Adaptive Succinct Garbled RAM
or: How To Delegate Your Database[⋆]

Ran Canetti[1,2], Yilei Chen[1], Justin Holmgren[3], and Mariana Raykova[4]

[1] Boston University
{canetti,chenyl}@bu.edu
[2] Tel Aviv University & CPIIS
[3] MIT
holmgren@mit.edu
[4] Yale University & SRI
mariana.raykova@yale.edu

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. The garbled database and programs reveal only the outputs of the programs when run in sequence on the database. Still, 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 somewhat-regular collision-resistant hash functions. 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, we give the first scheme for efficiently outsourcing a large database and computations on the database to an untrusted server, then delegating computations on this database, where these computations may update the database.
Our scheme extends the non-adaptive RAM garbling scheme of Canetti and Holmgren [ITCS 2016]. We also define and use a new primitive of independent interest, called adaptive accumulators. The primitive extends the positional accumulators of Koppula et al [STOC 2015] and somewhere statistical binding hashing of Hubáček and Wichs [ITCS 2015] to an adaptive setting.

1 Introduction

Database delegation. Alice is embarking on a groundbreaking experiment that involves collecting huge amounts of data over several months and then querying and running analytics on the data in ways to be determined as the data accumulates. Alas she does not have sufficient storage and processing power. Eve, who runs a large competing lab, offers servers for rent, but charges proportionally to storage and computing time. Can Alice make use of Eve's offer while being

[⋆] This paper was presented jointly with "Delegating RAM Computations with Adaptive Soundness and Privacy" by Prabhanjan Ananth, Yu-Chi Chen, Kai-Min Chung, Huijia Lin and Wei-Kai Lin.

guaranteed that Eve does not learn or modify Alice's data and algorithms? Can she do it at a cost that's reasonably proportional to the size of the actual data and resource requirements?

The rich literature on verifiable delegation of computation, e.g. [20,12,25,31,4] [34,24], provides Alice with ways to guarantee the correctness of the results on her weak machines, while paying Eve only a relatively moderate cost. In particular, with [24] the cost is proportional only to the unprotected database size, the complexity of her unprotected queries and the security parameter. However, these schemes do not provide secrecy for Alice's data and computations. Searchable encryption schemes such as [6,32,26] provide varying levels of secrecy at a reasonable cost, but no verifiability.

So Alice turns to delegation schemes based on garbling. Such schemes, starting with [16], can indeed provide both verifiability and privacy. Here the client garbles its input and program (along with some authentication information) and hands them to the server, who evaluates the garbled program on the garbled input and returns the result to the client. In Alice's case the garbling scheme should be *persistent,* namely it should be possible to garble multiple programs that operate on the same garbled data, possibly updating the data over time. Alice would also like the scheme to be *succinct,* in the sense that the overhead of garbling each new query should be proportional to the description size of that query as a RAM program, independently of on the size of the database. Furthermore, the evaluation process should be *efficiency preserving,* namely it should preserve the RAM efficiency of the underlying computation.

Gennaro et al. [16] use the original Yao circuit-garbling scheme [35,33], which is neither succinct, efficiency-preserving, nor persistent. [29,17,15,18] describe garbling schemes that operate on persistent memory, improve on efficiency, but haven't yet achieved succinctness. Succinct, efficiency-preserving and persistent garbling schemes, based on indistinguishability obfuscation for circuits and one way functions are constructed in [9,11], building on techniques from [5,10,28].

However, the security of these schemes is only analyzed in a static setting, where all the queries and data updates are fixed beforehand. Given the dynamic and on-going character of Alice's research, a static guarantee is hardly adequate. Instead, Alice needs to consider a setting where new queries and updates may depend on the public information released so far. The dependence may be arbitrary and potentially adversarially influenced. Adaptive security is considered in [20,3,22] in the context of *one-time, non-succinct* garbling. An adaptive garbling scheme for Turing machines is constructed in [2]. Still, adaptive security has not been achieved in the pertinent setting of succinct and persistent RAM garbling.

1.1 This work

We construct an adaptively secure, efficiency-preserving, succinct and persistent garbling scheme for RAM programs. 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 also have local output. It is guaranteed that:

(1) Running the garbled programs one after another in sequence on the garbled memory results in the same sequence of outputs as 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 initial database size and sequence of 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 garbled programs seen so far.
(3) The time to garble the memory is proportional to the plaintext memory. 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 usage of each garbled machine are comparable to those of the plaintext machine.

Given such a scheme, constructing a database delegation scheme as specified above is straightforward: Alice sends Eve a garbled version of her database. To delegate a query, she garbles the program that executes the query. To guarantee (public) verifiability Alice can use the following technique from [18]: Each program signs its outputs using an embedded signing key, and Alice publishes the corresponding public key. To hide the query results from the server, the program encrypts its output under a secret key known to Alice. We provide herein a more complete definition (within the UC framework), as well as an explicit construction and analysis.

1.2 Overview of the construction

Our starting point is the statically-secure garbling scheme of Canetti and Holmgren [9]. 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 four 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 the RAM computation steps 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 the garbled machines $\tilde{M_1}$, $\tilde{M_2}$ and the garbled inputs $\tilde{x_1}$, $\tilde{x_2}$ are computationally indistinguishable: $(\tilde{M_1}, \tilde{x_1}) \approx (\tilde{M_2}, \tilde{x_2})$. This step closely follows the scheme of Koppula, Lewko and Waters [28] for garbling of Turing machines. The garbled program is essentially an obfuscated RAM/CPU-step circuit, which takes as input a local state and a memory symbol, and outputs an updated local state, as well as a memory operation. 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 extends the construction to *fixed-access garbling scheme*, which allows the intermediate local states of the two transcripts to differ while everything else stays the same. This is achieved by encrypting the state in an obfuscation-friendly way. The third 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. Here they apply the same type of encryption used for the local state also to the memory content. The final 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 [13].)

The challenge of adaptive security. The first (and biggest) challenge has to do with the *positional accumulator*, which is an iO-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 local state 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" the accumulator value ac to contents v at location L. Intuitively, the main security property is that it should be computationally infeasible to open an accumulator to an incorrect value.

However, to be useful with indistinguishability obfuscation, the accumulator needs an additional property, called *enforceability.* In [28], 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^* at location L^* to any value other than v^*.[5]

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

[5] 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. We would like to argue that $\mathsf{i}\mathcal{O}(C_0') \approx \mathsf{i}\mathcal{O}(C_1')$; however, we cannot do it directly using a plain Merkle hash tree, since collisions exist and so C_0' and C_1' have very different functionalities. Positional accumulators get around this difficulty: Using enforceability 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.

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.

Adaptive Accumulators. We get around this problem by defining and constructing a new primitive, called *adaptive accumulators,* which are an adaptive alternative to 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 accumulators from adaptive 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 memory 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 verification 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 usable in the Merkle-Damgård construction, 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.

Our construction of an AP hash ensemble can be understood in two steps.

1. First we construct a c-bounded AP hash ensemble from any c-bounded hash ensemble $\{h_k\}$. This is done as follows: The public key is a 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 $y = f_{h_k}(x^*)$, 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 [7].
2. Next we construct AP hash functions which are 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. Suppose we start with a function $h' : \{0,1\}^{\lambda+1} \to \lambda$, and build $h : \{0,1\}^{2\lambda} \to \{0,1\}^\lambda$. A verification key vk for h is an obfuscated circuit C which takes x and y, and directly checks that $h(x) = y$.

The proof of security involves a sequence of hybrids, in which C is modified to contain a verification key for h'. This implies that in the real world, C must also be padded to this same size. In other words, the verification key vk must be as large as twice-obfuscated circuit computing h'. We note that it is possible to avoid this overhead by instead distributing λ different verification keys for h', but we avoid this approach for conceptual simplicity.

From adaptive accumulators to adaptively secure fixed-transcript garbling. We return to the challenges encountered when trying to use the [9] construction in our adaptive setting. With adaptive accumulators in hand, the additional modifications made on the use of iterator and splittable signatures are relatively local. Since these primitives do not access the long-lived shared memory, it suffices to generate a fresh instance of each primitive for each new query.

Adaptively secure fixed-access and fixed-address garbling. Next we upgrade the next two layers in the [9] construction, namely the fixed-access and fixed-address garbling schemes, to adaptively secure ones. This is done with relatively local changes from the original construction. Specifically we include the index and time step in the domain of puncturable PRF that is used to derive the randomness of the one-time-pad-like encryption on the state and memory. The technical details can be found in the main construction.

Adaptive full garbling. Recall that in [9] full garbling is achieved by applying an Oblivious RAM scheme on top of the fixed-access garbling. The randomness for the ORAM 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. Later, the key needs to be punctured at a point that may depend on the PRF values. This leads to another adaptivity problem.

We get around this problem 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 $\boldsymbol{A}_i = \boldsymbol{a_{i1}}, \ldots, \boldsymbol{a_{im}}$ 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 $\boldsymbol{a_{ij}}$ depends only on $R_{I_{ij}}$, i.e., the part of the randomness R indexed with I_{ij}; furthermore, $\boldsymbol{a_{ij}}$ 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, each virtual access consists of $O(\log S)$ phases, each of whose physical addresses are determined by two randomness intervals. Since the number of intervals in the range $[1, \ldots, |R|]$ is only polynomial in the security parameter, the reduction can guess the intervals for a phase (and therefore the points to puncture at) with only polynomial security loss.

In contrast, the localized randomness property used in [9] differs in property 1 above, requiring only that each $\mathbf{A}_i$ depends on polylogarithmically many bits of R. This does not suffice for us, because there are superpolynomially many possible dependencies, and so the reduction cannot guess correctly with any non-negligible probability.

Concurrent and independent work. A potential alternative to our adaptive positional accumulators is to build on the *somewhere statistically binding (SSB) hash* of Hubáček and Wichs [23] or Okamoto et al. [30]. SSB hashes have a similar flavor to positional accumulators, but they allow rigging to be statistically binding at a hidden location L^*. However it turns out that SSB hashes alone do not suffice for positional accumulators, even in the non-adaptive case! In concurrent and independent work, Ananth et al. [1] give a stronger definition of SSBs which *does* suffice, and then show that a known construction [30] satisfies this stronger property. Their reduction can then be made adaptive by guessing L^*, at the price of reducing the reduction's winning probability by a factor proportional to the database size. In all, their construction uses a somewhat stronger assumption than ours (DDH vs. discrete log) and their security reduction is somewhat less efficient than ours.

Organization. The rest of the paper is organized as follows. Section 2 provides definitions of RAM and adaptively secure garbled RAM. Sections 3, 4 and 5 define and construct bounded hashing, adaptively puncturable hashing and adaptively secure positional accumulator. Sections 6, 7, 8 and 9 provide the definitions and constructions of fixed-transcript, fixed-access, fixed-address, and fully secure garbling. Section 10 includes the definition of secure database delegation within the UC framework and our construction and proof.

Due to the page limitation, some missing details are only available in the full version of this paper [8]. Those missing details include (1) The other primitives used in our work; (2) The proofs of fixed-transcript, fixed-access, fixed-address and fully secure garbling; (3) A construction of a (stateful) reusable GRAM with persistent data.

2 Definitions

RAM Programs. A RAM M is defined as a tuple (Σ, Q, Y, C), where Σ is the set of memory symbols, Q is the set of possible local states, Y is the output space, and C is the transition function.

Memory Configurations A memory configuration on alphabet Σ is a function $s : \mathbb{N} \to \Sigma \cup \{\epsilon\}$, where ϵ denotes the contents of an empty memory cell. 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, where $s_x(i) = x_i$ if $i \leq |x|$ and $s_x(i) = \epsilon$, otherwise. 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.

Execution A RAM $M = (\Sigma, Q, Y, C)$ is executed on an initial memory $s_0 \in \Sigma^{\mathbb{N}}$ to obtain $M(s_0)$ by iteratively computing $(q_i, a_i, v_i) = C(q_{i-1}, s_{i-1}(a_{i-1}))$, where $a_0 = 0$, and defining $s_i(a) = v$ if $a = a_i$ and $s_i(a) = s_{i-1}$ otherwise.

When $M(s_0) \neq \perp$, it is convenient to define the following functions:

$\mathsf{Time}(M, s_0)$: *runtime* of M on s_0, i.e., the number of iterations of C.

$\mathsf{Space}(M, s_0)$: *space usage* of M on s_0, i.e., $\max_{i=0}^{t-1}(\|s_i\|_\infty)$.

$\mathcal{T}(M, s_0)$: *execution transcript* of M on s_0 defined as $((q_0, a_0, v_0), \ldots, (q_{t-1}, a_{t-1}, v_{t-1}), y)$.
$\mathsf{Addr}(M, s_0)$: *addresses accessed* by M on s_0, i.e, $(a_0, \ldots, a_{t-1})$.
$\mathsf{NextMem}(M, s_0)$: *resultant memory configuration* s_t after executing M on s_0.

Garbled RAM

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

$\mathsf{Setup}(1^\lambda, S)$ takes the security parameter λ in unary and a space bound S, and outputs a secret key SK.
$\mathsf{GbMem}(SK, s)$ takes a secret key SK and a memory configuration s, and then outputs a memory configuration $\tilde{s}$.
$\mathsf{GbPrg}(SK, M_i, T_i, i)$ takes a secret key SK, a RAM machine M_i, a running time bound T_i, and a sequence number i, and outputs a garbled RAM machine $\tilde{M}_i$.
$\mathsf{Eval}(\tilde{M}, \tilde{x})$: takes a garbled RAM $\tilde{M}$ and gabled input $\tilde{x}$ and evaluates the machine on the input, which we denote $\tilde{M}(\tilde{x})$.

Remark 1. The index number i given as input to GbPrg enforces defines a fixed order, so that $M_1, \ldots, M_\ell$ cannot be executed in any other order.

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 $t = \text{poly}(\lambda)$

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

where

- $\sum T_i \leq \text{poly}(\lambda)$, $|s_0| \leq S \leq \text{poly}(\lambda)$;
- $\mathsf{Space}(M_i, s_{i-1}) \leq S$ 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. Setup, 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 $\mathsf{Space}(\tilde{M}_i, \tilde{s}_{i-1}) = \tilde{O}(S)$ and $\mathsf{Time}(\tilde{M}_i, \tilde{s}_{i-1}) = \tilde{O}(\mathsf{Time}(M_i, s_{i-1}))$ (hiding polylogarithmic factors in S).

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

Definition 1. *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 bounded byT_i and S. We denote $y_i = M_i(s_{i-1})$, $s_i = \mathsf{NextMem}(M_i, s_{i-1})$, and $t_i = \mathsf{Time}(M_i, s_{i-1})$.*

```
Experiment $REAL_{\mathcal{A}}(1^\lambda)$
$(s_0, S) \leftarrow \mathcal{A}(1^\lambda)$
$SK \leftarrow \mathsf{Setup}(1^\lambda, S), \tilde{s}_0 \leftarrow \mathsf{GbMem}(SK, s_0)$
$(M_1, 1^{T_1}) \leftarrow \mathcal{A}(\tilde{s}_0)$
$\tilde{M}_1 \leftarrow \mathsf{GbPrg}(SK, M_1, T_1, 1)$
for $i = 1$ to $\ell = \mathsf{poly}(\lambda)$
  $(M_{i+1}, 1^{T_{i+1}}) \leftarrow \mathcal{A}(\tilde{M}_{[i,\ldots,1]}, \tilde{s}_0)$
  $\tilde{M}_{i+1} \leftarrow \mathsf{GbPrg}(SK, M_{i+1}, T_{i+1}, i+1)$
Output : $b \leftarrow \mathcal{A}(\tilde{M}, \tilde{s}_0)$
```

```
Experiment $IDEAL_{\mathcal{A}}(1^\lambda)$
$(s_0, S) \leftarrow \mathcal{A}(1^\lambda)$
$\tilde{s}_0 \leftarrow \mathsf{Sim}(1^\lambda, \|s_0\|_0)$
$(M_1, 1^{T_1}) \leftarrow \mathcal{A}(\tilde{s}_0)$
$\tilde{M}_1 \leftarrow \mathsf{Sim}(y_1, |M_1|, t_1)$
for $i = 1$ to $\ell = \mathsf{poly}(\lambda)$
  $(M_{i+1}, 1^{T_{i+1}}) \leftarrow \mathcal{A}(\tilde{M}_{[i,\ldots,1]}, \tilde{s}_0)$
  $\tilde{M}_{i+1} \leftarrow \mathsf{Sim}(y_{i+1}, |M_{i+1}|, t_{i+1})$
Output : $b' \leftarrow \mathcal{A}(\tilde{M}, \tilde{s}_0)$
```

*The garbling scheme $\mathcal{GRAM}$ is $\epsilon(\cdot)$-*adaptively secure *if*

$$\left|\mathsf{Pr}[1 \leftarrow REAL_{\mathcal{A}}(1^\lambda)] - \mathsf{Pr}[1 \leftarrow IDEAL_{\mathcal{A}}(1^\lambda)]\right| < \epsilon(\lambda).$$

3 *c*-Bounded Collision-Resistant Hash Functions

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

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

That is, with high probability, every element in the codomain of h has at most $c(\lambda)$ pre-images. In our adaptively secure garbling scheme, we need $c(\cdot)$ to be any polynomial (smaller is better for the security reduction), and we need $D_\lambda = \{0,1\}^{\lambda'}$ and $R_\lambda = \{0,1\}^{\lambda'-1}$ for some $\lambda' = \mathrm{poly}(\lambda)$. For both of the constructions in this section, we obtain constant $c(\cdot)$.

The starting point for our constructions is the construction of [14], using a claw-free pair of permutations (π_0, π_1) on a domain $\mathcal{D}_\lambda$, where for some fixed y_0, the hash $h(x)$ is defined as $(\pi_{x_0} \circ \cdots \circ \pi_{x_n})(y_0)$. Unfortunately, while this construction allows an arbitrarily-compressing hash function, it in general may not be poly(n)-bounded even if $n = \log|\mathcal{D}_\lambda| + O(1)$.

However, a slight modification of this construction allows us to take any injective functions $\iota_{in} : \{0,1\}^n \hookrightarrow \mathcal{D}_\lambda$ and $\iota_{out} : \mathcal{D}_\lambda \hookrightarrow \{0,1\}^m$, and produce a 2^k-bounded collision-resistant function mapping $\{0,1\}^{n+k} \to \{0,1\}^m$. As long there is such injections exist with $m-n = O(\log\lambda)$, this yields a poly(λ)-bounded collision-resistant hash family.

Theorem 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 p be a random λ-bit prime, and let g and h be randomly chosen generators of $\mathbb{Z}_p^*$. Our hash function is keyed by p, g, h. It is well-known that the permutations $\pi_0(x) = g^x$ and $\pi_1(x) = g^x h$ are claw-free. 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.

Theorem 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 $\mathcal{H} = \{\mathcal{H}_\lambda\}_{\lambda\in\mathbb{N}}$ where $\mathcal{H}_\lambda$ consists of functions mapping $\{0,1\}^{2\lambda+1} \to \{0,1\}^{2\lambda}$.*

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. $\iota_1(x)$ is therefore well-defined as the smallest of $\{3x, 3x+1, 3x+2\} \cap \mathbb{Z}_n^*$. 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 [21], 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 $\iota_{out} \circ \pi_{y_5} \circ \cdots \circ \pi_{y_1}$ per possible y value. 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 $x \mapsto \iota_{in}(x)^2 \pmod N$ follows from the fact that the two square roots are nontrivially related, i.e., neither is the negative of the other, so given both it would be possible to factor N.

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

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

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, and $h \in \mathcal{H}$, $\mathsf{Verify}(\mathsf{vk}, x, y) = 1$ iff $y = h(x)$, where $\mathsf{vk} \leftarrow \mathsf{GenVK}(1^\lambda, h)$.

Forced Verification
For all x^* and $h \in \mathcal{H}$, let $y^* = h(x^*)$. $\mathsf{Verify}(\mathsf{vk}, x, y^*) = 1$ iff $x = x^*$, where $\mathsf{vk} \leftarrow \mathsf{ForceGenVK}(1^\lambda, h, x^*)$.

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

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

Theorem 3. *If* $\mathsf{i}\mathcal{O}$ *exists and there is a* $\mathrm{poly}(\lambda)$*-bounded CRHF ensemble mapping* $\{0,1\}^{\lambda'+1} \to \{0,1\}^{\lambda'}$*, 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 $\mathrm{poly}(\lambda)$-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. 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

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

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 C given to the adversary from C_i to C_{i+1}.

1. We first change C so that when $y = y^*$, it computes the intermediate value $y' = h^{\lambda-i-1}(x_{i+2} \| \cdots \| x_{2\lambda})$ and outputs 1 if:
   - $h(x_{i+1} \| y') = y_{\lambda-i}$
   - For all $1 \leq j \leq i$, $x_i = x^*_i$.

   When $y \neq y^*$, the behavior of C is unchanged.
   This change preserves functionality (we only introduced a name y' for an intermediate value in the computation) 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. The challenger computes $y_{\lambda-i-1} = h^{\lambda-i-1}(x^*_{i+2}\| \cdots \|x^*_{2\lambda})$ 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.
   - Because $\mathcal{H}_\lambda$ is 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 $\mathsf{i}\mathcal{O}$.

5 Adaptively Secure Positional Accumulators

In this section we define and construct adaptive positional accumulators (APA). We use this primitive for memory authentication in our garbling construction. A garbled program will be an obfuscated functionality where one input is a succinct commitment ac to some memory contents, another is a piece of data v allegedly resulting from a memory operation op, and another is a commitment ac', allegedly to the resulting memory configuration. Informally, APAs provide a way for the garbled program to check the consistency of v and ac' with ac (given a short proof),

As described so far, Merkle trees satisfy our needs, and indeed our construction is built around a Merkle tree. However, we require more. As in the positional accumulators of [28], we need a way to indistinguishably "rig" the public parameters so that for some ac and op, there is exactly one (ac', v) with any accepting proof. We differ from [27] by separating the parameters used for proof verification from those used for updating the accumulator, and allowing the rigged $(\mathsf{ac}, \mathsf{op})$ to be chosen adaptively as an adversarial function of the update parameters.

We now formally define the algorithms of the APA primitive.

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

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

The verification 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 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, and 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, when sampling

```
$\mathsf{PP}, \mathsf{ac}_0, \mathsf{store}_0 \leftarrow \mathsf{SetupAcc}(1^\lambda, S)$
$\mathsf{vk} \leftarrow \mathsf{SetupVerify}(\mathsf{PP})$
For $i = 0, \ldots, k$:
    $\mathsf{store}_{i+1}, \mathsf{ac}_{i+1}, v_i, \pi_i \leftarrow \mathsf{Update}(\mathsf{PP}, \mathsf{store}_i, \mathsf{op}_i)$
    $b_i \leftarrow \mathsf{Verify}(\mathsf{vk}, \mathsf{ac}_i, \mathsf{op}_i, \mathsf{ac}_{i+1}, v_i, \pi_i)$
```

it holds (with probability 1) that for all $j \in \{0, \ldots, k\}$, $v_j = v_j^*$ and $b_j = 1$

Enforcing

Enforcing requires that for all space bounds S, all sequences of operations $\mathsf{op}_0, \ldots, \mathsf{op}_{k-1}$, when sampling

```
$\mathsf{PP}, \mathsf{ac}_0, \mathsf{store}_0 \leftarrow \mathsf{SetupAcc}(1^\lambda, S)$
$\mathsf{vk} \leftarrow \mathsf{SetupEnforceVerify}(\mathsf{PP}, (\mathsf{op}_0, \ldots, \mathsf{op}_{k-1}))$
For $i = 0, \ldots, k-1$
    $\mathsf{store}_{i+1}, \mathsf{ac}_{i+1}, v_i, \pi_i \leftarrow \mathsf{Update}(\mathsf{PP}, \mathsf{store}_i, \mathsf{op}_i)$
```

it holds (with probability 1) that for all accumulators $\hat{\mathsf{ac}}$, all values $\hat{v}$, and all proofs $\hat{\pi}$, if $\mathsf{Verify}(\mathsf{vk}, \mathsf{ac}_{k-1}, \mathsf{op}_{k-1}, \hat{\mathsf{ac}}, \hat{v}, \hat{\pi}) = 1$, then $(\hat{v}, \hat{\mathsf{ac}}) = (v_{k-1}, \mathsf{ac}_k)$

Indistinguishability of Enforcing Verify

Now we require that the output of $\mathsf{SetupVerify}(\mathsf{PP})$ is indistinguishable from the output of $\mathsf{SetupEnforceVerify}(\mathsf{PP}, (\mathsf{op}_1, \ldots, \mathsf{op}_k))$, 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 $\mathcal{A}_2$,

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

Efficiency

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

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

Theorem 4. *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 $\bot$, 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 $\mathsf{PP} = h$, $\mathsf{ac}_0 = h(\bot \| \bot)$, and store_0 to be a root node with value $h(\bot \| \bot)$.

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

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

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

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

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

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.

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

Computes the store_{k-1} which would result from processing $\mathsf{op}_1, \ldots, \mathsf{op}_{k-1}$. 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 $i \in \{1, \ldots, d\}$, 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 $i \in \{d + 1, \ldots, \log S\}$, just sample $\mathsf{vk}_i \leftarrow \mathsf{GenVK}(1^\lambda, h)$.

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

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

6 Fixed-Transcript Garbling

Next we present the first step in our construction, a garbling scheme that provides adaptive security for RAM programs that have the same transcript. The notion extends the first stage of Canetti-Holmgren scheme into the adaptive setting, and the construction employs the adaptive positional accumulators plus local changes in the other primitives.

We define fixed-transcript security via the following game.

1. The challenger samples $\mathsf{SK} \leftarrow \mathsf{Setup}(1^\lambda, S)$ and $b \leftarrow \{0, 1\}$.
2. The adversary sends a memory configuration s to the challenger. The challenger sends back $\mathsf{GbMem}(\mathsf{SK}, s)$.
3. The adversary repeatedly sends pairs of RAM programs (M_i^0, M_i^1) along with a time bound T_i, and the challenger sends back $\tilde{M}_i^b \leftarrow \mathsf{GbPrg}(\mathsf{SK}, M_i^b, T_i, i)$. 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 $M_1^1, \ldots, M_\ell^1$.
– Each M_i^b runs in time at most T_i and space at most S.
– For each i, $|M_i^0| = |M_i^1|$.

Definition 2. *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 5. *Assuming the existence of indistinguishability obfuscation and an adaptive positional accumulator, there is a fixed-transcript secure garbling scheme.*

Proof. Our construction follows closely the fixed-transcript garbling scheme of [9], using our *adaptive* positional accumulator in place of [28]'s positional accumulator. We also rely on puncturable PRFs (PPRFs), splittable signatures and cryptographic iterators defined in the full version.

$\mathsf{Setup}(1^\lambda, S) \to \mathsf{SK}$**:** sample $(\mathsf{Acc.PP}, \mathsf{ac}_{\mathsf{init}}, \mathsf{store}_{\mathsf{init}}) \leftarrow \mathsf{Acc.Setup}(1^\lambda, S)$, a PPRF F. Set $\mathsf{SK} = (\mathsf{Acc.PP}, \mathsf{ac}_{\mathsf{init}}, \mathsf{store}_{\mathsf{init}}, \mathsf{Itr.PP}, \mathsf{itr}_{\mathsf{init}}, \mathsf{F})$, and $(\mathsf{Itr.PP}, \mathsf{itr}_{\mathsf{init}}) \leftarrow \mathsf{Itr.Setup}(1^\lambda)$.

$\mathsf{GbMem}(\mathsf{SK}, s) \to \tilde{s}$ **:** GbMem updates the APA $(\mathsf{ac}_{\mathsf{init}}, \mathsf{store}_{\mathsf{init}})$ to set the underlying memory to s (via a sequence of calls to Update) and let $\mathsf{ac}_0, \mathsf{store}_0$ denote the result. It then generates $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; \mathsf{F}(1, 0))$, where $(1, 0)$ represents the initial index number i and initial time-step number 0[6]. Finally, GbMem computes $\sigma_0 \leftarrow \mathsf{Spl.Sign}(\mathsf{sk}, (\bot, \bot, \mathsf{ac}_0, \mathsf{ReadWrite}(0 \mapsto 0)))$. Here the first $\bot$ represents an initial local state q_0 for M_1, and the second $\bot$ represents an initial iterator value itr_0. GbMem outputs $\tilde{s} = (\sigma_0, \mathsf{ac}_0, \mathsf{store}_0)$.

$\mathsf{GbPrg}(\mathsf{SK}, M_i, T_i, i) \to \tilde{M}_i$**:** GbPrg 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, GbPrg defines and outputs a RAM machine $\tilde{M}_i$, which has $\tilde{C}_i$ hard-coded as part of its transition function, such that $\tilde{M}_i$ does the following:
1. Reads $(\mathsf{ac}_0, \sigma_0)$ from memory. Define $\mathsf{op}_0 = \mathsf{ReadWrite}(0 \mapsto 0)$, $q_0 = \bot$, and $\mathsf{itr}_0 = \bot$.
2. For $t = 0, 1, 2, \ldots$:
   (a) Compute $\mathsf{store}_{t+1}, \mathsf{ac}_{t+1}, v_t, \pi_t \leftarrow \mathsf{Acc.Update}(\mathsf{Acc.PP}, \mathsf{store}_t, \mathsf{op}_t)$.
   (b) Compute $\mathsf{out}_t \leftarrow \tilde{C}_i(t, q_t, \mathsf{itr}_t, \mathsf{ac}_t, \mathsf{op}_t, \sigma_t, v_t, \mathsf{ac}_{t+1}, \pi_t)$.
   (c) If out_t parses as (y, σ), then write $(\mathsf{ac}_{t+1}, \sigma)$ to memory, output y, and terminate.
   (d) Otherwise, parse out_t as $(q_{t+1}, \mathsf{itr}_{t+1}, \mathsf{ac}_{t+1}, \mathsf{op}_{t+1}), \sigma_{t+1}$ or terminate if out_t is not of this form.

We note that GbPrg can efficiently produce $\tilde{M}_i$ 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 $\tilde{M}_i$.

[6] Looking ahead, all the intermediate $(\mathsf{sk}, \mathsf{vk})$ key pairs are generated by applying F to the (index, time-step) tuple.

```
Input: Time $t$, state $q$, iterator $\mathsf{itr}$, accumulator $\mathsf{ac}$, operation $\mathsf{op}$, signature $\sigma$,
       memory value $v$, new accumulator $\mathsf{ac}'$, proof $\pi$
Data: Puncturable PRF $\mathsf{F}$, RAM machine $M_i$ with transition function $\delta_i$,
      Accumulator verification key $\mathsf{vk}_{\mathsf{Acc}}$, index $i$, iterator public parameters
      $\mathsf{Itr.PP}$, time bound $T_i$
1  $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; \mathsf{F}(i, t))$;
2  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 $\perp$;
3  $\mathsf{out} \leftarrow \delta_i(q, v)$;
4  if $\mathsf{out} \in Y$ then
5      $(\mathsf{sk}', \mathsf{vk}') \leftarrow \mathsf{Spl.Setup}(1^\lambda; \mathsf{F}(i+1, 0))$;
6      return $\mathsf{out}, \mathsf{Sign}(\mathsf{sk}', (\perp, \perp, \mathsf{ac}', \mathsf{ReadWrite}(0 \mapsto 0))$
7  else
8      Parse $\mathsf{out}$ as $(q', \mathsf{op}')$;
9      $\mathsf{itr}' \leftarrow \mathsf{Itr.Iterate}(\mathsf{Itr.PP}, (q, \mathsf{itr}, \mathsf{ac}, \mathsf{op}))$;
10     $(\mathsf{sk}', \mathsf{vk}') \leftarrow \mathsf{Spl.Setup}(1^\lambda; \mathsf{F}(i, t+1))$;
11     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.

$\mathsf{Eval}(\tilde{M}, \tilde{s})$ The evaluation algorithm runs $\tilde{M}$ on the garbled memory $\tilde{s}$, and outputs $\tilde{M}(\tilde{s})$.

Correctness and efficiency are easy to verify. For the proof of security we refer the readers to the full version.

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 3 (Fixed-access security).
We define fixed-access 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, 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 $M_1^0, \dots, M_\ell^0$ on initial memory configuration s yields the same transcript as executing $M_1^1, \dots, M_\ell^1$, 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 [9]: xoring the state with a pseudorandom function applied on the local time t. The PRF keys used in different machines are sampled independently.

Theorem 6. *If there is a fixed-transcript garbling scheme, then there is a fixed-access garbling scheme.*

Proof. From a fixed-transcript garbling scheme $(\mathsf{Setup}', \mathsf{GbMem}', \mathsf{GbPrg}', \mathsf{Eval}')$, we construct a fixed-access garbling scheme $(\mathsf{Setup}, \mathsf{GbMem}, \mathsf{GbPrg}, \mathsf{Eval})$.

$\mathsf{Setup}(1^\lambda, S)$ 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, outputs $\tilde{M}_i' \leftarrow \mathsf{GbPrg}'(SK', M_i', T_i, i)$, where M_i' is defined as in Algorithm 2. If M_i's initial state is q_0, the initial state of M_i' is $(0, q_0 \oplus F_i(0))$.
$\mathsf{Eval}(\tilde{M}, \tilde{s})$ outputs $\mathsf{Eval}'(\tilde{M}', \tilde{s}')$.

```
**Input**: State $(t, c_q)$, memory symbol $\sigma$
**Data**: RAM machine $M_i$, puncturable PRF $F_i$
1 $q \leftarrow c_q \oplus F_i(t)$;
2 $\mathsf{out} \leftarrow M_i(q, \sigma)$;
3 **if** $\mathsf{out} \in Y$ **then return** $\mathsf{out}$;
4 Parse $\mathsf{out}$ as $(q', \mathsf{op})$;
5 **return** $((t+1, q' \oplus F_i(t+1)), \mathsf{op})$;
```
Algorithm 2: M_i', the modified version of M_i which encrypts its state.

For the proof that this construction satisfies the requisite security, we refer the readers to the full version.

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 4 (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 are the same as when 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.*
- *For each i, $|M_i^0| = |M_i^1|$.*

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 [9], with a slight modification at the way to "encrypt" the memory configuration. In [9], 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 [9], 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 7. Suppose $(\mathsf{Setup}', \mathsf{GbMem}', \mathsf{GbPrg}', \mathsf{Eval}')$ is a fixed-access garbling scheme, we construct a fixed-address garbling scheme $(\mathsf{Setup}, \mathsf{GbMem}, \mathsf{GbPrg}, \mathsf{Eval})$:

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

$\mathsf{GbMem}(SK, s)$ outputs $\tilde{s}'_0 \leftarrow \mathsf{GbMem}'(SK', s'_0)$, 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 $\tilde{M}'_i \leftarrow \mathsf{GbPrg}'(SK', M'_i, T_i, i)$, where M'_i is defined as in Algorithm 3. If the initial state of M_i was q_0, the initial state of M'_i is $(0, q_0, q_0)$.

$\mathsf{Eval}(\tilde{M}, \tilde{s})$ outputs $\mathsf{Eval}'(\tilde{M}', \tilde{s}'_0)$.

```
  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 $F_A$, $F_B$
1 out ← $M_i(q_A, F_A(i_{in}, t_{in}) \oplus c_A)$;
2 if out $\in Y$ then return out;
3 Parse out as $(q', \mathsf{ReadWrite}(\mathsf{addr}' \mapsto v'))$;
4 op' := $\mathsf{ReadWrite}(\mathsf{addr}' \mapsto (i, t_q, F_A(i, t_q) \oplus v', F_B(i, t_q) \oplus v')$);
5 return $(t_q + 1, q', q')$, op';
```

Algorithm 3: M'_i: Modified version of M_i which encrypts its memory twice in parallel.

Theorem 8. *If* $(\mathsf{Setup}', \mathsf{GbMem}', \mathsf{GbPrg}')$ *is a fixed-access garbling scheme, then Construction 7 is a fixed-address garbling scheme.*

Proof. For the proof of security we refer the readers to the full version.

9 Full Garbling

In order to construct a fully secure garbling scheme, we will need to make use of an oblivious RAM (ORAM) [19] to hide the addresses accessed by the machine.

9.1 Oblivious RAMs with strong localized randomness

We require that the ORAM has a strong localized randomness property[7], which is satisfied by the ORAM construction of [13]. Below we give a brief definition of ORAM and the property we need.

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

[7] This notion is similar but stronger to the "localized randomness" defined in [9]

M_{op}. More generally, for a RAM machine M, we can define $\mathsf{OProg}(1^\lambda, M, S)$ as the (probabilistic) machine 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.

Security (Strong Localized Randomness).

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 $\boldsymbol{A} = \boldsymbol{a}_1 \| \cdots \| \boldsymbol{a}_t$. There should be a natural way to decompose each $\boldsymbol{a}_i$ (in the Chung-Pass ORAM, we consider each recursive level of the construction) such that we can write $\boldsymbol{a}_i = \boldsymbol{a}_{i,1} \| \cdots \| \boldsymbol{a}_{i,m}$. Our notion of strong localized randomness requires that (after having fixed $\mathsf{op}_1, \ldots, \mathsf{op}_t$), each $\boldsymbol{a}_{i,j}$ depends on some small substring of R, which does not influence any other $\boldsymbol{a}_{i',j'}$. In other words:

- There is some $\alpha_{i,j}, \beta_{i,j} \in \mathbb{N}$ such that $0 < \beta_{i,j} - \alpha_{i,j} \leq \text{poly}(\log S)$ and such that $\boldsymbol{a}_{i,j}$ 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.

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
  $$\boldsymbol{A}_1 \| \cdots \| \boldsymbol{A}_t \leftarrow \mathsf{addr}(M^{R_1}_{\mathsf{op}_1}; \ldots; M^{R_t}_{\mathsf{op}_t}, \epsilon^{\mathbb{N}})$$
  where $R = R_1 \| \cdots \| R_t$ denotes the randomness used by the oblivious accesses and each $\boldsymbol{A}_i$ denotes the addresses accessed by $M^{R_i}_{\mathsf{op}_i}$, then $(\boldsymbol{A}_t)_{[\tau_j, \tau_{j+1})} = C_j(R_{I_j})$ 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 $R_{\mathbb{N} \setminus I_j}$, $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_{t-1}$ 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.

A full exposure, including the full definition and proof that Chung-Pass ORAM satisfies the strong localized randomness property can be found in the full version.

9.2 Full Garbling Construction

Theorem 9. *If there is an efficient fixed-address garbling scheme, then there is an efficient full garbling scheme.*

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

$\mathsf{Setup}(1^\lambda, T, S)$ 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 $[2^{2\lambda}]$.

$\mathsf{GbMem}(\mathsf{SK}, s_0)$ outputs $\tilde{s}'_0 \leftarrow \mathsf{GbMem}'(\mathsf{SK}', \mathsf{OMem}(1^\lambda, s_0, S))$.

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

$\mathsf{Eval}(\tilde{M}, \tilde{s})$ outputs $\mathsf{Eval}'(\tilde{M}', \tilde{s}'_0)$.

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 is given S and n, and 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 is given $y_i = M_i(s_{i-1})$ and $t_i = \mathsf{Time}(M_i, s_{i-1})$, where $s_i = \mathsf{NextMem}(M_i, s_{i-1})$, 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 4, 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
      $C_1, \ldots, C_m$ guaranteed by localized randomness
1 for $t = 1, \ldots, t_i$ do
2     for $k = 1, \ldots, m$ do
3         $r_k \leftarrow G_i(t, k)$;
4         Access addresses given by $C_k(r_k)$
5 return $y_i$.
```

Algorithm 4: 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.

We refer the readers to the full version for the proof.

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 stress that the secrecy requirement for the case of a corrupted third party is incomparable to the secrecy requirement in the case of an honest third party. In particular, the case of corrupted third parties guarantees secrecy even when the entire evaluation and verification processes are completely exposed.)

More precisely, a database delegation scheme (or, protocol) consists of the following algorithms:

$\mathsf{DBDelegate}$: Initial delegation of the database. Takes as input a plain 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 $\mathsf{sk}_{\mathsf{enc}}$ 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 $\mathsf{sk}_{\mathsf{enc}}$ and outputs either an answer value or $\perp$.

Security. The security requirement from a database delegation scheme $\mathcal{S} = (\mathsf{DBDelegate}, \mathsf{Query}, \mathsf{Eval}, \mathsf{AnsDecVer})$ is that it UC-realize the *database delegation ideal functionality* $\mathcal{F}_{dd}$ defined as follows. (For simplicity we assume that the database owner is uncorrupted, and that the communication channels are authenticated.)

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 $\|D\|_0$ 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 $|M|$, the running time of M, and the length of a to the adversary. If the adversary returns `ok` then output (M, a) to P.

To make the requirements imposed by $\mathcal{F}_{dd}$ more explicit, we also provide an alternative (and equivalent) formulation of 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 $\mathsf{sk}_{\mathsf{enc}}^i$ and $\tilde{M}_i$. $\mathcal{A}$ obtains $\tilde{M}_i$, and in the dishonest case also the decryption key $\mathsf{sk}_{\mathsf{enc}}^i$.
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 $\mathsf{AnsDecVer}(\mathsf{vk}, \mathsf{sk}_{\mathsf{enc}}, \mathsf{out}_i)$.

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

1. $\mathcal{A}$ provides a database D, receives the output of $\mathsf{Sim}(\|D\|_0)$.
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 5. *A delegation scheme $\mathcal{S} = (\mathsf{DBDelegate}, \mathsf{Query}, \mathsf{Eval}, \mathsf{AnsDecVer})$ is secure if it UC-realizes $\mathcal{F}_{dd}$. Equivalently, it 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 with $\mathcal{S}$ or in the ideal interaction with* Sim.

Theorem 10. *If there exist adaptive succinct garbled RAMs 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 $(\mathsf{Setup}, \mathsf{GbMem}, \mathsf{GbPrg}, \mathsf{Eval})$ 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 $(\mathsf{vk_{sign}}, \mathsf{sk_{sign}})$ for the signature scheme. Set $\mathsf{msk} \leftarrow (SK, \mathsf{sk_{sign}})$ and $\mathsf{vk} \leftarrow \mathsf{vk_{sign}}$.

$\mathsf{Query}(M_i, \mathsf{msk}, \mathsf{pk})$**:** Generate a symmetric encryption key $\mathsf{sk_{enc}}$. Generate the extended version of M_i' of M_i as in Algorithm 5.
Output $\tilde{M} \leftarrow \mathsf{GbPrg}(SK, M_i'[\mathsf{sk_{sign}}, \mathsf{sk_{enc}}], i)$

```
**Input**: State $q$, memory value $v$
**Data**: RAM program $M_i$ with transition function $\delta_i$ and output space $Y$, and
      signing and encryption keys $\mathsf{sk_{sign}}$, $\mathsf{sk_{enc}}$
1 $\mathsf{out} \leftarrow \delta_i(q, v)$;
2 if $\mathsf{out} \in Y$ then
3     $\mathsf{ct_{out}} \leftarrow \mathsf{Enc}(\mathsf{sk_{enc}}, \mathsf{out})$
4     $\sigma_{\mathsf{out}} \leftarrow \mathsf{Sign}(\mathsf{sk_{sign}}, \mathsf{ct_{out}} \| i)$
5     return $(\mathsf{ct_{out}}, \sigma_{\mathsf{out}})$;
6 return out
```

Algorithm 5: 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 $\mathsf{sk_{sign}}, \mathsf{vk_{sign}}$. Sim runs the simulator $\mathsf{Sim_{GRAM}}$ for a GRAM scheme to obtain a simulated garbled database $\tilde{D}$. It provides $\tilde{D}$ and $\mathsf{vk_{sign}}$ 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 $\mathsf{sk_{enc}}$. It computes $\mathsf{ct} = \mathsf{Enc}(\mathsf{sk_{enc}}, a)$, $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk_{sign}}, \mathsf{ct}\|i)$ and runs the simulator $\mathsf{Sim_{GRAM}}$ with inputs $(\mathsf{ct}\|i, \sigma)$ to obtain simulated garbled RAM $\tilde{M}_i$. It returns $\tilde{M}_i$ and $\mathsf{sk_{enc}}$ to $\mathcal{A}$.
  If Sim is executed with inputs (s, t) on the i-th iteration, it generates a random value ct, computes $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk_{sign}}, \mathsf{ct}\|i)$ and runs the simulator $\mathsf{Sim_{GRAM}}$ with inputs $(\mathsf{ct}\|i, \sigma)$ to obtain simulated garbled RAM $\tilde{M}_i$. It returns $\tilde{M}_i$ to $\mathcal{A}$.
- $\mathsf{AnsDecVer}$: If Sim executes on input out_i then it outputs $\mathsf{AnsDecVer}(\mathsf{vk}, \mathsf{sk_{enc}}, \mathsf{out}_i)$.

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 $\mathsf{Sim_{GRAM}}$ to generate simulated database $\tilde{D}'$. We generate the signature scheme keys $(\mathsf{vk_{sign}}, \mathsf{sk_{sign}})$ honestly. We also use $\mathsf{Sim_{GRAM}}$ to generate the garbling for the programs M'_i given inputs $\mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}_{\mathsf{enc}}, \mathsf{out})\|i$, $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}_{\mathsf{sign}}, \mathsf{ct}_i)$ 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.

Acknowledgments

We would like to thank Oxana Poburinnaya. Although she preferred not to co-author this paper, her constructive suggestions and criticisms played an essential role throughout the creation of this work.

This work is supported by US NSF grants 1413920, 1218461, 1012798, 1012910, 1421102, 1562888, 1565208, 1633282, ISF grant 1523/14, and DARPA SafeWare W911NF-15-C-0236. Part of the research by Y.C. was conducted while at SRI funded by the NSF grant 1421102.

References

1. Prabhanjan Ananth, Yu-Chi Chen, Kai-Min Chung, Huijia Lin, and Wei-Kai Lin. Delegating ram computations with adaptive soundness and privacy. Cryptology ePrint Archive, Report 2015/1082, 2015.
2. Prabhanjan Ananth and Amit Sahai. Functional encryption for turing machines. *IACR Cryptology ePrint Archive*, 2015:776, 2015.
3. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In *ASIACRYPT*, pages 134–153, 2012.
4. Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. The hunting of the SNARK. *IACR Cryptology ePrint Archive*, 2014:580, 2014.

5. 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.
6. C. T. Bösch, P. H. Hartel, W. Jonker, and A. Peter. A survey of provably secure searchable encryption. *ACM computing surveys*, 47(2):18:1–18:51, August 2014.
7. Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. In *TCC*, pages 52–73, 2014.
8. Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Succinct adaptive garbled ram. Cryptology ePrint Archive, Report 2015/1074, 2015.
9. Ran Canetti and Justin Holmgren. Fully succinct garbled ram. In *ITCS*, 2016.
10. Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Indistinguishability obfuscation of iterated circuits and ram programs. Cryptology ePrint Archive, Report 2014/769, 2014.
11. 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.
12. Kai-Min Chung, Yael Tauman Kalai, and Salil P. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO, 2010*, pages 483–501, 2010.
13. Kai-Min Chung and Rafael Pass. A simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013.
14. Ivan Bjerre Damgård. Collision free hash functions and public key signature schemes. In *EUROCRYPT*, pages 203–216, 1988.
15. Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled ram from one-way functions. In *STOC*, 2015.
16. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO, 2010*, pages 465–482, 2010.
17. Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT*, pages 405–422, 2014.
18. Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private ram computation. In *FOCS*, 2014.
19. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.
20. 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.
21. 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.
22. Brett Hemenway, Zahra Jafargholi, Rafail Ostrovsky, Alessandra Scafuro, and Daniel Wichs. Adaptively secure garbled circuits from one-way functions.
23. Pavel Hubáček and Daniel Wichs. On the communication complexity of secure function evaluation with long output. ITCS, 2015.
24. Yael Tauman Kalai and Omer Paneth. Delegating ram computations. Cryptology ePrint Archive, Report 2015/957, 2015.
25. Yael Tauman Kalai, Ran Raz, and Ron D. Rothblum. How to delegate computations: the power of no-signaling proofs. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 485–494, 2014.
26. Seny Kamara. Encrypted search. *ACM Crossroads*, 21(3):30–34, 2015.

27. Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. Cryptology ePrint Archive, Report 2014/925, 2014.
28. Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In *STOC*, 2015.
29. Steve Lu and Rafail Ostrovsky. How to garble ram programs? In *EUROCRYPT*. 2013.
30. 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.
31. Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO, 2011*, pages 91–110, 2011.
32. Raluca A. Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 85–100, 2011.
33. Phillip Rogaway. *The round complexity of secure protocols*. PhD thesis, Massachusetts Institute of Technology, 1991.
34. Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them. *Commun. ACM*, 58(2):74–84, 2015.
35. Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, 1986.