Indistinguishability Obfuscation of Iterated Circuits and RAM programs

Ran Canetti* Justin Holmgren† Abhishek Jain‡ Vinod Vaikuntanathan §

September 30, 2014

Abstract

A key source of inefficiency in existing obfuscation schemes is that they operate on programs represented as Boolean circuits or (with stronger assumptions and costlier constructs) as Turing machines.

We bring the complexity of obfuscation down to the level of RAM programs. That is, assuming injective one way functions and indistinguishability obfuscators for all circuits, we construct indistinguishability obfuscators for RAM programs with the following parameters, up to polylogarithmic factors and a multiplicative factor in the security parameter: (a) The space used by the obfuscated program, as well as the initial size of the program itself, are proportional to the maximum space s used by the plaintext program on any input of the given size. (b) On each input, the runtime of the obfuscated program is proportional to s plus the runtime of the plaintext program on that input. The security loss is proportional to the number of potential inputs for the RAM program.

Our construction can be plugged into practically any existing use of indistinguishability obfuscation, such as delegation of computation, functional encryption, non-interactive zero-knowledge, and multiparty computation protocols, resulting in significant efficiency gains. It also gives the first succinct and efficient *one-time* garbled RAM scheme. The size of the garbled RAM is proportional to the maximum space s used by the RAM machine, and its evaluation time is proportional to the running time of the RAM machine on plaintext inputs.

At the heart of our construction is a mechanism for succinctly obfuscating "iterated circuits", namely circuits that run in iterations, and where the output of an iteration is used as input to the next. As contributions of independent interest, we also introduce (a) a new cryptographic tool called Asymmetrically Constrained Encapsulation (ACE), that allows us to succinctly and asymmetrically puncture both the encapsulation and decapsulation keys; and (b) a new program analysis tool called Inductive Properties (IP), that allows us to argue about computations that are locally different, but yet globally the same.

*Tel Aviv University and Boston University. Email: canetti@tau.ac.il. Supported by the Check Point Institute for Information Security, ISF grant 1523/14, NSF MACS project, and an NSF Algorithmic foundations grant 1218461.

†MIT. Email: holmgren@mit.edu.

‡Johns Hopkins University. Email: abhishekjain.itbhu@gmail.com.

§MIT. Email: vinodv@mit.edu. Research supported in part by DARPA Grant number FA8750- 11-2-0225, an Alfred P. Sloan Research Fellowship, the Northrop Grumman Cybersecurity Research Consortium (CRC), Microsoft Faculty Fellowship, and a Steven and Renee Finn Career Development Chair from MIT.

1 Introduction

The ability to cryptographically obfuscate general programs holds great prospects for securing the future digital world. However, current general-purpose obfuscation mechanisms are highly inefficient. One of the main sources of inefficiency is the fact that the existing mechanisms work in different models of computation than those used to write modern computer programs. Specifically, the candidate indistinguishability obfuscator of Garg et. al [GGH+13] and most other general purpose obfuscators in the literature are designed for boolean circuits, and incur a polynomial overhead in both the size and the depth of the circuit. Assuming circuit-obfuscators that satisfy a stronger security property (differing input obfuscation, or DIO), Boyle et. al [BCP14] and Ananth et al. [ABG+13] show how to transform the obfuscator of Garg et. al into one that operates directly on Turing machines, where both the size of the obfuscated program and its runtime on each input are polynomially related to the size and runtime of the input program.

However, working in either the circuit model or the Turing machine model does not allow taking advantage of the fact that realistic computations are invariably the result of relatively short programs written for RAM machines, where the program is executed on CPU with random access to large amount of memory. In other words, when applying such an obfuscator to a RAM program (ie a program written for a RAM machine), one has to first translate the program to a circuit or a Turing machine. Such translation may incur unreasonable overhead in of itself, even before applying the obfuscator. Furthermore, since the obfuscated program is now a circuit or a Turing machine, one cannot meaningfully exploit the advantages of the RAM model in running it.

We present an indistinguishability obfuscator for RAM programs, that roughly preserves the time and space requirements of the input program. Time is preserved on a per-input basis, whereas space is preserved only in worst-case. More precisely, our obfuscator $\mathcal{O}$ takes as input a RAM program π and a length bound n, and outputs a circuit π' of size $\tilde{O}(s)\mathsf{poly}(k)$ and depth $\mathsf{poly}(k)$, where k is the security parameter and s is the maximum space taken by π on any n-bit input.

Given an n-bit input x, π' outputs a RAM program π'_x of size $|\pi'_x| = \mathsf{poly}(k)$ and an initialized $\tilde{O}(s)$-cells random-access memory, where each cell is of size $\mathsf{poly}(k)$. π'_x runs for $\tilde{O}(t)$ steps and outputs $\pi(x)$, where t is the runtime of π on x.

As for security, we guarantee that $\mathcal{O}(\pi_1, n) \approx \mathcal{O}(\pi_2, n)$ for any two programs π_1, π_2 that have the same runtime (more precisely, same same number of memory accesses) on all n-bit inputs x, and furthermore $\pi_1(x) = \pi_2(x)$.

Notice that the program π'_x is in essence a garbled RAM program. This improves on the garbled RAM scheme of Gentry et al. [**?**], where the size of the garbled program is proportional to the runtime of the plaintext progam.

Our construction starts from any indistinguishability obfuscator for circuits, and in addition assumes only existence of injective one way functions. We incur a loss in security that's linear in the number of different inputs on which π runs. That is, if the underlying obfuscator guarantees distinguishing probability $\epsilon(k)$, and π takes ℓ different inputs, then our obfuscator guarantees distinguishing probability $\ell\mathsf{poly}(k)\epsilon(k)$.

Applicability. Our obfuscation mechanism can be used in practically any place where indistinguishability obfuscators for circuits have been used, with the commensurate efficiency gains. We remark that most computations have relatively low space requirements (say, linear in the input size), thus making the dependence on space less critical. A number of immediate applications include:

Publicly verifiable non-interactive delegation of computation: To delegate the computation of $\pi(x)$ for a RAM program π and input x, the delegator samples a pair (sk, vk) of signature and verification keys of a digital signature scheme, and creates the program $\pi' = \mathcal{O}(\hat{\pi}_x)$ where $\hat{\pi}$ is the program that runs π on input x, obtains the output y and outputs $(y, Sign(sk, y))$. It then sends π', M, vk to the worker. The result is accepted if the signature verifies with respect to vk. (Non-adaptive) soundness is straightforward to argue. Note that, for space bounded computations, this protocol provides a publicly verifiable alternative to the recent delegation scheme of Kalai et. al [KRR13]. Furthermore, the worker

only incurs overhead relative to the RAM complexity of π. Note that here $\mathcal{O}$ operates on a single-input program so the security loss in the reduction is minimal.

Functional encryption. Plugging our construction instead of the indistinguishability obfuscators in the functional encryption scheme of Garg et al. [GGH+13], we obtain functional encryption where the complexity of decryption is proportional only to the RAM complexity of the underlying function.

NIZK. Plugging our construction instead of the indistinguishability obfuscators in the non-interactive zero knowledge protocol of Sahai and Waters [SW14], we obtain the first general NIZK where the complexity of the prover is proportional only to the RAM complexity of the underlying relation.

Multiparty computation. Plugging our construction instead of the indistinguishability obfuscators in the multiparty computation protocol of Garg et al. [GGHR14], which in turn uses the underlying protocol of Gordon et al. [GKK+12], we obtain the first constant-round multiparty protocol where the overhead of the parties is proportional only to the RAM complexity of the underlying function.

1.1 Our techniques

Succinct obfuscation of iterated functionalities. The key component in our solution approach is a mechanism for succinctly obfuscating iterated computations. Consider the following problem: We are given a function (in form of a circuit c) from $\{0,1\}^n$ to $\{0,1\}^n$. We would like to create an indistinguishability obfuscation of the function c^t for some t. Clearly, one can just directly obfuscate the circuit c_t which consists of t copies of c composed in sequence. However, can one do better? Specifically, can one come up with an obfuscation mechanism that, given c and t, outputs an obfuscated circuit $\mathcal{O}(c) = c'$, whose size is only polylogarithmic in t, such that $c'^t(x) = c(x)^t$ on all x, and such that c' is obfuscated in the sense that no intermediate results are revealed by c'? Furthermore, can this be done assuming only indistinguishability obfuscation for circuits?

We show how to do this using a mechanism for encapsulating the "state" information that's passed from one iteration of c' to the next. The basic idea is simple: have c' increment an "iteration number" in each iteration, and then encrypt and authenticate the state information via keys that are known only to c'. (The idea has been used before in different contexts, eg in [CGH04, CPS13].) Here however proving security of this process is tricky, since the encryption and authentication keys are only protected by indistinguishability obfuscation. In fact, the argument appears to be circular: indistinguishability of the obfuscation hinges on the fact that c' rejects fake "state information from the previous round", but rejecting such fake information hinges on the obfuscation being secure.

We get around this circularity using a primitive that we call *Asymmetrically Constrained Encapsulation (ACE)*. The idea is to have determinstic authenticated encryption where one can puncture keys only at the reciving end, without puncturing them at the sending end. See formal definition within. We construct an ACE scheme using the Sahai-Waters encapsulation mechanism on top of constrained (puncturable) PRFs [SW14], along with any indistinguishability obfuscator.

Using ACE we show that $\mathcal{O}(c_1, t) \approx \mathcal{O}(c_2, t)$ for any two circuits c_1, c_2 such that $c_1^t(x) = c_2^t(x)$ for all x, even if, say, $c_1^{t'}(x) \neq c_2^{t'}(x)$ for some $t' < t$. However, this comes with a caveat: At each iteration, we incur a security loss that is proportional to the number of values in the image of c (namely, 2^n in this case). This technique is thus most powerful when applied to circuits with small range.

Obfuscating RAM programs with fixed memory access pattern. Our next step is to obfuscate RAM programs where the memory access pattern is fixed in advance, or in other words construct an obfuscator $\mathcal{O}$ with the right efficiency parameters, and where $\mathcal{O}(\pi_1) = \mathcal{O}(\pi_2)$ for any π_1, π_2 such that (a) $\pi_1(x) = \pi_2(x)$ for all x, and (b) π_1, π_2 have the same runtime and same memory access pattern for all x. In a sense, this step can be seen as an extension of the generic iterated circuits obfuscation mechanism described above to the case where the circuit takes its inputs in "small pieces". Indeed, here the security loss incurred is significantly smaller.

We do this in two steps. First we construct a "single input obfuscation" mechanism, or in other words a "succinct garbling" mechanism for such programs. The mechanism is straightforward: Given a RAM program π and input x, $\mathcal{O}$ first chooses key a for an ACE scheme. It then prepares an initial memory map μ. μ consists of $s(|x|)$ memory elements, where each element is the result of applying ACE with the chosen key a to a "plaintext cell", where the contents of the plaintext cells are defined as follows. (Recall, $s(|x|)$ is the maximum memory size for inputs of size $|x|$.) Each plaintext memory cell contains its address (namely, its index in the sequence of cells) and a timestamp initialized to 1. In addition, the first n plaintext cells contain the corresponding bits of the input, and the remaining cells contain 0.

Next $\mathcal{O}$ prepares the garbled program π'. π' is the result of applying an indistinguishability obfuscator to the following program π'', represented as a circuit. π'' has the key a. It then runs the underlying π in the natural way where the memory accesses made by π are translated to the actual, encapsulated memory. In doing so, π'' verifies the validity of the decapsulated information and updates the time counter values of the written cells. Once π generates output, π'' outputs whatever π outputs.

Correctness under honest execution and the complexity parameters of $\mathcal{O}$ are easy to verify. For security, we show a strong simulatability property: The information gathered by any adversary given $\mathcal{O}(\pi, x)$ is simulatable given only $\pi(x)$, together with the (fixed) memory access pattern of π. This is done in t steps, where t is the runtime of $\pi(x)$. In step i we consider the hybrid execution where the memory is initialized to its contents at step i of the computation, and π'' is modified so that in the first i steps of the computation it essentially idles and only updates the time counters of the accessed memory locations. The actual computation then resumes from step i. We then reduce the indistinguishability of any two consecutive hybrids to the security of the underlying ACE scheme.

The security loss in this reducion is minimal. On the down side, our proof of security crucially uses the fact that $\mathcal{O}$ generates the entire memory map in advance; this is what causes the initial obfuscation to be proportional to the space requirements of the plaintext RAM program.

This single-input obfuscation can be extended in a number of ways to an obfuscation that takes multiple inputs. Perhaps the most direct way is to use the approach of Gentry et al. [GHRW14], namely to have $\tilde{\mathcal{O}}(\pi)$ be the indistinguishability obfuscation of the circuit that has a puncturable PRF f hardwired, and given and x, runs the obfuscator $\mathcal{O}$ described above on π, x, and random input $f(\pi, x)$. Alternatively, keep the program π' unchanged, and then add to the obfuscation another program ν, which is the result of applying an indistinguishability obfuscator to the circuit that has the same ACE key as π', and on input x generates the initial encapsulated memory map μ that corresponds to x. Here care should be taken to prevent "mix and match" attacks that combine memory maps from different executions.

Obfuscating general RAM programs. Our final step addresses RAM programs that have arbitrary memory access patterns. Here we employ an Oblivious RAM (ORAM) mechanism [**?**] for hiding the memory access pattern of the plaintext program, as well as a "freshness guarntor (FG) mechanism" that keeps track of the last time that a memory cell was written to. We employ both mechanisms directly to the plaintext program π. The randomness for the ORAM is obtained by applying a puncturable PRF to the actual memory contents read or written. Next, we apply the single-input mechanism from the previous step.

For our scheme to work, we need ORAM and FG mechanisms with special properties. For efficiency, we need that the computational overhead of both mechanism will be only polylogarithmic per read or write operation *in the worst case* (as opposed to amortized). For security, we need the following strong security property from the ORAM: For any i, the sequence of actual memory accesses that correspond to the ith virtual memory access is indistingushable from an actual access pattern that comes from some predetermined distribution. Furthermore, this is so even given all the actual access patterns in the execution prior to the ith virtual memory access. For the FG mechanism we need the inductive property stating essentially that the correctness of the result of the mechanism for a given memory cell depends only on the past write accesses to that very cell.

We then demonstrate that existing ORAM and FG mechanisms satisfy the desired properties. Specifically, we use the ORAM mechanism of Chung and Pass [CP13] and the FG mechanism of [GHRW14].

We note that the proof of security of this scheme is delicate and requires much care. Again, the main

source of difficulty results from the fact that one cannot directly argue that the random choices of the ORAM are hidden from the adversary. Rather, the hiding and validity properties need to be argued via a long sequence of hybrids - even for moving between two consecutive computational steps in the underlying program. Indeed, the number of hybrids (hence the security loss) is polynomial in t.

Organization. Section 2 recalls indistinguishability obfuscation and other basic definitions. Section 3 presents and constructs Asymmetrically Constrained Encapsulation (ACE). Section 4 presents our succinct obfuscation mechanism for iterated circuits. We note that this mechanism is not directly used in the construction of RAM obfuscation; rather, it is a separate result which is of interest on its own. Section 5 presents two general structural theorems regarding the application of ACE schemes to circuits of a special form. These theorems simplify and clarify the analysis of the scheme for obfuscating RAM programs, presented in Section 6.

2 Preliminaries

We will use λ to denote the security parameter. Two distribution ensembles $\mathcal{A} = \{A_\lambda\}_{\lambda>0}$ and $\mathcal{B} = \{B_\lambda\}_{\lambda>0}$ are computationally indistinguishable if for every probabilistic polynomial time (PPT) distinguisher D, there is a negligible function $\mu(\cdot)$ such that for every λ, $\left|\Pr[x \leftarrow A_\lambda : D(x) = 1] - \Pr[x \leftarrow B_\lambda : D(x) = 1]\right| \leq \mu(\lambda)$. In this case, we say that $\mathcal{A} \approx_c \mathcal{B}$. Throughout the paper, we will use OWF to denote an injective one-way function.

2.1 Injective Pseudorandom Generators

A pseudorandom generator (PRG) is a polynomial-time algorithm G that maps n-bit strings into m-bit strings where $m > n$, and $G(U_n) \approx_c U_m$. We will make use of pseudorandom generators that are injective functions from $\{0,1\}^n \to \{0,1\}^m$. Such PRGs can be based on injective one-way functions.

2.2 Puncturable Pseudorandom Functions

A puncturable family of PRFs are a special case of constrained PRFs [BW13, BGI14, KPTZ13], where the PRF is defined on all input strings except for a set of size polynomial in the security parameter. Below we recall their definition, as given by [SW14].

Syntax A *puncturable* family of PRFs is defined by a tuple of algorithms $(\mathsf{Gen}_{\mathsf{PRF}}, \mathsf{Puncture}_{\mathsf{PRF}}, \mathsf{Eval}_{\mathsf{PRF}})$ and a pair of polynomials $n(\cdot)$ and $m(\cdot)$:

- **Key Generation** $\mathsf{Gen}_{\mathsf{PRF}}$ is a PPT algorithm that takes as input the security parameter λ and outputs a PRF key K
- **Punctured Key Generation** $\mathsf{Puncture}_{\mathsf{PRF}}(K, S)$ is a PPT algorithm that takes as input a PRF key K, a set $S \subset \{0,1\}^{n(\lambda)}$ and outputs a punctured key $K\{S\}$
- **Evaluation** $\mathsf{Eval}_{\mathsf{PRF}}(K, x)$ is a deterministic algorithm that takes as input a (punctured or regular) key K, a string $x \in \{0,1\}^{n(\lambda)}$ and outputs $y \in \{0,1\}^{m(\lambda)}$

Definition 1. *A family of PRFs* $(\mathsf{Gen}_{\mathsf{PRF}}, \mathsf{Puncture}_{\mathsf{PRF}}, \mathsf{Eval}_{\mathsf{PRF}})$ *is* puncturable *if it satisfies the following properties :*

- **Functionality preserved under puncturing.** *Let* $K \leftarrow \mathsf{Gen}_{\mathsf{PRF}}$, $K\{S\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K, S)$. *Then, for all* $x \notin S$, $\mathsf{Eval}_{\mathsf{PRF}}(K, x) = \mathsf{Eval}(K\{S\}, x)$.

- **Pseudorandom at punctured points.** *For every PPT adversary (A_1, A_2) such that $A_1(1^\lambda)$ outputs a set $S \subset \{0,1\}^{n(\lambda)}$ and $x \in S$, consider an experiment where $K \leftarrow \mathsf{Gen}_{\mathsf{PRF}}$ and $K\{S\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K, S)$. Then*

$$\left|Pr[A_2(K\{S\}, x, \mathsf{Eval}_{\mathsf{PRF}}(K, x)) = 1] - Pr[A_2(K\{S\}, x, U_{m(\lambda)}) = 1]\right| \leq \mathsf{negl}(\lambda)$$

where U_ℓ denotes the uniform distribution over ℓ bits.

As observed by [BW13, BGI14, KPTZ13], the GGM construction [GGM86] of PRFs from one-way functions yields puncturable PRFs.

Theorem 1 ([GGM86, BW13, BGI14, KPTZ13])**.** *If one-way functions exist, then for all polynomials $n(\lambda)$ and $m(\lambda)$, there exists a puncturable PRF family that maps $n(\lambda)$ bits to $m(\lambda)$ bits.*

Remark 1. *In the above construction, the size of the punctured key K_S grows linearly with the size of the puncture set S.*

Remark 2. *We will also be using injective puncturable PRFs in our constructions. Such PRFs can be based on injective one-way functions.*

2.3 Indistinguishability Obfuscation for Circuits

Here we recall the notion of indistinguishability obfuscation that was defined by Barak et al. [BGI+01]. Intuitively speaking, we require that for any two circuits C_1 and C_2 that are "functionally equivalent" (i.e., for all inputs x in the domain, $C_1(x) = C_2(x)$), the obfuscation of C_1 must be computationally indistinguishable from the obfuscation of C_2. Below we present the formal definition following the syntax of [GGH+13].

Definition 2 (Indistinguishability Obfuscation for Circuits)**.** *A uniform PPT machine $\mathsf{i}\mathcal{O}$ is called an indistinguishability obfuscator for a circuit class $\{\mathcal{C}_\lambda\}$ if the following holds:*

- **Correctness:** *For every $\lambda \in \mathbb{N}$, for every $C \in \mathcal{C}_\lambda$, for every input x in the domain of C, we have that*

$$\Pr[C'(x) = C(x) : C' \leftarrow \mathsf{i}\mathcal{O}(C)] = 1.$$

- **Efficiency:** *There exists a polynomial P such that for every $\lambda \in \mathbb{N}$, for every $C \in \mathcal{C}_\lambda$, $|\mathsf{i}\mathcal{O}(C)| \leq P(\lambda, |C|)$.*

- **Indistinguishability:** *For every $\lambda \in \mathbb{N}$, for all pairs of circuits $C_0, C_1 \in \mathcal{C}_\lambda$, if $C_0(x) = C_1(x)$ for all inputs x, then for all PPT adversaries $\mathcal{A}$, we have:*

$$|\Pr[\mathcal{A}(\mathsf{i}\mathcal{O}(C_0)) = 1] - \Pr[\mathcal{A}(\mathsf{i}\mathcal{O}(C_1)) = 1]| \leq \mathsf{negl}(\lambda).$$

3 Asymmetrically Constrained Encapsulation

We define and construct a new primitive that we call Asymmetrically Constrained Encapsulation (ACE). Essentially, an ACE scheme is a deterministic authenticated secret key encryption scheme, with the following additional properties:

1. For each message and each key, there is at most a single string that correctly decapsulates to m under key k.

2. The full decapsulation algorithm is indistinguishable from a constrained version of this algorithm, where the key is punctured at a possibly large set of points (namely at some $S \subset M$ which is decidable by a small circuit). Furthermore, this is so even when given the corresponding *unpunctured* encapsulation algorithm, as long as no message that corresponds to an encoding $c \in S$ is known.

This property will be central in our analysis of iterated circuit obfuscation.

3.1 Definition

Let $\mathcal{M} = \{0,1\}^n$ denote the message space, where $n = \mathsf{poly}(\lambda)$. We will have two parameters for a ACE scheme: a *ciphertext security* parameter ℓ and a *set description size* parameter s that will be described later.

An (ℓ, s)-asymmetrically constrained encapsulation scheme consists of five (deterministic or randomized) algorithms $(\mathsf{Setup}, \mathsf{GenEK}, \mathsf{GenDK}, \mathsf{Enc}, \mathsf{Dec})$ described as follows:

- **Setup:** $\mathsf{Setup}(1^\lambda)$ is a randomized algorithm that takes as input the security parameter λ and outputs a secret key SK.
- **(Constrained) Key Generation:** Let $S \subset \mathcal{M}$ be any set whose membership is decidable by a circuit C_S. We say that S is *admissible* if $|C_S| \leq s$. Intuitively, the set size parameter s denotes the upper bound on the size of circuit description C_S for any set $S \subset M$ that is input to the encapsulation and decapsulation key generation algorithms described below.
  - $\mathsf{GenEK}(SK, C_S)$ takes as input the secret key SK of the scheme and the description of circuit C_S for an admissible set S. It outputs an encapsulation key $EK\{S\}$. We write EK to denote $EK\{\emptyset\}$.
  - $\mathsf{GenDK}(SK, C_S)$ also takes as input the secret key SK of the scheme and the description of circuit C_S for an admissible set S. It outputs a decapsulation key $DK\{S\}$. We write DK to denote $DK\{\emptyset\}$.

  Unless mentioned otherwise, we will only consider admissible sets $S \subset \mathcal{M}$.
- **Encapsulation:** $\mathsf{Enc}(EK', m)$ is a deterministic algorithm that takes as input an encapsulation key EK' (that may be constrained) and a message $m \in \mathcal{M}$ and outputs a ciphertext c or the reject symbol $\perp$.
- **Decapsulation:** $\mathsf{Dec}(DK', c)$ is a deterministic algorithm that takes as input a decapsulation key DK' (that may be constrained) and a ciphertext c and outputs a message $m \in M$ or the reject symbol $\perp$.

(ℓ, s)-Efficiency. We require that the algorithm Setup is polynomial-time in the security parameter. Furthermore, we require that the size of any encapsulation key EK' and decapsulation key DK' (either of which may be constrained) is bounded by some polynomial in λ, ℓ, and s. Similarly, the running time of each of the algorithms GenEK, GenDK, Enc, and Dec is bounded by some polynomial in λ, ℓ, and s.

Correctness. An ACE scheme is correct if the following properties hold:

1. *Correctness of Decapsulation*: For all sets $S, S' \subset M$ all $m \notin S \cap S'$,

$$\Pr[SK \leftarrow \mathsf{Setup}(1^\lambda), \mathsf{Dec}(\mathsf{GenDK}(SK, C_S), \mathsf{Enc}(\mathsf{GenEK}(SK, C_{S'}), m)) = m] = 1.$$

   Informally this says that $\mathsf{Dec} \circ \mathsf{Enc}$ is the identity on messages which are in neither of the punctured sets.

2. *Equivalence of Constrained Encapsulation*: For all $S \subset M$ and all $m \notin S$,

$$\Pr[SK \leftarrow \mathsf{Setup}(1^\lambda), \mathsf{Enc}(\mathsf{GenEK}(SK, C_S), m) = \mathsf{Enc}(\mathsf{GenEK}(SK, \emptyset), m)] = 1.$$

   This says that the only functional difference between a punctured $EK\{S\}$ and an unpunctured EK is on the punctured set S.

3. *Safety of Constrained Decapsulation*: For all $S \subset M$ and all encapsulations c,

$$\Pr[SK \leftarrow \mathsf{Setup}(1^\lambda), DK \leftarrow \mathsf{GenDK}(SK, C_S), m \leftarrow \mathsf{Dec}(DK, c); m \in S] = 0$$

   This says that a punctured $DK\{S\}$ will never decapsulate to a message in S.

4. *Equivalence of Constrained Decapsulation*: For all $S \subset M$ and all encapsulations c,

$$\Pr\left[\begin{array}{l} SK \leftarrow \mathsf{Setup}(1^\lambda), DK \leftarrow \mathsf{GenDK}(SK, \emptyset), DK\{S\} \leftarrow \mathsf{GenDK}(SK, C_S), \\ m \leftarrow \mathsf{Dec}(DK, c), m' \leftarrow \mathsf{Dec}(DK\{S\}, c);\ \text{either } m \in S \text{ or } m' = m \end{array}\right] = 1,$$

This says that the only functional difference between a punctured $DK\{S\}$ and an unpunctured DK is on encapsulations of the punctured set S.

Unique Encapsulations. Let $SK \leftarrow \mathsf{Setup}(1^\lambda)$. For any message $m \in \mathcal{M}$, there exists an encapsulation c_m such that

$$\Pr[\mathsf{Dec}(\mathsf{GenDK}(SK, \emptyset), c_m) = m] = 1$$

and

$$c \neq c_m \implies \Pr[\mathsf{Dec}(\mathsf{GenDK}(SK, \emptyset), c) = m] = 0$$

Security of Constrained Decapsulation. We describe security of constrained decapsulation as a multi-stage game between an adversary $\mathcal{A}$ and a challenger.

- *Setup:* $\mathcal{A}$ choose sets S_0, S_1, U s.t. $S_0 \subseteq S_1 \subseteq U \subseteq M$ and sends their circuit descriptions (C_{S_0}, C_{S_1}, C_U) to the challenger. $\mathcal{A}$ also sends arbitrary polynomially many messages $m_1, \ldots, m_l$ such that $m_i \notin S_1 \setminus S_0$.

  The challenger computes $SK \leftarrow \mathsf{Setup}(1^\lambda)$ and $EK\{U\} \leftarrow \mathsf{GenEK}(SK, C_U)$. Further, it chooses a bit $b \in \{0,1\}$ and computes $DK\{S_b\} \leftarrow \mathsf{GenDK}(SK, C_{S_b})$. It sends $EK\{U\}, DK\{S_b\}$ to the challenger, along with $\mathsf{Enc}(EK, m_i)$ for each $i \in \{1, \ldots, l\}$.

- *Guess:* $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$.

The advantage of $\mathcal{A}$ in this game is defined as $\mathsf{adv}_{\mathcal{A}} = \Pr[b' = b] - \frac{1}{2}$. We require that there exists a function $\epsilon_{S,S'}(\cdot)$ s.t. $\mathsf{adv}_{\mathcal{A}} = \epsilon_{S,S'}(\lambda)$.

Remark 3. *In the above definition, we do not necessarily require ϵ_{S_0,S_1} to be negligible. Looking ahead, in our construction of ACE, for any S_0, S_1, we have $\epsilon_{S_0,S_1}(\lambda) = \mathsf{poly}(|S_1 \setminus S_0|, \lambda) \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{i\mathcal{O}}(\lambda))$. When $|S_1 \setminus S_0|$ is super-polynomial, this is negligible assuming subexponential hardness of one-way functions as well as $i\mathcal{O}$. As we will see later, this weaker definition suffices for the applications of ACE in this paper.*

(Selective) ℓ-Indistinguishability of Ciphertexts. We describe (selective) security of ciphertexts of a asymmetrically constrained encapsulation scheme as a multi-stage game between an adversary $\mathcal{A}$ and a challenger.

- *Setup:* $\mathcal{A}$ chooses sets $S, U \subset M$ and ℓ challenge message pairs $(m_1^0, m_1^1), \ldots, (m_\ell^0, m_\ell^1)$, where every $m_i^b \in S \cap U$. $\mathcal{A}$ sends the message pairs along with circuits (C_S, C_U) to the challenger.

  The challenger chooses a bit $b \in \{0,1\}$ and computes the following: (a) $SK \leftarrow \mathsf{Setup}(1^\lambda)$, (b) $EK \leftarrow \mathsf{GenEK}(SK, \emptyset)$, (c) $c_i^b \leftarrow \mathsf{Enc}(EK, m_i^b)$, $c_i^{1-b} \leftarrow \mathsf{Enc}(EK, m_i^{1-b})$ for every $i \in [\ell]$, and (d) $EK\{U\} \leftarrow \mathsf{GenEK}(SK, C_U)$, $DK\{S\} \leftarrow \mathsf{GenDK}(SK, C_S)$. Finally, it sends the following tuple to $\mathcal{A}$:

$$\left(EK\{S\}, DK\{U\}, c_1^b, \ldots, c_\ell^b\right).$$

- *Guess:* $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$.

The advantage of $\mathcal{A}$ in this game is defined as $\mathsf{adv}_{\mathcal{A}} = \Pr[b' = b] - \frac{1}{2}$.

We require that for all PPT adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ s.t. $\mathsf{adv}_{\mathcal{A}} = \mathsf{negl}(\lambda)$.

Definition 3. *A (ℓ, s)-ACE scheme is* ℓ-secure *if it satisfies the properties of correctness, unique ciphertexts, security of constrained decapsulation and selective ℓ-indistinguishability of ciphertexts.*

3.2 Construction of ACE

We now present a construction of an asymmetrically constrainable encapsulation scheme. Our scheme is based on the "hidden triggers" mechanism in the deniable encryption scheme of [SW14], and additionally makes use of indistinguishability obfuscation.

Notation. Let $\mathcal{F}_1 = \{F_{1,k}\}_{k\in\{0,1\}^\lambda}$ be a puncturable *injective* pseudorandom function family, where $F_{1,k} : \{0,1\}^n \to \{0,1\}^{2n}$. Let $\mathcal{F}_2 = \{F_{2,k}\}_{k\in\{0,1\}^\lambda}$ be another puncturable pseudorandom function family, where $F_{2,k} : \{0,1\}^{2n} \to \{0,1\}^n$. Let $\mathsf{i}\mathcal{O}$ be an indistinguishability obfuscator for all circuits.

Let ℓ be the ciphertext security parameter and s denote the set size parameter for ACE. Let $p = \mathsf{poly}(\lambda, \ell, s)$ be a parameter to be determined later.

Construction. We now proceed to describe our scheme $\mathcal{ACE} = (\mathsf{Setup}, \mathsf{GenEK}, \mathsf{GenDK}, \mathsf{Enc}, \mathsf{Dec})$.

$\mathsf{Setup}(1^\lambda)$**:** The setup algorithm first samples fresh keys $K_1 \leftarrow \mathsf{Gen}_{\mathsf{PRF}}(1^\lambda)$ and $K_2 \leftarrow \mathsf{Gen}_{\mathsf{PRF}}(1^\lambda)$ for the puncturable PRF families $\mathcal{F}_1$ and $\mathcal{F}_2$ respectively.

Abusing notation, we will sometimes use F_i to refer to F_{i,K_i}.

$\mathsf{GenEK}((K_1, K_2), C_S)$**:** The encapsulation key generation algorithm takes as input keys K_1, K_2 and the circuit description C_S of an admissible set S. It prepares a circuit representation of $\mathcal{G}_{\mathsf{enc}}$ (Algorithm 1), padded to be of size p. Next, it computes the encapsulation key $EK\{S\} \leftarrow \mathsf{i}\mathcal{O}(\mathcal{G}_{\mathsf{enc}})$ and outputs the result.

Algorithm 1: (Constrained) Encapsulation $\mathcal{G}_{\mathsf{enc}}$

Data: $K1$, $K2$, circuit C_S
Input: message m
if $C_S(m)$ **then return** $\perp$;
else
  $\alpha \leftarrow F_1(m)$;
  $\beta \leftarrow F_2(\alpha) \oplus m$;
  return $\alpha \| \beta$
end

$\mathsf{GenDK}((K_1, K_2), C_S)$**:** The decapsulation key generation algorithm takes as input keys K_1, K_2 and the circuit description C_S of an admissible set S. It prepares a circuit representation of $\mathcal{G}_{\mathsf{dec}}$ (Algorithm 2), padded to be of size p. It then computes the decapsulation key $DK\{S\} \leftarrow \mathsf{i}\mathcal{O}(\mathcal{G}_{\mathsf{dec}})$ and outputs the result.

Algorithm 2: (Constrained) Decapsulation $\mathcal{G}_{\mathsf{dec}}$

Data: $K1$, $K2$, circuit C_S
Input: encapsulation c
parse c as $\alpha \| \beta$;
$m \leftarrow F_2(\alpha) \oplus \beta$;
if $C_S(m)$ **then return** $\perp$;
else if $\alpha \neq F_1(m)$ **then return** $\perp$;
else return m;

$\mathsf{Enc}(EK', m)$**:** The encapsulation algorithm simply runs the encapsulation key program EK' on message m to compute the ciphertext $c \leftarrow EK'(m)$.

$\mathsf{Dec}(DK', c)$**:** The decapsulation algorithm simply runs the decapsulation key program DK' on the input ciphertext c and returns the output $DK'(c)$.

This completes the description of our construction of $\mathcal{ACE}$.

(ℓ, s)**-Efficiency.** It follows from the description that Setup runs in time $\mathsf{poly}(\lambda)$. Now, note that both $\mathcal{G}_{\mathsf{enc}}$ and $\mathcal{G}_{\mathsf{dec}}$ are of size $p = \mathsf{poly}(\lambda, \ell, s)$. From the efficiency property of $\mathsf{i}\mathcal{O}$, it follows any (constrained) encapsulation key EK' and any (constrained) decapsulation key DK' is of size $\mathsf{poly}(\lambda, \ell, s)$. It then follows from the description of the scheme that GenEK, GenDK, Enc, and Dec run in time $\mathsf{poly}(\lambda, \ell, s)$.

Theorem 2. *Assuming indistinguishability obfuscation for all circuits and one-way functions, for all* $\ell, s \in \mathsf{poly}(\lambda)$*, the proposed scheme* $\mathcal{ACE} = (\mathsf{Setup}, \mathsf{GenEK}, \mathsf{GenDK}, \mathsf{Enc}, \mathsf{Dec})$ *is* ℓ*,*s*-efficient.*

3.3 Proof of Security

Correctness. We now argue correctness:

1. *Correctness of Decapsulation* Now, let $S, S' \in M$ and let $m \notin S' \cap S$ be an input to the encapsulation key program. Then because of line 1 in $\mathcal{G}_{\mathsf{enc}}$ and line 3 in $\mathcal{G}_{\mathsf{dec}}$, and the correctness of $\mathsf{i}\mathcal{O}$, the correctness of decapsulation is inherited from the correctness of the unpunctured, unobfuscated scheme, which is easy to check.

2. *Equivalence of Constrained Encapsulation*: This follows from the fact that the only difference between a constrained and an unconstrained encapsulation key is in line 1 of Algorithm 1.

3. *Safety of Constrained Decapsulation*: This follows from line 3 of Algorithm 2.

4. *Equivalence of Constrained Decapsulation*: This follows from the fact that the only difference between a constrained and an unconstrained decapsulation key is in lines 4-5 of Algorithm 2.

Uniqueness of Encapsulations. The unique encapsulation property follows from the injectivity of F_1.

Security of Constrained Decapsulation. We now prove that $\mathcal{ACE}$ satisfies security of constrained decapsulation.

Lemma 1. *The proposed scheme* $\mathcal{ACE}$ *satisfies security of constrained decapsulation.*

Proof. Let S_0, S_1, U be arbitrary subsets of M s.t. $S_0 \subseteq S_1 \subseteq U \subseteq M$ and let C_{S_0}, C_{S_1}, C_U be their circuit descriptions. Let $SK \leftarrow \mathsf{Setup}(1^\lambda)$ and $EK\{U\} \leftarrow \mathsf{GenEK}(SK, C_U)$. Further, for $b \in \{0, 1\}$, let $DK\{S_b\} \leftarrow \mathsf{GenDK}(SK, C_{S_b})$. We now argue that no PPT distinguisher can distinguish between $EK\{U\}, DK\{S_0\}$ and $EK\{U\}, DK\{S_1\}$ with advantage more than $\epsilon_{S_0,S_1} = |S_1 \setminus S_0| \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$.

We will prove this via a sequence of $|S_1 \setminus S_0|$ main hybrid experiments H_i where in experiment H_0, the decapsulation key is $DK\{S_0\}$ whereas in experiment $H_{|S_1 \setminus S_0|}$, the decapsulation key is $DK\{S_1\}$. For every i, we will prove that H_i and H_{i+1} are indistinguishable.

We now proceed to give details. Let u_i denote the lexicographically ith element of $S_1 \setminus S_0$. Throughout the experiments, we will refer to the encapsulation key and decapsulation key programs given to the distinguisher as EK' and DK' respectively. Similarly, (unless stated otherwise) we will refer to the unobfuscated algorithms underlying EK' and DK' as $\mathcal{G}'_{\mathsf{enc}}$ and $\mathcal{G}'_{\mathsf{dec}}$, respectively.

Hybrid H_i**:** In the ith hybrid, the decapsulation key program DK' first checks whether $m \in S_1$ *and* $m \leq u_i$. If this is the case, then it simply outputs $\perp$. Otherwise, it behaves in the same manner as $DK\{S_0\}$. The underlying unobfuscated program $\mathcal{G}'_{\mathsf{dec}}$ is described in Algorithm 3.

By convention, $u_0 = -\infty$. Therefore, in experiment H_0, DK' has the same functionality as $DK\{S_0\}$, and in $H_{|S_1 \setminus S_0|}$, DK' has the same functionality as $DK\{S_1\}$.

We now construct a series of intermediate hybrid experiments $H_{i,0}, \ldots, H_{i,7}$ where $H_{i,0}$ is the same as H_i and $H_{i,7}$ is the same as H_{i+1}. For every j, we will prove that $H_{i,j}$ is computationally indistinguishable from $H_{i,j+1}$, which will establish that H_i and H_{i+1} are computationally indistinguishable.

Hybrid $H_{i,0}$**:** This is the same as experiment H_i.

Algorithm 3: (Constrained) Decapsulation $\mathcal{G}'_{\mathsf{dec}}$ in Hybrid i

Input: ciphertext c
Data: $K1$, $K2$
1 parse c as $\alpha \| \beta$;
2 $m \leftarrow F_2(\alpha) \oplus \beta$;
3 **if** $m < u_i$ *and* $m \in S_1$ **then return** $\perp$;
4 **else if** $m \in S_0$ **then return** $\perp$;
5 **else if** $\alpha \neq F_1(m)$ **then return** $\perp$;
6 **else return** m;

Hybrid $H_{i,1}$**:** This is the same as experiment $H_{i,0}$ except that we modify $\mathcal{G}'_{\mathsf{dec}}$ as follows. If the decapsulated message $m = u_i$, then instead of checking whether $\alpha \neq F_1(m)$ in line 5, $\mathcal{G}'_{\mathsf{dec}}$ now checks whether $PRG(\alpha) \neq PRG(F_1(u_i))$, where PRG is an injective pseudo random generator.

Hybrid $H_{i,2}$**:** This is the same as experiment $H_{i,1}$ except that we modify $\mathcal{G}'_{\mathsf{dec}}$ as follows:

- Hardwire the value $z = PRG(F_1(u_i))$ in $\mathcal{G}'_{\mathsf{dec}}$. Now, when the decapsulated message $m = u_i$, then $\mathcal{G}'_{\mathsf{dec}}$ simply checks whether $PRG(\alpha) \neq z$.
- The PRF key K_1 in $\mathcal{G}'_{\mathsf{dec}}$ is punctured at u_i, i.e., K_1 is replaced with $K_1\{u_i\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_1, u_i)$.

Hybrid $H_{i,3}$**:** This is the same as experiment $H_{i,2}$ except that we modify the (unobfuscated) program $\mathcal{G}'_{\mathsf{enc}}$ underlying EK' such that the PRF key K_1 hardwired in $\mathcal{G}_{\mathsf{enc}}$ is replaced with punctured key $K_1\{u_i\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_1, u_i)$.

Hybrid $H_{i,4}$**:** This is the same as experiment $H_{i,3}$ except that the hardwired value z in $\mathcal{G}'_{\mathsf{dec}}$ is now computed as $PRG(r)$ where r is a randomly chosen string.

Hybrid $H_{i,5}$**:** This is the same as experiment $H_{i,4}$ except that the hardwired value z in $\mathcal{G}'_{\mathsf{dec}}$ is now set to be a randomly chosen string r'.

Hybrid $H_{i,6}$**:** This is the same as experiment $H_{i,5}$ except that we now modify $\mathcal{G}'_{\mathsf{dec}}$ such that it outputs $\perp$ when the decapsulated message $m = u_i$. An equivalent description of $\mathcal{G}'_{\mathsf{dec}}$ is that in line 3, it now checks whether $m < u_{i+1}$ instead of $m < u_i$.

Hybrid $H_{i,7}$**:** This is the same as experiment $H_{i,6}$ except that the PRF key corresponding to F_1 is unpunctured in both $\mathcal{G}'_{\mathsf{enc}}$ and $\mathcal{G}'_{\mathsf{dec}}$. That is, we replace $K_1\{u_i\}$) with K_1 in both $\mathcal{G}'_{\mathsf{enc}}$ and $\mathcal{G}'_{\mathsf{dec}}$. Note that experiment $H_{i,7}$ is the same as experiment H_{i+1}.

This completes the description of the hybrid experiments. We now argue their indistinguishability.

Indistinguishability of $H_{i,0}$ **and** $H_{i,1}$**.** Since PRG is injective, we have that the following two checks are equivalent: $\alpha \neq F_1(m)$ and $PRG(\alpha) \neq PRG(F_1(m))$. Then, we have that the algorithms $\mathcal{G}'_{\mathsf{dec}}$ in $H_{i,0}$ and $H_{i,1}$ are functionally equivalent. Therefore, the indistinguishability of $H_{i,0}$ and $H_{i,1}$ follows from the security of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Indistinguishability of $H_{i,1}$ **and** $H_{i,2}$**.** Let $\mathcal{G}'_{\mathsf{dec}}$ (resp., $\mathcal{G}''_{\mathsf{dec}}$) denote the unobfuscated algorithms underlying the decapsulation key program DK' in experiments $H_{i,1}$ (resp., $H_{i,2}$). We will argue that $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ are functionally equivalent. The indistinguishability of $H_{i,0}$ and $H_{i,1}$ then follows from the security of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Let c_i denote the *unique* ciphertext such that $\mathsf{Dec}(DK, c_i) = u_i$ (where DK denotes the unconstrained decapsulation key program). First note that on any input $c \neq c_i$, both $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ have identical behavior, except that $\mathcal{G}'_{\mathsf{dec}}$ uses the PRF key K_1 while $\mathcal{G}''_{\mathsf{dec}}$ uses the punctured PRF key $K_1\{u_i\}$. Since the punctured PRF scheme preserves functionality under puncturing, we have that $\mathcal{G}'_{\mathsf{dec}}(c) = \mathcal{G}''_{\mathsf{dec}}(c)$. Now, on input c_i, after decapsulating c_i to obtain u_i, $\mathcal{G}'_{\mathsf{dec}}$ computes $PRG(F_1(u_i))$ and then checks whether $PRG(\alpha_i) \neq$

$PRG(F_1(u_i))$ whereas $\mathcal{G}''_{\mathsf{dec}}$ simply checks whether $PRG(\alpha_i) \neq z$. But since the value z hardwired in $\mathcal{G}''_{\mathsf{dec}}$ is equal to $PRG(F_1(u_i))$, we have that $\mathcal{G}'_{\mathsf{dec}}(c_i) = \mathcal{G}''_{\mathsf{dec}}(c_i)$.

Thus we have that $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ are functionally equivalent.

Indistinguishability of $H_{i,2}$ and $H_{i,3}$. Let $\mathcal{G}'_{\mathsf{enc}}$ (resp., $\mathcal{G}''_{\mathsf{enc}}$) denote the unobfuscated algorithms underlying the encapsulation key program EK' in experiments $H_{i,1}$ and $H_{i,2}$. Note that the only difference between $\mathcal{G}'_{\mathsf{enc}}$ and $\mathcal{G}''_{\mathsf{enc}}$ is that the former contains the PRF key K_1 while the latter contains the punctured PRF key $K_1\{u_i\}$. However, note that on input u_i, both $\mathcal{G}'_{\mathsf{enc}}$ and $\mathcal{G}''_{\mathsf{enc}}$ output $\perp$. Then, since the punctured PRF preserves functionality under puncturing, we have that $\mathcal{G}'_{\mathsf{enc}}$ and $\mathcal{G}''_{\mathsf{enc}}$ are functionally equivalent. The indistinguishability of $H_{i,2}$ and $H_{i,3}$ follows from the security of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Indistinguishability of $H_{i,3}$ and $H_{i,4}$. From the security of the punctured PRF, it follows immediately that $H_{i,3}$ and $H_{i,4}$ are computationally indistinguishable.

Indistinguishability of $H_{i,4}$ and $H_{i,5}$. From the security of PRG, it follows immediately that $H_{i,4}$ and $H_{i,5}$ are computationally indistinguishable.

Indistinguishability of $H_{i,5}$ and $H_{i,6}$. Let $\mathcal{G}'_{\mathsf{dec}}$ (resp., $\mathcal{G}''_{\mathsf{dec}}$) denote the unobfuscated algorithms underlying the decapsulation key program DK' in experiments $H_{i,5}$ and $H_{i,6}$. We will argue that with all but negligible probability, $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ are functionally equivalent. The indistinguishability of $H_{i,5}$ and $H_{i,6}$ then follows from the security of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Let c_i denote the *unique* ciphertext corresponding to the message u_i. We note that with overwhelming probability, the random string z (hardwired in both $H_{i,5}$ and $H_{i,6}$) is not in the image of the PRG. Thus, except with negligible probability, there does not exist an α_i such that $PRG(\alpha_i) = z$. This implies that except with negligible probability, $\mathcal{G}'_{\mathsf{dec}}(c_i) = \perp$. Since $\mathcal{G}''_{\mathsf{dec}}$ also outputs $\perp$ on input c_i and $\mathcal{G}'_{\mathsf{dec}}, \mathcal{G}''_{\mathsf{dec}}$ behave identically on all other input ciphertexts, we have that $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ are functionally equivalent.

Indistinguishability of $H_{i,6}$ and $H_{i,7}$. This follows in the same manner as the indistinguishability of experiments $H_{i,2}$ and $H_{i,3}$. We omit the details.

Completing the proof. Note that throughout the hybrids, we use the security of three cryptographic primitives: injective PRG, (injective) puncturable PRFs, and indistinguishability obfuscation. In total, (ignoring constant multiplicative factors) we have $|S_1 \setminus S_0|$ hybrids. Thus, overall, we get that no adversary can distinguish between $EK\{U\}, DK\{S_0\}$ and $EK\{U\}, DK\{S_1\}$ with advantage more than $\epsilon_{S_0,S_1} = |S_1 \setminus S_0| \cdot (\mathsf{adv}_{PRG}(\lambda) + \mathsf{adv}_{PRF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$. Replacing $\mathsf{adv}_{PRG}(\lambda) + \mathsf{adv}_{PRF}(\lambda)$ with adv_{OWF}, where OWF is the one-way function used to construct the PRG ad puncturable PRF, we get $\epsilon_{S_0,S_1} = |S_1 \setminus S_0| \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$ as required. □

Selective ℓ-Indistinguishability of Ciphertexts. We now prove that $\mathcal{ACE}$ satisfies indistinguishability of ciphertexts.

Lemma 2. *The proposed scheme $\mathcal{ACE}$ satisfies ℓ-indistinguishability of ciphertexts.*

Proof. The proof of the lemma proceeds in a sequence of hybrid experiments where we make indistinguishable changes to $EK\{U\}$, $DK\{S\}$, and the ciphertexts (c_i^b, c_i^{1-b}).

Hybrid H_0. This is the real world experiment. For completeness (and to ease the presentation of the subsequent hybrid experiments), we describe the actions of the challenger here. Let $S, U \subset M$ be the sets chosen by the adversary and C_S, C_U be their corresponding circuit descriptions. Further, let $(m_1^0, m_1^1), \ldots, (m_\ell^0, m_\ell^1)$ be the ℓ challenge message pairs chosen by the adversary. Then the challenger performs the following steps:

1. Sample PRF keys $K_1 \leftarrow \mathsf{Gen}_{\mathsf{PRF}}(1^\lambda)$, $K_2 \leftarrow \mathsf{Gen}_{\mathsf{PRF}}(1^\lambda)$.
2. For every $i \in [\ell]$, $t \in \{0,1\}$, compute $\alpha_i^t \leftarrow F_1(m_i^t)$, $\gamma_i^t \leftarrow F_2(\alpha_i^t)$ and $\beta_i^t = \gamma_i^t \oplus m_i^t$. Let $c_i^t = (\alpha_i^t, \beta_i^t)$.
3. Compute $EK\{U\} \leftarrow \mathsf{i}\mathcal{O}(\mathcal{G}'_{\mathsf{enc}})$ where $\mathcal{G}'_{\mathsf{enc}}$ is described in Algorithm 4.

4. Compute $DK\{S\} \leftarrow \mathsf{i}\mathcal{O}(\mathcal{G}'_{\mathsf{dec}})$ where $\mathcal{G}'_{\mathsf{dec}}$ is described in Algorithm 5.

5. Choose a random bit b and send the following tuple to the adversary:

$$\left(EK\{S\}, DK\{U\}, c_1^b, \ldots, c_\ell^b\right).$$

Algorithm 4: $\mathcal{G}'_{\mathsf{enc}}$ in Hybrid H_0

Data: $K1$, $K2$, circuit C_U
Input: message m
if $C_U(m)$ **then return** $\perp$;
else
  $\alpha \leftarrow F_1(m)$;
  $\beta \leftarrow F_2(\alpha) \oplus m$;
  return $\alpha \| \beta$
end

Algorithm 5: $\mathcal{G}'_{\mathsf{dec}}$ in Hybrid H_0

Data: $K1$, $K2$, circuit C_S
Input: ciphertext c
parse c as $\alpha \| \beta$;
$m \leftarrow F_2(\alpha) \oplus \beta$;
if $C_S(m)$ **then return** $\perp$;
else if $\alpha \neq F_1(m)$ **then return** $\perp$;
else return m;

Hybrid H_1. Modify $\mathcal{G}'_{\mathsf{enc}}$: the hardwired PRF key K_1 is replaced with a punctured key $K_1\{\Sigma_1\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_1, \Sigma_1)$, where Σ_1 is the list of messages $(m_1^0, m_1^1), \ldots, (m_\ell^0, m_\ell^1)$ sorted lexicographically.

Hybrid H_2. Modify $\mathcal{G}'_{\mathsf{dec}}$: the hardwired PRF key K_1 is replaced with a punctured key $K_1\{\Sigma_1\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_1, \Sigma_1)$, where Σ_1 is the list of messages $(m_1^0, m_1^1), \ldots, (m_\ell^0, m_\ell^1)$ sorted lexicographically.

Hybrid H_3. Modify $\mathcal{G}'_{\mathsf{dec}}$: Perform the following check in the beginning. If input ciphertext $c = c_i^t$ for any $i \in [\ell], t \in \{0,1\}$ then output $\perp$. The modified $\mathcal{G}'_{\mathsf{dec}}$ is described in Algorithm 6

Algorithm 6: $\mathcal{G}'_{\mathsf{dec}}$ in Hybrid 3

Data: $K1$, $K2$, circuit C_S, $\{(c_1^0, c_1^1), \ldots, (c_\ell^0, c_\ell^1)\}$
Input: ciphertext c
if $c = c_i^t$ *for any* $i \in [\ell], t \in \{0,1\}$ **then return** $\perp$;
parse c as $\alpha \| \beta$;
$m \leftarrow F_2(\alpha) \oplus \beta$;
if $C_S(m)$ **then return** $\perp$;
else if $\alpha \neq F_1(m)$ **then return** $\perp$;
else return m;

Hybrid H_4. Modify challenge ciphertexts $c_i^t = (\alpha_i^t, \beta_i^t)$: For every $i \in [\ell]$, $t \in \{0,1\}$, replace α_i^t with a truly random string $\tilde{\alpha}_i^t$.

Hybrid H_5. Modify $\mathcal{G}'_{\mathsf{enc}}$: the hardwired PRF key K_2 is replaced with a punctured key $K_2\{\Sigma_2\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_2, \Sigma_2)$, where Σ_2 is the list of strings $(\tilde{\alpha}_1^0, \tilde{\alpha}_1^1), \ldots, (\tilde{\alpha}_\ell^0, \tilde{\alpha}_\ell^1)$ sorted lexicographically.

Hybrid H_6. Modify $\mathcal{G}'_{\mathsf{dec}}$: we change the check performed in line 1 of Algorithm 6. For any input ciphertext $c = (\alpha, \beta)$, if $\alpha = \alpha_i^t$ for any $i \in [\ell], t \in \{0,1\}$ (where $c_i^t = (\alpha_i^t, \beta_i^t)$), then output $\perp$. Note that $\mathcal{G}'_{\mathsf{dec}}$ only has $\tilde{\alpha}_i^t$ hardwired as opposed to c_i^t. The modified $\mathcal{G}'_{\mathsf{dec}}$ is described in Algorithm 7.

Algorithm 7: $\mathcal{G}'_{\mathsf{dec}}$ in Hybrid 6

Data: $K1$, $K2$, circuit C_S, $\{(\tilde{\alpha}_1^0, \tilde{\alpha}_1^1), \ldots, (\tilde{\alpha}_\ell^0, \tilde{\alpha}_\ell^1)\}$
Input: ciphertext c
1 parse c as $\alpha \| \beta$;
2 **if** $\alpha = \tilde{\alpha}_i^t$ *for any* $i \in [\ell], t \in \{0,1\}$ **then return** $\perp$;
3 $m \leftarrow F_2(\alpha) \oplus \beta$;
4 **if** $C_S(m)$ **then return** $\perp$;
5 **else if** $\alpha \neq F_1(m)$ **then return** $\perp$;
6 **else return** m;

Hybrid H_7. Modify $\mathcal{G}'_{\mathsf{dec}}$: the hardwired PRF key K_2 is replaced with a punctured key $K_2\{\Sigma_2\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_2, \Sigma_2)$, where Σ_2 is the list of strings $(\tilde{\alpha}_1^0, \tilde{\alpha}_1^1), \ldots, (\tilde{\alpha}_\ell^0, \tilde{\alpha}_\ell^1)$ sorted lexicographically.

Hybrid H_8. Modify challenge ciphertexts $c_i^t = (\tilde{\alpha}_i^t, \beta_i^t)$: For every $i \in [\ell]$, $t \in \{0,1\}$, set $\beta_i^t = \tilde{\gamma}_i^t \oplus m_i^t$, where $\tilde{\gamma}_i^t$ is a random string.

This completes the description of the hybrid experiments. We will now first prove indistinguishability of experiments H_i and H_{i+1} for every i. We will then prove that the adversary can guess bit b in the final hybrid H_8 with probability at most $\frac{1}{2}$. This suffices to prove the claim.

Indistinguishability of H_0 and H_1. Let $\mathcal{G}'_{\mathsf{enc}}$ and $\mathcal{G}''_{\mathsf{enc}}$ denote the algorithms underlying the encapsulation key program $EK\{U\}$ in H_0 and H_1 respectively. Note that due to the check performed in line 1, both $\mathcal{G}'_{\mathsf{enc}}$ and $\mathcal{G}''_{\mathsf{enc}}$ output $\perp$ on each challenge message m_i^t. In particular, line 4 is *not* executed in both $\mathcal{G}'_{\mathsf{enc}}$ and $\mathcal{G}''_{\mathsf{enc}}$ for every input message m_i^t. (In fact, line 4 is only executed when the input message $m \notin S$.) Then, since the punctured PRF scheme preserves functionality under puncturing, we have that $\mathcal{G}'_{\mathsf{enc}}$ (using K_1) and $\mathcal{G}''_{\mathsf{enc}}$ (using $K_1\{\Sigma_1\}$) are functionally equivalent. The indistinguishability of H_0 and H_1 follows from the indistinguishability of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Indistinguishability of H_1 and H_2. Let $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ denote the algorithms underlying the decapsulation key program $DK\{S\}$ in H_1 and H_2 respectively. Note that due to the check performed in line 3, both $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ output $\perp$ on every challenge ciphertext c_i^t. In particular, line 4 is is *not* executed in both $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ for every m_i^t. Then, since the punctured PRF scheme preserves functionality under puncturing, we have that $\mathcal{G}'_{\mathsf{dec}}$ (using K_1) and $\mathcal{G}''_{\mathsf{dec}}$ (using $K_1\{\Sigma_1\}$) are functionally equivalent. As a consequence, the indistinguishability of H_0 and H_1 follows from the indistinguishability of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Indistinguishability of H_2 and H_3. Let $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ denote the algorithms underlying the decapsulation key program $DK\{S\}$ in H_2 and H_3 respectively. Note that the only difference between $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ is that $\mathcal{G}''_{\mathsf{dec}}$ performs an additional check whether the input ciphertext c is equal to any challenge ciphertext c_i^t. However, note that due to line 3, $\mathcal{G}'_{\mathsf{dec}}$ also outputs $\perp$ on such input ciphertexts. Thus, $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ are functionally equivalent and the indistinguishability of H_2 and H_3 follows from the indistinguishability of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Indistinguishability of H_3 and H_4. This follows immediately from the security of the punctured PRF family F_1.

Indistinguishability of H_4 and H_5. Note that with overwhelming probability, each of the random string $\tilde{\alpha}_i^t$ is not in the range of the F_1. Therefore, except with negligible probability, there does not exist a message

m such that $F_1(m) = \alpha_i^t$, for any $i \in [\ell], t \in \{0,1\}$. Since the punctured PRF scheme preserves functionality under puncturing, $\mathcal{G}'_{\mathsf{enc}}$ (using K_2) and $\mathcal{G}''_{\mathsf{enc}}$ (using $K_2\{\Sigma_2\}$) behave identically on all input messages, except with negligible probability. The indistinguishability of H_2 and H_3 follows from the indistinguishability of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Indistinguishability of H_5 and H_6. Let $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ denote the algorithms underlying the decapsulation key program $DK\{S\}$ in H_5 and H_6 respectively. Note that the only difference between $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ is their description in line 1: $\mathcal{G}''_{\mathsf{dec}}$ outputs $\perp$ on every ciphertext $c = (\tilde{\alpha}_i^t, \beta)$ (for arbitrary β) while $\mathcal{G}'_{\mathsf{dec}}$ only outputs $\mathcal{G}'_{\mathsf{dec}}$ only outputs $\perp$ on every $c_i^t = (\tilde{\alpha}_i^t, \beta_i^t)$. In particular, the execution of $\mathcal{G}'_{\mathsf{dec}}$ continues onward from line 2 for every $c = (\tilde{\alpha}_i^t, \beta)$ such that $\beta \neq \beta_i^t$ for every $i \in [\ell], t \in \{0,1\}$. However, note that with overwhelming probability, each of the random string $\tilde{\alpha}_i^t$ is not in the range of the F_1. Thus in line 5, $\mathcal{G}'_{\mathsf{dec}}$ will also output $\perp$ on every $c = (\tilde{\alpha}_i^t, \beta)$, except with negligible probability. As a consequence, we have that except with negligible probability, $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ have identical input/output behavior, and therefore, the indistinguishability of H_5 and H_6 follows from the indistinguishability of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Indistinguishability of H_6 and H_7. Let $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ denote the algorithms underlying the decapsulation key program $DK\{S\}$ in H_6 and H_7 respectively. Note that due to the check performed in line 2 (see Algorithm 7), line 3 is not executed in both $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ whenever the input ciphertext c is of the form (α_i^t, β). Then, since the the punctured PRF scheme preserves functionality under puncturing, $\mathcal{G}'_{\mathsf{dec}}$ (using K_2) and $\mathcal{G}''_{\mathsf{dec}}$ (using $K_2\{\Sigma_2\}$) are functionally equivalent and the indistinguishability of H_2 and H_3 follows from the indistinguishability of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Indistinguishability of H_7 and H_8. This follows immediately from the security of the punctured PRF family F_1.

Finishing the proof. Observe that in experiment H_8, every challenge ciphertext c_i^t consists of random strings $(\tilde{\alpha}_i^t, \tilde{\beta}_i^t)$ that information theoretically hide the bit b. Further, $EK\{U\}$ and $DK\{S\}$ are also independent of bit b. Therefore, the adversary cannot guess bit b with probability better than $\frac{1}{2}$.

□

4 Indistinguishability Obfuscation of Iterated Circuits

In this section we define and construct an indistinguishability obfuscator for iterated circuits. Note that, while this construction serves as a good "warmup" for the construction of garbled RAMs in later sections, it is not formally used there. Still we believe it is of independent interest.

Consider two *iterated* circuits of the form C_0^T, C_1^T for some circuits C_0, C_1 and polynomial T such that C_0^T and C_1^T are functionally equivalent[1] Roughly speaking an indistinguishability obfuscator $\mathsf{i}\mathcal{O}_{\mathsf{IC}}$ for iterated functions guarantees that for any such pair of iterated circuits (C_0^T, C_1^T), $\mathsf{i}\mathcal{O}_{\mathsf{IC}}(C_0, T)$ is computationally indistinguishable from $\mathsf{i}\mathcal{O}_{\mathsf{IC}}(C_1, T)$.

Note that one could achieve this effect by simply using a standard indistinguishabilty obfuscator for circuits to obfuscate C_0^T and C_1^T. However, this may incur a blowup of size $\mathsf{poly}(T \cdot |C_0|)$ (resp., $\mathsf{poly}(T \cdot |C_1|)$). The crucial requirement from an indistinguishability obfuscator for iterated circuits is that it that the size of the obfuscated circuit should be only $\mathsf{poly}(|C|, \log(T))$ for iterated circuit C^T.

Syntax. An indistinguishability obfuscator for iterated circuits consists of a pair of algorithms $(\mathsf{i}\mathcal{O}_{\mathsf{IC}}, \mathsf{Eval}_{\mathsf{IC}})$. The obfuscation algorithm $\mathsf{i}\mathcal{O}_{\mathsf{IC}}$ takes as input the security parameter λ, a circuit C and an iteration bound T and outputs another circuit $\tilde{C}$. The evaluation algorithm $\mathsf{Eval}_{\mathsf{IC}}$ is a stateful algorithm that takes as input the obfuscated program $\tilde{C}$, the iteration bound T and an input x and returns an output y.

Definition 4. *A pair of algorithms $(\mathsf{i}\mathcal{O}_{\mathsf{IC}}, \mathsf{Eval}_{\mathsf{IC}})$ is called an indistinguishability obfuscator for iterated circuits for a circuit class $\{\mathcal{C}_\lambda\}$ if the following holds:*

[1] Note that C_0^T and C_1^T could be functionally equivalent even if C_0^i and C_1^i are not, for some $i < T$.

- **Correctness:** *For every $\lambda \in \mathbb{N}$, every $C \in \mathcal{C}_\lambda$, every $T \in \mathsf{poly}(\lambda)$, and every input x, we have that:*

$$\Pr[\mathsf{Eval}_{\mathsf{IC}}(\tilde{C}, T, x) = C^T(x) : \tilde{C} \leftarrow \mathsf{i}\mathcal{O}_{\mathsf{IC}}(1^\lambda, C, T)] = 0.$$

- **Efficiency:** *For every $\lambda \in \mathbb{N}$, every $C \in \mathcal{C}_\lambda$, every $T \in \mathsf{poly}(\lambda)$, and every input x, $\mathsf{Eval}_{\mathsf{IC}}(\tilde{C}, T, x)$ runs in time $O(|\tilde{C}| \cdot T)$, where $\tilde{C} \leftarrow \mathsf{i}\mathcal{O}_{\mathsf{IC}}(1^\lambda, C, T)$.*

- **Succinctness:** *For every $\lambda \in \mathbb{N}$, every $C \in \mathcal{C}_\lambda$, every $T \in \mathsf{poly}(\lambda)$, the size of obfuscated program $|\mathsf{i}\mathcal{O}_{\mathsf{IC}}(1^\lambda, C, T)| = \mathsf{poly}(\lambda, |C|, \log(T))$.*

- **ϵ-Indistinguishability:** *For every $\lambda \in \mathbb{N}$, every $C_0, C_1 \in \mathcal{C}_\lambda$, every $T \in \mathsf{poly}(\lambda)$, if $C_0^T(x) = C_1^T(x)$ for all inputs x and $|C_0| = |C_1|$, then for all PPT adversaries $\mathcal{A}$, we have:*

$$\left|\Pr[\mathcal{A}(\mathsf{i}\mathcal{O}_{\mathsf{IC}}(1^\lambda, C_0, T) = 1] - \Pr[\mathcal{A}(\mathsf{i}\mathcal{O}_{\mathsf{IC}}(1^\lambda, C_1, T) = 1]\right| \leq \epsilon(\lambda)$$

Remark 4. *Similar to the definition of security of constrained decapsulation (see Section 3), here we do not necessarily require ϵ to be negligible. Looking ahead, our construction in Section 4.1 will achieve ϵ-indistinguishability for $\epsilon = |\mathcal{I}|^2 \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$, where $\mathcal{I}$ is the input and output space of the circuit being obfuscated. When $|\mathcal{I}|$ is super-polynomial, this is negligible assuming sub exponential hardness of one-way functions as well as $\mathsf{i}\mathcal{O}$.*

The rest of this section is organized as follows. We present a construction of iterated function obfuscation using ACE in Section 4.1. In Section 4.2, we prove the security of our construction.

4.1 Our Construction

Here we provide a construction of an indistinguishability obfuscator for iterated circuits. Our scheme is based on standard indistinguishability obfuscation for circuits and asymmetrically constrained encapsulation.

Notation. Let $\{\mathcal{C}_\lambda\}$ be a family of polynomial-sized circuits. Let $\mathcal{I} \in \{0,1\}^n$ denote the input and output space of $C \in \mathcal{C}_\lambda$ for some $n = \mathsf{poly}(\lambda)$. Let T denote the iteration bound. Let $\mathsf{i}\mathcal{O}$ be a standard indistinguishability obfuscator for $\{\mathcal{C}_\lambda\}$. Finally, let $\mathcal{ACE} = (\mathsf{Setup}, \mathsf{GenEK}, \mathsf{GenDK}, \mathsf{Enc}, \mathsf{Dec})$ be an asymmetrically constrainable encapsulation scheme for message space M consisting of messages of the form $m = (m_1, m_2, m_3)$ where m_1 and m_3 are n-bit strings, and m_2 is a non-negative integer which is at most T.

Construction. We now proceed to describe our iterated circuits indistinguishability obfuscator $\mathcal{O} = (\mathsf{i}\mathcal{O}_{\mathsf{IC}}, \mathsf{Eval}_{\mathsf{IC}})$.

$\mathsf{i}\mathcal{O}_{\mathsf{IC}}(1^\lambda, C, T)$**:** Sample $SK \leftarrow \mathsf{Setup}(1^\lambda)$. Let $Z = \{\star, 0, \star\}$ denote the set of all messages $m = (m_1, m_2, m_3)$ in M where $m_2 = 0$, and let C_Z be the circuit description of Z. Compute $EK\{Z\} \leftarrow \mathsf{GenEK}(SK, C_Z)$ and $DK\{Z\} \leftarrow \mathsf{GenDK}(SK, C_Z)$. Compute $\tilde{C} \leftarrow \mathsf{i}\mathcal{O}(\mathcal{G}_{\mathsf{IC}})$ where $\mathcal{G}_{\mathsf{IC}}$ is described in Algorithm 8. Here, $\mathcal{G}_{\mathsf{IC}}$ is padded with sufficient zeros so that $|\mathcal{G}_{\mathsf{IC}}| = |\mathcal{G}_{\mathsf{IC}}^{i,j}|$, where $\mathcal{G}_{\mathsf{IC}}^{i,j}$ is described in Algorithm 10 in the security proof.

The output is $\tilde{C}$.

Algorithm 8: Algorithm $\mathcal{G}_{\mathsf{IC}}$

Input: ciphertext ct OR plain input x
Data: $EK\{Z\}$, $DK\{Z\}$, T, C
if *given a plain input* x **then**
  $t \leftarrow 0$;
  $q \leftarrow x$;
else
  $m \leftarrow \mathsf{Dec}(DK, \mathsf{ct})$;
  if $m = \perp$ **then return** $\perp$;
  else parse m as (x, t, q) ;
if $t > T$ **then return** $\perp$;
if $t = T$ **then return** q;
return $\mathsf{Enc}(EK, (x, t+1, C(q)))$

$\mathsf{Eval}_{\mathsf{IC}}(\tilde{C}, T, x)$*:* Let $y_0 = x$. For every $i \in [T]$, compute $y_i \leftarrow \tilde{C}(y_{i-1})$. The output is y_T.

This completes the description of $\mathcal{O} = (\mathsf{i}\mathcal{O}_{\mathsf{IC}}, \mathsf{Eval}_{\mathsf{IC}})$. Correctness and efficiency of $\mathcal{O}$ follows from the description of the algorithms $\mathsf{i}\mathcal{O}_{\mathsf{IC}}$ and $\mathsf{Eval}_{\mathsf{IC}}$ together with the correctness of $\mathsf{i}\mathcal{O}$. We will argue the succinctness property in Section 4.3. In the next subsection, we argue ϵ-indistinguishability.

4.2 Proof of Security

Let $C_0, C_1 \in \mathcal{C}_\lambda$ and $T \in \mathsf{poly}(\lambda)$ be such that:(a) C_0^T and C_1^T are functionally equivalent, and (b) $|C_0| = |C_1|$. We will argue that $\mathsf{i}\mathcal{O}_{\mathsf{IC}}(C_0)$ and $\mathsf{i}\mathcal{O}_{\mathsf{IC}}(C_1)$ are $\epsilon = |\mathcal{I}|^2 \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$-indistingushable through a sequence of $|\mathcal{I}| \cdot O(T)$ experiments $H_{i,j}$, where $i \in \mathcal{I}$ and $j \in \{0, \ldots, 2T\}$.

Below, we first describe the experiments $H_{i,0}$, where $H_{0,0}$ will correspond to $\mathsf{i}\mathcal{O}_{\mathsf{IC}}(C_0)$ and $H_{\mathcal{I},0}$ will correspond to $\mathsf{i}\mathcal{O}_{\mathsf{IC}}(C_1)$. Next, to transition from $H_{i,0}$ to $H_{i+1,0}$, we describe the experiments $H_{i,j}$ such that $H_{i,2T}$ is the same as $H_{i+1,0}$. For convenience, when presenting a hybrid experiment, we underline the difference between it and the previous hybrid.

Hybrid $H_{i,0}$*:* In this experiment, the adversary is given $\mathsf{i}\mathcal{O}_{\mathsf{IC}}(\mathcal{G}_{\mathsf{IC}}^{i,0})$, where $\mathcal{G}_{\mathsf{IC}}^{i,0}$ is described in Algorithm 9.

Algorithm 9: $\mathcal{G}_{\mathsf{IC}}^{i,0}$

Input: ciphertext ct OR plain input x
Data: $EK\{Z\}$, $DK\{Z\}$, T, C_0, C_1
if *given a plain input* x **then**
  $t \leftarrow 0$;
  $q \leftarrow x$;
else
  $m \leftarrow \mathsf{Dec}(DK\{Z\}, \mathsf{ct})$;
  if $m = \perp$ **then return** $\perp$;
  else parse m as (x, t, q) ;
if $t > T$ **then return** $\perp$;
if $t = T$ **then return** q;
if $x < i$ **then return** $\mathsf{Enc}(EK\{Z\}, (x, t+1, C_1(q)))$;;
else return $\mathsf{Enc}(EK\{Z\}, (x, t+1, C_0(q)))$;

Hybrid $H_{i,j}$: In this experiment, the adversary is given $\mathsf{i}\mathcal{O}_{\mathsf{IC}}(\mathcal{G}_{\mathsf{IC}}^{i,j})$, where $\mathcal{G}_{\mathsf{IC}}^{i,j}$ is described in Algorithm 10 .

Algorithm 10: $\mathcal{G}_{\mathsf{IC}}^{i,j}$

Input: ciphertext ct OR plain input x
Data: ct_j^*, $EK\{S_j\}$, $DK\{S_j\}$, T, C_0, C_1, where $S_j = Z \cup \{(x,t,q) \mid x = i \text{ and } t < j \text{ and } q \neq \bot\}$ and $\mathsf{ct}_j^* = \mathsf{Enc}(EK\{S_j\}, (i, j, C_0^j(i)))$

if *given a plain input* x **then**
  $t \leftarrow 0$;
  $q \leftarrow x$;
else
  $m \leftarrow \mathsf{Dec}(DK\{S_j\}, \mathsf{ct})$;
  if $m = \bot$ **then return** $\bot$;
  else parse m as (x, t, q) ;
if $t > T$ **then return** $\bot$;
if $t = T$ **then return** q ;
if $x < i$ **then return** $\mathsf{Enc}(EK\{S_j\}, (x, t+1, C_1(q)))$;
if $x = i$ **then**
  if $t < j-1$ **then return** $\mathsf{Enc}(EK\{S_j\}, (x, t+1, \bot))$;
  else if $t = j-1$ **then return** ct_j^* ;
  else return $\mathsf{Enc}(EK\{S_j\}, (x, t+1, C_0(q)))$;
return $\mathsf{Enc}(EK\{S_j\}, (x, t+1, C_0(q)))$

Lemma 3. *Experiments $H_{i,j}$ and $H_{i,j+1}$ are $|\mathcal{I}| \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$-distinguishable.*

Proof. This is the core of our security proof. In order to prove Lemma 3, we will describe a series of 9 intermediate hybrids $H_{i,j:0}, \ldots, H_{i,j:8}$, where $H_{i,j:0}$ is the same as $H_{i,j}$ and $H_{i,j:8}$ is the same as $H_{i,j+1}$. For clarity, in each hybrid $H_{i,j:k}$, the (unobfuscated) function $\mathcal{G}_{\mathsf{IC}}$ is denoted by $\mathcal{G}_{\mathsf{IC}}^{i,j:k}$.

Hybrid $H_{i,j:0}$: Same as $H_{i,j}$.

Hybrid $H_{i,j:1}$: Same as $H_{i,j:0}$ except that we modify $\mathcal{G}_{\mathsf{IC}}^{i,j}$ in the following manner: let S denote the subset of all messages in M of the form $(i, j, \star)$. The encapsulation key $EK\{S_j\}$ hardwired in $\mathcal{G}_{\mathsf{IC}}^{i,j}$ is replaced with $EK\{S_j \cup S\}$ that is constrained at the set $S_j \cup S$.

The modified function $\mathcal{G}_{\mathsf{IC}}^{i,j:1}$ is described in algorithm 11.

Algorithm 11: $\mathcal{G}_{\mathsf{IC}}^{i,j:1}$

Input: ciphertext ct OR plain input x

Data: ct_j^*, $EK\{S_j \cup S\}$, $DK\{S_j\}$, T, C_0, C_1, where $S_j = Z \cup \{(x,t,q) \mid x = i \text{ and } t < j \text{ and } q \neq \bot\}$, $S = \{i, j, \star\}$ and $\mathsf{ct}_j^* = \mathsf{Enc}(EK\{S_j\}, (i, j, C_0^j(i)))$

if *given a plain input* x **then**
  $t \leftarrow 0$;
  $q \leftarrow x$;
else
  $m \leftarrow \mathsf{Dec}(DK\{S_j\}, \mathsf{ct})$;
  if $m = \bot$ **then return** $\bot$;
  else parse m as (x, t, q) ;
if $t > T$ **then return** $\bot$;
if $t = T$ **then return** q ;
if $x < i$ **then return** $\mathsf{Enc}(EK\{S_j \cup S\}, (x, t+1, C_1(q)))$;
if $x = i$ **then**
  if $t < j - 1$ **then return** $\mathsf{Enc}(EK\{S_j \cup S\}, (x, t+1, \bot)$;
  else if $t = j - 1$ **then return** ct_j^* ;
  else return $\mathsf{Enc}(EK\{S_j\}, (x, t+1, C_0(q)))$;
return $\mathsf{Enc}(EK\{S_j \cup S\}, (x, t+1, C_0(q)))$

Hybrid $H_{i,j:2}$*:* Same as $H_{i,j:1}$ except that we modify $\mathcal{G}_{\mathsf{IC}}^{i,j:1}$ as follows:

- Let $\mathsf{ct}_{j+1}^* = \mathsf{Enc}(EK\{S_j\}, (i, j+1, C_0^{j+1}(i)))$. Hardwire ct_{j+1}^* in $\mathcal{G}_{\mathsf{IC}}^{i,j:1}$ and whenever the input is ct_j^*, then directly output ct_{j+1}^*.
- Constrain the decapsulation key further to include the point $(i, j, C_0^j(i))$ as well. That is, replace $DK\{S_j\}$ with the constrained key $DK\{S_j \cup (i, j, C_0^j(i))\}$.

The modified function $\mathcal{G}_{\mathsf{IC}}^{i,j:2}$ is described in Algorithm 12.

Algorithm 12: $\mathcal{G}_{\mathsf{IC}}^{i,j:2}$

Input: ciphertext ct OR plain input x

Data: ct_j^*, $EK\{S_j \cup S\}$, $DK\{S_j \cup (i,j,C_0^j(i))\}$, T, C_0, C_1, where $S_j = Z \cup \{(x,t,q) \mid x = i \text{ and } t < j \text{ and } q \neq \bot\}$, $S = \{i,j,\star\}$, $\mathsf{ct}_j^* = \mathsf{Enc}(EK\{S_j\},(i,j,C_0^j(i)))$, and $\mathsf{ct}_{j+1}^* = \mathsf{Enc}(EK\{S_j\},(i,j+1,C_0^{j+1}(i)))$

```
if given a plain input x then
    t ← 0;
    q ← x;
if ct = ct*_j then return ct*_{j+1};
else
    m ← Dec(DK{S_j ∪ (i,j,C_0^j(i))}, ct);
    if m = ⊥ then return ⊥;
    else parse m as (x,t,q) ;
if t > T then return ⊥ ;
if t = T then return q ;
if x < i then return Enc(EK{S_j ∪ S}, (x,t+1,C_1(q))) ;
if x = i then
    if t < j − 1 then return Enc(EK{S_j ∪ S}, (x,t+1,⊥)) ;
    else if t = j − 1 then return ct*_j ;
    else return Enc(EK{S_j ∪ S}, (x,t+1,C_0(q)));
return Enc(EK{S_j ∪ S}, (x,t+1,C_0(q)))
```

Hybrid $H_{i,j:3}$: Same as $H_{i,j:2}$ except that we constrain the decapsulation key further to include all the messages $S = \{i,j,\star\}$. That is, replace $DK\{S_j \cup (i,j,C_0^j(i))\}$ with $DK\{S_j \cup S\}$.

Hybrid $H_{i,j:4}$: Same as $H_{i,j:3}$ except that the hardwired ciphertext ct_j^* is now set to the value $\mathsf{Enc}(EK\{S_j\},(i,j,\bot))$.

Hybrid $H_{i,j:5}$: Same as $H_{i,j:4}$ except that the decapsulation key is now only constrained at the set S_{j+1}. That is, replace $DK\{S_j \cup S\}$ with $DK\{S_{j+1}\}$.

Hybrid $H_{i,j:6}$: Same as $H_{i,j:5}$ except that we modify $\mathcal{G}_{\mathsf{IC}}^{i,j:5}$ as follows: instead of directly returning ct_{j+1}^* when the input is ct_j^*, we now first decapsulate the ciphertext (as every other input) and if $x = i$, $t = j$, then output ct_{j+1}^*.

The modified function $\mathcal{G}_{\mathsf{IC}}^{i,j:6}$ is described in Algorithm 13.

Algorithm 13: $\mathcal{G}_{\mathsf{IC}}^{i,j:6}$

Input: ciphertext ct OR plain input x

Data: ct_j^*, $EK\{S_j \cup S\}$, $DK\{S_{j+1}\}$, T, C_0, C_1, where $S_j = Z \cup \{(x,t,q) \mid x = i \text{ and } t < j \text{ and } q \neq \bot\}$, $S = \{i, j, \star\}$, $\mathsf{ct}_j^* = \mathsf{Enc}(EK\{S_j\}, (i,j,\bot))$, and $\mathsf{ct}_{j+1}^* = \mathsf{Enc}(EK\{S_j\}, (i, j+1, C_0^{j+1}(i)))$

1 **if** *given a plain input* x **then**
2 $\quad t \leftarrow 0$;
3 $\quad q \leftarrow x$;
4 **else**
5 $\quad m \leftarrow \mathsf{Dec}(DK\{S_{j+1}\}, \mathsf{ct})$;
6 $\quad$ **if** $m = \bot$ **then return** $\bot$;
7 $\quad$ **else** parse m as (x,t,q) ;
8 **if** $t > T$ **then return** $\bot$;
9 **if** $t = T$ **then return** q ;
10 **if** $x < i$ **then return** $\mathsf{Enc}(EK\{S_j \cup S\}, (x, t+1, C_1(q)))$;
11 **if** $x = i$ **then**
12 $\quad$ **if** $t < j-1$ **then return** $\mathsf{Enc}(EK\{S_j \cup S\}, (x, t+1, \bot))$;
13 $\quad$ **else if** $t = j-1$ **then return** ct_j^* ;
14 $\quad$ **else if** $t = j$ **then return** ct_{j+1}^*;
15 $\quad$ **else return** $\mathsf{Enc}(EK\{S_j \cup S\}, (x, t+1, C_0(q)))$;
16 **return** $\mathsf{Enc}(EK\{S_j \cup S\}, (x, t+1, C_0(q)))$

Hybrid $H_{i,j:7}$*:* Same as $H_{i,j:6}$ except that the encapsulation key is now only constrained at the set S_{j+1}. That is, replace $EK\{S_j \cup S\}$ with $EK\{S_{j+1}\}$.

Hybrid $H_{i,j:8}$*:* Same as $H_{i,j:7}$ except that we modify $\mathcal{G}_{\mathsf{IC}}^{i,j:7}$ as follows: instead of directly outputting the hardwired value ct_j^* in line 14, we now return $\mathsf{Enc}(EK\{S_{j+1}\}, (i,j,\bot))$.

The modified function $\mathcal{G}_{\mathsf{IC}}^{i,j:8}$ is described in Algorithm 14. Note that is is the same as experiment $H_{i,j+1}$.

Algorithm 14: $\mathcal{G}_{\mathsf{IC}}^{i,j:8}$

Input: ciphertext ct OR plain input x

Data: ct_j^*, $EK\{S_{j+1}\}$, $DK\{S_{j+1}\}$, T, C_0, C_1, where $S_j = Z \cup \{(x,t,q) \mid x = i \text{ and } t < j \text{ and } q \neq \bot\}$, $\mathsf{ct}_j^* = \mathsf{Enc}(EK\{S_{j+1}\}, (i,j,\bot))$, and $\mathsf{ct}_{j+1}^* = \mathsf{Enc}(EK\{S_{j+1}\}, (i,j+1,C_0^{j+1}(i)))$

if *given a plain input* x **then**
  $t \leftarrow 0$;
  $q \leftarrow x$;
else
  $m \leftarrow \mathsf{Dec}(DK\{S_{j+1}\}, \mathsf{ct})$;
  if $m = \bot$ **then return** $\bot$;
  else parse m as (x,t,q) ;
if $t > T$ **then return** $\bot$;
if $t = T$ **then return** q ;
if $x < i$ **then return** $\mathsf{Enc}(EK\{S_{j+1}\}, (x,t+1,C_1(q)))$;
if $x = i$ **then**
  if $t < j+1$ **then return** $\mathsf{Enc}(EK\{S_{j+1}\}, (x,t+1,\bot))$;
  else if $t = j$ **then return** ct_{j+1}^*;
  else return $\mathsf{Enc}(EK\{S_{j+1}\}, (x,t+1,C_0(q)))$;
return $\mathsf{Enc}(EK\{S_{j+1}\}, (x,t+1,C_0(q)))$

This completes the description of the hybrid experiments.

Indistinguishability of $H_{i,j:0}$ and $H_{i,j:1}$: We note that in both $\mathcal{G}_{\mathsf{IC}}^{i,j:0}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:1}$, the encapsulation key is never evaluated on messages of the form $(i,j,\star)$. In particular, on any input ciphertext ct that decapsulates to m of the form $(i,j-1,\star)$, both return the same hardwired value ct_j^*. Thus, replacing $EK\{S_j\}$ with $EK\{S_j \cup S\}$ does not change the functionality of $\mathcal{G}_{\mathsf{IC}}^{i,j:0}$ and we have that $\mathcal{G}_{\mathsf{IC}}^{i,j:0}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:1}$ are functionally equivalent. The indistinguishability of $H_{i,j:0}$ and $H_{i,j:1}$ follows from the security of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Indistinguishability of $H_{i,j:1}$ and $H_{i,j:2}$: We consider the input/output behavior of $\mathcal{G}_{\mathsf{IC}}^{i,j:1}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:2}$ in the following two cases:

- On input ct_j^*, $\mathcal{G}_{\mathsf{IC}}^{i,j:1}$ decapsulates it using $DK\{S_j\}$ and returns $\mathsf{Enc}(EK\{S_j \cup S\}, (i,j+1,C_0(q)))$ where $q = C_0^j(i)$. In contrast, $\mathcal{G}_{\mathsf{IC}}^{i,j:2}$ directly returns ct_{j+1}^* (without using the decapsulation key). However, we have that $\mathsf{ct}_{j+1}^* = \mathsf{Enc}(EK\{S_j \cup S\}, (i,j+1,C_0^{j+1}(i)))$. Thus, $\mathcal{G}_{\mathsf{IC}}^{i,j:1}(\mathsf{ct}_j^*) = \mathcal{G}_{\mathsf{IC}}^{i,j:2}(\mathsf{ct}_j^*)$.
- It is easy to see that $\mathcal{G}_{\mathsf{IC}}^{i,j:1}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:2}$ behave identically on any plain input x. Further, it follows from the correctness of constrained decapsulation property of ACE that on any input ciphertext $\mathsf{ct} \neq \mathsf{ct}_j^*$, $\mathcal{G}_{\mathsf{IC}}^{i,j:1}$ (using $DK\{S_j\}$) and $\mathcal{G}_{\mathsf{IC}}^{i,j:2}$ (using $DK\{S_j \cup (i,j,C_0^j(i))\}$) have the same functionality.

Combining the above two cases, we have that $\mathcal{G}_{\mathsf{IC}}^{i,j:1}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:2}$ are functionally equivalent. Then, the indistinguishability of $H_{i,j:1}$ and $H_{i,j:2}$ follows from the security of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

ϵ-Indistinguishability of $H_{i,j:2}$ and $H_{i,j:3}$: Let $U = S_j \cup S$, $S_0^* = S_j \cup (i,j,C_0^j(i)$ and $S_1^* = S_j \cup S$. Thus, we have that $S_0^* \subset S_1^* \subseteq U \subset M$. In $H_{i,j:2}$, the encapsulation and decapsulation key pair hardwired in $\mathcal{G}_{\mathsf{IC}}^{i,j:2}$ is $EK\{U\}, DK\{S_0^*\}$ while in $H_{i,j:3}$, the encapsulation and decapsulation key pair hardwired in $\mathcal{G}_{\mathsf{IC}}^{i,j:3}$ is $EK\{U\}, DK\{S_1^*\}$. Then, from the security of constrained decapsulation of ACE, we have that $H_{i,j:2}$ and $H_{i,j:3}$ are at most $|S_1^* \setminus S_0^*| \cdot \mathsf{negl}(\lambda)$-distinguishable. Since $S_1^* \setminus S_0^* = S$ and $|S| = |\mathcal{I}|$, we have that $H_{i,j:2}$ and $H_{i,j:3}$ are at most $|\mathcal{I}| \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$-distinguishable.

Indistinguishability of $H_{i,j:3}$ and $H_{i,j:4}$: Note that both the encapsulation and decapsulation keys hardwired in $\mathcal{G}_{\mathsf{IC}}^{i,j:3}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:4}$ are constrained at the set $S_j \cup S$ where S includes both $m_0^* = (i,j,C_0^j(i))$ and $m_1^* = (i,j,\bot)$.

The only difference between $H_{i,j:3}$ and $H_{i,j:4}$ is that in the former, $\mathcal{G}_{\mathsf{IC}}^{i,j:3}$ contains $\mathsf{ct}_j^* = \mathsf{Enc}(EK\{S_j\}, m_0)$ while in the latter, $\mathcal{G}_{\mathsf{IC}}^{i,j:4}$ contains $\mathsf{ct}_j^* = \mathsf{Enc}(EK\{S_j\}, m_1)$. Then, the indistinguishability of $H_{i,j:3}$ and $H_{i,j:4}$ follows from the selective indistinguishability of ciphertexts of ACE.

Indistinguishability of $H_{i,j:4}$ and $H_{i,j:5}$: This follows in the same manner as the ϵ-distinguishability of $H_{i,j:2}$ and $H_{i,j:3}$. Specifically, let $U = S_j \cup S$, $S_0^* = S_j \cup S$ and $S_1^* = S_{j+1}$. Thus, we have that $S_1 \subset S_0 \subseteq U \subset M$. In $H_{i,j:4}$, the encapsulation and decapsulation key pair hardwired in $\mathcal{G}_{\mathsf{IC}}^{i,j:4}$ is $EK\{U\}, DK\{S_0^*\}$ while in $H_{i,j:5}$, the encapsulation and decapsulation key pair hardwired in $\mathcal{G}_{\mathsf{IC}}^{i,j:5}$ is $EK\{U\}, DK\{S_1^*\}$. Then, from the security of constrained decapsulation of ACE, we have that $H_{i,j:4}$ and $H_{i,j:5}$ are at most $|S_0^* \setminus S_1^*| \cdot \mathsf{negl}(\lambda)$-distinguishable. Since $|S_0^* \setminus S_1^*| = 1$ (namely, the point $(i, j, \bot)$), we have that $H_{i,j:4}$ and $H_{i,j:5}$ are $(\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$-distinguishable.

Indistinguishability of $H_{i,j:5}$ and $H_{i,j:6}$: The input/output behavior of $\mathcal{G}_{\mathsf{IC}}^{i,j:5}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:6}$ is identical on every input $\mathsf{ct} \neq \mathsf{ct}_j^*$. When the input is ct_j^*, then $\mathcal{G}_{\mathsf{IC}}^{i,j:5}$ directly outputs the hardwired value ct_{j+1}^* while $\mathcal{G}_{\mathsf{IC}}^{i,j:6}$ first decapsulates ct_j^* and if the decapsulateed message m^* is of the form $(i, j, \cdot)$, then it outputs ct_{j+1}^*. From the correctness of (constrained) decapsulation property of ACE, we have that the m^* is indeed of the form $(i, j, \cdot)$, and therefore, we have that $\mathcal{G}_{\mathsf{IC}}^{i,j:5}(\mathsf{ct}_j^*) = \mathcal{G}_{\mathsf{IC}}^{i,j:6}(\mathsf{ct}_j^*)$. Summing up, we have that $\mathcal{G}_{\mathsf{IC}}^{i,j:5}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:6}$ are functionally equivalent and the indistinguishability of $H_{i,j:5}$ and $H_{i,j:6}$ follows from the security of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Indistinguishability of $H_{i,j:6}$ and $H_{i,j:7}$: We note that in both $\mathcal{G}_{\mathsf{IC}}^{i,j:6}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:7}$, the encapsulation key is never evaluated on messages of the form $(i, j+1, \star)$. In particular, on any input ciphertext ct that decapsulates to m of the form $(i, j, \star)$, both return the same hardwired value ct_{j+1}^*. Thus, replacing $EK\{S_j \cup S\}$ with $EK\{S_{j+1}\}$ does not change the functionality of $\mathcal{G}_{\mathsf{IC}}^{i,j:6}$ and we have that $\mathcal{G}_{\mathsf{IC}}^{i,j:6}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:7}$ are functionally equivalent. The indistinguishability of $H_{i,j:6}$ and $H_{i,j:7}$ follows from the security of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Indistinguishability of $H_{i,j:7}$ and $H_{i,j:8}$: The input/output behavior of $\mathcal{G}_{\mathsf{IC}}^{i,j:7}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:8}$ is identical on every input ct that decapsulates to a message m not of the form $(i, j-1, \cdot)$. Let ct' be an input ciphertext such that it decapsulates to a message of the form $(i, j-1, \cdot)$. Note that on input ct', $\mathcal{G}_{\mathsf{IC}}^{i,j:7}$ decapsulates ct' and then outputs ct_j^* as per line 13, while $\mathcal{G}_{\mathsf{IC}}^{i,j:8}$ decapsulates ct' and then outputs $\mathsf{Enc}(EK\{S_{j+1}\}, (i, j, \bot))$ as per line 12. However, $\mathsf{ct}_j^* = \mathsf{Enc}(EK\{S_{j+1}\}, (i, j, \bot))$. Thus, we have that $\mathcal{G}_{\mathsf{IC}}^{i,j:7}(\mathsf{ct}') = \mathcal{G}_{\mathsf{IC}}^{i,j:8}(\mathsf{ct}')$. Summing up, we have that $\mathcal{G}_{\mathsf{IC}}^{i,j:7}$ and $\mathcal{G}_{\mathsf{IC}}^{i,j:8}$ are functionally equivalent and the indistinguishability of $H_{i,j:7}$ and $H_{i,j:8}$ follows from the security of the indistinguishability obfuscator $\mathsf{i}\mathcal{O}$.

Completing the proof of Lemma 3. Combining the above indistinguishability claims, we have that $H_{i,j}$ and $H_{i,j+1}$ are $|\mathcal{I}| \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$-distinguishable. □

Completing the proof of ϵ-indistinguishability of $\mathcal{O}$. We now observe that when $j = T$, then on input i, C_0 is not evaluated at all. Then, since $C_0^T(i) = C_1^T(i)$, we can replay the same hybrids in reverse order to replace C_0 by C_1. Thus, effectively, we have $O(2T)$ hybrids between hybrid $H_{i,0}$ and $H_{i+1,0}$. Since $T = \mathsf{poly}(\lambda)$, from Lemma 3, we have that $H_{i,0}$ and $H_{i+1,0}$ are $|\mathcal{I}| \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$-distinguishable. Since i ranges over all the values in $\mathcal{I}$, we have that $H_{0,0}$ and $H_{\mathcal{I},0}$ are $\epsilon = |\mathcal{I}|^2 \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$-distinguishable, as required.

4.3 Succinctness of $\mathcal{O}$

We now argue that $\mathcal{O}$ satisfies the succinctness property. From the efficiency property of $\mathsf{i}\mathcal{O}$, we have that $|\mathsf{i}\mathcal{O}_{\mathsf{IC}}(C, T)| = P(\lambda, |\mathcal{G}_{\mathsf{IC}}|)$ for some polynomial P, where $|\mathcal{G}_{\mathsf{IC}}| = \mathsf{poly}(\lambda, |EK\{Z\}|, |DK\{Z\}|, \log(T), |C|)$. From the (ℓ, s)-efficiency property of ACE, we have that $|EK\{Z\}|$ and $|DK\{Z\}|$ are bounded by $\mathsf{poly}(\lambda, \ell, s)$, where ℓ is the ciphertext security parameter and s is the set description size parameter. We first note that it suffices to set $\ell = 1$ since in our security proof in Section 4.2, we only require indistinguishability of

one ACE ciphertext at any time. Further, at any point, the encapsulation and decapsulation key programs constrained at set $S_j = Z \cup \{(x,t,q) \mid x = i \text{ and } t < j \text{ and } q \neq \bot\}$ that can be described by a circuit of size $\log(|I|, T)$ for any $i \in I, j \in T$, where $|I|$ is at most 2^n.[2] Thus, putting everything together, we have that $|\mathcal{G}_{\mathsf{IC}}| = \mathsf{poly}(\lambda, |C|, \log(T))$, and therefore $|\mathsf{i}\mathcal{O}_{\mathsf{IC}}(C,T)| = \mathsf{poly}(\lambda, |C|, \log(T))$.

5 Inductive Properties

When we move to RAM machines, the main abstraction in our proof of security will be *inductive properties* of programs. These are properties about the encapsulated inputs of a program which are proven by induction on the number of times the program is called. Just as in usual induction, the inductive hypothesis may be stronger than the property being proven. In our case, we call the inductive hypothesis an *invariant*. We care about invariants because we can indistinguishably puncture ACE keys so that plaintexts satisfy the invariant. Here, the size s of the circuit deciding the invariant is a key parameter because it determines the size of the punctured ACE keys. Although the puncturing of keys happens only in the hybrids, it happens inside of an $\mathsf{i}\mathcal{O}$-obfuscated program. Because $\mathsf{i}\mathcal{O}$ only implies indistinguishability of ciruits of equal size, the magnitude of s increases the size even of the real-world program.

In this section, we accomplish two things. We first give a number of basic claims which form a calculus for showing inductive properties with an invariant of small size s. For example, we have a notion of one predicate "inductively implying" another predicate, and a corresponding claim which is analogous to modus ponens. We also have a composition property which allows us to prove inductive properties of a large circuit from inductive properties of a smaller subcircuit.

Next, we consider a generalization of the context in which we will apply ACE. Namely, we are given a circuit C and some inputs $m_1, \ldots, m_n$, and we transform it into a circuit $\tilde{C}$ which takes encapsulated inputs and produces encapsulated outputs. We will want to show that $\mathsf{i}\mathcal{O}(\tilde{C}\|0^p), \mathsf{Enc}(m_1), \ldots, \mathsf{Enc}(m_n)$ is indistinguishable from some other $\mathsf{i}\mathcal{O}(\tilde{C'}\|0^p), \mathsf{Enc}(m'_1), \ldots, \mathsf{Enc}(m'_n)$. We show that this is the case (with only a small amount of padding p) in three different cases: (a) C and C' are "inductively equivalent" and $m_i = m'_i$ (b) C and C' "inductively differ only by a diamond" or (c) $C(m_i) = C'(m_i)$ and $C(x) = C'(x)$ everywhere else.

In the above discussion, we implicitly assumed that a circuit C takes only a single encapsulated input m. In order to go beyond this, we introduce some notation to give a special structure to the domain and range of C - in other words, describing how C parses its input and structures its output. We consider a collection $\mathcal{F}$ of parametrized domains[3], and say that C maps from $F(M)$ to $G(M)$ for some $F, G \in \mathcal{F}$. In our case we will take $\mathcal{F}$ to be disjoint unions of products of sets, which are either M or some other "base" set.

As an example, suppose A is some set (for instance $\mathbb{Z}_p$). Then one instance of an $F \in \mathcal{F}$ is $(\cdot \times A) \sqcup (\cdot \times \cdot)$. Then $F(M)$ would be $(M \times A) \sqcup (M \times M)$. Elements of $\mathcal{F}$ are useful for describing the transformation of a circuit C whose inputs and outputs are plaintexts into a circuit whose inputs and outputs are ciphertexts. (For general F, $\tilde{C}$ would map from $F(X) \to G(X)$, where X is the set of ciphertexts). It will also be important for us to refer to the different parts of $x \in F(M)$ which parse as elements of M. We call these M-components of x.

Definition 5. *We define the M-components of an element of disjoint unions or products of sets as follows:*

- *The M-components of $x \in S$ are defined as $\{x\}$ if $S = M$, and $\emptyset$ if S is another base set (that is, not expressed as a product or disjoint union of smaller sets).*
- *The M-components of*

$$x = (x_1, \ldots, x_n) \in \prod_{i=1}^{n} S_i$$

are defined as the union of the M-components of each $x_i \in S_i$.

[2] Actually in our proof, we also constrain the encapsulation and decapsulation key programs at sets $S_j \cup S$ that contains one extra point than S_{j+1}. But the circuit description of this set has the same size as that of S_j.

[3] For the category-theoretically informed, $\mathcal{F}$ is a functor in Set

- *The M-components of*

$$x = (i, x') \in \bigsqcup_{i=1}^{n} S_i$$

are defined as the M-components of $x' \in S_i$.

Remark 5. *The notion of M-components allows us to handle things like RAM programs, which take one or two encapsulated inputs, and output one or zero encapsulated inputs, together with something unencapsulated.*

Definition 6 (Extension of predicates)**.** *Given a predicate ϕ on M, we define $\bar{\phi}$ on $F(M)$ as:*

$$\bar{\phi}(x) = \textit{every } M\textit{-component of } x \textit{ satisfies } \phi$$

Definition 7 (Invariants)**.** *We say that a predicate ϕ on M is an invariant of $f : F(M) \to G(M)$ if whenever all M-components of $x \in F(M)$ satisfy ϕ, then all M-components of $f(x)$ satisfy ϕ. In other words,*

$$\bar{\phi}(x) \implies \bar{\phi}(f(x))$$

where $\bar{\phi}$ is the extension of ϕ which acts on $F(M)$.

Definition 8 (Inductive predicates)**.** *We say that a predicate ϕ on M is s-inductive with respect to a function $f : F(M) \to G(M)$ given $S_0 \in M^*$ if there is an invariant ϕ' of f such that:*

- $\forall m \in M, \phi'(m) \implies \phi(m)$.
- $\forall m \in S_0, \phi'(m)$
- *There is a circuit $C_{\phi'}$ deciding ϕ' with $|C_{\phi'}| \leq s$*

Claim 1 (Modus ponens)**.** *If ϕ and ψ are predicates on M such that $\phi \implies \psi$ and ϕ is s-inductive, then ψ is s-inductive.*

Proof. The invariant for ϕ can be used to prove inductiveness of ψ. □

Claim 2 (Conjunction)**.** *If ϕ and ψ are predicates on M such that ϕ is s-inductive and ψ is s'-inductive w.r.t. f given S_0, then $(\phi \wedge \psi)$ is $(s + s' + 1)$-inductive w.r.t. f given $S_0 \in M^*$*

Proof. The conjunction of the invariants ϕ' and ψ' suffices to prove the $s + s' + 1$-inductiveness of $\phi \wedge \psi$. □

Definition 9 (Inductive implication)**.** *We say that a predicate ϕ on $F(M)$ inductively implies a predicate ψ on M with respect to $f : F(M) \to G(M)$ given $S_0 \in M^*$ if there is an invariant ψ' of f such that:*

- $\forall m \in M, \psi'(m) \implies \psi(m)$
- $\forall m \in S_0, \psi'(m)$
- $\forall x \in F(M), \phi(x) \wedge \bar{\psi}'(x) \implies \bar{\psi}'(f(x))$
- *there is a circuit $C_{\psi'}$ deciding ψ' with $|C'_{\psi}| \leq s$*

Claim 3 (Inductive modus ponens)**.** *If ϕ and ψ are predicates on M such that ϕ is s-inductive w.r.t. f given S_0, and $\bar{\phi}$ s'-inductively implies ψ w.r.t. f given S_0, then ψ is $(s + s' + 1)$-inductive w.r.t. f given S_0.*

Proof. The conjunction of the invariants ϕ' and ψ' again suffices to show inductiveness. □

Definition 10 (Black Box Attributes)**.** *Given $\pi_Q : M \to Q$ and $C : F(M) \to G(M)$, and $D : F'(Q) \to G'(Q)$, we say that Q is a black box attribute of M in C with respect to D if:*

- *For all inputs $x \in F(M)$, and all M-components m' of $C(x)$, there exists some $x_{m'}$ such that $\pi_Q(m') = D(x_{m'})$, where every Q-component q_i of $x_{m'}$ is equal to $\pi_Q(m_i)$ where m_i is an M-component of x.*

The collection of all $x_{m'}$s above are called the inputs to D in C on x.

Claim 4 (Inductive properties of black-box attributes)**.** *Suppose that ϕ is an s-inductive predicate of $D : F'(Q) \to G'(Q)$ with respect to $\pi_Q(S_0)$, and Q is a black box attribute of M in $C : F(M) \to G(M)$ with respect to D. Then $\phi \circ \pi_Q$ is an $s + |\pi_Q|$-inductive property of C with respect to S_0.*

Proof. If ϕ' is an invariant of D which implies ϕ, then $\phi' \circ \pi_Q$ is an invariant which implies $\phi \circ \pi_Q$. □

Claim 5 (Inductive implication of black-box predicates)**.** *Suppose that Q is a black-box attribute of M, C is a circuit mapping $F(M) \to G(M)$, D is a circuit mapping $F'(Q) \to G'(Q)$, χ is an s-inductive property on M with respect to C, ϕ is a predicate on $F'(Q)$ and ψ is a predicate on Q such that ϕ s'-inductively implies ψ.*

If $\chi(x)$ implies that all inputs to D on x satisfy ϕ then $\psi \circ \pi_Q$ is $s + s' + |\pi_Q|$-inductive with respect to C.

Proof. $\chi' \wedge (\psi' \circ \pi_Q)$ is the required invariant, where χ' and ψ' are the invariants for χ and ψ respectively. □

Definition 11 (Graded Posets)**.** *A poset M is graded if there is a function $\rho : M \to \mathbb{N}$ such that for $x, y \in M$, $x \prec y \iff \rho(x) < \rho(y)$. $\rho(x)$ is called the* grade *of x.*

Definition 12 (Strictly Ascending Functions)**.** *If M is a poset, we say that a function $f : F(M) \to G(M)$ is strictly ascending in M if for all $x \in F(M)$, all M-components m of x, and all M-components m' of $f(x)$, we have $m' \succ m$*

The next three theorems are the raison d'être of inductive properties:

Theorem 3 (Indistinguishability of Inductively Equivalent Programs)**.** *Suppose that M is a graded poset, $S_0 \in M^*$, and C_0 and C_1 are circuits mapping $F(M) \to G(M)$ such that:*

- *C_0 and C_1 are strictly ascending in M*
- *ϕ is s-inductive w.r.t. both C_0 and C_1 given $S_0 \in M^*$*
- *$|C_0| = |C_1|$*
- *$C_0(x) = C_1(x)$ whenever either the M-components of x all satisfy ϕ or some M-component of x has rank at least T.*

Then $(\mathsf{i}\mathcal{O}(\mathtt{AddACE}(C_0)\|0^p), \mathsf{Enc}(S_0)) \approx (\mathsf{i}\mathcal{O}(\mathtt{AddACE}(C_1)\|0^p, \mathsf{Enc}(S_0))$, where the amount of padding p is $\mathsf{poly}(\log T, s, \lambda)$ and $\mathsf{Enc}(S_0)$ denotes the element-wise encapsulation of S_0 using the same ACE encapsulation key as is used in $\tilde{C}_b$.

Remark 6. *We will consider C_0 and C_1 which are designed to be run T times, and in particular expect never to receive an input with timestamp T or greater. If they do, they output $\perp$, which means that indeed if the rank of an M-component of x is at least T, then $C_0(x) = C_1(x)$.*

Proof. We show a sequence of indistinguishable hybrids $H_0, \ldots, H_{2T+1}$ such that $H_0 = \mathsf{i}\mathcal{O}(\tilde{C}_0)$, and $H_{2T+1} = \mathsf{i}\mathcal{O}(\tilde{C}_1)$.

For $i \leq T$, the ith hybrid H_i is $\tilde{C}_{0,i}$, where $\tilde{C}_{0,i}$ is defined identically to $\tilde{C}_0$, but with ACE encapsulation key and decapsulation key punctured at

$$A_i = \{m \in M : \mathrm{rank}(m) < i \wedge \neg\phi'(m)\}$$

were ϕ' is the invariant for ϕ. Note that $A_0 = \emptyset$, so indeed H_0 is the same as $\tilde{C}_0$.

For $i > T$, the ith hybrid H_i is $\mathsf{i}\mathcal{O}(\tilde{C}_{1,i})$, where $\tilde{C}_{1,i}$ is defined identically to $\tilde{C}$ but with the ACE encapsulation and decapsulation key punctured at A_{2T+1-i}. When $i = 2T + 1$, the punctured set is again $\emptyset$, so hybrid H_{2T+1} is the same as $\mathsf{i}\mathcal{O}(\tilde{C}_1)$. We now just need to show that H_i is indistinguishable from H_{i+1}.

First, let us show that $H_T \approx H_{T+1}$. The decapsulation keys for both H_T and H_{T+1} are punctured on the same set (A_T). So if any X-component of an input x to H_T fails to decapsulate, then $H_T(x) = \perp = H_{T+1}(x)$.

On the other hand, if all the X-components of input x to H_T decapsulate properly to a message m, then there are two possibilities. Either $\text{rank}(m) \geq T$ for some M-component m of the decapsulated input, or all of the M-components of the decapsulated input satisfy ϕ. In either case, we are guaranteed that the output of C_0 is the same as the output of C_1 on the decapsulated input. By the security of $i\mathcal{O}$, $H_T \approx H_{T+1}$.

We now show that for $0 \leq i < T$, $H_i \approx H_{i+1}$. We use an intermediate hybrid $H_i.A = i\mathcal{O}(\tilde{C}_{0,i}.A)$, in which the decapsulation key is punctured at A_i, but the encapsulation key is punctured at A_{i+1}.

H_i is indistinguishable from $H_i.A$ by $i\mathcal{O}$. Because $A_i \subset A_{i+1}$, $\tilde{C}_{0,i}$ and $\tilde{C}_{0,i}.A$ differ in functionality only if the copy of C_0 embedded in $\tilde{C}_{0,i}$ outputs $m \in A_{i+1} \setminus A_i$, i.e. $rank(m) = i \wedge \neg\phi'(m)$. But this can't happen, because if the rank of an output of C_0 is i, then the rank of all inputs to C_0 must be less than i. And by the puncturing of the decapsulation key, if all inputs decapsulate correctly and have rank less than i, then all the decapsulations satisfy ϕ'. But because ϕ' is an invariant of C_0, the outputs of C_0 must satisfy ϕ' as well.

$H_i.A$ is indistinguishable from H_{i+1} by the constrainability of decapsulation keys (note that S_0 is disjoint from A_{i+1} because every element of S_0 satisfies ϕ), so we have shown that $H_i \approx H_{i+1}$.

The case for $i > T$ is proved identically, which concludes the proof of the theorem.. □

We give one more theorem which allows us to switch from encapsulating S_0 to encapsulating a different S_0'.

Theorem 4. *Let $S_0 = (s_1, \ldots, s_n)$, and $S_0' = (s_1', \ldots, s_n')$ such that none of these elements is ever an M-output of C (C is a circuit mapping $F(M) \to G(M)$). If changing an M-input of C from s_i to s_i' never changes the output of C, then $(i\mathcal{O}(\tilde{C}\|0^p), \mathsf{Enc}(S_0)) \approx (i\mathcal{O}(\tilde{C}\|0^p), \mathsf{Enc}(S_0'))$, where p, the amount of padding is $\mathsf{poly}(|s_i|)$ but is independent of n.*

Proof. The proof will proceed in $n+1$ indistinguishable hybrids $H_0, \ldots, H_n$.

$H_i = (i\mathcal{O}(\tilde{C}\|0^p), \mathsf{Enc}(s_1', \ldots, s_i', s_{i+1}, \ldots, s_n))$. We show that H_i is indistinguishable from H_{i+1} in a few steps.

1. First, we puncture EK at s_{i+1} and at s_{i+1}', which is indistinguishable by $i\mathcal{O}$ because each of them have rank 0 and thus are never an output of C.
2. Next, we puncture DK at s_{i+1} and s_{i+1}', hard-coding the decapsulations. This is indistinguishable by $i\mathcal{O}$.
3. Next, we swap the hard-coded values c_{i+1} and c_{i+1}' wherever they appear. Note that this includes changing $\mathsf{Enc}(S_0)$ from having an encapsulation of s_{i+1} to having an encapsulation of s_{i+1}'. This change is indistinguishable by the indistinguishability of ciphertexts given a punctured key.
4. Now we "fix" the hard-coded decapsulation, so that c_{i+1} and c_{i+1}' map to s_{i+1} and s_{i+1}' respectively. This change is indistinguishable by $i\mathcal{O}$ because C produces the same output whenever an input of s_{i+1} is replaced with an input of s_{i+1}'.
5. We can now remove the hard-coding and unpuncture the encapsulation and decapsulation keys by $i\mathcal{O}$, which results in Hybrid H_{i+1}.

The padding p we need to use is just the size cost of puncturing a decapsulation key at two points plus the cost of hard-coding two decapsulations. This padding is $\mathsf{poly}(|s_i|)$. □

The following theorem is similar in spirit to Theorem 4, but applies to changing functionality purely inside the obfuscated program, and not changing the encapsulated inputs.

Theorem 5 (Diamond)**.** *Suppose M is a graded poset, $C_0, C_1 : F(M) \to G(M)$ with $|C_0| = |C_1|$, C_0 and C_1 are strictly ascending in M, $S_0 \in M^*$, $m_0, m_1 \in M$, and suppose there is some predicate ϕ on M which is s-inductive with respect to both C_0 and C_1 given S_0, and suppose that there exists $x^* \in F(M)$ such that for all $x \in F(M)$ satisfying $\bar{\phi}$:*

- *m_1 is not an M-component of $C_0(x)$*
- *m_0 is not an M-component of $C_1(x)$*
- *The following three propositions are equivalent:*
  - *m_0 is an M-component of $C_0(x)$*
  - *m_1 is an M-component of $C_1(x)$*
  - *$x = x^*$*
- *If m_0 is an M-component of x, then*

$$C_0(x) = C_1(x[m_0 \mapsto m_1])$$

  Here, $x[m_0 \mapsto m_1]$ denotes x where m_0 is replaced by m_1.
- *If $x \neq x^*$ and neither m_0 nor m_1 are M-components of x, then $C_0(x) = C_1(x)$*

Then $(i\mathcal{O}(\texttt{AddACE}(C_0)\|0^p), \mathsf{Enc}(S_0)) \approx (i\mathcal{O}(\texttt{AddACE}(C_1)\|0^p), \mathsf{Enc}(S_0))$, where p is an amount of padding which is $\mathsf{poly}(s)$.

Remark 7. *There is a simple special case, illustrated by Figure 1 which is when $F(M) = M$ and $G(M) = M \cup \{\bot\}$. C_0 and C_1 each induce a directed graph on M. Then Theorem 5 is applicable when the graphs of C_0 and C_1 differ "only by a diamond":*

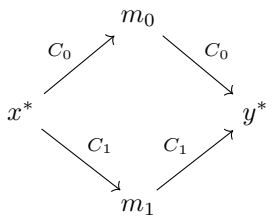

Figure 1: The namesake of the diamond theorem

We now prove Theorem 5.

Proof. We make a sequence of indistinguishable changes which transform $i\mathcal{O}(\texttt{AddACE}(C_0))$ into $i\mathcal{O}(\texttt{AddACE}(C_1))$. All of our changes happen before the obfuscation.

1. First, using Theorem 3, we can add an assertion to the inputs of C_0 that the inputs satisfy the invariant ϕ' for ϕ (recall $\phi' \implies \phi$).
2. We hard-code the output of $\texttt{AddACE}(C_0)$ so that on input x^*, it outputs $c^* = \mathsf{Enc}(m_0)$. This change is indistinguishable by the security of $i\mathcal{O}$ because the functionality is the same. Indeed, $C_0(x^*) = m_0$, and so $\texttt{AddACE}(C_0)(x^*) = c^*$.
3. We puncture EK to $EK\{m_0, m_1\}$. This is indistinguishable by $i\mathcal{O}$ because the functionality is the same: Because of our assertion that the input always satisfies ϕ, C_0 never outputs m_1, and x^* is the only input on which m_0 would be output. So because of our hard-coding, we never need to encapsulate either m_0 or m_1.
4. We hard-code the decapsulation of c^* as m_0 and puncture DK to $DK\{m_0\}$. This is again indistinguishable by $i\mathcal{O}$ because our hard-coding implies that we never need to use DK to decapsulate m_0.

5. We puncture $DK\{m_0\}$ to $DK\{m_0, m_1\}$. This is indistinguishable by the constrainability of decapsulation keys, since EK is already punctured at $\{m_0, m_1\}$, and our only auxiliary ciphertext is an encapsulation of m_0.

6. We replace c^* by $\mathsf{Enc}(m_1)$ (indistinguishability of ciphertexts)

7. We unpuncture $DK\{m_0, m_1\}$ to $DK\{m_0\}$, remove the hard-coded decapsulation of c^*, and replace C_0 by C_1. This is indistinguishable by $i\mathcal{O}$ because functionality is preserved. To see this, consider three types of inputs $\tilde{x}$:

   - If $\tilde{x}$ has a component which is an encapsulation of m_0, then decapsulation fails and thus $\perp$ is output, with or without this change.
   - Otherwise, if $\tilde{x}$ has a component which is an encapsulation of m_1 (that is, c^*), then this change says that instead of computing $C_0(x[m_1 \mapsto m_0])$, we compute $C_1(x)$, which produces the same result.
   - If $\tilde{x}$ contains neither an encapsulation of m_0 nor an encapsulation of m_1, then $C_0(x) = C_1(x)$.

8. We can now unpuncture $DK\{m_0\}$ to DK by the constrainability of decapsulation keys.

9. We unpuncture $EK\{m_0, m_1\}$ to EK and un-hardcode c^* as the output on input x^* by $i\mathcal{O}$.

10. Finally, we remove the assertion that we inserted in step 1 by applying Theorem 3 again.

□

6 Obfuscating Bounded-Space RAM Programs

Now given a RAM program M, we apply some of the same techniques to garble and obfuscate M more efficiently than converting M into a circuit, or even an iterated circuit.

6.1 RAM Programs

We now formally define RAM programs. We assume that for each RAM program M we are given a bound S_M on the amount of space used by M, as well as a smaller bound n_M on the length of the inputs to M. We also assume we are given a worst-case bound T_M on the running time of M.

Definition 13. *A M is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, and Γ are finite sets encoded by binary strings of lengths l_Q, l_Σ, and l_Γ respectively.*

- *Q is the set of states*
- *Γ is the memory alphabet, including the blank symbol $\sqcup$.*
- *$\Sigma \subset \Gamma$ is the input alphabet, with $\sqcup \notin \Sigma$.*
- *δ is a circuit mapping $Q \times R \to (Q \times A) \cup \{\perp\}$, where $R = \Gamma \sqcup \{\perp\}$ is the set of read and write responses, while $A = [s] \sqcup ([s] \times \Gamma)$ is the set of read and write accesses. When we say that δ inputs or outputs elements of Q or Γ, we assume it uses the encodings in $\{0,1\}^{l_Q}$ or $\{0,1\}^{l_\Gamma}$.*
- *$q_0 \in Q$ is the initial state.*
- *q_{accept} and q_{reject} are designated states in Q for accepting and rejecting an input, respectively, where $q_{accept} \neq q_{reject}$.*
- *n_M is the length of inputs. Inputs take the form Σ^{n_M}.*

- $S_M \geq n_M$ *is a bound on the space used by* M *- that is, its memory is represented as an element of* Γ^{S_M}.

A bounded RAM program M is encoded by the tuple $(\delta, q_0, q_{accept}, q_{reject}, n_M, S_M)$, and we say that the size $|M|$ is the size of the encoding of this tuple. In particular, we don't need to give a (giant) encoding of the sets Q, Σ, or Γ.

Definition 13 will be useful for us as a canonical form for RAM programs, but we will not directly specify RAM programs using Definition 13. Instead, we will specify them with imperative code which uses $\texttt{Get}(i)$ and $\texttt{Put}(i, x)$ to denote outputting an access. Whenever we consider a RAM program M, we assume the space bound is known and is denoted S_M. The compilation from imperative code is straight-forward and is deferred to Appendix **??**. One key property is that the size of the circuit δ depends only polynomially on the length of the program, unlike the running time of the program.

6.1.1 Interactive RAM Machines

Our construction will involve layering an ORAM and a "predictably timed writes" data structure on top of the original machine M, and then obfuscating that composite machine. In order to formally define composition, we introduce the notion of interactive RAM machines. More generally, we can consider interactive RAM machines (which can receive many inputs, invoke subroutines, and return many values, all while keeping state around) by partitioning the set of states into three disjoint types.

Definition 14. *An interactive machine* M *is a tuple* $(Q_I, I, Q_V, V, Q_W, W, A, q_0, \delta_I, \delta_V, \delta_W, \alpha, \nu)$*, where* Q_I*,* I*,* Q_V*,* V*,* Q_W*,* W*, and* A *are finite sets which are all encoded by disjoint subsets of* $\{0,1\}^l$.

- Q_I *is the set of states which are "ready for input"*
- I *is the set of inputs*
- Q_V *is the set of states which "have completed with a return value"*
- V *is the set of return values* V.
- $\nu : Q_V \to V$ *is a function which extracts the return value from a state .*
- Q_W *is the set of states which are "waiting for a subroutine to return"*
- W *is the set of return values a subroutine is allowed to return.*
- A *is the set of arguments which can be passed to a subroutine.*
- $q_0 \in Q_I$ *is the starting state of* M
- α *is a function from* Q_W *to* A *which extracts the argument intended for a subroutine from the state.*
- δ_I*,* δ_V*, and* δ_W *are all transition functions whose type signatures reflect that different information is needed to proceed in different types of states.* $\delta_I : Q_I \times I \to Q_W \cup \{\perp\}$*,* $\delta_W : Q_W \times W \to Q_W \cup Q_V \cup \{\perp\}$*, and* $\delta_V : Q_V \to Q_I \cup \{\perp\}$.

Note that a non-interactive s-bounded RAM machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ is just a special case of s-bounded interactive RAM machines. `Get`s and `Put`s are regarded as a single subroutine whose argument set A is tuples of the form (READ, i) and (WRITE, i, x), and whose W is $\{\perp\} \cup \Gamma$. Formally:

- Q_I is the singleton set $\{q_0\}$. I is just $\{\perp\}$.
- $Q_V = \{q_{accept}, q_{reject}\}$, $V = \{$"accept", "reject"$\}$
- $\nu : Q_V \to V$ is defined in the obvious way.

- $Q_W = (Q \setminus \{q_{accept}, q_{reject}, q_0\}) \times A$, where $A = \{(\mathsf{READ}, i) : i \in [s]\} \cup \{(\mathsf{WRITE}, i, x) : i \in [s] \wedge x \in \Gamma\}$. $W = \Gamma \cup \{\perp\}$.
- The function $\alpha : Q_W \to A$ is just projection onto the second component.
- $\delta_I(q_0, \perp) = \delta(q_0, \perp)$.
- $\delta_W((q, a), w) = \delta(q, w)$.
- $\delta_V(q_{accept}) = \delta_V(q_{reject}) = \perp$.

6.1.2 Composition

Suppose that M and N are interactive machines such that $M.A = N.I$ and $M.W = N.V$. We define their composition $M \circ N$ by defining:

- $Q_I = M.Q_I \times N.Q_I$
- $I = M.I$
- $Q_V = M.Q_V \times N.Q_I$
- $V = M.V$
- $Q_W = M.Q_W \times N.Q_W$
- $W = N.W$
- $A = N.A$
- $q_0 = M.Q_0 \times N.Q_0$
- $\alpha((q_M, q_N)) = \alpha(q_N)$
- $\nu((q_M, q_N)) = \nu(q_M)$
- $\delta_I((q_M, q_N), w) = (\delta_I(q_M, w), \delta_I(q_N, \alpha(\delta_I(q_M, w))))$
- $\delta_V((q_M, q_N)) = (\delta_V(q_M), q_N)$
- δ_W is defined in Algorithm 15:

Essentially we are just letting M use N as a subroutine whose argument is in A and whose return value is in W.

6.2 ORAM

Our construction of a succinct one-time Garbled RAM will rely heavily on Oblivious Random-Access Memory (ORAM) schemes, which we now define. Informally, an ORAM is a way of hiding the pattern of memory locations accessed by a RAM machine. We formalize the security property we need in Section 6.2.3. We note that the property that we need is stronger than the standard one; still, known schemes have this property. See Sections 6.2.3 and 6.2.4 for more details.

6.2.1 Syntax

An ORAM is a data structure parameterized by an underlying word set W, and a memory size s. Given these parameters, the ORAM will have a private state set Q, a word set $\tilde{W}$ and memory size $\tilde{s}$. Syntactically, an ORAM has two operations: `Init` and `OAccess`, both of which are probabilistic.

Algorithm 15: $\delta_W : Q_W \times W \to Q_W \sqcup Q_V \sqcup \{\perp\}$ for the composition $M \circ N$

Input: State (q_M, q_N), word w
Output: State q'

$q'_N := N.\delta_W(q_N, w)$;
if $q'_N \in Q_W$ **then return** (q_M, q'_N);
else if $q'_N = \perp$ **then return** $\perp$;
// else $q'_N \in Q_V$
$v := \nu(q'_N)$;
$q'_N := \delta_V(q'_N)$;
$q'_M := \delta_W(q_M, v)$;
if $q'_M \in Q_V$ **then return** (q'_M, q'_N);
else if $q'_M = \perp$ **then return** $\perp$;
// else $q'_M \in Q_W$
$q'_N := \delta_I(q'_N, \alpha(q'_M))$;
return (q'_M, q'_N);

Init. $\texttt{Init}(D_0, 1^\lambda) \to (q_0, \tilde{D}_0)$ initializes an ORAM given an input D_0 — a sequence of words $w_1, \ldots, w_s$ which describes the underlying initial memory contents — and a security parameter λ. `Init` outputs $q_0 \in Q$ where q_0 is an initial private state, as well as $\tilde{D}_0$, a sequence of ORAM words $\tilde{w}_1, \ldots, \tilde{w}_{\tilde{s}}$ describing the initial physical memory of the data structure.

OAccess. We think of `OAccess` as an interactive machine. Its input consists of an underlying access instruction $a \in \{\mathsf{READ}, \mathsf{WRITE}\}$, an underlying memory location j_i to be accessed, and if $a = \mathsf{WRITE}$ then it also has a value x_i to be written. `OAccess` also outputs physical accesses as subroutine calls, and receives responses. `OAccess` outputs a response to its input access. If the input was a READ, then the output is the retrieved value. Otherwise, the output is $\perp$. To avoid confusion, we refer to the memory read and write operations performed by `OAccess` as $\texttt{Get}(i)$ and $\texttt{Put}(i, x)$.

The *underlying memory access pattern* in a given execution (i.e., sequence of activations) of an ORAM scheme is the sequence of reads $\mathsf{READ}(i)$ and writes $\mathsf{WRITE}(i, x)$ which are given to `OAccess` as input. In other words, these are the locations in the underlying memory that the RAM machine accesses. The *physical memory access pattern* of an execution of `OAccess` is the sequence of $\texttt{Get}(i)$ and $\texttt{Put}(i, x)$ instructions which are generated by `OAccess`.

6.2.2 Correctness

An ORAM is correct with high probability if there is a negligible function $\epsilon(\cdot)$ such that for all underlying memory access patterns $a_0, \ldots, a_t$ ($t = \mathsf{poly}(\lambda)$), all initial underlying memory contents D_0, and all $j \in [s]$,

$$\Pr[(q_0, \tilde{D}_0) \leftarrow \texttt{Init}(D_0, 1^\lambda), (q_{i+1}, \tilde{D}_{i+1}) \leftarrow \texttt{OAccess}(q_i, a_i; \tilde{D}_i), \\ x \leftarrow \texttt{OAccess}(q_{t+1}, \mathsf{READ}(j); \tilde{D}_{t+1}); x \neq x_j] \leq \epsilon(\lambda)$$

Here x_j is the content of memory location j in the underlying machine, and is defined in the natural way: If t_j is the largest $i \leq t$ such that a_i is of the form $\mathsf{WRITE}(j, *)$, and in particular if a_{t_j} is of the form $\mathsf{WRITE}(j, x^*)$, then x_j is x^*. If there is no such a_t, then x_j is $D_0[j]$.

6.2.3 Security

The standard notion of security for an ORAM says that the physical accesses produced by any two sequences of underlying memory accesses are indistinguishable. However, this definition is only useful when we have

the luxury of a black-box CPU so that we can hide the secrets and state that were used in the generation of the physical access pattern. Since we want to use indistinguishability obfuscation, we require a stonger notion of security. We require the existence of a dummy physical access sequence distribution, as well as the ability to simulate a tuple of a final state together with a final set of memory contents $(q_{\mathsf{final}}, \tilde{D}_{\mathsf{final}})$ consistent with given physical and virtual access sequences.

For all t, all D_0, and all virtual access sequences $a_1, \ldots, a_t$, let $s_{j,t}$ be the largest $s \leq t$ such that a_s is an access (READ or WRITE) to location j. In words, $s_{j,t}$ is the last time that location j was accessed. Let $\hat{a}_{j,t}$ be the tuple $(s_{j,t}, \tilde{D}_{s_{j,t}}[j])$.

Strong Simulation. We say that an ORAM is *strongly simulatable* if there exist probabilistic algorithms $\mathtt{OSample}$ and Sim satisfying the following property:

For all physical access patterns (excluding the values written) $I_1, \ldots, I_{t-1}$ with $I_i \in \mathrm{Range}(\mathtt{OSample}(i; \cdot))$, the following two distributions of $(I_t, q_{t+1}, \tilde{D}_{t+1})$ are indistinguishable.

1. Sample $q_t, \tilde{D}_t \leftarrow \mathsf{Sim}(\hat{a}_{1,t-1}, \ldots, \hat{a}_{s,t-1}, I_1, \ldots, I_{t-1})$. Then $I_t, q_{t+1}, \tilde{D}_{t+1} \leftarrow \mathtt{OAccess}(a_t, q_t; \tilde{D}_t)$
2. Sample $I_t \leftarrow \mathtt{OSample}(t)$. Then $q_{t+1}, \tilde{D}_{t+1} \leftarrow \mathsf{Sim}(\hat{a}_{1,t}, \ldots, \hat{a}_{s,t}, I_1, \ldots I_t)$.

This property says that making $t-1$ fake accesses and one real access is indistinguishable from making t fake accesses, even if the adversary sees all the state (q_{t+1} and $\tilde{D}_{t+1}$) afterwards. Obviously the state $(q_{t+1}, \tilde{D}_{t+1})$ reveals the virtual memory contents at time $t+1$, but the fact that Sim only depends on the *most recent* access to a virtual location is as close as we can come to saying that this is all that is revealed.

6.2.4 ORAM Construction

We show that a slight modification of the ORAM scheme constructed by Shi, Chen, Stefanov and Li [SCSL11], and simplified by Chung and Pass [CP13] satisfies our correctness and strong simulation security properties. We first review the construction of [CP13] and then argue why it satisfies the strong simulation definition.

The CP/SCSL Construction. Starting with a RAM machine Π that uses N memory words, the construction transforms it into a machine Π' that uses $N' = N \cdot \mathsf{poly}(\log N, \lambda)$ memory words. While the eventual goal is to store $\mathsf{poly}(\log N)$ words in the local state of Π', Chung and Pass start with a "basic" construction wherein the local state of Π' consists of a "position map" $\mathsf{pos} : [N/\alpha] \to [N/\alpha]$ for some $\alpha = \mathsf{polylog}(N)$.

The N underlying memory locations are divided into N/α "blocks" each storing α underlying memory words. The external memory is organized as a complete binary tree of N/α leaves. The semantics of the position map is that the i^{th} block of memory maps to the leaf labeled $\mathsf{pos}(i)$. Let $d = \log(N/\alpha)$. The CP/SCSL invariant is that:

> *"Block i is stored in some node on the path from the root to the leaf labeled with* $\mathsf{pos}(i)$*."*

Each internal node of the tree stores a few memory blocks. In particular, each internal node, labeled by a string $\gamma \in \{0,1\}^{\leq d}$ is associated with a "bucket" of β blocks for some $\beta = \mathsf{polylog}(N)$.

The reads and writes to a location $r \in [N]$ in the CP/SCSL ORAM proceed as follows:

- Fetch: Let $b = \lfloor r/\alpha \rfloor$ be the block containing the memory location r, and let $i = r \bmod \alpha$ be the component within block b containing the location r. We first look up the leaf corresponding to block b using the (locally stored) position map. Let $p = \mathsf{Pos}(b)$.

  Next, we traverse the tree from the roof to the leaf p, reading and writing the bucket associated to each internal node exactly once. In particular, we read the content once, and then we either write it back *permuting the blocks in each bucket*, or we erase a block once it is found, and write it back *permuting the blocks*.

- Update Position Map: Pick a uniformly random leaf $p' \leftarrow [N/\alpha]$ and set (in the local memory) $\mathsf{Pos}(b) = p'$.

- Write Back: In the case of a READ, add the tuple (b, p', v) to the root of the tree. In the case of a WRITE, add the tuple (b, p', v') where v' is the new value to be written. If there is not enough space in the bucket associated with the root, output overflow and abort. (Jumping ahead, we note that [CP13, SCSL11] show that, setting the parameters appropriately, the probability that the overflow event happens is negligible).

- Flush the Block: Pick a uniformly random leaf $p^* \leftarrow [N/\alpha]$ and traverse the tree from the roof to the leaf p^* making exactly one read and one write operation for every memory cell associated with the nodes along the path so as to implement the following task: "push down" each tuple $(\tilde{b}, \tilde{p}, \tilde{v})$ read in the nodes traversed as far as possible along the path to p^* while ensuring that the tuple is still on the path to its associated leaf $\tilde{p}$ (i.e., maintaining the CP/SCSL invariant). In other words, the tuple ends up in the node γ = the longest common prefix of p^* and $\widetilde{p}$. If at any point some bucket is about to overflow, abort outputting overflow.

The following observation is central to the correctness and security of the CP/SCSL ORAM:

> *Each oblivious READ and WRITE operation traverses the the tree along two randomly chosen paths, independent of the history of operations so far.*

The key observation follows from the facts that (1) Each position in the position map is used exactly once in a traversal (and before this traversal, no information about the position is used in determining what nodes to traverse), and (2) the flushing, by definition, traverses a random path, independent of the history.

This basic ORAM construction has the drawback that the local state of Π' stores $O(N)$ memory words for the position map. However, we can outsource the storage of the position map to the memory, recursively. Recall that each invocation of an oblivious READ and WRITE requires reading just one position in the position map and updating its value to a random leaf; that is, we need to perform a single recursive oblivious READ or WRITE call to emulate the position map. At the base case of the recursion, when position map is of constant size, we use the basic ORAM construction which simply stores the position map in the registers.

Strong Simulation of the CP/SCSL Construction. Informally, the first property holds because the state is just the assigned location for each block, as well as the physical location of that block. The assigned location of each block is uniformly random and independent of everything. The physical location of the block depends only on when that block was last written, as well as all the accesses which come after, which are again independently uniformly random.

So the simulator can just randomly assign locations to each block, randomly choose an access pattern, and then compute the physical locations of each block.

We first show the strong simulation security of the basic construction (with a local memory of size $O(N)$). We will then extend this to the recursive construction (with a constant-size local memory). In more detail, we construct the simulator algorithms `OSample` and Sim as follows.

`OSample` samples and outputs two uniformly random paths in the tree.

Sim takes as input a tuple $(\hat{a}_{1,t}, \ldots, \hat{a}_{N,t})$ and sequences of memory accesses $(I_1, \ldots, I_t)$ and does the following. For every memory block $b \in [N/\alpha]$, let $\tau_b \leq t$ be the last time when block b was read or written to. Let $I_j = (I_j^{\mathsf{read}}, I_j^{\mathsf{flsh}})$ be the pair of paths that comprise each I_j.

- For each block b, pick a uniformly random leaf $p = p_b \leftarrow \lfloor N/\alpha \rfloor$. Compute the unique internal node γ_b such that γ_b is the largest common prefix between p and each of $I_{\tau_b}^{\mathsf{flsh}}, \ldots, I_t^{\mathsf{flsh}}$.

- Construct the state q_{t+1} consisting of the position map by letting $\mathsf{Pos}(b) = p_b$.

- Construct the final memory $\tilde{D}_t$ by writing each memory block b together with its value to the internal node γ_b.

Finally, the simulator for the recursive construction simply runs the level-1 simulator described above to get q_t, the memory content for the second level ORAM. It then runs the level-1 simulator with q_t as input to get the state for the second level of the recursion, and so on.

We show that $(\texttt{OSample}, \mathsf{Sim})$ satisfies strong simulation security. We start by observing that whenever we READ or WRITE a block b, we remove all trace of it from the memory, eventually rewriting it to the root. The block then gets flushed down in subsequent accesses to blocks $b' \neq b$.

To show security, fix any sequence of physical access patterns $(I_1, \ldots, I_{t-1})$, last-access-time pairs $\hat{a}_{1,t-1}, \ldots, \hat{a}_{s,t-1}$, and an access a_t. We would like to show that the following two distributions of $(I_t, q_{t+1}, \tilde{D}_{t+1})$ are indistinguishable.

1. Sample $q_t, \tilde{D}_t \leftarrow \mathsf{Sim}(\hat{a}_{1,t-1}, \ldots, \hat{a}_{s,t-1}, I_1, \ldots, I_{t-1})$. Then $I_t, q_{t+1}, \tilde{D}_{t+1} \leftarrow \texttt{OAccess}(a_t, q_t; \tilde{D}_t)$
2. Sample $I_t \leftarrow \texttt{OSample}(t)$. Then $q_{t+1}, \tilde{D}_{t+1} \leftarrow \mathsf{Sim}(\hat{a}_{1,t}, \ldots, \hat{a}_{s,t}, I_1, \ldots I_t)$.

The second distribution differs from the first in the a_t^{th} location of the state q_{t+1}, the location of the a_t^{th} block in the memory $\tilde{D}_{t+1}$ as well as I_t. In both cases, I_t is a pair of uniformly random paths, and the a_t^{th} location of the state q_{t+1} is a third uniform and independent path. Finally, in both cases, the block a_t gets written to the root of the tree and gets flushed down to the longest common prefix of $\mathsf{Pos}(a_t)$ and I_t^{flsh}. Thus, the two distribution are identical.

6.3 Freshness Guarantor

Recall that in our $\mathsf{i}\mathcal{O}_{\mathsf{IC}}$ scheme, we used ACE to ensure that if an evaluator tries to give the wrong values to the garbled RAM circuit, the circuit will just output $\perp$. In particular, each value that our program writes to memory is written in a tuple with the location and time it is being written. In order to ensure stale values or values from the future are not accepted, our program must ensure that on every read of a location it knows what time that location was last written.

We achieve this with a data structure that we call a freshness guarantor, described in [GHRW14] but reproduced here for completeness. This data structure will have as parameters an index set $\mathcal{I}_0$ which is the set of locations in memory whose "last-written" times are being tracked, as well as T, the maximum time we are interested in. The data structure itself will be stored in a separate area of memory whose locations are indexed by a set $\mathcal{I}_{FG}$.

The data structure consists of a balanced binary tree with one leaf per two elements of $\mathcal{I}_0$. We will index the nodes of this tree by binary strings with length up to d, where $d = \log |I_0|$. The root node is denoted by n_ϵ, where ϵ is the empty string. The left child of a node n_i is $n_{i\|0}$, while the right child is $n_{i\|1}$. Each node n_i has fields t, t_0, and t_1, where $t_b \in [T]$ is supposed to store $n_{i\|b}.t$.

The freshness guarantor has three operations: `Init`, `PreRead`, and `PreWrite`.

Init. `Init` sets every field of every node to 0.

PreRead. `PreRead` is described in Algorithm 16. It takes as arguments an element of I_0, represented as a binary string $i = i_0\| \cdots \|i_d$, and a time t. It accesses each node n_{p_j}, where $p_j = i_0\| \cdots \|i_{j-1}$, checking that $n_\epsilon.t = t - 1$ and $n_{p_j}.t_{i_j} = n_{p_{j+1}}.t$. If each of these checks passes, then `PreRead` updates $n_\epsilon.t_0$ to be t outputs $n_{p_j}.t_{i_j}$. Otherwise, PreRead outputs $\perp$.

PreWrite. `PreWrite` is described in Algorithm 17. It takes as arguments an element of $\mathcal{I}_0$, represented as a binary string $i = i_0\| \cdots \|i_d$, and a time t. It accesses each node n_{p_j}, where $p_j = i_0\| \cdots \|i_{j-1}$, checking that $n_{p_j}.t_{i_j} = n_{p_{j+1}}.t$. It also writes back each node n_{p_j}, updating $n_{p_j}.t_{i_j}$ and $n_{p_j}.t$ to be t. `PreWrite` does not have a return value.

It is easy to see that each operation on a freshness guarantor preserves the following invariants:

- Each node's expected timestamp for its child matches that child's timestamp. That is, for each node n_s, $n_s.t_b = n_{s\|b}.t$.

Algorithm 16: PreRead

Input: $i = i_0 \| \cdots \| i_d \in \mathcal{I}_0$, $t \in [T]$

$t_{exp} := t - 1$;
for $j := 0$ **to** d **do**
  $n := \texttt{Get}(i_0 \| \cdots \| i_{j-1})$; `// Here,` $i_0 \cdots i_{-1}$ `means the empty string` ϵ
  assert $t_{exp} = n.t$;
  $t_{exp} := n.t_{i_j}$;
`// Now write back the root with an updated timestamp.`
$root := \texttt{Get}(\epsilon)$;
$root.t := t$;
$\texttt{Put}(\epsilon, root)$;
return t_n

Algorithm 17: PreWrite

Input: $i = i_0 \| \cdots \| i_d \in \mathcal{I}_0$, $t \in [T]$

$t_{exp} := t - 1$;
for $j := 0$ **to** d **do**
  $n := \texttt{Get}(i_0 \| \cdots \| i_{j-1})$; `// Here,` $i_0 \cdots i_{-1}$ `means the empty string`
  assert $t_{exp} = n.t$;
  $t_{exp} := n.t_{i_j}$;
  $n.t := t$ `// Write back with updated timestamp and expectation for child`
  $n.t_{i_j} := t$;
  $\texttt{Put}(i_0 \| \cdots \| i_{j-1}, n)$;

- For each $i_0 \| \cdots \| i_d \in \mathcal{I}_0$, $n_{i_0 \| \cdots \| i_{d-1}}.t_{i_d}$ is the largest t such that $\texttt{PreWrite}(i_0 \cdots i_d, t)$ was executed.

Just like for an ORAM, we will think of a freshness guarantor as an interactive machine. Again, it takes READs and WRITEs as inputs and produces READs and WRITEs to be passed to a subroutine.

6.3.1 Freshness Guarantor Inductive Properties

Let FG be a freshness guarantor on S words with timestamps bounded by T. Define $\phi_{t*,f}$ as a predicate on $(q, w) \in FG.Q_I \times FG.W$:

$$q.t \leq t^* \implies w = f(q.t)$$

In words, this says that the first t^* inputs to FG are given by $f(1), \ldots, f(t^*)$. Let S_0 denote the initial memory, and let $q_{FG}^{(0)}$ denote the initial state of FG.

Claim 6. *Suppose that $t \leq t^*$ and $f(t)$ is an access of location i, and before time t, location i was most recently written at time t'.*

If $f(t)$ is a WRITE*, then $\phi_{t^*,f}$* $\mathsf{poly}(\log T, \log S)$*-inductively implies the following predicate on $q \in FG.Q_V$:*

$$q.t > t \text{ and } f(q.t) = \mathsf{READ}(i) \implies \nu(q) \geq t \tag{1}$$

If $f(t)$ is a READ*, then $\phi_{t*,f}$* $\mathsf{poly}(\log T, \log S)$*-inductively implies the following predicates on $q \in FG.Q_V$:*

$$q.t = t \implies \nu(q) = t' \tag{2}$$

Proof. We separately prove the inductiveness of each predicate. Recall the notation where i is written as a binary string $i_0 \| \cdots \| i_d$, and p_j denotes $i_0 \| \cdots \| i_{j-1}$.

Predicate 1 Recall that for each physical access that FG makes, its state has a variable t_{exp} which stores the expected timestamp for the current access. If this doesn't match, then the `PreRead` or `PreWrite` aborts and outputs $\perp$. Recall also that the value $\nu(q)$ comes directly from an access of a FG tree leaf on the previous state, so a property about the FG leaf corresponding to i 0-inductively implies a property about $\nu(q)$.

We claim that if n_{p_j} has timestamp at least t, then $n_{p_j}.t_{i_j}$ is also at least t. In order to make this actually an invariant, we need something a bit stronger. We need to add that if $q_{FG}.i = i$ and $q_{FG}.t > t$, then $q_{FG}.t_{exp} \geq t$. The conjunction of these two properties is in fact an invariant and is decidable by a circuit of size $\mathsf{poly}(|q_{TT}|) = \mathsf{poly}(\log T, \log S)$, and therefore Predicate 1 is $\mathsf{poly}(\log T, \log S)$-inductive.

Predicate 2 We use Predicate 1 to establish a lower bound: $\nu(q) \geq t'$. We must also establish a matching upper bound (and then we will use the fact that conjunctions of inductive predicates are inductive). It is trivial to establish a loose upper bound of $\nu(q) \leq t$, because `PreRead` and `PreWrite` return a prediction smaller than the current time. We know that for all times $t' + 1, \ldots, t - 1$, the leaf for i is not accessed (technically this property is $O(\log T)$-inductively implied by $\phi_{t^*,f}$). Hence the leaf node of the FG tree never predicts that location i's last-written timestamp is between $t' + 1$ and $t - 1$. Therefore $\nu(q) = t'$ is the only remaining possibility. We used invariants whose total size is $\mathsf{poly}(\log T, \log S)$, so Predicate 2 is $\mathsf{poly}(\log T, \log S)$-inductive.

□

6.4 Space-Efficient One-time RAM Garbling

We now define and construct a space-efficient one-time RAM garbling scheme.

6.4.1 Definition

A garbling scheme for a family $\mathcal{M}$ of RAM machines consists of two algorithms, Garble and Eval.

$\mathsf{Garble}(M, x, S_M, 1^\lambda)$ takes as input $M \in \mathcal{M}$ and an input x, as well as the space S_M used by M, and the security parameter 1^λ. It outputs a garbling $\tilde{y}$.

$\mathsf{Eval}(\tilde{M}, \tilde{x})$ takes a garbling $\tilde{y}$ and produces an output value y.

Correctness. A garbling scheme is *correct* if for all $M \in \mathcal{M}$ and all correctly-sized inputs x and all λ,

$$\Pr[\tilde{y} \leftarrow \mathsf{Garble}(M, x, S_M, 1^\lambda), y \leftarrow \mathsf{Eval}(\tilde{y}); y = M(x)] = 1$$

Space Efficiency A garbling scheme is *space efficient* if Garble runs in time $\mathsf{poly}(|M|, S_m, \lambda)$, where S_m is the space usage of M.

One-time Simulation Security. A garbling scheme is *simulation secure* if there is a simulator Sim such that for all $M \in \mathcal{M}$, all correctly-sized inputs x, and all λ:

$$\mathsf{Sim}(M(x), S_M, T_M, 1^\lambda) \approx_C \mathsf{Garble}(M, x, S_M, 1^\lambda)$$

Here, $\approx_C$ denotes computational indistinguishability.

6.4.2 Construction

First, assume that the machine M erases memory before terminating, and assume that M's terminal states contain only the result. As long as M reads its entire input, then this assumption incurs no additional asymptotic overhead on the running time of M, because the size of memory is at most the running time of M. Of course, there is also no additional space overhead.

Next, we introduce interactive RAM machine of our ORAM and freshness guarantor data structures, and call them $ORAM_K$ and FG. K denotes a puncturable pseudorandom function key, since `OAccess` needs randomness but interactive RAM machines are deterministic. The freshness guarantor's `PreRead` and `PreWrite` are deterministic, so FG does not require a pseudorandom function key.

We now describe Garble in Algorithm 18. Garble is not constrained to be deterministic, so it makes sense for it to randomly sample parameters using $\mathsf{Gen}_{\mathsf{PRF}}$, GenEK, GenDK. Garble then runs $\texttt{Init}_{ORAM}$ and $\texttt{Init}_{FG}$ to get an interactive machine $ORAM$, an interactive machine FG, and initial memory contents $\tilde{D}_0$. Garble then obfuscates the composition of M, $ORAM$, and FG to produce $\tilde{M}$. It also outputs the encapsulation of a starting state $\tilde{q}$ and a word-by-word encapsulated initial memory $\tilde{x}$. Eval is described in Algorithm 19.

Algorithm 18: Garble

Input: M, input x, security parameter 1^λ

$K \leftarrow \mathsf{Gen}_{\mathsf{PRF}} PPRF(1^\lambda)$;
$SK \leftarrow \mathsf{Setup}_{ACE}(1^\lambda)$;
$EK \leftarrow \mathsf{GenEK}_{ACE}(SK, \emptyset)$;
$DK \leftarrow \mathsf{GenDK}_{ACE}(SK, \emptyset)$;
$D_0 \leftarrow x$ padded on the right with $\llcorner$ to be S_M words long ;
$(q^{(0)}_{ORAM}, D_0) \leftarrow \texttt{Init}_{ORAM}(D_0, 1^\lambda)$;
$(q^{(0)}_{FG}, D_0) \leftarrow \texttt{Init}_{FG}(D_0)$;
$\tilde{M} \leftarrow \mathsf{i}\mathcal{O}(\texttt{AddACE}(M \circ ORAM_K \circ FG))$;
for $i \in |D_0|$ **do**
  $\tilde{x}[i] := \mathsf{Enc}(EK, D_0[i])$;
$\tilde{q} \leftarrow \mathsf{Enc}(EK, (q^{(0)}_M, q^{(0)}_{ORAM}, q^{(0)}_{FG}))$;
return $(\tilde{M}, \tilde{q}, \tilde{x})$;

Algorithm 19: Eval

Input: $(\tilde{M}, \tilde{q}, \tilde{x})$. We treat $\tilde{x}$ as a mutable array of ciphertexts
Output: y

$p := \tilde{M}(\tilde{q}, \perp)$;
while *p is not a return value* **do**
  Parse p as (c_q, a) where c_q is an encapsulated state and a is an access;
  if *a is of the form* (READ, i) **then**
    $p := \tilde{M}(c_q, \tilde{x}[i])$;
  else
    Parse a as (WRITE, i, c_v);
    $\tilde{x}[i] := c_v$;
    $p := \tilde{M}(c_q, \perp)$;
return p;

6.5 Proof of Security

Claim 7. (Garble, Eval) *satisfies correctness and space efficiency.*

Proof. This is easy to see directly. □

The more difficult property to show is the one-time simulation security. For this we must show that $\mathsf{Garble}(M, x)$ and $\mathsf{Sim}(M(x))$ produce indistinguishable distributions of $\tilde{M}, \tilde{x}$. At a high-level, our hybrids progressively make $\tilde{M}$ do more dummy steps and fewer computation steps, while $\tilde{x}$ encapsulates a hard-coded state from progressively higher times. The final state is a function only of $M(x)$.

There are a couple of difficulties in achieving this compared to the iterated functions case. One is that a transition function for a RAM machine inherently takes two different inputs - its state and the contents of memory it just accessed. This means that a malicious evaluator could supply inputs that are individually valid, but are combined in the wrong way. For example, a malicious evaluator could supply the memory contents at location i' when the machine requested location i. More deviously, the evaluator could supply memory contents from location i which are not the most recently written one. To solve this problem, we use a consistency checking transformation which ensures that each state will only be accepted with a single corresponding input word.

Another difficulty is that our circuit must output the locations it wishes to access in the clear, which could reveal something about the underlying machine. An ORAM would solve this problem if we had a black-box CPU, but the security of an ORAM typically relies on the ability to generate secret randomness and have secret state. Indistinguishability obfuscation guarantees neither of these things. We introduce a stronger notion of security for ORAMs, and show that existing constructions satisfy this property. This stronger notion of security reduces the requirements on the secrecy of randomness and the secrecy of state in a sort of "worst-case forward security" way. Once we have this security notion, puncturable pseudorandom functions inside of $\mathsf{i}\mathcal{O}$ provide sufficient randomness-hiding.

Theorem 6. $(\mathsf{Garble}, \mathsf{Eval})$ *satisfies one-time simulation security.*

Proof. To prove one-time simulation security of our garbling scheme, we use a hybrid argument to show that the real distribution $\tilde{y} \leftarrow \mathsf{Garble}(M, x, 1^\lambda)$ is indistinguishable from a simulated $\tilde{y} \leftarrow \mathsf{Sim}(M(x), 1^\lambda)$ by presenting a sequence of hybrid distributions $H_0, \ldots, H_{T_M}$, where H_0 is the same as the real distribution, H_{T_M} is the same as the simulated distribution, and H_i is indistinguishable from H_{i+1}.

In H_i, $\tilde{y}$ is the output of Garble_i, described in Algorithm 20. Note that in the final hybrid, $\tilde{y}$ depends only on $M(x)$, because of two assumptions about the structure of M which we can make without loss of generality:

- M erases its memory in sequential order at the end of its execution
- The end state of M $\left(q_M^{(T_M)}\right)$ is a function only of $M(x)$

It is therefore easy to define Sim in terms of Garble_{T_M}: Sim produces the same output distribution as Garble_{T_M}, but takes as input $M(x)$ instead of (M, x).

In order to show that $H_i = \mathsf{Garble}_i(M, x)$ is indistinguishable from $H_{i+1} = \mathsf{Garble}_{i+1}(M, x)$, we give two intermediate hybrids $H_{i,1}$ and $H_{i,2}$ such that

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

In Hybrid $H_{i,1}$, $\tilde{q}$ and $\tilde{x}$ are the same as in H_i, but $\tilde{M}$ is defined differently as $\mathsf{i}\mathcal{O}(\mathtt{AddACE}(N_{i,1} \circ FG))$, where $N_{i,1}$ is described in Algorithm 22. Essentially $N_{i,1}$ hard-codes its behavior for the $i+1$th access of M. This hard-coded behavior consists of reading $\iota_1, \ldots, \iota_\eta$, ($\eta$ is the overhead of the ORAM), checking that the responses are $(w_1, \ldots, w_\eta)$ (outputting $\perp$ otherwise), and then writing back $(w'_1, \ldots, w'_\eta)$ to the same locations. Here ι_l, w_l, and w'_l are all hard-coded constants.

In Hybrid $H_{i,2}$, we change each of $\tilde{M}$, $\tilde{q}$, and $\tilde{x}$ compared to their values in $H_{i,1}$.

- $\tilde{M}$ is $\mathsf{i}\mathcal{O}(\mathtt{AddACE}(N_{i,2} \circ FG))$, where $N_{i,2}$ is described in Algorithm 23. The code for $N_{i,2}$ still has $\iota_1, \ldots, \iota_\eta$ hard-coded, but instead of having hard-coded input and output words and states, $N_{i,2}$ just writes back the same value, just as a dummy access would do.
- $\tilde{x}$ is defined so that for each $l \in \{1, \ldots, \eta\}$, $\tilde{x}[\iota_l]$ is an encapsulation of $(0, \iota_l, w'_l)$ instead of an encapsulation of $(0, \iota_l, w_l)$.

Algorithm 20: Garble_i

Input: RAM Program M, input x, security parameter 1^λ

Let $D_M^{(0)}$ denote x padded on the right with $\sqcup$ until it is S_M words long;
Let $D_M^{(i)}$ denote the memory contents if M is executed for i steps starting on D_0;
Let $q_M^{(i)}$ denote the state of M after running for i steps starting on D_0;
Let $a_1, \ldots, a_T$ denote the underlying access pattern induced by M with starting memory D_0;
Let $s_{i,t}$ denote the largest s with $s \leq t$ such that a_s accesses location i, or 0 if there is no such access;
Let $\hat{a}_{i,t}$ denote $(s_{i,t}, D_M^{(s_{i,t})}[i])$;
$K \leftarrow \mathsf{Gen}_{\mathsf{PRF}}(1^\lambda)$;
Let $I_j = \mathtt{OSample}(j, \mathsf{Eval}_{\mathsf{PRF}}(K, j))$;
$(q_{ORAM}^{(i)}, D_{ORAM}^{(i)}) \leftarrow \mathsf{Sim}_{ORAM}(\hat{a}_{1,i}, \ldots, \hat{a}_{s,i}, I_1, \ldots, I_i, 1^\lambda)$;
$(q_{FG}^{(0)}, D_{FG}^{(0)}) \leftarrow \mathtt{Init}_{FG}(D_{ORAM}^{(i)})$;
$SK \leftarrow \mathsf{Setup}_{ACE}(1^\lambda)$;
$EK \leftarrow \mathsf{GenEK}_{ACE}(SK, \emptyset)$;
$DK \leftarrow \mathsf{GenDK}_{ACE}(SK, \emptyset)$;
$\tilde{M} \leftarrow i\mathcal{O}(\mathtt{AddACE}(N_i \circ FG))$ where N_i in Algorithm 21 has $ORAM_K$, K, and M as constants;
for $j \in \{1, \ldots, |D_{FG}|\}$ **do**
    $\tilde{x}[j] := \mathsf{Enc}(EK, D_{FG}^{(0)}[j]))$;
$\tilde{q} \leftarrow \mathsf{Enc}(EK, (q_M^{(i)}[t \mapsto 0], q_{ORAM}^{(i)}[t \mapsto 0], q_{FG}^{(0)}))$;
return $(\tilde{M}, \tilde{q}, \tilde{x})$;

Algorithm 21: δ_W for Interactive Machine N_i

Data: M, $ORAM_K$, K
Input: State q, word w
Output: State q'

Parse q as (q_M, q_{ORAM});
if $q.t < (i, 0)$ **then**
    $q' := q[t \mapsto t + 1]$;
    $\iota := \mathtt{OSample}(PRF_K(q_M.t))\,[\lfloor q_{ORAM}.t/2 \rfloor]$;
    if $q_{ORAM}.t$ *is even* **then return** $(q', \mathsf{READ}(\iota))$;
    else return $(q', \mathsf{WRITE}(\iota, w))$;
else return $(M \circ ORAM).\delta_W(q, w)$;

- $\tilde{q}$ is also changed to be $q_{2\eta}[t \mapsto 0]$ instead of $q_0[t \mapsto 0]$.

Algorithm 22: δ_W for Interactive Machine $N_{i,1}$

Data: M, $ORAM_K$, K, and locations $\iota_0, \ldots, \iota_{\eta-1}$, input words $w_0, \ldots, w_{\eta-1}$, output words $w'_0, \ldots, w'_{\eta-1}$, states $q_0, \ldots, q_{2\eta}$ which correspond to $q_M.t = i$ if running N_i honestly
Input: State q, word w
Output: State q'

if $q = q_{2k}$ *for* $k \in \{0, \ldots, \eta-1\}$ **then** **return** $(q_{2k+1}, \mathsf{READ}(\iota_k))$;
else if $q = q_{2k+1}$ *for* $k \in \{0, \ldots, \eta-1\}$ **then**
  assert $w = w_k\}$;
  return $(q_{2k+2}, \mathsf{WRITE}(\iota_k, w'_k))$;
Parse q as (q_M, q_{ORAM});
if $q.t < (i, 0)$ **then**
  $q' := q[t \mapsto t+1]$;
  $\iota := \mathtt{OSample}(PRF_K(q_M.t))\,[\lfloor q_{ORAM}.t/2 \rfloor]$;
  if $q_{ORAM}.t$ *is even* **then return** $(q', \mathsf{READ}(\iota))$;
  else return $(q', \mathsf{WRITE}(\iota, w))$;
else if $q.t \geq (i+1, 0)$ **then**
  return $(M \circ ORAM).\delta_W(q, w)$;

Algorithm 23: δ_W for Interactive Machine $N_{i,2}$

Data: M, $ORAM_K$, K, locations $\iota_1, \ldots, \iota_\eta$
Input: State q, word w
Output: State q'

Parse q as (q_M, q_{ORAM});
$q' := q[t \mapsto t+1]$;
if $q.t < (i, 0)$ **then**
  $\iota := \mathtt{OSample}(PRF_K(q_M.t))\,[\lfloor q_{ORAM}.t/2 \rfloor]$;
  if t_2 *is even* **then return** $(q', \mathsf{READ}(\iota))$;
  else return $(q', \mathsf{WRITE}(\iota, w))$;
else if $q.t = (i, j)$ *for some* j **then**
  if $q_{ORAM}.t$ *is even* **then return** $(q', \mathsf{READ}(\iota_{q_{ORAM}.t/2}))$;
  else return $(q', \mathsf{WRITE}(\iota_{\lfloor q_{ORAM}.t/2 \rfloor}, w))$;
else if $q.t \geq (i+1, 0)$ **then**
  return $(M \circ ORAM).\delta_W(q, w)$;

Timestamps Our indistinguishably arguments will use Theorems 3, 4, and 5. As such, we need to define a graded poset structure on our message space M (which is $(Q_M \times Q_{ORAM}) \sqcup W$). Towards this goal, we assign a timestamp to every element of this set. Recall that Q_M and Q_{ORAM} both have a timestamp component, which ranges from 0 to T for Q_M and 0 to $\eta - 1$ for Q_{ORAM} (recall η is the multiplicative time-overhead of our ORAM). Given $q = (q_M, q_{ORAM})$, we will say that q's timestamp $q.t$ is $\eta q_M.t + q_{ORAM}.t$. Elements w of W are also tagged with a timestamp $w.t$ which is equal to that of the state at which they were written. Sometimes we will write a timestamp t as (t_1, t_2), where $t = \eta t_1 + t_2$, and $0 \leq t_2 < \eta$. (t_1, t_2) is uniquely defined by division with remainder by η.

One final note about our graded poset is that it does not suffice to let the rank ρ of an element $m \in M$ be given by $m.t$. This would allow some $q \in Q_M \times Q_{ORAM}$ to have the same rank as some $w \in W$. Instead we will disambiguate by saying (somewhat arbitrarily) that if $w.t = q.t$, then $w \succ q$. In terms of ranks, this can be achieved by defining

$$\rho(m) = \begin{cases} 2(m.t) + 1 & \text{if } m \in W \\ 2(m.t) & \text{if } m \in Q_M \times Q_{ORAM} \end{cases}$$

Indistinguishability of H_i and $H_{i,1}$ We now show that H_i is indistinguishable from $H_{i,1}$ via intermediate hybrids: $H_i \approx H_{i,0,0} \approx \cdots \approx H_{i,0,\eta} \approx H_{i,1}$. In $H_{i,0,j}$, $\tilde{q}$ and $\tilde{x}$ are the same as they are in H_i, but $\tilde{M}$ is $\mathsf{i}\mathcal{O}(\mathtt{AddACE}(N_{i,0,j} \circ FG))$, where $N_{i,0,j}$ hard-codes its first j accesses (to locations $\iota_1, \ldots, \iota_j$) and is described in Algorithm 24.

We will show indistinguishability of $H_{i,0,j}$ from $H_{i,0,j+1}$ by using Theorem 3. Let $S_0 \in (Q \sqcup W)^*$ be the sequence of plaintexts which are encapsulated by $(\tilde{q}, \tilde{x})$. We give a predicate ϕ_j which is $\mathsf{poly}(\log S, \log T)$-inductive with respect to both $N_{i,0,j} \circ FG$ and $N_{i,0,j+1} \circ FG$ given S_0 such that $N_{i,0,j} \circ FG$ is functionally the same as $N_{i,0,j+1} \circ TT$ when restricted to inputs satisfying $\bar{\phi_j}$. Once we show this, Theorem 3 implies that

$$(\mathsf{i}\mathcal{O}(\mathtt{AddACE}(N_{i,0,j})), \tilde{q}, \tilde{x}) \approx (\mathsf{i}\mathcal{O}(\mathtt{AddACE}(N_{i,0,j+1})), \tilde{q}, \tilde{x})$$

which is a restatement of the claim that $H_{i,0,j} \approx H_{i,0,j+1}$.

Algorithm 24: δ_W for Interactive Machine $N_{i,0,j}$

Data: M, $ORAM_K$, K, and locations $\iota_0, \ldots, \iota_{j-1}$, input words $w_0, \ldots, w_{j-1}$, output words $w'_0, \ldots, w'_{j-1}$, states $q_0, \ldots, q_{2j}$.
Input: State q, word w
Output: State q'

```
if q = q_{2k} for k ∈ {0, ..., j − 1} then return (q_{2k+1}, READ(ι_k)) ;
else if q = q_{2k+1} for k ∈ {0, ..., j − 1} then
    assert w = w_k ;
    return (q_{2k+2}, WRITE(ι_k, w'_k));
Parse q as (q_M, q_ORAM);
if q.t < (i, 0) then
    q' := q[t ↦ t + 1];
    ι := OSample(PRF_K(q_M.t)) [⌊q_ORAM.t/2⌋];
    if t_2 is even then return (q', READ(ι));
    else return (q', WRITE(ι, w)) ;
else
    return (M ∘ ORAM).δ_W(q, w);
```

We define ϕ_j as the conjunction of several predicates, which are themselves proven to be $\mathsf{poly}(\log S, \log T)$-inductive in Claims 8, 9, and 10.

Let q_j denote the unique state (q_M, q_{ORAM}) of $N_i.j$ such that $q_M.t = i$ and $q_{ORAM}.t = j$, and q_j occurs in an honest execution of N_i. Let w_j denote the word (if non-$\perp$) that q_j is paired with as an input.

Claim 8. *The following predicate on $w \in W$ is $\mathsf{poly}(\log S, \log T)$-inductive with respect to both $N_{i,0,j} \circ FG$ and $N_{i,0,j+1} \circ FG$.*

$$w.t = t_j \wedge w.i = \iota_j \implies w = w_j \tag{3}$$

Proof. If $t_j \geq (i, 0)$ (that is, location ι_j was most-recently written by a hard-coded access), then the predicate is immediately $\mathsf{poly}(\log S, \log T)$-inductive (as it is itself an invariant). This is because regardless of input, only one word with timestamp t_j is ever output, and that word is hard-coded as w_j.

If $t_j < (i, 0)$, an invariant which implies this property is

$$w.t < (i, 0) \wedge w.i = \iota_j \implies w.x = x_{\iota_j}$$

This holds for S_0 because $S_0 \cap W$ is exactly the values which are encapsulated in $\tilde{x}$. Of these values, only one takes the form $(\star, \iota_j, \star)$, and that word is $(0, \iota_j, x_{\iota_j})$.

This is invariant because if $t < (i, 0)$, a word with timestamp t of the form (t, ι, x) is output only if the a word of the form (t', ι, x) with $t' < t$ is input. Finally, this implies the desired predicate (3) because a word (just a tuple) is fully determined by its values of t, i, and x. □

Claim 9. *The following predicate on $(q_M, q_{ORAM}, q_{FG}) \in Q$ is $\mathsf{poly}(\log S, \log T)$-inductive with respect to both $N_{i,0,j} \circ FG$ and $N_{i,0,j+1} \circ FG$.*

$$q_M.t = i \wedge q_{ORAM}.t = j \wedge q_{FG}.t = \log S \implies q_{FG}.t_{exp} = t_j \wedge q_{FG}.i = \iota_j \tag{4}$$

Recall that $q_{FG}.t = \log S$ only when FG is accessing underlying memory, and then $q_{FG}.t_{exp}$ and $q_{TT}.i$ are the expected timestamp and location tags on the next word read from memory and passed to N_i.

Proof. We will prove the claim by applying Claim 5 and Claim 6. Claim 6 tells us that if the first t^* inputs to FG are given by a function f, then for any $t \in \{1, \ldots, t^*\}$, we have an inductive property that FG will expect the correct timestamp and location on the t-th access. FG is a sub-circuit in $N_i.j \circ TT$, so we first show an inductive property claiming that indeed the first t^* inputs given to TT in $N_i.j \circ TT$ are given by f. Claim 5 then allows us to put these two properties together. A more formal proof follows.

First, Q_{FG} is a black-box component of $Q_M \times Q_{ORAM} \times Q_{FG}$ in $N_i.j \circ FG$, where the projection π_Q is defined as $\pi_Q(q_M, q_{ORAM}, q_{TT}) = qTT$.

Recall that Claim 6 says that $\phi_{t^*,f}$ $\mathsf{poly}(\log S, \log T)$-inductively implies ψ_{t^*} with respect to FG, where $\phi_{t^*,f}$ is a predicate on $(q, w) \in FG.Q_I \times FG.W$:

$$q.t \leq t^* \implies w = f(q.t)$$

and ψ_{t^*} is a predicate on Q_{FG} which says that FG expects the correct timestamp on the t^*-th access.

Now we claim χ is $\mathsf{poly}(\log S, \log T)$-inductive, where

$$\chi(x) = \text{all inputs to } D \text{ in } C \text{ on } x \text{ satisfy } \phi_{2\eta i+j,f}$$

and

$$f(2\eta i' + j') = \begin{cases} \mathtt{OSample}(\mathsf{Eval}_{\mathsf{PRF}}(K, i'))[j'] & \text{if } i' < i \\ \iota_j & \text{otherwise} \end{cases}$$

because χ is its own invariant, and its size is $\mathsf{poly}(\log S, \log T)$ because η is $\mathsf{poly}(\log S, \log T)$.

The claim now follows from Claim 5. □

Claim 10. *The following predicate on $(q_M, q_{ORAM}, q_{FG}) \in Q$ is $\mathsf{poly}(\log S, \log T)$-inductive with respect to both $N_{i,0,j} \circ FG$ and $N_{i,0,j+1} \circ FG$.*

$$q_M.t = i \wedge q_{ORAM}.t = j \implies (q_M, q_{ORAM}) = q_j \tag{5}$$

Proof. We prove that (3) and (4) inductively imply (5) with respect to both $N_i.j \circ FG$ and $N_i.(j+1) \circ FG$. This is almost trivial for the latter case. (5) is an invariant of $N_{i,0,j+1} \circ FG$ because N_i only ever outputs one state satisfying $(q_M.t = i \wedge q_{ORAM}.t = j)$, and this state is hard-coded as q_j. Composition with TT does not cause any different states of N_i to be output.

For $N_{i,0,j} \circ FG$, first suppose $j > 0$. Then we notice that a state with timestamp (i, j) is only output when the input state has timestamp $(i, j-1)$ But such a state can only be q_{j-1}. By (3) and (4), w_{j-1} is the only word w for which (q_{j-1}, w) as input will not induce $\perp$ as output. We know the output state will then be q_j by definition.

If $j = 0$, parse q_0 as $(q_M^{(0)}, q_{ORAM}^{(0)})$. Then we use a different invariant to show that there is only one possible state with time timestamp $(i, 0)$:

$$(q_M.t, q_{ORAM}.t) \leq (i, 0) \implies q_M = q_M^{(0)}[t \mapsto q_M.t] \wedge q_{ORAM} = q_{ORAM}^{(0)}[t \mapsto q_{ORAM}.t])$$

This is an invariant because states with timestamp at most $(i, 0)$ are produced by incrementing the timestamp of the input state. □

This says that the only difference between q_0 and all states preceding it is the timestamps. The conjunction of (3), (4), and (5) imply that $N_i \circ FG$ and $N_i' \circ FG$ are functionally equivalent. So Theorem 3 shows that H_i and $H_i.A$ are indistinguishable.

Indistinguishability of $H_{i,1}$ and $H_{i,2}$ We now show that $H_{i,1}$ is indistinguishable from $H_{i,2}$. Here we will repeatedly apply the Diamond Theorem (5) to progress through hybrids:

$$H_{i,1} \approx H_{i,1,0} \approx \cdots \approx H_{i,1,\eta i} \approx H_{i,2}$$

$H_{i,1,j}$ is defined such that $\tilde{q}$ and $\tilde{x}$ are the same as in $H_{i,1}$, but $\tilde{M}$ is $\mathsf{i}\mathcal{O}(\texttt{AddACE}(N_{i,1,j} \circ FG))$, where $N_{i,1,j}$ is given in Algorithm 25. $N_{i,1,j}$ differs from $N_{i,1}$ in that it has a different set of hard-coded constants. In particular, we define (q_ℓ, w_ℓ) as the inputs which induce the first access to location ι_ℓ that (in an honest execution of $N_{i,1}$) happens at time at least $2(\eta i - j)$. w_ℓ' is defined the same way in $H_{i,1,j}$ as it is in $H_{i,1}$. It is easy to verify that $H_{i,1,0} = H_{i,1}$ by $\mathsf{i}\mathcal{O}$.

It is also easy to show that $H_{i,1,2\eta i} \approx H_{i,2}$ by using Theorem 4, which we now do.

Claim 11. $H_{i,1,2\eta i} \approx H_{i,2}$

Proof. In Hybrid $H_{i,1,2\eta i}$, the hard-coded inputs are all values which appear in S_0. Theorem 3 implies that we might as well let $N_{i,1,2\eta i}$ do the same thing on S_0' as it does on S_0, and then we can use Theorem 4 η times to allow us to replace S_0 by S_0', while also changing it in $\tilde{M}$. Finally, we can use Theorem 3 to change $\tilde{M}$ so that our hybrid is identical to $H_{i,2}$. □

What remains is to show that $H_{i,1,j} \approx H_{i,1,j+1}$, and we show this by using our Diamond Theorem (5). Note that $N_{i,1,j}$ and $N_{i,1,j+1}$ compute different states on only one value of ℓ: call this value ℓ_j. Call the changed values in $N_{i,1,j+1}$ $\tilde{q}_{\iota_j}$ and $\tilde{w}_{\iota_j}$. In order to apply the Diamond Theorem to change q_{ι_j} into $\tilde{q}_{\iota_j}$, we need to show a property ϕ which is $\mathsf{poly}(\log S, \log T)$-inductive with respect to both $H_{i,1,j}$ and $H_{i,1,j+1}$ such that for all x satisfying $\bar{\phi}$:

1. There is only one input x^* which causes either $N_{i,1,j}$ to output q_{ι_j} or causes $N_{i,1,j+1}$ to output $\tilde{q}_{\iota_j}$, and this x^* causes both to happen. For this to be true, it suffices for ϕ needs to imply that there is only one possible state with the (shared) preceding timestamp, and only one word which will be accepted together with this state. We already know that this property is $\mathsf{poly}(\log S, \log T)$-inductive.

2. $N_{i,1,j}$ never outputs $\tilde{q}_{\iota_j}$ and $N_{i,1,j+1}$ never outputs q_{ι_j}. The same inductive property as above implies this.

3. $N_{i,1,j}(q_{\iota_j}, \cdot)$ is equivalent to $N_{i,1,j+1}(\tilde{q}_{\iota_j}, \cdot)$. It suffices for ϕ to imply that $H_{i,1,j}(q_{\iota_j}, w_{\iota_j}) = H_{i,1,j+1}(\tilde{q}_{\iota_j}, \tilde{w}_{\iota_j})$ and for all $w \neq w_{\iota_j}$, $H_{i,1,j}(q_{\iota_j}, w) = \perp$ and for all $w \neq \tilde{w}_{\iota_j}$, $H_{i,1,j+1}(\tilde{q}_{\iota_j}, w) = \perp$. This is again easy to show $\mathsf{poly}(\log S, \log T)$-inductively (the property that words at a specific location after a bunch of dummy accesses retains its value is invariant).

Algorithm 25: δ_W for Interactive Machine $N_{i,1,j}$

Data: M, $ORAM_K$, K, and locations $\iota_0, \ldots, \iota_{\eta-1}$, input words $w_0, \ldots, w_{\eta-1}$, output words $w'_0, \ldots, w'_{\eta-1}$, states $q_0, \ldots, q_{2\eta-1}$, q^*_{in}, q^*_{out}. Here $q_{2\ell}$ denotes the first state (in an honest execution of $N_{i,1}$) with timestamp at least $2\eta i - j$ which also outputs a read of location ι_ℓ

Input: State q, word w

Output: State q'

```
if q = q*_in then q' := q*_out ;
else q' := q[t ↦ t + 1] ;
if q = q_{2k} with k ∈ {0, ..., η − 1} then return (q', READ(ι_k)) ;
else if q = q_{2k+1} with k ∈ {0, ..., η − 1} then
    assert w = w_k ;
    return (q', WRITE(ι_k, w'_k));
else if q.t ≥ 2(ηi − j) then
    q' := q[t ↦ t + 1];
    if q.t = (i, 2k) with k ∈ {0, ..., η − 1} then return (q', READ(ι_k)) ;
    else if q.t = (i, 2k + 1) with k ∈ {0, ..., η − 1} then return (q', WRITE(ι_k, w)) ;
    else return (M ∘ ORAM).δ_W(q, w) ;
else return N_{i,1}(q, w) ;
```

4. For $q \notin \{q_{\iota_j}, \tilde{q}_{\iota_j}\}$, $H_{i,1,j}(q, \cdot) \equiv H_{i,1,j+1}(q, \cdot)$. This is clear because $H_{i,1,j}$ and $H_{i,1,j+1}$ are *defined* to only differ when $q \in \{q_{\iota_j}, \tilde{q}_{\iota_j}\}$.

Changing w_j to $\tilde{w}_j$ is similar, but showing (3) uses the second property of the freshness guarantor. In particular, in order to show that $H_{i,1,j}(\cdot, w_j)$ is the same as $H_{i,1,j+1}(\cdot, \tilde{w}_j)$, we will show that there is only one state which will be accepted with w_j or $\tilde{w}_j$, which reduces our task to looking at the function on specific inputs. This is because the freshness guarantor tells us that there is only one value for $q.t$ (call this t^*) at which a word whose timestamp and location are $w_j.t$ and $w_j.i$ respectively will be accepted. Another inductive property says that there is only state with timestamp t^*, since all the accesses before t^* only increment the timestamp of q.

Applying ORAM security property We now show that $H_{i,1}$ is indistinguishable from H_{i+1}. This follows directly from the simulability of our ORAM. The difference between $H_{i,1}$ and H_{i+1} is in the starting ORAM memory $D^{(i+1)}_{ORAM}$, the starting ORAM state $q^{(i+1)}_{ORAM}$, and the hard-coded locations $I_{i+1} = \iota_1, \ldots, \iota_\eta$. The distributions of these values in $H_{i,1}$ and H_{i+1} are exactly the two distributions which are indistinguishable by the simulability property. Once the hard-coded locations are the same as would be generated by $\mathtt{OSample}$, we can use $i\mathcal{O}$ to replace them with a normal dummy access.

□

6.6 Obfuscation

Finally, we turn our (one-time) succinct RAM garbling scheme into an indistinguishability obfuscation scheme for RAM machines, proving our main theorem:

Theorem 7. *Let $S = S(n)$ be any polynomial space bound function. Assuming sub-exponentially secure indistinguishability obfuscation for circuits, there is an indistinguishability obfuscator for RAM machines that run in space bound S, where*

- *The size of the obfuscated RAM machine M is $S \cdot \mathsf{poly}(\lambda, n) + \mathsf{poly}(|M|, \lambda, n)$; and*

- *The running time of the obfuscated RAM machine M is $S \cdot \mathsf{poly}(\lambda, n) + T \cdot \mathsf{poly}(|M|, \log S, \log T, \lambda, n)$, where T is the running time bound of the machine M.*

Proof. The idea for the RAM obfuscation is simple. Consider a (deterministic) circuit $C_{K,M}$ where M is a RAM machine and K is a key for a puncturable PRF, that takes as input x, and does the following.

- Compute $\rho \leftarrow \mathsf{PRF}_K(M, x, 1^{|S|})$; and
- Outputs $(\tilde{M}, \tilde{q}, \tilde{x}) \leftarrow \mathsf{Garble}(M, x, 1^S; \rho)$.

Since Garble is a circuit of size $\mathsf{poly}(|M|, S, \lambda)$, we know that $C_{K,M}$ is also of size $\mathsf{poly}(|M|, S, \lambda)$. The RAM obfuscation $\tilde{C}_M$ is simply an IO obfuscation of the circuit $C_{K,M}$ for a uniformly random $K \leftarrow \{0,1\}^\lambda$.

This gives a succinct way to obfuscate RAM programs. The evaluation procedure $\mathsf{Eval}_{Obf}(\tilde{C}_M, x)$ just evalutes $\tilde{C}_M$ with input x to obtain $(\tilde{M}, \tilde{q}, \tilde{x})$. Then it executes $\mathsf{Eval}_{Garble}(\tilde{M}, \tilde{q}, \tilde{x})$ to obtain $M(x)$.

To show indistinguishability of obfuscations of machines M_0 and M_1, we proceed through a sequence of $2^{|x|} + 1$ hybrids $H_0, \ldots, H_{2^{|x|}}$. H_i is the obfuscation of the circuit that

- On input $x \leq i$, the circuit works exactly as C_{K,M_0} would.
- On input $x > i$, the circuit works exactly as C_{K,M_1} would.

We show indistinguishability between hybrids H_i and H_{i+1} by invoking the security of the punctured PRF in a by-now standard way (see, [SW14], for example). Since there are $2^{|x|} + 1 \leq 2^{n+1}$ hybrids, by a standard complexity leveraging argument, we set the size of the obfuscation and the size of the key of the puncturable PRF to be a sufficiently large polynomial in S, and obtain security under subexponential IO for circuits and subexponentially hard OWF. We note that a hybrid argument with exponentially many such hybrids has been used recently in several works on obfuscation, including [PST14, GLSW14]. □

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.

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

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, pages 1–18, 2001.

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

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

[CGH04] Ran Canetti, Oded Goldreich, and Shai Halevi. On the random-oracle methodology as applied to length-restricted signature schemes. In *TCC*, pages 40–57, 2004.

[CLTV13] Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. Probabilistic indistinguishability obfuscation. In *Personal Communication*, 2013.

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

[CPS13] Kai-Min Chung, Rafael Pass, and Karn Seth. Non-black-box simulation from one-way functions and applications to resettable security. In *STOC*, pages 231–240, 2013.

[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.

[GGHR14] Sanjam Garg, Craig Gentry, Shai Halevi, and Mariana Raykova. Two-round secure MPC from indistinguishability obfuscation. In *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, pages 74–94, 2014.

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

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

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *ACM CCS*, pages 513–524, 2012.

[GLSW14] Craig Gentry, Allison B. Lewko, Amit Sahai, and Brent Waters. Indistinguishability obfuscation from the multilinear subgroup elimination assumption. *IACR Cryptology ePrint Archive*, 2014:309, 2014.

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

[KRR13] Yael Tauman Kalai, Ran Raz, and Ron D. Rothblum. Delegation for bounded space. In *STOC*, pages 565–574, 2013.

[PST14] Rafael Pass, Karn Seth, and Sidharth Telang. Indistinguishability obfuscation from semantically-secure multilinear encodings. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, pages 500–517, 2014.

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

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