-to-date state of the database following all the updates made so far.

Secrecy of database and queries: For queries made by the database owner and honest third parties, the adversary does not learn anything other than the size of the database, the sizes and runtimes of the queries, and the sizes of the answers. This holds even if the answers to the queries become partially or fully known by other means.

For queries made by adversarially controlled third parties, the adversary learns in addition only the answers to the queries.

(We note that these two secrecy requirement are incomparable and complementary. In the case of honest third parties the adversary does not see the processing done on the receiver side, but then she should not learn anything. In the case of dishonest third parties the adversary sees all the computation involved in the evaluation of the query, but then the answer is not protected. This distinction will become clearer in the actual definition.)

More precisely, a database delegation scheme consists of the following algorithms:

$\mathsf{DBDelegate}$**:** Initial delegation of the database. Takes as input a plaintext database, and outputs an encrypted database (to be sent to the server), public verification key vk and private master key msk to be kept secret.

Query**:** Delegation of a query or database update. Takes a RAM program and the master secret key msk, and outputs a delegated program to be sent to the server and a secret key $\mathsf{sk_{enc}}$ that allows recovering the result of the evaluation from the returned response.

Eval**:** Evaluation of a query or update. Takes a delegated database $\tilde{D}$ and a delegated program $\tilde{M}$, runs $\tilde{M}$ on $\tilde{D}$. Returns a response value a and an updated database $\tilde{D}'$.

$\mathsf{AnsDecVer}$**:** Local processing of the server's answer. Takes the public verification key vk, the private decryption key $\mathsf{sk_{enc}}$ and outputs either an answer value or $\perp$.

Security. Essentially, the security requirement is that the scheme UC-emulates the *database delegation* ideal functionality $\mathcal{F}_{dd}$ defined as follows. (For simplicity, it is assumed that the database owner is uncorrupted.)

1. When activated for the first time, $\mathcal{F}_{dd}$ expects to obtain from the activating party (the database owner) a database D. It then records D and discloses $|D|$ to the adversary.
2. In each subsequent activation by the owner, that specifies a program M and party P, run M on D, obtain an answer a and a modified database D', store D' and disclose P and the length of a to the adversary. If the adversary returns $\mathtt{ok}$ then output (M, a) to P.

To make the requirements imposed by $\mathcal{F}_{dd}$ more explicit, we also formulate the definition in terms of a distinguishability game. Specifically, we require that there exists a simulator Sim such that no adversary (environment) $\mathcal{A}$ will be able to distinguish whether it is interacting with the real or the ideal games as described here:

Real game $REAL_{\mathcal{A}}(1^{\lambda})$**:**

1. $\mathcal{A}$ provides a database D, receives the public outputs of $\mathsf{DBDelegate}(D)$.
2. $\mathcal{A}$ repeatedly provides a program M_i and a bit that indicates either *honest* or *dishonest*. In response, Query is run to obtain $\mathsf{sk}^i_{\mathsf{enc}}$ and $\tilde{M}_i$. $\mathcal{A}$ obtains $\tilde{M}_i$, and in the dishonest case also the decryption key $\mathsf{sk}^i_{\mathsf{enc}}$.
3. In the honest case $\mathcal{A}$ provides the server's output out_i for the execution of M_i, and obtains in response the result of $\mathsf{AnsDecVer}(\mathsf{vk}, \mathsf{sk_{enc}}, \mathsf{out}_i)$.

Ideal game $IDEAL_{\mathcal{A}}(1^\lambda)$:

1. $\mathcal{A}$ provides a database D, receives the output of $\mathsf{Sim}(|D|)$.
2. $\mathcal{A}$ repeatedly provides a program M_i and either *honest* or *dishonest*. In response, M_i runs on the current state of the database D to obtain output a and modified database D'. D' is stored instead of D. In the case of dishonest, $\mathcal{A}$ obtains $\mathsf{Sim}(a, s, t)$, where s is the description size of M and t is the runtime of M. In the case of honest, $\mathcal{A}$ obtains $\mathsf{Sim}(s, t)$.
3. In the honest case $\mathcal{A}$ provides the server's output out_i for the execution of M_i, and obtains in response $\mathsf{Sim}(out_i)$, where here $\mathsf{Sim}(out_i)$ can take one out of only two values: either a or $\perp$.

Definition 10.1. We say that a delegation scheme $(\mathsf{DBDelegate}, \mathsf{Query}, \mathsf{Eval}, \mathsf{AnsDecVer})$ is secure if there exists a simulator Sim such that no $\mathcal{A}$ can guess with non-negligible advantage whether it is interacting in the real interaction or in the ideal interaction with Sim.

Theorem 10.1. If there exist adaptive succinct garbled RAM schemes with persistent memory, unforgeable signature schemes and symmetric encryption schemes with pseudorandom ciphertexts, then there exist secure database delegation schemes with succinct queries and efficient delegation, query preparation, query evaluation, and response verification.

Proof. Let $(\mathsf{Setup}, \mathsf{GbMem}, \mathsf{GbPrg})$ be an adaptively secure garbling scheme for RAM with persistent memory. We construct a database delegation scheme as follows:

$\mathsf{DBDelegate}(1^\lambda)$**:** Run $SK \leftarrow \mathsf{Setup}(1^\lambda, D)$ and $\tilde{D} \leftarrow \mathsf{GbMem}(SK, D, |D|)$. Generate signing and verification keys $(\mathsf{vk}_{\mathsf{sign}}, \mathsf{sk}_{\mathsf{sign}})$ for the signature scheme. Set $\mathsf{msk} \leftarrow (SK, \mathsf{sk}_{\mathsf{sign}})$ and $\mathsf{vk} \leftarrow \mathsf{vk}_{\mathsf{sign}}$.

$\mathsf{Query}(M_i, \mathsf{msk}, \mathsf{pk})$**:** Generate a symmetric encryption key $\mathsf{sk}_{\mathsf{enc}}$. Generate the extended version of M'_i of M_i as in Algorithm 11.
Output $\tilde{M} \leftarrow \mathsf{GbPrg}(SK, M'_i[\mathsf{sk}_{\mathsf{sign}}, \mathsf{sk}_{\mathsf{enc}}], i)$

Input: State q, memory value v
Data: RAM program M_i with transition function δ_i and output space Y, and signing and encryption keys $\mathsf{sk}_{\mathsf{sign}}$, $\mathsf{sk}_{\mathsf{enc}}$

$\mathsf{out} \leftarrow \delta_i(q, v)$;
if $\mathsf{out} \in Y$ **then**
  $\mathsf{ct}_{\mathsf{out}} \leftarrow \mathsf{Enc}(\mathsf{sk}_{\mathsf{enc}}, \mathsf{out})$
  $\sigma_{\mathsf{out}} \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathsf{sign}}, \mathsf{ct}_{\mathsf{out}} \| i)$
  return $(\mathsf{ct}_{\mathsf{out}}, \sigma_{\mathsf{out}})$;
return out

Algorithm 11: M'_i: modified version of M_i which encrypts and signs its final output

Eval**:** Run $\tilde{M}$ on $\tilde{D}$ and return the output value a and an updated database $\tilde{D}'$.

$\mathsf{AnsDecVer}(i, \mathsf{out}, \mathsf{vk}, \mathsf{sk})$**:** Parse $\mathsf{out} = (\mathsf{ct}, \sigma)$. If $\mathsf{Verify}(\mathsf{vk}, \mathsf{ct}\|i, \sigma) \neq 1$, output $\perp$. Else output $\mathsf{Dec}(\mathsf{sk}, \mathsf{ct})$.

We construct a simulator Sim for the delegation scheme as follows:

$\mathsf{DBDelegate}$: Sim generates signing and verifications keys $\mathsf{sk}_{\mathsf{sign}}, \mathsf{vk}_{\mathsf{sign}}$. Sim runs the simulator $\mathsf{Sim}_{\mathsf{GRAM}}$ for a GRAM scheme to obtain a simulated garbled database $\tilde{D}$. It provides $\tilde{D}$ and $\mathsf{vk}_{\mathsf{sign}}$ as output to the adversary $\mathcal{A}$.

Query: If Sim is executed with inputs (a, s, t) on the i-th iteration, it generates symmetric encryption key $\mathsf{sk}_{\mathsf{enc}}$. It computes $\mathsf{ct} = \mathsf{Enc}(\mathsf{sk}_{\mathsf{enc}}, a)$, $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathsf{sign}}, \mathsf{ct} \mid i)$ and runs the simulator $\mathsf{Sim}_{\mathsf{GRAM}}$ with inputs $(\mathsf{ct} \mid i, \sigma)$ to obtain simulated garbled RAM $\tilde{M}_i$. It returns $\tilde{M}_i$ and $\mathsf{sk}_{\mathsf{enc}}$ to $\mathcal{A}$.

If Sim is executed with inputs (s, t) on the i-th iteration, it generates a random value ct, computes $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathsf{sign}}, \mathsf{ct} \mid i)$ and runs the simulator $\mathsf{Sim}_{\mathsf{GRAM}}$ with inputs $(\mathsf{ct} \mid i, \sigma)$ to obtain simulated garbled RAM $\tilde{M}_i$. It returns $\tilde{M}_i$ to $\mathcal{A}$.

AnsDecVer: If Sim executes on input out_i then it outputs $\mathsf{AnsDecVer}(\mathsf{vk}, \mathsf{sk}_{\mathsf{enc}}, \mathsf{out}_i)$.

To show validity of Sim, We construct the following hybrids.

$\mathbf{H_0}$: This is the real world execution.

$\mathbf{H_1}$: In this hybrid we start using the simulator for the GRAM $\mathsf{Sim}_{\mathsf{GRAM}}$ to generate simulated database $\tilde{D}'$. We generate the signature scheme keys $(\mathsf{vk}_{\mathsf{sign}}, \mathsf{sk}_{\mathsf{sign}})$ honestly. We also use $\mathsf{Sim}_{\mathsf{GRAM}}$ to generate the garbling for the programs M_i' given inputs $\mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}_{\mathsf{enc}}, \mathsf{out}) \mid i$, $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathsf{sign}}, \mathsf{ct}_i)$ and out is the result of the evaluation of M_i with the memory state after the previous $i-1$ evaluations.

The indistinguishability of $\mathbf{H_0}$ and $\mathbf{H_1}$ follows from the simulation security of the GRAM scheme.

$\mathbf{H_2}$: In this hybrid for all honest executions for machines M_i where the adversary $\mathcal{A}$ does not get $\mathsf{sk}_{\mathsf{enc}}$, we run $\mathsf{Sim}_{\mathsf{GRAM}}$ to generate the garbling for the programs M_i' with inputs $\mathsf{ct}_i \leftarrow r$, where r is a random value, and $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathsf{sign}}, \mathsf{ct}_i \| i)$.

The indistinguishability of $\mathbf{H_1}$ and $\mathbf{H_2}$ follows from the pseudorandom property of symmetric encryption ciphertexts.

Now, consider the event where, in execution $\mathbf{H_2}$, the adversary provides a value out_i such that $\mathsf{AnsDecVer}(\mathsf{vk}, \mathsf{sk}_{\mathsf{enc}}, \mathsf{out}_i) = a'$ and $a \neq a' \neq \perp$, where a is the correct answer for the i-th query in this execution. We argue that:

- Conditioned on this event not happening, $\mathcal{A}$'s view of $\mathbf{H_2}$ is identical to its view in the ideal interaction.
- The event happens with at most negligible probability. Otherwise $\mathcal{A}$ can be used to break the unforgeability of the signature scheme. To see this consider an interaction between $\mathcal{A}$ and Sim that is the same as $\mathbf{H_2}$ except that Sim queries the signature scheme challenger $\mathcal{C}$ to obtain verification key $\mathsf{vk}_{\mathsf{sign}}$ and signatures σ_i for the values ct_i. Then out_i, which $\mathcal{A}$ returns, contains a signature of a message that Sim has not queried. Hence, Sim breaks the unforgeability property of the signature scheme.

□

11 Reusable GRAM with Persistent Memory

Our basic construction of adaptively secure garbled RAM can naturally support program/input reusability. That is, the garbler is able to garble the RAM machines or the inputs once, and reuse them for many times. In the previous works [GKP+13, GHRW14], reusability is viewed as a security feature. The techniques are built up in order to reuse the resulting garbled program.

In this work, we are not trying to reuse the garbled program. Instead, we take the advantage of the persistent memory, and simply store the plaintext code of the RAM program/input into the memory. To evaluate, garble an universal RAM machine that executes the specific machine and input in the memory. Since the size of universal RAM machine is independent of the sizes of the input and program it takes [CR73], the size of garbled universal RAM is only dependent on the security parameter. So the reusability we achieve is essentially an efficiency feature, which shares the same spirit with previous works but instantiated in a way that is closer to the real world scenario.

Definition 11.1 (Reusable garbled RAM)**.** A garbling scheme is said to be *reusable* if the total time of garbling the program M to be reused for d times is $O(|M|, \text{poly}(\lambda)) + d \cdot \text{poly}(\lambda)$; the total time of garbling the input x to be reused for d times is $O(|x|, \text{poly}(\lambda)) + d \cdot \text{poly}(\lambda)$.

Our construction use the basic adaptively secure garbled scheme $(\mathsf{Setup}', \mathsf{GbMem}', \mathsf{GbPrg}')$ as a black-box. Without loss of generality, we store the plaintext code of machines and inputs under an index system $\mathcal{I} = (\mathcal{I}.\mathsf{setup}, \mathcal{I}.\mathsf{store}, \mathcal{I}.\mathsf{fetch})$, where $\mathcal{I}.\mathsf{setup}$ outputs initialization parameter ι, $\mathcal{I}.\mathsf{store}(i, z)$ stores z under index i, $\mathcal{I}.\mathsf{fetch}(i)$ outputs the content stored at index i. There is a data structure where the size of $\mathsf{store}(i, z)$ and $\mathsf{fetch}(i)$ are $\log(|i|)$, and they run in time $\log(|i|) + |z|$.

Construction 11.1 (Reusable garbled RAM)**.** Let $(\mathsf{Setup}', \mathsf{GbMem}', \mathsf{GbPrg}')$ be an adaptively secure garbled RAM with persistent memory, we construct one that is reusable $(\mathsf{Setup}, \mathsf{GbMem}, \mathsf{GbInp}, \mathsf{GbPrg}, \mathsf{GbExe})$ as follows:

- $\mathsf{Setup}(1^\lambda, S)$ runs $SK' \leftarrow \mathsf{Setup}(1^\lambda, S)$, $\iota \leftarrow \mathcal{I}.\mathsf{setup}$.
- $\mathsf{GbMem}(SK, \perp, S)$ by default, runs $\tilde{s}'_0 \leftarrow \mathsf{GbMem}'(SK', \perp, S)$ to garble an empty memory for initialization.
- $\mathsf{GbInp}(SK, x_i, i)$ runs $\tilde{W}'_i \leftarrow \mathsf{GbPrg}'(SK', W_{x,i}, i)$, where the functionality of $W_{x,i}$ is "$\mathcal{I}.\mathsf{store}(i, x_i)$, outputs $\perp$." Then evaluate $\tilde{W}'_i$ on memory $\tilde{s}'_{i-1}$.
- $\mathsf{GbPrg}(SK, M_j, j)$ runs $\tilde{W}'_j \leftarrow \mathsf{GbPrg}'(SK', W_{M,j}, j)$, where the functionality of $W_{M,j}$ is "$\mathcal{I}.\mathsf{store}(j, M_j)$, outputs $\perp$." Then evaluate $\tilde{W}'_j$ on memory $\tilde{s}'_{j-1}$.
- $\mathsf{GbExe}(SK, i, j, g)$ runs $\tilde{U}'_{i,j,g} \leftarrow \mathsf{GbPrg}'(SK', U_{i,j}, g)$, evaluate $\tilde{U}'_{i,j,g}$ on the garbled memory configuration $\tilde{s}'_{g-1}$. The functionality of $U_{i,j}$ is "$U(\mathcal{I}.\mathsf{fetch}(j), \mathcal{I}.\mathsf{fetch}(i))$", where U a universal RAM machine.

Correctness, (simulation-based) adaptive security, persistence of the memory follows directly from the basic scheme. The time cost of garbling a program or input is the same with the basic scheme. That is, the running time of the garbler is linear w.r.t. the size of the program or input. The total size of garbled programs produced to reuse a machine M for d times is

$$|\mathsf{GbPrg}(SK, M_j, j)| + |\mathsf{GbExe}(SK, i, j, g)| = O(|M|, \text{poly}(\lambda)) + d \cdot \text{poly}(\lambda)$$

The total size of garbled programs produced to reuse an input x for d times is

$$|\mathsf{GbPrg}(SK, x_i, i)| + |\mathsf{GbExe}(SK, i, j, g)| = O(|x|, \text{poly}(\lambda)) + d \cdot \text{poly}(\lambda)$$

The definition and construction can be easily generalized to the setting where a universal RAM machine takes more than 1 RAM machine, 1 input (cf. the decomposable garbling studied in [AIK11, AIKW15]).

Acknowledgments

We would like to thank Oxana Poburinnaya. Although she refused to co-author this paper, her constructive suggestions and criticisms played an essential role throughout the creation of this work.

References

[ABG+13] Prabhanjan Ananth, Dan Boneh, Sanjam Garg, Amit Sahai, and Mark Zhandry. Differing-inputs obfuscation and applications. *IACR Cryptology ePrint Archive*, 2013:689, 2013.

[ABSV15] Prabhanjan Ananth, Zvika Brakerski, Gil Segev, and Vinod Vaikuntanathan. From selective to adaptive security in functional encryption. In *CRYPTO*, 2015.

[AIK11] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. How to garble arithmetic circuits. In *FOCS*, 2011.

[AIKW15] Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. Encoding functions with constant online rate, or how to compress garbled circuit keys. *SIAM J. Comput.*, 44(2):433–466, 2015.

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In *STOC*, pages 99–108, 1996.

[AS15] Prabhanjan Ananth and Amit Sahai. Functional encryption for turing machines. *IACR Cryptology ePrint Archive*, 2015:776, 2015.

[BCP14] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. In *TCC*, pages 52–73, 2014.

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

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *PKC*, pages 501–519, 2014.

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

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *ASIACRYPT*, pages 134–153, 2012.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *ASIACRYPT*, pages 280–300, 2013.

[CCC+15] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Computation-trace indistinguishability obfuscation and its applications. *IACR Cryptology ePrint Archive*, 2015.

[CH16] Ran Canetti and Justin Holmgren. Fully succinct garbled ram. In *ITCS (to be appeared)*, 2016.

[CHJV14] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Indistinguishability obfuscation of iterated circuits and ram programs. Cryptology ePrint Archive, Report 2014/769, 2014.

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for ram programs. In Ronitt Rubinfeld, editor, *STOC*, 2015.

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

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

[Dam88] Ivan Bjerre Damgård. Collision free hash functions and public key signature schemes. In *EUROCRYPT*, pages 203–216, 1988.

[GGH96] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Collision-free hashing from lattice problems. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 3, pages 236–241, 1996.

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

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

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

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private ram computation. In *FOCS*, 2014.

[GKP$^+$13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *STOC*, pages 555–564, 2013.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In *Proceedings of the 28th Annual Conference on Cryptology: Advances in Cryptology*, CRYPTO 2008, 2008.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled ram from one-way functions. In *STOC*, 2015.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

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

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *STOC*, 2008.

[HW15] Pavel Hubáček and Daniel Wichs. On the communication complexity of secure function evaluation with long output. ITCS, 2015.

[IK00] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *FOCS*, pages 294–304, 2000.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In *STOC*, 2015.

[KMR14] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. Flexor: Flexible garbling for XOR gates that beats free-xor. In *CRYPTO*, pages 440–457, 2014.

[KP15] Yael Tauman Kalai and Omer Paneth. Delegating ram computations. Cryptology ePrint Archive, Report 2015/957, 2015.

[KPTZ13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *ACM CCS*, pages 669–684, 2013.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, pages 486–498, 2008.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble ram programs? In *EUROCRYPT*. 2013.

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *TCC*, pages 329–346, 2011.

[MR07] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on Gaussian measure. *SIAM Journal on Computing*, 37(1):267–302, 2007.

[MR13] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In *CRYPTO*, pages 36–53, 2013.

[OPWW15] Tatsuaki Okamoto, Krzysztof Pietrzak, Brent Waters, and Daniel Wichs. New realizations of somewhere statistically binding hashing and positional accumulators. *IACR Cryptology ePrint Archive*, 2015:869, 2015.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), 2009.

[Rog91] Phillip Rogaway. *The round complexity of secure protocols*. PhD thesis, Massachusetts Institute of Technology, 1991.

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

[Wat15] Brent Waters. A punctured programming approach to adaptively secure functional encryption. In *CRYPTO*, pages 678–697, 2015.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, 1986.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, pages 220–250, 2015.