Succinct Garbled RAM from Indistinguishablity Obfuscation

by

Justin Lee Holmgren

S.B., Massachusetts Institute of Technology (2013)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2015

Author .....
Department of Electrical Engineering and Computer Science
February 1, 2015

Certified by .....
Shafi Goldwasser
RSA Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Professor Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Succinct Garbled RAM from Indistinguishablity Obfuscation

by

Justin Lee Holmgren

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I give the first construction of a succinct garbling scheme for RAM programs. For a program requiring space S and time T to compute, the size of its garbling is $\tilde{O}(S)$ instead of $\mathsf{poly}(T)$. This construction relies on the existence of indistinguishability obfuscation, as well as the existence of injective one-way functions.

As a building block, I introduce and construct a primitive called asymmetrically constrained encryption (ACE). This primitive is an encryption system for which keys can be punctured on succinctly described sets of plaintexts. For programs acting on ACE-encrypted values, I give a natural and general condition for their obfuscations to be indistinguishable, using the fact that the encryption and decryption keys can be separately punctured.

This succinct garbling scheme serves as a drop-in replacement for the ubiquitous garbled circuits of Yao, but with better asymptotic parameters. In some cases, these improved parameters allow qualitatively new applications.

Thesis Supervisor: Shafi Goldwasser
Title: RSA Professor of Electrical Engineering and Computer Science

Acknowledgments

First, I wish to thank Shafi Goldwasser for being a caring and helpful advisor. Thank you for setting me straight when I was lost, and for letting me switch into the magical world of cryptography.

I thank my coauthors Ran Canetti, Abhishek Jain, and Vinod Vaikuntanathan for their helpful collaboration. In particular, I thank Ran for his continued mentorship and for all the free food and I thank Abhishek for initiating a research project with me when I had nothing to prove myself.

I thank Elette Boyle for teaching me about Path ORAMs, without which this thesis would only garble iterated functions.

Finally, I thank my family. Any and all of my success is, ultimately, due to my upbringing.

Contents

Chapter 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+13a] 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), Boyle et al. [BCP14] and Ananth et al. [ABG+13a] show how to transform these obfuscators to act directly on Turing machines.

However, working in either the circuit model or the Turing machine model does not take 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 amounts of memory. When applying obfuscators 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 show how to obfuscate RAM programs directly, without paying the overhead associated with transforming to Turing machines or circuits. The central piece of our construction and the focus of this thesis is an efficient method for *garbling* RAM programs.

Garbled RAM programs. *Garbling* is a way to take a function f and an input x, and encode them as $\tilde{f}, \tilde{x} \leftarrow \mathsf{Garble}(f, x)$ in a way which reveals nothing more than $f(x)$. One obviously desirable property is efficiecy: computing $\tilde{f}$ and $\tilde{x}$ should be easier than just computing $f(x)$. The efficiency of a scheme depends on the representation of f; if f is given as a circuit, then just reading f takes as much time as computing $f(x)$. We give the first construction of an efficient garbling scheme in which functions are represented as RAM programs. As one of several easy applications, we then show how to transform circuit obfuscators into RAM program obfuscators.

Given a RAM program Π, a memory configuration x, and security parameter λ, Garble outputs a RAM program $\tilde{\Pi}$ of size $\mathsf{poly}(|\Pi|, \lambda)$ and an encoded input $\tilde{x}$ of size $O(|x| \cdot \log^{1+\epsilon}(\lambda))$ (for any positive constant ϵ). Running $\tilde{\Pi}$ with $\tilde{x}$ as initial memory gives $\Pi(x)$, and $\tilde{\Pi}$ and $\tilde{x}$ reveal nothing more than $\Pi(\vec{x})$ and the running time (as well as the sizes of Π and $\vec{x}$). More precisely, there is an efficient probabilistic algorithm Sim such that, for any Π and any x, $\mathsf{Garble}(\Pi, x) \approx \mathsf{Sim}(\Pi(x), |x|, |\Pi|, T_{\Pi,x})$, where $T_{\Pi,x}$ is the running time of Π on x.

Our construction uses as building blocks an injective one way function and an indistinguishability obfuscator for circuits.

Applicability. Our garbling scheme for RAM programs can be used in practically any place where garbling schemes for circuits have been used, with commensurate efficiency gains. We remark that our garbling scheme represents RAM program inputs as memory configurations. While this is the most general case, we do incur a loss of efficiency when the input is smaller than the computation's memory usage. However, many computations (particularly in crytography) have low memory requirements (say, linear in the input size), thus making the dependence on space less critical. Below we

give a brief overview of some applications.

Indistinguishability obfuscation for RAM programs. Our main application of our RAM garbling scheme is in converting an *indistinguishability obfuscator* for circuits to one for RAM programs. An indistinguishability obfuscator takes a function description f and outputs a functionally equivalent $\mathcal{O}(f)$. If f_0 and f_1 are functionally equivalent and have equally sized representations, then $\mathcal{O}(f_0)$ and $\mathcal{O}(F_1)$ must be indistinguishable. For RAM programs, there is another prerequisite: the two RAM programs must have the same running time on every input. Indistinguishably obfuscation was introduced as a weak notion of obfuscation to circumvent the impossibility results of Barak et al. [BGI+01], and a candidate indistinguishability obfuscator for *circuits* (which we will denote by $\mathsf{i}\mathcal{O}$) was recently given by Garg et al. [GGH+13a]. Since then, indistinguishability obfuscation for circuits has been shown to have extensive applications [SW14].

Our obfuscator ($\mathsf{i}\mathcal{O}_{RAM}$) has the following parameters. It relies on, as building blocks, subexponentially-hard indistinguishability obfuscators for circuits and subexponentially-hard injective one-way functions.[1] Suppose the input is a RAM program Π taking n-bit inputs and using $S \geq n$ bits of space. $\mathsf{i}\mathcal{O}_{RAM}$ outputs a circuit Π' of size $\mathsf{poly}(S, |\Pi|, \lambda)$, where λ is the security parameter. Evaluating $\Pi(x)$ given only Π' and x takes time $|\Pi'| + T_{\Pi,x} \cdot \mathsf{poly}(n, \lambda)$, where $T_{\Pi,x}$ is the running time of Π on x.

Publicly Verifiable Delegation of Computation: Our garbled RAM scheme yields a simple 2-round publicly verifiable delegation scheme: to delegate the computation of $\Pi(x)$ for a RAM program Π and input x, the delegator first samples a pair (sk, vk) of signature and verification keys of a digital signature scheme. He then garbles a RAM program Π' that on input x, runs Π to obtain $\Pi(x)$, and then outputs $(\Pi(x), Sign(sk, \Pi(x)))$. Finally he sends $\mathsf{Garble}(\Pi', x)$ to the worker and publishes vk. The worker will evaluate the garbling to obtain y, σ. This result is verified by

[1] Subexponential hardness assumptions essentially state that an adversary cannot win a security game with substantially better probability than guessing the secret key (or randomness).

checking that σ is a valid signature of y with respect to vk. Soundness follows immediately from the security of the garbled RAM and unforgeability of the signature scheme.

Functional Encryption Functional encryption is a strengthening of public-key encryption, in which restricted decryption keys SK_f can be issued, associated with a function f. $\mathsf{Dec}(SK_f, \mathsf{Enc}(m))$ yields $f(m)$, and DK_f does not allow the computation of anything else. Gentry et al. [GHRW14] show how a RAM garbling scheme can be used to construct a functional encryption scheme in which functions are represented by RAM programs. With our RAM garbler, this gives the following parameters. The key generation time and size of a decryption key SK_Π for a RAM program Π is $(\mathsf{poly}(|\Pi|) + S_\Pi) \cdot \mathsf{poly}(\lambda, \log T_\Pi, \log S_\Pi)$. Here T_Π is the worst-case run-time of Π and S_Π is the space usage of Π. Decrypting a ciphertext corresponding to a plaintext x using SK_Π takes time $(S_\Pi + T_{\Pi,x}) \cdot \mathsf{poly}(\lambda, \log T_\Pi, \log S_\Pi)$ where $T_{\Pi,x}$ is the run-time of Π on input x.

Reusable Garbled RAM Gentry et al. [GHRW14] also gave a reduction showing that functional encryption for RAM programs can be used to build a reusable garbling scheme for RAM programs. When this is built using our RAM garbler, the resulting reusable garbled RAM scheme is also *succinct*.

Multiparty Computation of RAM Programs. Gentry et al. [GHRW14] observed that any efficient garbled RAM can be used in any standard MPC protocol to obtain a new protocol with improved efficiency. To compute a functionality f on inputs $\vec{x}$, the parties jointly compute $\mathsf{Garble}(\Pi, \vec{x})$, where Π is a RAM program computing f. One party then evaluates the garbling and sends the result to all the other parties. When this protocol is instantiated with our RAM garbler, the evaluator takes time which is $\tilde{O}(S_\Pi + T_{\Pi,\vec{x}})$, where S_Π is space used by Π and $T_{\Pi,\vec{x}}$ is the running time on $\vec{x}$.

1.1 Our techniques

1.1.1 RAM Garbling from Weak Garbling and Oblivious Ram

Our ultimate goal is to fully garble a RAM program Π on an input x, revealing nothing more than the final output $\Pi(x)$. We will build a full RAM garbling scheme out of another primitive, which we call weak garbling, and an oblivious RAM.

Weak Garbling We first construct a somewhat weaker primitive which we call *weak garbling*. Weak garbling is syntactically the same as garbling: it takes a program Π and an input x, and outputs $\tilde{\Pi}, \tilde{x}$. Weak garbling guarantees that $\tilde{\Pi}_0, \tilde{x}_0$ and $\tilde{\Pi}_1, \tilde{x}_1$ are indistinguishable if Π_0 and Π_1 on x_0 and x_1 access exactly the same locations, which are succinctly described by a circuit for the first j steps, and if after these j steps, they have the same internal state and external memory configuration.

We then apply weak garbling to a RAM program which has been transformed to be *oblivious* with special additional properties.

Oblivious RAM An oblivious RAM program Π' on input x' simulates execution of an underlying program Π on an underlying input x, such that the addresses accessed by Π' are independent of the underlying addresses accessed by Π. We require two other natural properties:

1. For all j, the distribution of addresses accessed by Π' on the j^{th} underlying access is independent of the previously accessed addresses *and* the underlying access pattern, and is efficiently sampleable.

2. For any fixed underlying access pattern, one can efficiently sample the distribution of ORAM states (including external memory) that are consistent with a preceding sequence of accessed addresses.

We show that the ORAM of Shi et al. [SCSL11], simplified by Chung and Pass [CP13] has these properties. We denote Π', x' by $\mathsf{AddORAM}(\Pi, x)$.

Construction and Reduction Garble is constructed by composing our weak garbling scheme with such an oblivious RAM. That is,

$$\mathsf{Garble}(\Pi, x) \equiv \mathsf{WkGarble}(\mathsf{AddORAM}(\Pi, x)).$$

We show that $\mathsf{Garble}(\Pi, x) \approx \mathsf{Sim}(\Pi(x))$ by a simple hybrid argument. The i^{th} hybrid is $\mathsf{WkGarble}(\Pi'_i, x'_i)$ for some Π'_i, x'_i. Π'_i is a program which does i dummy steps accessing addresses $I_1, \ldots, I_i$. Each I_j consists of several addresses and is sampled independently according to the distribution of addresses accessed by the ORAM on time j. Π'_i then runs Π through the ORAM until Π halts. x'_i is the memory configuration of Π when executed for i steps on x, encoded to be compatible with ORAM accesses. This memory configuration could be encoded in a number of ways, each resulting from some random ORAM setup, and randomness used for each ORAM access. We pick the encoding at random, conditioned on the ORAM accesses accessing locations $I_1, \ldots, I_i$. Our extra ORAM properties allow us to efficiently sample this distribution.

In order to switch from the i^{th} hybrid to the $i+1^{th}$ hybrid in the proof of security for Garble, we need to indistinguishably change the behavior of Π'_i on the $i+1^{th}$ access. We proceed in three steps.

First, weak garbling allows us to hard-code the locations accessed at the $i+1^{th}$ timestep: if the time is $i+1$, Π'_i accesses a hard-coded list of addresses I_{i+1}, without writing new values to these locations. We simultaneously switch x'_i to an encoding of the memory configuration resulting from executing Π for $i+1$ steps on x. This change is easily seen to satisfy the conditions under which weak garbling guarantees indistinguishability.

Next, we sample I_{i+1} according to $\mathsf{OSample}(i+1)$ independently of the rest of Π'_i, and we sample x'_{i+1} to be consistent with this I_{i+1} (and $I_1, \ldots, I_i$). Our ORAM properties guarantee that changing I_{i+1} and x'_{i+1} in this way is indistinguishable.

Finally, we apply weak garbling again to remove the hard-coding of the locations accessed at time $i+1$.

1.1.2 Weak Garbling from Verifiable Reads

We will weakly garble a RAM program Π with input x by first applying a *verifiable reads* transformation to obtain a different RAM program Π' and memory x'. Then we encrypt each word of x', yielding $\tilde{x}$, and we augment Π''s transition function so that it acts on encrypted state and encrypted memory words. Finally, we $i\mathcal{O}$-obfuscate Π''s transition function to obtain $\tilde{\Pi}$. The weak garbling $\mathsf{WkGarble}(\Pi, x)$ is defined as $\tilde{\Pi}, \tilde{x}$.

Verifiable Reads Suppose one is given an external memory which is *semi-malicious*: on any access, the memory can return any value previously written to memory, not necessarily the most recent value. Then one can simulate a semi-malicious memory up to aborts by storing message authentication codes (MACs) with every value written to a fully malicious memory which is computationally bounded. A natural question is whether one can also simulate an honest external memory. This seems to be a necessary component of a secure RAM garbler, because a distinguisher might "evaluate" the garbled RAM program by answering memory accesses arbitrarily. Indeed, Gentry et al. [GHL+14] give a method for simulating an honest external memory up to aborts, given a semi-mqqalicious external memory.

While this abstraction suffices to prove security of Gentry et al.'s non-succinct garbled RAM construction, our proof relies on specific properties of their construction. We describe this construction further, as well as the properties we need, in Chapter 7.

Asymmetrically Constrained Encryption This construction involves an indistinguishability obfuscated circuit that has encryption and decryption keys hard-coded. We only prove security with a special type of encryption which we call Asymmetrqically Constrained Encryption (ACE). ACE is a secret-key deterministic encryption system in which either an encryption key or a decryption key can be punctured at a *succinctly*-described set - that is, a set for which membership in the set can be decided by a small circuit.

ACE must also satisfy a few security properties. A decryption key punctured on

a set S should be indistinguishable from an unpunctured key. However, using such a punctured decryption key to decrypt should *never* yield a message in S. Finally, given keys punctured at a set S, encryptions of messages in S should be indistinguishable.

Security of Weak Garbling To show security of weak garbling, we will indistinguishably change the weak garbling $\mathsf{WkGarble}(\Pi, x)$ into the obfuscation of a program which only executes "dummy" steps for the first j steps of computation.

Consider the transition function $\delta_{\Pi'}$ for the verifiable reads-transformed RAM program Π'. This transition function takes a state q and a memory word w as input, and produces (among other things) a state q' and a memory word w' as output. Suppose that S_Q and S_W are "invariant" sets of Π': whenever q is in S_Q and w is in S_W then q' is also in S_Q and w' is also in S_W. If S_Q and S_W are also succinctly described, then we show that we can indistinguishably change Π' so that it outputs $\perp$ whenever $q \notin S_Q$ or $w \notin S_W$. The indistinguishability of this change relies crucially on the properties of ACE; we alternately puncture encryption keys using $\mathsf{i}\mathcal{O}$ and then apply ACE's indistinguishability of punctured decryption keys.

We will change $\delta_{\Pi'}$ on the first j steps into a dummy computation's transition function, one timestep at a time. We use the so-called "punctured programming" technique of Sahai and Waters [SW14], for which we need the above property of ACE. The idea of punctured programming is the following steps for changing the behavior of $\delta_{\Pi'}$ at some time j. Hard code the behavior of $\delta_{\Pi'}$ when its inputs are the "correct" (ciphertext) inputs at time j. Similarly, hard-code these ciphertexts as outputs instead of ever actually encrypting the corresponding messages. Then puncture the encryption and decryption keys at all messages corresponding to time j, and use ciphertext indistinguishability to change the hard-coded ciphertexts to encryptions of something else. Finally, unpuncture the encryption and decryption keys and un-hard-code the ciphertexts.

One may then raise an objection. We can't puncture the encryption and decryption keys at all possible messages corresponding to time j, because we have only hard-coded ciphertexts for one pair of messages. To resolve this, we construct invari-

ant sets S_Q and S_W such that when we restrict our attention to S_Q and S_W, $\delta_{\Pi'}$ has the desired properties. Namely, there is only one state q at time j and one corresponding w such that $\delta_{\Pi'}(q, w) \neq \perp$. By first indistinguishably restricting $\delta_{\Pi'}$ to S_Q and S_W, the punctured programming argument can be made to work.

The proof of security for our weak garbling scheme is presented in more detail in Chapter 7.

1.2 Previous Work

Garbling schemes were first introduced by Yao [Yao82] to implement secure two-party computation. Yao's construction allowed n parties holding inputs $x_1, \ldots, x_n$ to evaluate a function $f(x_1, \ldots, x_n)$, while learning nothing about each other's inputs. The complexity of Yao's protocol is proportional to the size of the circuit computing f. Subsequent multiparty computation protocols [GMW87, BOGW88, AL11] have for the most part continued to represent functionalities as circuits. One notable exception is the work of Boyle et al. [BGT13] which focuses on multiparty RAM computations with sublinear running times.

Many other areas of cryptography study computing on encrypted data. For example, there has been a large body of work on fully homomorphic encryption [Gen09, vDGHV09, BV11, BGV12], attribute-based encryption [SW05, GPSW06, GVW13, GGH+13b] and functional encryption [SS10, AGVW13], all of which use the Boolean circuit model of computation. An exception is the work of Waters [Wat12] that constructs an attribute-based encryption scheme for finite state machines, a uniform model of computation.

Barak et al. [BGI+01] showed that a strong notion of program obfuscation known as virtual black-box obfuscation is unconditionally impossible to achieve. One definition of obfuscation they proposed as possibly achievable was indistinguishability obfuscation. In 2013, Garg et al. [GGH+13a] introduced a candidate construction of an indistinguishability obfuscator for circuits. Subsequently, many applications [SW14, CGP14, Wat14, BPR14, GGG+14] of indistinguishability obfuscation for cir-

cuits have been found.

In a recent paper, [GLOS14] show how to transform a RAM machine with *worst*-case running time t into a garbled RAM program whose size and running time are polynomially related to t. Their construction only relies on the existence of one-way functions. In our work we reduce the size of the garbled RAM program to depend only logarithmically on t, at the cost of assuming not only one-way functions, but also indistinguishability obfuscators for circuits. We also achieve input-specific running times instead of worst-case.

[GHRW14] assume the existence of various types of obfuscators, and construct reusable garbled RAM schemes. Our succinct garbled RAM strictly improves upon their construction based on only indistinguishability obfuscation.

1.3 Other Related Work

The work in this thesis was first published in [CHJV14] as joint work with Ran Canetti, Abhishek Jain, and Vinod Vaikuntanathan. Concurrently and independently, Bitansky et al. [BGT14] and Lin and Pass [LP14] gave constructions for garbling and indistinguishability obfuscation of Turing machines, assuming only indistingushability obfuscation and one way functions. While the general thrusts of the three works is similar, the technical approaches taken are very different. Furthermore, the specific results obtained are incomparable. Specifically, they provide a generic mechanism for using indistingushability obfuscation to transform any "locally computable" garbling mechanism (i.e., a garbling mechanism where each portion of the garbled program can be generated "locally", without knowledge of the other portions) into a succinct one. The mechanism is essentially the same as our garbling-to-obfuscation transformation: they obfuscate a circuit that generates a portion of the garbled program, where the randomness for the garbling mechanism is obtained by applying a puncturable PRF to the input. They then apply this simple-but-powerful general mechanism to the Yao garbling scheme for circuits.

These two approaches achieve very similar parameters, but are still incomparable.

Our scheme is slightly more succinct, while their scheme relies on a slightly weaker assumptions. In our scheme a garbled RAM program is of size $\mathsf{poly}(\lambda, \log T, \log S)$ and an encoded input is of size $O(S \cdot \log^{1+\epsilon}(\lambda))$. In their scheme a garbled RAM program is of size $S \cdot \mathsf{poly}(\lambda, \log T, \log S)$ and an encoded input is of size $S \cdot \mathsf{poly}(\lambda)$. In our scheme we need to $\mathsf{i}\mathcal{O}$-obfuscate circuits with depth $O(\log(\lambda))$ and input length $O(\log^{1+\epsilon}(\lambda))$. In Bitansky et al.'s scheme, they $\mathsf{i}\mathcal{O}$-obfuscate circuits with depth $O(\log(\lambda))$ and input length $O(\log(\lambda))$. Our scheme also requires injective one-way functions, while their scheme uses any one-way function.

Another difference between the two approaches, which is perhaps more significant than the difference in parameters, is the approach: while their approach is to apply obfuscation on top of existing garbling mechanisms (which may in of themselves be composed of multiple components), our approach is to try to use the power of obfuscation to the fullest, with few other primitives, and with minimal modification to the structure of the underlying program. Indeed, our resulting garbled program has a very similar structure to the original program and can potentially be run on a standard random access computer with minimal adjustments.

In a follow-on work, Koppula et al. [KLW14] apply techniques somewhat similar to ours to construct a fully succinct garbling scheme for Turing machines - that is, their garblings are of size $\tilde{O}(1) \cdot \mathsf{poly}(\lambda)$, and their runtime is $\tilde{O}(T) \cdot \mathsf{poly}(\lambda)$, where T is the Turing machine runtime. Notably, they do not achieve RAM runtimes for a garbling scheme.

Chapter 2

Background

In this chapter we review some standard cryptographic primitives, as well as the definition of $\mathsf{i}\mathcal{O}$ obfuscation. 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 $\mathsf{negl}(\cdot)$ such that for all λ,

$$\Pr\left[D(1^\lambda, x_b) = b \middle| x_0 \leftarrow A_\lambda, x_1 \leftarrow B_\lambda, b \leftarrow \{0,1\}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

In this case, we say that $\mathcal{A} \approx \mathcal{B}$.

2.1 Injective Non-interactive Bit Commitment

An injective non-interactive bit commitment scheme is a pair of polynomials $n(\cdot)$ and $m(\cdot)$ and an ensemble of efficiently computable injective functions $\mathsf{Commit}_\lambda : \{0,1\} \times \{0,1\}^{n(\lambda)} \to \{0,1\}^{m(\lambda)}$ such that for all polynomial time adversaries $\mathcal{A}$,

$$\Pr\left[\mathcal{A}(1^\lambda, \mathsf{Commit}_\lambda(b; r)) = b \middle| b \leftarrow \{0,1\}, r \leftarrow \{0,1\}^{n(\lambda)}\right] < \frac{1}{2} + \mathsf{negl}(\lambda)$$

We can construct an injective non-interactive commitment scheme given an injective one-way function $f : n'(\lambda) \to m'(\lambda)$, and we give the construction here without proof. Without loss of generality f has a hard-core bit because the Goldreich-Levin

[GL89] transformation of a one-way function into one with an explicit hard-core bit preserves injectivity. Then define $n(\lambda) = n'(\lambda)$, $m(\lambda) = m'(\lambda) + 1$, and

$$\mathsf{Commit}_\lambda(b; r) = f(r) \| (b \oplus h(r))$$

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 $K \leftarrow \mathsf{Gen}_{\mathsf{PRF}}$, and $K\{S\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K, S)$. Then, for all $x \notin S$, $\mathsf{Eval}_{\mathsf{PRF}}(K, x) = \mathsf{Eval}_{\mathsf{PRF}}(K\{S\}, x)$.
- **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 punctured set S.

Remark 2. We will also use statistically injective puncturable PRF families in our constructions. These are families of PRFs for which with high probability over the choice of K, $\mathsf{Eval}_{\mathsf{PRF}}(K, \cdot)$ is injective. Any PRF family mapping $\{0,1\}^n \to \{0,1\}^{2n+\omega(\log \lambda)}$ can be made statistically injective with high probability by XOR-ing with a family of pairwise independent hash functions [SW14].

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+13a].

Definition 2 (Indistinguishability Obfuscation for all circuits)**.** A uniform PPT machine $\mathsf{i}\mathcal{O}$ is called an indistinguishability obfuscator if the following holds:

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

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

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

- **Indistinguishability:** For every $\lambda \in \mathbb{N}$, for all pairs of circuits C_0, C_1, if $C_0(x) = C_1(x)$ for all inputs x and $|C_0| = |C_1|$, then

$$\mathsf{i}\mathcal{O}(1^\lambda, C_0) \approx \mathsf{i}\mathcal{O}(1^\lambda, C_1)$$

2.4 Garbling Schemes

A garbling scheme is an algorithm Garble which takes as inputs a function f, an input x, and the security parameter 1^λ in unary. Garble outputs a garbled function $\tilde{f}$ and a garbled input $\tilde{x}$.

The representation of functions and their inputs will be important when we discuss succinctness. Classical garbling schemes represent f as a circuit and x as a bitstring; more recent ones and the construction of this work represent f as a RAM program. In this work we represent inputs as an initial memory configuration for the RAM program.

Garble must satisfy the following properties:

– **Correctness**: For all functions f and inputs x,

$$\Pr\left[\tilde{f}(\tilde{x}) = f(x) \middle|\ \tilde{f}, \tilde{x} \leftarrow \mathsf{Garble}(f, x, 1^\lambda)\ \right] = 1$$

– **Security**: There is a PPT algorithm Sim such that for all functions f and inputs x,

$$\mathsf{Sim}(f(x), |f|, |x|, 1^\lambda) \approx \mathsf{Garble}(f, x, 1^\lambda)$$

Garbling schemes (including ours) commonly have another property:

– **Separability**: Garble can be decomposed into three algorithms: KeyGen, Garble_{Prog} and Garble_{In}. KeyGen takes the security parameter in unary and gen-

erates a garbling key, which is shared between Garble_{Prog} and Garble_{In}. Conditioned on K, Garble_{Prog} garbles the function independently of Garble_{In} garbling the input.

That is,

$$\mathsf{Garble}(f, x, 1^\lambda) \equiv \mathsf{Garble}_{Prog}(K, f), \mathsf{Garble}_{In}(K, x)$$

when $K \leftarrow \mathsf{KeyGen}(1^\lambda)$.

Remark 3. Separability is a property which can apply to any transformation of a RAM program and corresponding input. The other transformations we use in this work – namely, an Oblivious RAM and a weak garbling scheme – will also satisfy separability. One can easily see that the composition of two separable transformations is itself separable.

The last property of our RAM garbling scheme, and the one most unique to our work, is **succinctness**. This is nothing more than requiring that Garble_{Prog} and Garble_{In} both run in polynomial time, *when functions are represented as RAM programs.*

Chapter 3

Techniques Overview

3.1 Asymmetrically Constrained Encapsulation

One of the contributions of this work is the definition of a primitive that we call Asymmetrically Constrained Encryption (ACE), which greatly enhances our ability to show indistinguishability of obfuscated programs. Informally, this is a *deterministic* and *secret-key* encryption scheme which guarantees both confidentiality and authenticity. Crucially, in addition to this, the scheme allows to constrain both the encryption key EK and the decryption key DK. That is, for a set S, one can compute a constrained encryption key $EK\{S\}$ that can encrypt all messages outside of S, and a constrained decryption key $DK\{S\}$ that can decrypt all ciphertexts that decrypt to messages outside S.

Using ACE. Our constructions are $\mathsf{i}\mathcal{O}$-obfuscations of circuits that decrypt their inputs (which are ciphertexts) and encrypt their outputs. We would like to argue that the $\mathsf{i}\mathcal{O}$ obfuscation of two circuits C_0 and C_1 are indistinguishable if they differ only on a set of "bad inputs" S which the adversary cannot encrypt. But $\mathsf{i}\mathcal{O}$ is not that strong a notion. It guarantees indistinguishability only if encryptions of bad inputs *do not exist*. Previous works avoid this issue by assuming a stronger notion of obfuscation called extractability obfuscation [BCP14] or differing inputs obfuscation [ABG+13b], which we wish to avoid.

Our solution tries to achieve non-existence of bad ciphertexts to the extent possible. Correctness of the encryption scheme means that every input, *even a bad one*, must have a valid encryption. Our first idea is to *puncture* the decryption key so ciphertexts of bad messages never decrypt. To move between these two worlds unnoticeably, we also require that the real decryption key DK is computationally indistinguishable from a "punctured" decryption key $DK\{S\}$.

However, this last requirement seems to raise concerns. If the set of bad inputs is known and the encryption key is known, then a punctured decryption key *cannot* be indistinguishable from an unpunctured one. Unfortunately, our obfuscated circuits contain the encryption key EK, which makes this a real concern. Our second idea is to puncture the encryption key also, so that it cannot encrypt bad inputs any more. Once this is done, one can expect indistinguishability of punctured versus unpunctured decryption keys. Indeed, this is exactly what we achieve. Namely, the punctured decryption key $DK\{S\}$ and the real decryption key DK are indistinguishable given the punctured encryption key $EK\{S'\}$ for any set $S' \supseteq S$, as well as ciphertexts of any messages that lie outside of S.

Finally, the set of bad inputs cannot be arbitrary if we require the encryption and decryption keys to be small. Indeed, the keys could otherwise be used to compress arbitrary data, which is information theoretically impossible. Our solution is to only consider sets of bad inputs S for which set membership is decidable by a small circuit C_S that takes as input x and decides if $x \in S$.

3.1.1 Construction

Our ACE construction is heavily inspired by the "hidden sparse triggers" technique of Sahai and Waters [SW14], and is identical to the "puncturable deterministic encryption" of Waters [Wat14]. In particular, our ciphertexts for a message m are of the form

$$C = \big(F_1(m), F_2(F_1(m)) \oplus m\big)$$

where F_1 and F_2 are puncturable pseudorandom functions (PPRFs). Despite these similarities, ACE is a much more demanding primitive.

In particular, ACE is different from puncturable deterministic encryption in at least two respects. First, we define and crucially use the fact that the encryption and decryption keys can be punctured separately, on different sets. Secondly, and perhaps more importantly, a punctured decryption key $DK\{S\}$ is indistinguishable from $DK\{S'\}$ as long as one does not have access to ciphertexts of messages in the symmetric difference $S \Delta S'$.

Indistinguishability of Punctured Decryption: A Sketch. We will show how to puncture a decryption key at one additional point, and then use an easy sequence of hybrids to puncture the entire set S. Here we suffer a security loss of $|S|$, but in our applications S and in fact the entire message space M will have polynomially bounded size, so this is acceptable.

To puncture at a point m^*, we use an injective bit-commitment (constructed from an injective OWF) to indistinguishably sabotage a check in Dec. This check asserts that for a ciphertext $\alpha \| \beta$, $\alpha = F_1(m)$, where m is the tentative decrypted value. We will make this ($\mathsf{i}\mathcal{O}$-obfuscated) check always fail by making the following indistinguishable changes.

1. If $m = m^*$, the check is replaced by $\mathsf{Commit}(0; \alpha) = z$, where z is hard-coded as $\mathsf{Commit}(0; F_1(m^*))$. The indistinguishability of this change uses $\mathsf{i}\mathcal{O}$, where functional equivalence follows from Commit's injectivity.

2. The key for F_1 is punctured at m^* giving $K\{m^*\}$. Indistinguishability is by $\mathsf{i}\mathcal{O}$, where functional equivalence follows because F_1 is no longer ever evaluated at m^*.

3. z is hard-coded as $\mathsf{Commit}(0; r)$, where r is chosen uniformly at random. Indistinguishability is by the pseudorandomness at punctured points of F_1. We note in order for this step to work, *all* copies of F_1 seen by the adversary must be

punctured at m^*. In particular, this is where the argument breaks down if the adversary is given EK or an encryption of m^*.

4. z is hard-coded as $\mathsf{Commit}(1; r)$. Indistinguishability is by the computationally hiding property of the commitment.

5. The check that $\mathsf{Commit}(0; \alpha) = z$ is replaced by FALSE. Indistinguishability is by $i\mathcal{O}$, since by injectivity z is not in the image of $\mathsf{Commit}(0, \cdot)$, so no α will make the check return TRUE.

6. The key for F_1 is unpunctured at m^* by $i\mathcal{O}$.

3.2 Garbled RAM Overview

Our main application of ACE is the construction of a *succinct garbled RAM*. We give an overview of the construction and the proof ideas here. We refer the reader to Chapter 8 for formal definitions and proofs. Our construction is greatly simplified by defining a weaker primitive which we call weak garbling of RAM programs.

3.2.1 Weak Garbling

Definition

Syntactically, a weak garbling scheme (parameterized by s) is an algorithm $\mathsf{WkGarble}$. As input, $\mathsf{WkGarble}$ takes a RAM program Π and an initial memory $\vec{x}$, and as output it produces another RAM program Π' and initial memory $\vec{x}'$. In addition to correctness ($\Pi(\vec{x}) = \Pi'(\vec{x}')$) and efficiency ($\Pi'$ must take roughly the same time and space as Π and $\vec{x}'$ must be roughly the same size as $\vec{x}$), $\mathsf{WkGarble}$ must satisfy the following security property:

Suppose $i_1, \ldots, i_t$ are a sequence of memory addresses describable by a circuit Γ with $|\Gamma| \leq s$. Let Π^0 and Π^1 be RAM machines, and let $\vec{x}^0$ and $\vec{x}^1$ be initial memory configurations such that first t locations accessed in $\Pi^0|_{\vec{x}^0}$ and $\Pi^1|_{\vec{x}^1}$ are $i_1, \ldots, i_t$ (here $\Pi|_{\vec{x}}$ denotes the execution of Π on $\vec{x}$). Suppose that after t steps $\Pi^0|_{\vec{x}^0}$ and $\Pi^1|_{\vec{x}^1}$ both

output the same value, or they both have the same state and memory configuration. If Π^0 and Π^1's transition functions are also functionally equivalent on all states with timestamp larger than t, then $\mathsf{WkGarble}(\Pi^0, \vec{x}^0) \approx \mathsf{WkGarble}(\Pi^1, \vec{x}^1)$.

Construction

$\mathsf{WkGarble}$ first transforms Π and $\vec{x}$ to have a "verifiable reads" property against "semi-malicious memories". It then transforms the resulting $\vec{x}'$ and the transition function of Π' so that every word of memory is encrypted using ACE. Similarly, the state of Π' is also encrypted using ACE. Finally, the transition function is $\mathsf{i}\mathcal{O}$-obfuscated, and $\mathsf{WkGarble}$ outputs $(\Pi'', \vec{x}'')$. In the language of Chapter 5, $\Pi'', \vec{x}''$ is $\mathsf{Harden}((\Pi', (Q, W)), \vec{x}')$, where Q denotes the universe of states of Π' and W denotes the universe of words in $\vec{x}'$.

A *semi-malicious memory* is an adversarial external memory storing words from a universe W at addresses in $[N]$. When the semi-malicious memory receives a read request, rather than answering correctly, it can answer with any arbitrary previously written value. In order for this to differ meaningfully from a fully malicious memory (which can answer read requests completely arbitrarily), W should be larger than $\{0, 1\}$. Indeed in our applications W will be strings of $O(\log \lambda)$ length.

Π' having *verifiable reads* means that Π' can verify that the values read from memory are correct. Against a semi-malicious memory, this property is essentially the same as the predictably timed writes of [GHRW14], and in fact our construction uses basically the same generic transformation. A location tag and timestamp is added to each word written to memory. To ensure that each word read from memory has a predictable timestamp, we access memory via a binary tree structure, in which each node holds the timestamps of its children.

Security Proof

The security of the above construction would be straight-forward if instead of $\mathsf{i}\mathcal{O}$-obfuscation we could use a much stronger obfuscator, such as virtual black-box obfuscation. In Chapter 5, we develop general techniques for showing indistinguishability

of $i\mathcal{O}$-obfuscated programs which take ACE-encrypted inputs and outputs.

The proof of security for WkGarble shows a sequence of indistinguishable hybrids, the i^{th} of which replaces the first i steps of Π with dummy accesses, and gives as initial memory the memory configuration of Π after i steps of execution on $\vec{x}$. Showing indistinguishability then consists of applying Theorem 2 with a suitable invariant.

3.2.2 Full Garbling Construction

We garble a RAM program Π and an initial memory $\vec{x}$ by applying two transformations. We first apply the oblivious RAM transformation of Shi et al. [SCSL11], which was simplified by Chung and Pass [CP13]. We then weakly garble the result to obtain $(\tilde{\Pi}, \tilde{\vec{x}})$.

We note that both the ORAM and our weak garbling scheme are separable – They consist of a key generation phase, after which the program and initial memory configuration can be transformed separately. This implies that the full garbling scheme is also separable.

3.2.3 Overview of the Security Proof

Simulator

We would like to simulate $\mathsf{Garble}(\Pi, \vec{x})$ given only $\Pi(\vec{x})$, T (the running time of Π on $\vec{x}$). $\mathsf{Sim}(\Pi(\vec{x}))$ is defined as the weak garbling of $(\Pi', \vec{x}')$, where

- Π' first makes $\tilde{O}(T)$ dummy accesses, each of which "looks like" an ORAM executing a single underlying access. Then Π' outputs $\Pi(\vec{x})$.
- $\vec{x}'$ is a length-N' memory configuration containing just $\perp$. Here N' is the number of words that the ORAM needs to simulate a memory of size N.

We show that $\mathsf{Sim}(\Pi(\vec{x}))$ is indistinguishable from $\mathsf{Garble}(\Pi, \vec{x})$ by a hybrid argument. The main idea is that by using a suitable property of the oblivious RAM, we can bootstrap WkGarble into a full garbling scheme. In the hybrids, we replace

the real computation with a dummy computation one step at a time, starting at the beginning.

ORAM Properties

We require the following two natural properties from the ORAM:

1. The locations accessed by the ORAM at a time j are independent of the underlying accesses, and can be efficiently sampled.

2. The ORAM's state and memory configuration at time t can be efficiently sampled given the initial memory configuration, the underlying accesses, and the real accesses made by the ORAM.

3. The ORAM has low worst-case overhead. In particular, the ORAM simulates each underlying access with only $\mathsf{polylog}(N)$ physical accesses. Here N is the size of the underlying memory.

The ORAM of Shi et al. [SCSL11], simplified by Chung and Pass [CP13], has all of these properties.

These properties together with WkGarble allows us to move (in a long sequence of hybrids) to a hybrid in which all of the locations accessed by $\tilde{\Pi}$ are computable by a small circuit Γ. A last application of WkGarble indistinguishably switches the initial memory to encryptions of many $\perp$s. This hybrid is then simulatable given only $\Pi(\vec{x})$, $|\Pi|$, $|\vec{x}|$, and the running time T.

3.2.4 Optimizations

Eliminating double obfuscation In our construction as given, we have two layers of obfuscation - one to implement ACE, and an obfuscated circuit that contains ACE keys. This is only necessary for the modularity of our proof - the properties of ACE could be proved directly in our construction with only one layer of $i\mathcal{O}$ obfuscation.

Generating the encrypted initial memory We note that the $O(S) \cdot \mathsf{poly}(\lambda)$ dependence in our scheme is only in the running time of Encode, and does not use obfuscation. This is because this encryption can be done with the underlying PRF keys rather than the obfuscated encryption algorithm. In particular, we only obfuscate one small circuit.

Chapter 4

Asymmetrically Constrained Encryption

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

1. For each message m and key K, there is at most a single string that decrypts to m under key K.

2. The full decryption algorithm is indistinguishable from a version which is punctured at a succinctly described set of messages (namely at some $S \subset \mathcal{M}$ which is decidable by a small circuit). Furthermore, indistinguishably holds even when given ciphertexts and a *constrained* encryption algorithm, as long as the trivial attack doesn't work.

These properties will be central in our analysis of iterated circuit obfuscation.

4.1 Definition

An asymmetrically constrained encryption scheme consists of five polynomial-time algorithms $(\mathsf{Setup}, \mathsf{GenEK}, \mathsf{GenDK}, \mathsf{Enc}, \mathsf{Dec})$ described as follows:

- **Setup:** $\mathsf{Setup}(1^\lambda, 1^n, 1^s)$ is a randomized algorithm that takes as input the security parameter λ, the input length n, and a "circuit succinctness" parameter s, all in unary. Setup then outputs a secret key SK.

  Let $\mathcal{M} = \{0,1\}^n$ denote the message space, where $n = \mathsf{poly}(\lambda)$. While n and s are arguments of Setup, we will generally think of them as given.

- **(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 of sets on which encryption and decryption keys can be punctured.

  - $\mathsf{GenEK}(SK, C_S)$ 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 encryption 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 decryption key $DK\{S\}$. We write DK to denote $DK\{\emptyset\}$.

  Unless mentioned otherwise, we will only consider admissible sets $S \subset \mathcal{M}$.

- **Encryption:** $\mathsf{Enc}(EK', m)$ is a deterministic algorithm that takes as input an encryption key EK' (that may be constrained) and a message $m \in \mathcal{M}$ and outputs a ciphertext c or the reject symbol $\perp$.

- **Decryption:** $\mathsf{Dec}(DK', c)$ is a deterministic algorithm that takes as input a decryption key DK' (that may be constrained) and a ciphertext c and outputs a message $m \in \mathcal{M}$ or the reject symbol $\perp$.

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

1. *Correctness of Decryption*: For all n, all $m \in \mathcal{M}$, all sets $S, S' \subset \mathcal{M}$ s.t. $m \notin S \cup S'$,

$$\Pr\left[\mathsf{Dec}(DK, \mathsf{Enc}(EK, m)) = m \;\middle|\; \begin{array}{l} SK \leftarrow \mathsf{Setup}(1^\lambda), \\ EK \leftarrow \mathsf{GenEK}(SK, C_{S'}), \\ DK \leftarrow \mathsf{GenDK}(SK, C_S) \end{array}\right] = 1.$$

   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 Encryption*: Let $SK \leftarrow \mathsf{Setup}(1^\lambda)$. For any message $m \in \mathcal{M}$ and any sets $S, S' \subset \mathcal{M}$ with m not in the symmetric difference $S \Delta S'$,

$$\Pr\left[\mathsf{Enc}(EK, m) = \mathsf{Enc}(EK', m) \;\middle|\; \begin{array}{l} SK \leftarrow \mathsf{Setup}(1^\lambda), \\ EK \leftarrow \mathsf{GenEK}(SK, C_S), \\ EK' \leftarrow \mathsf{GenEK}(SK, C_{S'}) \end{array}\right] = 1.$$

3. *Unique Ciphertexts*: If $\mathsf{Dec}(DK, c) = \mathsf{Dec}(DK, c') \neq \perp$, then $c = c'$.

4. *Safety of Constrained Decryption*: For all strings c, all $S \subset \mathcal{M}$,

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

   This says that a punctured key $DK\{S\}$ will never decrypt a string c to a message in S.

5. *Equivalence of Constrained Decryption*: If $\mathsf{Dec}(DK\{S\}, c) = m \neq \perp$ and $m \notin S'$, then $\mathsf{Dec}(DK\{S'\}, c) = m$.

Security of Constrained Decryption. Intuitively, this property says that for any two sets S_0, S_1, no adversary can distinguish between the constrained key $DK\{S_0\}$ and $DK\{S_1\}$, even given additional auxiliary information in the form of a constrained encryption key EK' and ciphertexts $c_1, \ldots, c_t$. To rule out trivial attacks, EK' is

constrained at least on $S_0 \Delta S_1$. Similarly, each c_i is an encryption of a message $m \notin S_0 \Delta S_1$.

Formally, we describe security of constrained decryption 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. $S_0 \Delta S_1 \subseteq U \subseteq \mathcal{M}$ and sends their circuit descriptions (C_{S_0}, C_{S_1}, C_U) to the challenger. $\mathcal{A}$ also sends arbitrary polynomially many messages $m_1, \ldots, m_t$ such that $m_i \notin S_0 \Delta S_1$.

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

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

The advantage of $\mathcal{A}$ in this game is defined as $\mathsf{adv}_{\mathcal{A}} = \left|\Pr[b' = b] - \frac{1}{2}\right|$. We require that $\mathsf{adv}_{\mathcal{A}}(\lambda) \leq \mathsf{negl}(\lambda)$.

Remark 4. Looking ahead, in our construction of ACE, we have

$$\mathsf{adv}_{\mathcal{A}}(\lambda) = \mathsf{poly}(|S_0 \Delta S_1|, \lambda) \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$$

for any S_0, S_1. When $|S_0 \Delta S_1|$ is super-polynomial, we require something like subexponential hardness of one-way functions as well as subexponential hardness of $\mathsf{i}\mathcal{O}$ in order for $\mathsf{adv}_{\mathcal{A}}$ to be negligible. As we will see later, our garbled RAM construction has $\mathcal{M} = \{0,1\}^{O(\log \lambda)}$, which means $|S_0 \Delta S_1| \leq |\mathcal{M}| \leq \mathsf{poly}(\lambda)$ so we can rely on polynomial assumptions.

Selective Indistinguishability of Ciphertexts. Intuitively, this property says that no adversary can distinguish between encryptions of m_0 from encryptions of m_1, even given additional auxiliary information. The auxiliary information corresponds to constrained encryption and decryption keys EK', DK', as well as some ciphertexts

$c_1, \ldots, c_t$. In order to rule out trivial attacks, EK' and DK' should both be punctured on at least $\{m_0, m_1\}$, and none of $c_1, \ldots, c_t$ should be an encryption of m_0 or m_1.

Formally, we require that for all sets $S, U \subset \mathcal{M}$, for all $m_0^*, m_1^* \in S \cap U$, and all $m_1, \ldots, m_t \in \mathcal{M} \setminus \{m_0^*, m_1^*\}$, the distribution

$$EK\{S\}, DK\{U\}, c_0^*, c_1^*, c_1, \ldots, c_t$$

is indistinguishable from

$$EK\{S\}, DK\{U\}, c_1^*, c_0^*, c_1, \ldots, c_t$$

where $SK \leftarrow \mathsf{Setup}(1^\lambda)$, $EK \leftarrow \mathsf{GenEK}(SK, \emptyset)$, $EK\{S\} \leftarrow \mathsf{GenEK}(SK, C_S)$, $DK\{U\} \leftarrow \mathsf{GenDK}(SK, C_U)$, $c_b^* \leftarrow \mathsf{Enc}(EK, m_b^*)$, and $c_i \leftarrow \mathsf{Enc}(EK, m_i)$.

Definition 3. An ACE scheme is *secure* if it satisfies the properties of correctness, unique ciphertexts, security of constrained decryption and selective indistinguishability of ciphertexts.

4.2 Construction

We now present a construction of an asymmetrically constrainable encryption 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 $F_{1,k} : \{0,1\}^n \to \{0,1\}^{2n+\log^{1+\epsilon}(\lambda)}$. Let $\mathcal{F}_2 = \{F_{2,k}\}_{k \in \{0,1\}^\lambda}$ be another puncturable pseudorandom function family, where $F_{2,k} : \{0,1\}^{2n+\log^{1+\epsilon}(\lambda)} \to \{0,1\}^n$. Let $i\mathcal{O}$ be an indistinguishability obfuscator for all circuits.

Let s denote the set description size parameter for ACE. Let $p = \mathsf{poly}(n, \lambda, s)$ 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. We will write F_i to denote the function F_{i,K_i}.

$\mathsf{GenEK}((K_1, K_2), C_S)$**:** The encryption 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 encryption key $EK\{S\} \leftarrow \mathsf{i}\mathcal{O}(\mathcal{G}_{\mathsf{enc}})$ and outputs the result.

Constants: K_1, K_2, circuit C_S
Input: Message $m \in \{0,1\}^n$
if $C_S(m)$ **then return** $\perp$;
else
  $\alpha \leftarrow F_1(m)$;
  $\beta \leftarrow F_2(\alpha) \oplus m$;
  return $\alpha \| \beta$
end

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

$\mathsf{GenDK}((K_1, K_2), C_S)$**:** The decryption 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 decryption key $DK\{S\} \leftarrow \mathsf{i}\mathcal{O}(\mathcal{G}_{\mathsf{dec}})$ and outputs the result.

Constants: K_1, K_2, circuit C_S
Input: Ciphertext $c \in \{0,1\}^{3n+\log^{1+\epsilon}(\lambda)}$
parse c as $\alpha \| \beta$ with $\beta \in \{0,1\}^n$;
$m \leftarrow F_2(\alpha) \oplus \beta$;
if $C_S(m)$ **then return** $\perp$;
else if $\alpha \neq F_1(m)$ **then return** $\perp$;
else return m;

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

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

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

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

4.3 Proof of Security

Correctness. We first argue correctness:

1. *Correctness of Decryption*: This follow directly from the definitions of Algorithm 1 and Algorithm 2 and the perfect correctness of $\mathsf{i}\mathcal{O}$.

2. *Equivalence of Constrained Encryption*: This follows directly from the definition of Algorithm 1 and the perfect correctness of $\mathsf{i}\mathcal{O}$.

3. *Uniqueness of Encryptions*: A ciphertext $c = \alpha \| \beta$ decrypts to $m \neq \perp$ only if $\alpha = F_1(m)$ and $\beta = F_2(\alpha)$. So there can be only one ciphertext which decrypts to m.

4. *Safety of Constrained Decryption*: This follows directly from the definition of Algorithm 2 and the perfect correctness of $\mathsf{i}\mathcal{O}$.

5. *Equivalence of Constrained Decryption*: This follows directly from the definition of Algorithm 2 and the perfect correctness of $\mathsf{i}\mathcal{O}$.

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

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

Proof. Let S_0, S_1, U be arbitrary subsets of $\{0,1\}^n$ s.t. $S_0 \Delta S_1 \subseteq U$ and let C_{S_0}, C_{S_1}, C_U be their circuit descriptions. Let $m_1, \ldots, m_t$ be arbitrary messages such that every $m_i \in M \setminus (S_0 \Delta S_1)$. Let $SK \leftarrow \mathsf{Setup}(1^\lambda)$, $EK \leftarrow \mathsf{GenEK}(SK, \emptyset)$ and $EK\{U\} \leftarrow \mathsf{GenEK}(SK, C_U)$. For every $i \in [t]$, let $c_i \leftarrow \mathsf{Enc}(EK, m_i)$. 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 $(EK\{U\}, DK\{S_0\}, \{c_i\})$ and $(EK\{U\}, DK\{S_1\}, \{c_i\})$ with advantage more than $\epsilon_{S_0,S_1} = |S_0 \Delta S_1| \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{\mathsf{i}\mathcal{O}}(\lambda))$.

We will prove this via a sequence of $|S_0 \Delta S_1|$ indistinguishable hybrid experiments H_i where in experiment H_0, the decryption key is $DK\{S_0\}$ whereas in experiment $H_{|S_0 \Delta S_1|}$, the decryption key is $DK\{S_1\}$. Without loss of generality, we will suppose that $S_0 \subseteq S_1$. The general case follows because

$$DK\{S_0\} \approx DK\{S_0 \cap S_1\} \approx DK\{S_1\}$$

where we have omitted the auxiliary information of encryption keys and ciphertexts.

We now proceed to give details. Let u_i denote the lexicographically i^{th} element of $S_1 \setminus S_0$. Throughout the experiments, we will refer to the encryption key and decryption 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 i^{th} hybrid, the decryption key program $\mathcal{G}'_{\mathsf{dec}}$ 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.

For notational simplicity, set $u_0 = -\infty$. Therefore, in experiment H_0, $\mathcal{G}'_{\mathsf{dec}}$ has the same functionality as $DK\{S_0\}$, and in $H_{|S_1 \setminus S_0|}$, $\mathcal{G}'_{\mathsf{dec}}$ 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.

Hybrid $H_{i,1}$: This is the same as experiment $H_{i,0}$ except that we modify $\mathcal{G}'_{\mathsf{dec}}$ as in

Input: ciphertext $c \in \{0,1\}^{3n+\log^{1+\epsilon}(\lambda)}$
Constants: PPRF keys K_1, K_2, circuits C_{S_0}, C_{S_1}
1 Parse c as $\alpha \| \beta$;
2 $m \leftarrow F_2(\alpha) \oplus \beta$;
3 **if** $m \leq u_i$ *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;

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

Algorithm 4. If the decrypted message m is u_{i+1}, then instead of checking whether $\alpha \neq F_1(m)$ in line 5, $\mathcal{G}'_{\mathsf{dec}}$ now checks whether $\mathsf{Commit}(0;\alpha) \neq \mathsf{Commit}(0; F_1(u_i))$, where Commit is a perfectly binding injective commitment.

Input: ciphertext $c \in \{0,1\}^{3n+\log^{1+\epsilon}(\lambda)}$
Constants: PPRF keys K_1, K_2, circuits C_{S_0}, C_{S_1}, message u_{i+1},
$z = \mathsf{Commit}(0; F_1(u_{i+1}))$
1 Parse c as $\alpha \| \beta$;
2 $m \leftarrow F_2(\alpha) \oplus \beta$;
3 **if** $m \leq u_i$ *and* $m \in S_1$ **then return** $\perp$;
4 **else if** $m \in S_0$ **then return** $\perp$;
5 **else if** $m = u_{i+1}$ *and* $\mathsf{Commit}(0;\alpha) \neq z$ **then return** $\perp$;
6 **else if** $\alpha \neq F_1(m)$ **then return** $\perp$;
7 **else return** m;

Algorithm 4: $\mathcal{G}'_{\mathsf{dec}}$ in Hybrid $i, 1$

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

- The PRF key K_1 in $\mathcal{G}'_{\mathsf{dec}}$ is punctured at u_{i+1}, i.e., K_1 is replaced with $K_1\{u_{i+1}\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_1, u_{i+1})$.

Hybrid $H_{i,3}$: This is the same as experiment $H_{i,2}$ except that we modify the program $\mathcal{G}'_{\mathsf{enc}}$ underlying EK' such that the PRF key K_1 hardwired in $\mathcal{G}'_{\mathsf{enc}}$ is replaced with the same punctured key $K_1\{u_{i+1}\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_1, u_{i+1})$ as is used in $\mathcal{G}'_{\mathsf{dec}}$

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 $\mathsf{Commit}(0; r)$ where r is a randomly chosen string in

$\{0,1\}^{2n+\log^{1+\epsilon}(\lambda)}$.

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 $\mathsf{Commit}(1;r)$ where r is a randomly chosen string in $\{0,1\}^{2n+\log^{1+\epsilon}(\lambda)}$.

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 decrypted message m is u_{i+1}. An equivalent description of $\mathcal{G}'_{\mathsf{dec}}$ is that in line 3, it now checks whether $m \leq u_{i+1}$ instead of $m \leq u_i$.

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 $K_1\{u_{i+1}\}$ 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 Commit is injective, we have that the following two checks are equivalent: $\alpha \neq F_1(m)$ and $\mathsf{Commit}(0;\alpha) \neq \mathsf{Commit}(0;F_1(m))$. 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 decryption 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 $c_{i+1} = \alpha_{i+1}\|\beta_{i+1}$ denote the *unique* ciphertext such that $\mathsf{Dec}(DK, c_{i+1}) = u_{i+1}$ (where DK denotes the unconstrained decryption key program). First note that on any input $c \neq c_{i+1}$, 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 $K_1\{u_{i+1}\}$. 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+1}, after decrypting to obtain u_{i+1}, $\mathcal{G}'_{\mathsf{dec}}$ computes $\mathsf{Commit}(0;F_1(u_{i+1}))$ and then checks whether $\mathsf{Commit}(0;\alpha_{i+1}) \neq \mathsf{Commit}(0;F_1(u_{i+1}))$ whereas $\mathcal{G}''_{\mathsf{dec}}$ simply

checks whether $\mathsf{Commit}(0;\alpha_i) \neq z$. But since the value z hardwired in $\mathcal{G}''_{\mathsf{dec}}$ is equal to $\mathsf{Commit}(0;F_1(u_{i+1}))$, 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 encryption 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 $K_1\{u_{i+1}\}$. However, note that neither $\mathcal{G}'_{\mathsf{enc}}$ nor $\mathcal{G}''_{\mathsf{enc}}$ ever evaluate F_1 on u_{i+1}. Thus, 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 $H_{i,5}$. $H_{i,4}$ and $H_{i,5}$ are computationally indistinguishable because of the hiding properties of Commit.

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 decryption 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 commitments, pseudorandom functions, and indistinguishability obfuscation. In total, (ignoring constant multiplicative factors) we have $|S_0 \Delta S_1|$ hybrids where $S_0 \Delta S_1$ is the symmetric set difference. Thus, overall, we get that no adversary can distinguish between

$$EK\{U\}, DK\{S_0\}, c_1, \ldots, c_t$$

and

$$EK\{U\}, DK\{S_1\}, c_1, \ldots, c_t$$

with advantage more than

$$\epsilon_{S_0,S_1} = |S_0 \Delta S_1| \cdot (\mathsf{adv}_{COM}(\lambda) + \mathsf{adv}_{PRF}(\lambda) + \mathsf{adv}_{i\mathcal{O}}(\lambda)).$$

Replacing $\mathsf{adv}_{COM}(\lambda) + \mathsf{adv}_{PRF}(\lambda)$ with $\mathsf{poly}(\lambda) \cdot \mathsf{adv}_{OWF}$, where OWF is the injective one-way function used to construct the commitment and puncturable PRF, we get

$$\epsilon_{S_0,S_1} = |S_0 \Delta S_1| \cdot \mathsf{poly}(\lambda) \cdot (\mathsf{adv}_{OWF}(\lambda) + \mathsf{adv}_{i\mathcal{O}}(\lambda))$$

as required. □

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

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

Proof. The proof of the lemma proceeds in a sequence of hybrid distributions where we make indistinguishable changes to $EK\{U\}$, $DK\{S\}$, and the challenge ciphertexts (c_0^*, c_1^*). The "extra" ciphertexts $c_1, \ldots, c_t$ remain unchanged throughout the hybrids.

Hybrid H_0. This is the real world distribution. For completeness (and to ease the presentation of the subsequent hybrid distributions), we describe the sampling process

here. Let $S, U \subset \mathcal{M} = \{0,1\}^n$ be the sets chosen by the adversary and C_S, C_U be their corresponding circuit descriptions. Let m_0^*, m_1^* be the challenge messages in $S \cap U$ and $(m_1, \ldots, m_t)$ be the extra messages in $\{0,1\}^n$. Then

1. Sample PRF keys $K_1 \leftarrow \mathsf{Gen}_{\mathsf{PRF}}(1^\lambda)$, $K_2 \leftarrow \mathsf{Gen}_{\mathsf{PRF}}(1^\lambda)$, $b \leftarrow \{0,1\}$.

2. For $b \in \{0,1\}$, compute $\alpha_b^* \leftarrow F_1(m_b^*)$, $\gamma_b^* \leftarrow F_2(\alpha_b^*)$ and $\beta_b^* = \gamma_b^* \oplus m_b^*$. Let $c_b^* = \alpha_b^* \| \beta_b^*$.

3. For every $j \in [t]$, compute $\alpha_j \leftarrow F_1(m_j)$, $\gamma_j \leftarrow F_2(\alpha_j)$ and $\beta_j = \gamma_j \oplus m_j$. Let $c_j = \alpha_j \| \beta_j$.

4. Compute $EK\{U\} \leftarrow \mathsf{i}\mathcal{O}(\mathcal{G}'_{\mathsf{enc}})$ where $\mathcal{G}'_{\mathsf{enc}}$ is described in Algorithm 5.

5. Compute $DK\{S\} \leftarrow \mathsf{i}\mathcal{O}(\mathcal{G}'_{\mathsf{dec}})$ where $\mathcal{G}'_{\mathsf{dec}}$ is described in Algorithm 6.

6. Output the following tuple:

$$\left(EK\{S\}, DK\{U\}, c_b^*, c_{1-b}^*, c_1, \ldots, c_t\right).$$

Constants: $K1$, $K2$, 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: $\mathcal{G}'_{\mathsf{enc}}$ in Hybrid H_0

Hybrid H_1. Modify $\mathcal{G}'_{\mathsf{enc}}$: the hardwired PRF key K_1 is replaced with a punctured key $K_1\{m_0^*, m_1^*\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_1, \{m_0^*, m_1^*\})$.

Hybrid H_2. Modify $\mathcal{G}'_{\mathsf{dec}}$: the hardwired PRF key K_1 is replaced with a punctured key $K_1\{m_0^*, m_1^*\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_1, \{m_0^*, m_1^*\})$.

Constants: K_1, K_2, circuit C_S
Input: ciphertext c
1 Parse c as $\alpha \| \beta$;
2 $m \leftarrow F_2(\alpha) \oplus \beta$;
3 **if** $C_S(m)$ **then return** $\perp$;
4 **else if** $\alpha \neq F_1(m)$ **then return** $\perp$;
5 **else return** m;

Algorithm 6: $\mathcal{G}'_{\mathsf{dec}}$ in Hybrid H_0

Hybrid H_3**.** Modify $\mathcal{G}'_{\mathsf{dec}}$: Perform the following check in the beginning. If input ciphertext $c = c_b^*$ for $b \in \{0,1\}$ then output $\perp$. The modified $\mathcal{G}'_{\mathsf{dec}}$ is described in Algorithm 7

Constants: $K_1\{m_0^*, m_1^*\}$, K_2, circuit C_S, c_0^*, c_1^*
Input: ciphertext c
1 **if** $c = c_b^*$ *for* $b \in \{0,1\}$ **then return** $\perp$;
2 Parse c as $\alpha \| \beta$;
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;

Algorithm 7: $\mathcal{G}'_{\mathsf{dec}}$ in Hybrid 3

Hybrid H_4**.** Modify challenge ciphertexts $c_b^* = \alpha_b^* \| \beta_b^*$: Generate each α_b^* as a truly random string.

Hybrid H_5**.** Modify $\mathcal{G}'_{\mathsf{enc}}$: the hardwired PRF key K_2 is replaced with a punctured key $K_2\{\alpha_0^*, \alpha_1^*\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_2, \{\alpha_0^*, \alpha_1^*\})$. Here we assume the set $\{\alpha_0^*, \alpha_1^*\}$ is sorted lexicographically.

Hybrid H_6**.** Modify $\mathcal{G}'_{\mathsf{dec}}$: we change the check performed in line 1 of Algorithm 7. For any input ciphertext $c = \alpha \| \beta$, if $\alpha = \alpha_b^*$ for $b \in \{0,1\}$, then output $\perp$. Note that $\mathcal{G}'_{\mathsf{dec}}$ only has α_b^* hardwired as opposed to c_b^*. The modified $\mathcal{G}'_{\mathsf{dec}}$ is described in Algorithm 8.

Hybrid H_7**.** Modify $\mathcal{G}'_{\mathsf{dec}}$: the hardwired PRF key K_2 is replaced with the same a punctured key $K_2\{\alpha_0^*, \alpha_1^*\} \leftarrow \mathsf{Puncture}_{\mathsf{PRF}}(K_2, \{\alpha_0^*, \alpha_1^*\})$ as was used in $\mathcal{G}'_{\mathsf{enc}}$.

Hybrid H_8**.** Modify challenge ciphertexts $c_b^* = \alpha_b^* \| \beta_b^*$: For $b \in \{0,1\}$, generate β_b^* as

Constants: PPRF keys $K_1\{m_0^*, m_1^*\}$, K_2, circuit C_S, ciphertext prefixes α_0^*, α_1^*
Input: ciphertext c

1 parse c as $\alpha \| \beta$;
2 **if** $\alpha = \alpha_b^*$ *for* $b \in \{0,1\}$ **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;

Algorithm 8: $\mathcal{G}'_{\mathsf{dec}}$ in Hybrid 6

a truly 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 observe that no adversary can guess bit b in the final hybrid H_8 with probability better than $\frac{1}{2}$. 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 encryption 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_b^*. In particular, line 4 is *not* executed in both $\mathcal{G}'_{\mathsf{enc}}$ and $\mathcal{G}''_{\mathsf{enc}}$ for every input message m_b^*. (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 $K_1\{m_0^*, m_1^*\}$) are functionally equivalent. The indistinguishability of H_0 and H_1 follows from the security 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 decryption 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 either challenge ciphertext c_b^*. In particular, line 4 is *not* executed in both $\mathcal{G}'_{\mathsf{dec}}$ and $\mathcal{G}''_{\mathsf{dec}}$ for either m_b^*. 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 $K_1\{m_0^*, m_1^*\}$) are functionally equivalent. As a consequence, the indistinguishability of H_1 and H_2 follows from the security 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 decryption 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 either challenge ciphertext c^*_b. 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 security 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 $\mathcal{F}_1$. (Note that each ciphertext $c_1, \ldots, c_t$ can be generated using the punctured PRF key, because they are not encryptions of m^*_0 or m^*_1.)

Indistinguishability of H_4 and H_5. Note that with overwhelming probability, the random strings α^*_b are not in the range of the F_1. Therefore, except with negligible probability, there does not exist a message m such that $F_1(m) = \alpha^*_b$ for $b \in \{0, 1\}$. 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_4 and H_5 follows from the security 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 decryption 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: $\mathcal{G}''_{\mathsf{dec}}$ hard-codes an output of $\perp$ for every ciphertext of the form $c = \tilde{\alpha}^*_b \| \star$ while $\mathcal{G}'_{\mathsf{dec}}$ only hard-codes an output of $\perp$ for ciphertexts $c^*_b = \alpha^*_b \| \beta^*_b$. In particular, the execution of $\mathcal{G}'_{\mathsf{dec}}$ continues onward from line 2 for every $c = \alpha^*_b \| \beta$ such that $\beta \neq \beta^*_b$. However, note that with overwhelming probability, each of the random strings α^*_b is not in the range of the F_1. Thus in line 5, $\mathcal{G}'_{\mathsf{dec}}$ will also output $\perp$ on every $c = \alpha^*_b \| \beta$, 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 security 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 un-

derlying the decryption key program $DK\{S\}$ in H_6 and H_7 respectively. Note that due to the check performed in line 2 (see Algorithm 8), 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 $\alpha^* \| \star$. Then, since the the punctured PRF scheme preserves functionality under puncturing, $\mathcal{G}'_{\mathsf{dec}}$ (using K_2) and $\mathcal{G}''_{\mathsf{dec}}$ (using $K_2\{\alpha_0^*, \alpha_1^*\}$) are functionally equivalent and the indistinguishability of H_6 and H_7 follows from the security 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 $\mathcal{F}_2$ (note that with overwhelming probability, each ciphertext $c_1, \ldots, c_t$ can be generated with the punctured F_2, because the $\tilde{\alpha}_b^*$ are chosen randomly).

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

□

Remark 5. In our discussion, we used the security properties of $\mathsf{i}\mathcal{O}$ somewhat loosely. We basically said that if $C_0 \equiv C_1$ then $\mathsf{i}\mathcal{O}(C_0) \approx \mathsf{i}\mathcal{O}(C_1)$. In particular, we ignored the constraint that $|C_0| = |C_1|$. To formalize the proof, we pad $\mathcal{G}_{\mathsf{enc}}$ to be as large as $\mathcal{G}_{\mathsf{enc}}$ is in any of our hybrids, and we do a similar thing for $\mathcal{G}_{\mathsf{dec}}$. One can check that $\mathcal{G}_{\mathsf{enc}}$ and $\mathcal{G}_{\mathsf{dec}}$ are not too large in any of our hybrids. $|\mathcal{G}_{\mathsf{enc}}|$ and $|\mathcal{G}_{\mathsf{dec}}|$ are both bounded by $\mathsf{poly}(n, \lambda, s)$. Here n is the message length, λ is the security parameter, and s is the "succinctness" parameter – the maximum admissible size for a set's circuit description.

Chapter 5

How to use ACE

We'll develop techniques using ACE to garble computations consisting of repeated identical steps - for example, a Turing machine's transition function. The garbled functionality will consist of an obfuscated "hardened block" which executes this step, acting on an encrypted intermediate state. If one uses ACE to encrypt the state and $i\mathcal{O}$ to obfuscate the block, then this chapter develops a useful condition for two hardened blocks to be indistinguishable.

5.1 Blocks

We aim to instantiate a computational "block", to which we can apply a Harden transformation.

Definition 4. We say that $B = (C, (\mathcal{M}_1, \ldots, \mathcal{M}_\ell))$ is a block mapping

$$\mathcal{M}_{i_1} \times \cdots \times \mathcal{M}_{i_n} \to \mathcal{M}_{j_1} \times \cdots \times \mathcal{M}_{j_m}$$

if C is a circuit also mapping $\mathcal{M}_{i_1} \times \cdots \times \mathcal{M}_{i_n} \to \mathcal{M}_{j_1} \times \cdots \times \mathcal{M}_{j_m}$. Here each $\mathcal{M}_k$ is a message space, and we say that $\mathcal{M}_1, \ldots, \mathcal{M}_\ell$ are the *encapsulated types* of B.

Example 1. The principle block we will harden in our RAM garbling construction is $(\delta, (Q, W))$, where $\delta : Q \times W \to Q \times W \times [N] \times Y$ is the transition function for a RAM program.

Definition 5. Given a block $B = (C, (\mathcal{M}_1, \ldots, \mathcal{M}_\ell))$ mapping $\mathcal{M}_{i_1} \times \cdots \times \mathcal{M}_{i_n} \to \mathcal{M}_{j_1} \times \cdots \times \mathcal{M}_{j_m}$ as well as inputs $\vec{x}^i \in \mathcal{M}_i^*$, we define $\mathsf{Harden}(1^\lambda, B, \vec{x}^1, \ldots, \vec{x}^\ell)$ as:

1. Let $EK_i, DK_i \leftarrow \mathsf{Setup}_{ACE}(1^\lambda)$ for $i = 1, \ldots, l$ such that EK_i can be used to encrypt messages in $\mathcal{M}_i$.

   Let
   $$D_j(x) = \begin{cases} \mathsf{Dec}(DK_j, x) & \text{if } j \in \{1, \ldots, l\} \\ x & \text{otherwise} \end{cases}$$
   and
   $$E_j(x) = \begin{cases} \mathsf{Enc}(EK_j, x) & \text{if } j \in \{1, \ldots, l\} \\ x & \text{otherwise} \end{cases}$$

2. Define $C' = (E_{j_1} \| \cdots \| E_{j_m}) \circ C \circ (D_{i_1} \| \cdots \| D_{i_n})$ (here $f \| g$ denotes parallel composition of functions, defined as $(f \| g)(x, y) = f(x), g(y)$).

3. Harden outputs $\mathsf{i}\mathcal{O}(1^\lambda, C')$, $\mathsf{Enc}^*(EK_1, \vec{x}^1)$, $\ldots$, $\mathsf{Enc}^*(EK_\ell, \vec{x}^\ell)$, where Enc^* denotes component-wise encryption.

For ease of notation, we will often omit the security parameter and even $\vec{x}^1, \ldots, \vec{x}^\ell$, just write $\mathsf{Harden}(B)$ instead.

Remark 6. Harden is *separable*. Shared randomness can be used to compute the set of keys $\{EK_i, DK_i\}_{i=1}^\ell$, and then B can be hardened independently of the inputs $\vec{x}^1, \ldots, \vec{x}^\ell$.

Example 2. When Π is a RAM program with transition function $\delta : Q \times W \to Q \times W \times [N] \times Y$ and $\vec{x}$ is an initial memory configuration $(x_1, \ldots, x_N)$, $\mathsf{Harden}((\delta, (Q, W), \vec{x})$ consists of two parts. First is the $\mathsf{i}\mathcal{O}$ obfuscation of the circuit depicted in Figure 5-1. The other is the ACE-encrypted initial memory:

$$\mathsf{Enc}(EK_W, x_1), \ldots, \mathsf{Enc}(EK_W, x_N)$$

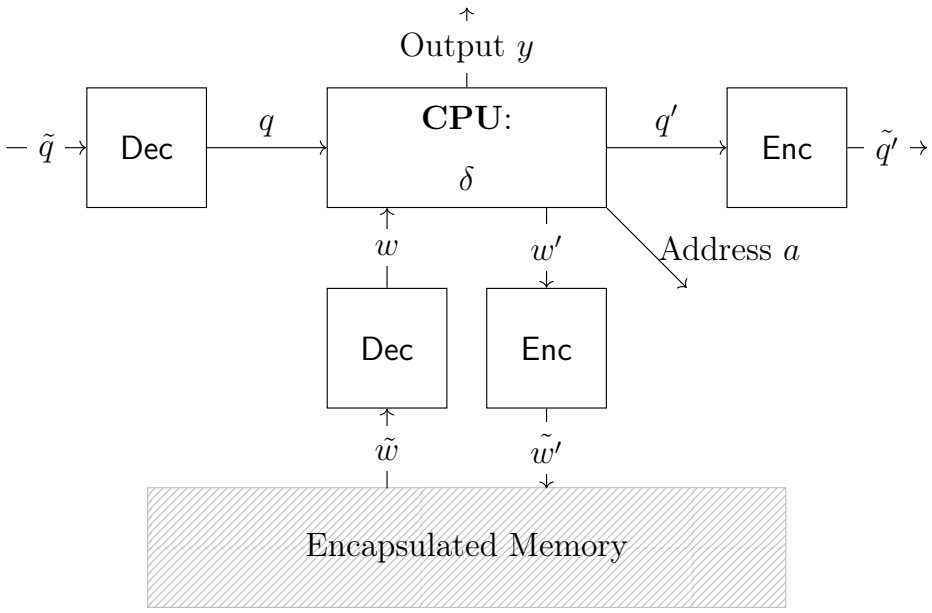

Figure 5-1: This circuit takes an encrypted state $\tilde{q}$ and an encrypted word $\tilde{w}$ as input. As outputs it may produce an encrypted state $\tilde{q'}$, an encrypted word $\tilde{w'}$, and a memory address a to access next. Alternatively, it may produce a final output y. w is interpreted as the word read from memory by q, while w' is the word written to memory by q. a is the location accessed by q'.

5.2 Preliminary Definitions

In this section we prove a general theorem giving conditions under which

$$\mathsf{Harden}(1^\lambda, (C_0, (\mathcal{M}_1, \ldots, \mathcal{M}_\ell)), (\vec{x}_0^1, \ldots, \vec{x}_0^\ell)$$

is indistinguishable from

$$\mathsf{Harden}(1^\lambda, (C_1, (\mathcal{M}_1, \ldots, \mathcal{M}_\ell)), (\vec{x}_1^1, \ldots, \vec{x}_1^\ell),$$

when C_0 and C_1 both map from the same domain $\mathcal{M}_{i_1} \times \cdots \mathcal{M}_{i_n}$ to the same codomain $\mathcal{M}_{j_1} \times \cdots \times \mathcal{M}_{j_m}$.

Definition 6 (Invariant Set)**.** Given a block B with encapsulated types $\mathcal{M}_1, \ldots, \mathcal{M}_\ell$,

as well as vectors $\vec{x}_1 \in \mathcal{M}_1^*, \ldots, \vec{x}_\ell \in \mathcal{M}_\ell^*$[1], we say that sets $(S_1 \subset \mathcal{M}_1, \ldots, S_\ell \subset \mathcal{M}_\ell)$ are *invariant* if:

1. For $i \in \{1, \ldots, \ell\}$, S_i contains each element of $\vec{x}_i$
2. $B(T_{i_1} \times \cdots \times T_{i_n}) \subset T_{j_1} \times \cdots T_{j_m}$, where we define

$$T_k = \begin{cases} S_k & \text{if } k \in \{1, \ldots, \ell\} \\ \mathcal{M}_k & \text{othrwise} \end{cases}$$

Remark 7. The minimal invariant sets are called the *reachable sets*, and have an intuitive definition: S_k is the set of all possible elements of $\mathcal{M}_k$ that can be reached by applying B to previously obtained inputs, if the unencapsulated inputs are allowed to be freely chosen.

The essential limitation of $\mathsf{i}\mathcal{O}$ compared to VBB obfuscation in our techniques is that we can only reason about succinctly described invariant sets, rather than precisely the reachable set. This is because we want our garbled programs to be succinct, and their size is determined by the size of the largest hybrid.

Definition 7. We say that a set S is *s-succinct* if membership in S is decidable by circuits with total size s.

Informally, the main condition for $\mathsf{Harden}(B_0)$ and $\mathsf{Harden}(B_1)$ to be indistinguishable is that B_0 and B_1 are in some sense "isomorphic" – that is, functionally equivalent up to permutations of the encapsulated types. Actually our theorem will apply when the isomorphism is succinctly described; we shall see that more interesting cases follow as a corollary by repeatedly applying this basic case.

Definition 8. We say that B_0 and B_1 with encapsulated types $\mathcal{M}_1, \ldots \mathcal{M}_\ell$ are *isomorphic* on invariant sets $S_1, \ldots, S_\ell$ by injections $\iota_j : S_j \hookrightarrow \mathcal{M}_j$ if on $T_{i_1} \times \cdots \times T_{i_n}$, if

$$(\iota_{j_1} \times \cdots \times \iota_{j_m}) \circ B_0 \equiv B_1 \circ (\iota_{i_1} \times \cdots \times \iota_{i_n})$$

[1]The notation S^* denotes the set of all finite sequences of elements in S.

where for $j \notin \{1, \ldots, \ell\}$, ι_j is defined as the identity function mapping $\mathcal{M}_j \to \mathcal{M}_j$. T_k is again defined as in Definition 6.

We will prove that $\mathsf{Harden}(B_0)$ and $\mathsf{Harden}(B_1)$ are indistinguishable if B_0 and B_1 are isomorphic on succinct invariant sets, with one caveat: each encapsulated type must have a timestamp, which is bounded by some polynomial in λ and increases on every execution of B_0 or B_1.

Definition 9. We say that a block B with encapsulated types $\mathcal{M}_1, \ldots \mathcal{M}_\ell$ is *ascending* if each encapsulated type has a timestamp attribute, and for all inputs to B, the encapsulated outputs always have larger timestamp than any encapsulated input. We say that B is T-*bounded* if B outputs $\perp$ whenever any of its inputs have timestamp greater than T.

5.3 Diamond Theorem

Theorem 2. *Suppose blocks B_0 and B_1 are isomorphic on s-succinct sets $S_1, \ldots, S_\ell$ by "singleton" injections $\iota_1, \ldots, \iota_\ell$. That is, each ι_j differs from the identity on at most one point. Suppose further that B_0 and B_1 are ascending and $\mathsf{poly}(\lambda)$-bounded, and $|\mathcal{M}_j| = \mathsf{poly}(\lambda)$ for each encapsulated type $\mathcal{M}_j$. Then*

$$\mathsf{Harden}(B_0, \vec{x}_1, \ldots, \vec{x}_\ell) \approx \mathsf{Harden}(B_1, \bar{\iota_1}(\vec{x}_1), \ldots, \bar{\iota_\ell}(\vec{x}_\ell)$$

where B_0 and B_1's circuits are both padded by an amount p which is polynomial in the succinctness parameter s, the security parameter λ, and the input / output lengths.

Proof. We give several indistinguishable hybrid distributions H_0 through H_6, starting with the left-hand side and ending with the right-hand side.

Hybrids Overview Let $S_1, \ldots, S_\ell$ be s-succinct invariant sets of B_0 and $\iota_1, \ldots, \iota_\ell$ be injections such that such that $(\iota_{j_1} \times \cdots \times \iota_{j_m}) \circ B_0 \equiv B_1 \circ (\iota_{i_1} \times \cdots \times \iota_{i_n})$ on $T_{i_1} \times \cdots \times T_{i_n}$.

1. First, we puncture the ACE keys (both encryption and decryption) for each encapsulated type $\mathcal{M}_j$ at $\mathcal{M}_j \setminus S_j$. This is an indistinguishable change by Lemma 3.

2. For each $j \in \{1, \ldots, \ell\}$, suppose ι_j maps $x_j \mapsto y_j$. We also puncture EK_j and DK_j at x_j and y_j. Where necessary to preserve functionality, we hard-code the correspondence $x_j \leftrightarrow \boxed{x_j}$ and $y_j \leftrightarrow \boxed{y_j}$. Here $\boxed{x_j}$ and $\boxed{y_j}$ denote $\mathsf{Enc}(EK_j, x_j)$ and $\mathsf{Enc}(EK_j, y_j)$ respectively. If $y_j \notin S_j$, we only hard-code $x_j \leftrightarrow \boxed{x_j}$ (not $y_j \leftrightarrow \boxed{y_j}$). This change is indistinguishable by $\mathsf{i}\mathcal{O}$.

3. In the hard-coded correspondence, we swap $\boxed{x_j}$ and $\boxed{y_j}$. We apply the same substitution to the element-wise encryptions of $\vec{x_1}, \ldots, \vec{x_\ell}$. This is indistinguishable by ACE's ciphertext indistinguishability, applied ℓ times – once for each encapsulated type.

4. In each hard-coded correspondence, we swap x_j and y_j. Simultaneously, we replace B_0 by B_1. This is indistinguishable by $\mathsf{i}\mathcal{O}$ because this is functionally equivalent to replacing B_0 by $\iota_{j_1}^{-1} \times \cdots \times \iota_{j_m}^{-1} \circ B_1 \circ \iota_{i_1} \times \cdots \times \iota_{i_n}$, where all encapsulated inputs inputs are guaranteed to be in the appropriate S_j.

5. For each $j \in \{1, \ldots, \ell\}$, we reduce the puncturing of EK_j and DK_j so that the keys for type $\mathcal{M}_j$ are punctured at $\mathcal{M}_j \setminus \iota_j(S_j)$ and remove the hard-coded correspondence $x_j \leftrightarrow \boxed{x_j}$ and $y_j \leftrightarrow \boxed{y_j}$. This is indistinguishable by $\mathsf{i}\mathcal{O}$.

6. For each $j \in \{1, \ldots, \ell\}$, we unpuncture EK_j and DK_j. Since $(\iota_1(S_1), \ldots, \iota_\ell(S_\ell))$ must be invariant sets of B_1 containing $\bar{\iota_1}(\vec{x}_1), \ldots, \bar{\iota_\ell}(\vec{x}_\ell)$, this is indistinguishable by Lemma 3. Now the distribution is exactly that of the right-hand side.

□

It now remains to prove indistinguishability of step 1 (and similarly, of step 6).

Lemma 3. *If B is an ascending block with s-succinct invariant sets $S_1, \ldots, S_\ell$ are containing $\vec{x}_1, \ldots, \vec{x}_\ell$, then*

$$\tilde{B}, \mathsf{Enc}^*(\vec{x}_1), \ldots, \mathsf{Enc}^*(\vec{x}_\ell) \approx \tilde{B}\{\mathcal{M}_j \setminus S_j\}, \mathsf{Enc}^*(\vec{x}_1), \ldots, \mathsf{Enc}^*(\vec{x}_\ell)$$

Here $\tilde{B}$ is constructed as in $\mathsf{Harden}(B)$, and $\tilde{B}\{\mathcal{M}_j \setminus S_j\}$ is the same but with the ACE keys for each encapsulated type $\mathcal{M}_j$ punctured at $\mathcal{M}_j \setminus S_j$.

Proof. We give a sequence of hybrid distributions starting with the left-hand side and ending with the right-hand side. In these hybrids we puncture the ACE keys on an increasing sequence of sets.

Definition of H_i. Define

$$Z_{j,i} = \{m \in \mathcal{M}_j : m\text{'s timestamp is less than } i \text{ and } m \notin S_j\}$$

Then H_i is defined as:

$$\tilde{B}\{Z_{1,i}, \ldots, Z_{\ell,i}\}, \mathsf{Enc}^*(\vec{x_1}), \ldots, \mathsf{Enc}^*(\vec{x_\ell})$$

It is easy to see that H_0 is the left-hand side of our desired indistinguishability, while H_{T+1} (where T is the maximum possible timestamp) is the right-hand side. So we just need to show that H_i is indistinguishable from H_{i+1}. This follows from the following two indistinguishable steps:

1. For each encapsulated type $\mathcal{M}_j$, we puncture its encryption key EK_j at $Z_{j,i+1}$. This is indistinguishable because in H_i, the decryption keys are punctured so that every input passed to B with timestamp less than i is in S_k for the appropriate k. By the invariance of S, every output of B with timestamp less than $i+1$ must also be in S_j. So nothing in $Z_{j,i+1}$ is ever encrypted. By ACE's equivalence of punctured encryption keys, puncturing the encryption key does not change functionality and is hence indistinguishable by $\mathsf{i}\mathcal{O}$.

2. For each encapsulated type $\mathcal{M}_j$, we puncture its decryption key DK_j at $Z_{j,i+1}$. This is indistinguishable by ACE's indistinguishability of punctured decryption keys, since the encryption key EK_j is already punctured at $Z_{j,i+1}$.

□

As stated earlier, isomorphism by singleton injections is a basic case which extends to isomorphism by other succinctly described injections. In particular, we have the following corollaries.

Corollary 1. *Let $\iota_1, \ldots, \iota_\ell$ be injections of size $p = O(1)$, meaning $\iota_j(x) \neq x$ for p values of x. If ascending blocks B_0 and B_1 are such that $(\iota_{j_1} \times \cdots \times \iota_{j_m}) \circ B_0 \equiv B_1 \circ (\iota_{i_1} \times \cdots \times \iota_{i_n})$ on s-succinct invariant sets of B_0, then*

$$\mathsf{Harden}(B_0, \vec{x_1}, \ldots, \vec{x_\ell}) \approx \mathsf{Harden}(B_1, \bar{\iota_1}(\vec{x_1}), \ldots, \bar{\iota_\ell}(\vec{x_\ell}))$$

where the padding of B_0 and B_1 is proportional to $s + p$.

Proof. This follows from the fact that any injection changing p points can be written as $\iota_1 \circ \cdots \circ \iota_p$ for injections $\iota_1, \ldots, \iota_p$, and then applying Theorem 2 p times. □

Corollary 2. *Suppose for each $j \in \{1, \ldots, \ell\}$, ι_j is an injection which is equal to $\theta_1 \circ \cdots \circ \theta_p$, where each θ_i is an injection of size $O(1)$ and p is $\mathsf{poly}(\lambda)$. Suppose further that for each i, $\theta_i \circ \cdots \circ \theta_p$ can be computed and inverted by circuits of size s. If ascending blocks B_0 and B_1 are isomorphic on s'-succinct invariant sets $S_1, \ldots, S_\ell$ by $\iota_1, \ldots, \iota_\ell$, then*

$$\mathsf{Harden}(B_0, \vec{x_1}, \ldots, \vec{x_\ell}) \approx \mathsf{Harden}(B_1, \bar{\iota_1}(\vec{x_1}), \ldots, \bar{\iota_\ell}(\vec{x_\ell}))$$

where the padding of $\tilde{B}_0$ and $\tilde{B}_1$ is proportional to $O(s + s')$.

Proof. This follows from applying Corollary 1 p times. □

Remark 8. While we have presented the diamond theorem for a single block, an analogous version for multiple blocks is provable using the same techniques. In spirit,

this is almost the same as combining the multiple blocks into a single block with an unencapsulated input selecting the block to execute.

Chapter 6

Garbling Iterated Functions

Our techniques are best understood in the application of garbling iterated functions, which may be of independent interest. In Chapter 8, we generalize the techniques presented here to garble RAM programs.

We want to garble a function of the form $f(x) = g^T(x)$, where g^T denotes g composed with itself T times. Garbling means encoding f as $\tilde{f}$ and x as $\tilde{x}$ such that:

1. (Correctness) One can compute $f(x)$ from $\tilde{f}, \tilde{x}$.

2. (Security) $\tilde{f}, \tilde{x}$ does not reveal side information other than T, $|g|$, and $|x|$. More precisely, there is a probabilistic polynomial-time algorithm Sim such that $\tilde{f}, \tilde{x}$ is computationally indistinguishable from $\mathsf{Sim}(f(x), T, |g|, |x|)$.

f and x are chosen selectively, which is why we can speak of $\tilde{f}, \tilde{x}$ as a distribution which depends only on f and x.

We will use the language of computational blocks which we developed in Chapter 5. An iterated function is just a computational block (g) repeated t times; our garbling will be the hardening of this block, along with the initial input $\boxed{x}$.

6.1 Construction

Given a function $f = g^T(x)$, we first define a block $B : \boxed{Q} \to \boxed{Q} \times X$,where X is the domain (and codomain) of the inner function g, and Q is $[T] \times X$. The notation

$\boxed{Q} \to \boxed{Q} \times X$ simply means that Q is the only encapsulated type.

$$B((t,x)) = \begin{cases} (t+1, g(x)), \bot & \text{if } t < T \\ \bot, q & \text{otherwise} \end{cases}$$

The garbling of f is $\mathsf{Harden}(B, (0,x))$ with $O(n + \log T)$ padding, where n is the bit-length of elements of X.

Correctness of this construction follows because evidently given $\tilde{B}$ and $\boxed{0, x}$, one can repeatedly evaluate $\tilde{B}$ obtaining $\boxed{1, g(x)}$, ..., $\boxed{T, g^T(x)}$, and finally $g^T(x) = f(x)$.

6.2 Security Proof

Theorem 3. *Assuming sub-exponentially secure $\mathsf{i}\mathcal{O}$ and sub-exponentially secure injective one-way functions, there is a PPT algorithm Sim such that for all circuits g, all T, all x, and all PPT $\mathcal{A}$,*

$$\Pr\left[\mathcal{A}(\tilde{f}_b, \tilde{x}_b) = b \,\middle|\, \begin{array}{l} b \leftarrow \{0,1\}; \tilde{f}_0, \tilde{x}_0 \leftarrow \mathsf{Garble}(g, T); \\ \tilde{f}_1, \tilde{x}_1 \leftarrow \mathsf{Sim}(g^T(x), T, |x|, |g|); \end{array}\right] < \frac{1}{2} + \mathsf{negl}(\lambda)$$

Proof. Let x_i denote $g^i(x)$. We show a sequence of indistinguishable hybrid distributions:

$$H_0 \approx H_1 \approx \cdots \approx H_T,$$

where H_T is efficiently sampleable given $g^T(x)$, $|x|$, $|g|$, and T.

Hybrid H_i. H_i is defined as $\mathsf{Harden}(B_i, (0, x_i))$, where

$$B_i((t,x)) = \begin{cases} (t+1, x), \bot & \text{if } t < i \\ (t+1, g(x)), \bot & \text{if } i \le t < T \\ \bot, x & \text{otherwise} \end{cases}$$

To show $H_i \approx H_{i+1}$, we introduce $i+1$ intermediate hybrids $H_{i,i}$ through $H_{i,0}$.

Hybrid $H_{i,j}$. $H_{i,j}$ is defined as $\mathsf{Harden}(B_{i,j}, (0, x_i))$, where

$$B_{i,j}((t,x)) = \begin{cases} (t+1, x), \bot & \text{if } t < j \text{ or } j < t < i+1 \\ (t+1, g(x)), \bot & \text{if } t = j \text{ or } i+1 \leq t < T \\ \bot, x & \text{otherwise} \end{cases}$$

Claim 1. *For each $i \in \{0, \ldots, T-1\}$, $H_i \approx H_{i,i}$.*

Proof. This follows from $\mathsf{i}\mathcal{O}$ because $B_{i,i}$ is functionally equivalent to B_i. □

Claim 2. *For each $i \in \{0, \ldots, T-1\}$ and $j \in \{1, \ldots, i\}$, $H_{i,j} \approx H_{i,j-1}$.*

Proof. We apply Theorem 2.

- Define

$$S = \{(t,x) \in Q : \text{if } t \leq j \text{ then } x = x_i \text{ and if } t = j+1 \text{ then } x = x_{i+1}\}$$

  S is clearly $O(n + \log T)$-succinct, and also $(0, x_i) \in S$.

- Let $\iota : S \hookrightarrow Q$ be the injection which maps (j, x_i) to (j, x_{i+1}), and is the identity elsewhere.

Invariance of S S is an invariant of $B_{i,j}$, as can be seen by casework. Suppose that $(t,x) \in S$ and that $B_{i,j}(t,x)$ is $q', \bot$.

1. If $t < j$, then x must be x_i. By definition of $B_{i,j}$, q' is $(t+1, x_i)$. $t+1$ is at most j, so q' is in S.
2. If $t = j$, then x must be x_i. By definition of $B_{i,j}$, q' is $(j+1, x_{i+1})$, which is in S.
3. If $t > j$, then q' has a timestamp whic is greater than $j+1$, so q' is vacuously in S.

Isomorphism of $B_{i,j}$ and $B_{i,j-1}$ on S For all $(t,x) \in S$, $((\iota \times \mathsf{id}) \circ B_{i,j})(t,x) = (B_{i,j-1} \circ \iota)(t,x)$, as can be seen by casework.

1. If $t < j-1$, then $B_{i,j}$ and $B_{i,j-1}$ are functionally identical, and outputs in Q that either produce have timestamp $t' < j$. This means that ι has no effect on either the input or the output to $B_{i,j-1}$, so $B_{i,j} \circ \iota$ and $(\iota \times \mathsf{id}) \circ B_{i,j-1}$ are functionally identical.

2. If $t = j-1$, then $x = x_i$. $((\iota \times \mathsf{id}) \circ B_{i,j})(t,x) = (j, x_{i+1}), \bot$. On the right-hand side, we compute $(B_{i,j-1} \circ \iota)(t,x)$. $\iota(t,x) = j-1, x_i$. $B_{i,j-1}(j-1, x_i) = (j, x_{i+1}), \bot$

3. If $t = j$, then $x = x_i$. $((\iota \times \mathsf{id}) \circ B_{i,j})(t,x) = (j+1, x_{i+1}), \bot$. On the right-hand side, we compute $(B_{i,j-1} \circ \iota)(t,x)$. $\iota((t,x)) = j, x_{i+1}$. $B_{i,j-1}(j, x_{i+1}) = (j+1, x_{i+1}), \bot$.

4. If $t > j$, then $B_{i,j}$ and $B_{i,j-1}$ are functionally identical, and outputs in Q that either produce have timestamp $t' > j$. This means that $\bar{\iota}$ has no effect on either the input or the output to $B_{i,j-1}$, so $B_{i,j}$ and $\bar{\iota} \circ B_{i,j-1} \circ \bar{\iota}$ are functionally identical.

Theorem 2 thus implies that $H_{i,j} \approx H_{i,j-1}$. □

Claim 3. *For each $i \in \{0, \ldots, T-1\}$, $H_{i,0} \approx H_{i+1}$.*

Proof. This follows from Theorem 2.

- Define
$$S = \{(t,x) : \text{if } t = 0 \text{ then } x = x_i\}.$$
S is clearly $O(n + \log T)$-succinct, and also $(0, x_i) \in S$.

- Let $\iota : S \hookrightarrow Q$ be the injection which maps $(0, x_i)$ to $(0, x_{i+1})$ and is the identity elsewhere.

Invariance of S. S is an invariant set of $B_{i,0}$ because $B_{i,0}$ never outputs an element of Q with timestamp 0, so $B_{i,0}$'s outputs in Q are always vacuously in S.

Isomorphism of $B_{i,0}$ and B_{i+1} on S. For all (t, x) in S, $((\iota \times \mathsf{id}) \circ B_{i,0})(t, x) = (B_{i+1} \circ \iota)(t, x)$, as can be seen by casework.

1. If $t = 0$, then $x = x_i$. Then $(\iota \times \mathsf{id})(B_{i,0}(t, x)) = (1, x_{i+1}), \perp$. On the other hand, we compute $(B_{i+1} \circ \iota)(t, x)$. $\iota((t, x)) = (0, x_{i+1})$. $B_{i+1}(0, x_{i+1}) = (1, x_{i+1}), \perp$.

2. If $t > 0$, $B_{i,0}$ and B_{i+1} are functionally equivalent and ι is the identity, so the equality is clear.

Theorem 2 then implies the claim. □

This concludes the security proof. □

Chapter 7

Weak RAM Garbling

In this chapter, we begin to consider RAM programs. We generalize the techniques from Chapter 6 to construct a weak notion of RAM garbling.

7.1 Definitions

7.1.1 RAM program

There are several ways to formalize what a RAM program is, usually by defining some succinct transition function. This transition function will have an input state and an output state, along with several inputs and outputs for reading or writing to memory. We simplify the transition function's definition by assuming that each memory access of the RAM program consists of a read followed by a write to the same address.

Defining the asymptotic behavior of a RAM program as input sizes grow is tricky, so we will suppose a RAM program is given with a fixed memory size N. The words stored at a memory address come from a word set W. W is commonly $\{0,1\}$, but when transforming RAM machines (as when garbling a RAM program), it will be useful to support a more general word set. When a RAM program terminates, it outputs an answer $y \in Y$. Y can be any set, but is often taken to be $\{0,1\}$.

Formally, we say that a RAM program Π is an integer N, finite sets Q, W, and Y, and a transition function $\delta_\Pi : Q \times W \to Q \times W \times [N] \sqcup Y$. A RAM program may

have an initial state $q_0 \in Q$, but without loss of generality, we will say that q_0 is $\perp$ (the real initial state can always be hard-coded in δ_Π).

7.1.2 Weak Garbling

Before constructing our full RAM garbling scheme, we will first construct a weak notion of garbling, satisfying the following property.

Definition 10. A weak garbling scheme is an algorithm $\mathsf{WkGarble}$ satisfying *correctness* and *indistinguishability security*, given below.

Correctness : For all RAM programs Π and inputs x,

$$\Pr\left[\tilde{\Pi}(\tilde{x}) = \Pi(x) \middle| \tilde{\Pi}, \tilde{x} \leftarrow \mathsf{WkGarble}(\Pi, x, 1^\lambda)\right] \geq 1 - \mathsf{negl}(\lambda)$$

Indistinguishability Security Let Π_0 and Π_1 be RAM programs, and let x_0 and x_1 be memory configurations. $\mathsf{WkGarble}(\Pi_0, x_0, 1^\lambda)$ and $\mathsf{WkGarble}(\Pi_1, x_1, 1^\lambda)$ are computationally indistinguishable if all of the following hold.

- The first t^* addresses $a_1, \ldots, a_{t^*}$ accessed by Π_0 on x_0 are the same as the first t^* addresses accessd by Π_1 on x_1. Furthermore, there is a small circuit Γ such that for $t < t^*$, $a_t = \Gamma(t)$.

- Either:

  $\Pi_0(\vec{x}_0)$ and $\Pi_1(\vec{x}_1)$ yield identical internal states and external memory contents immediately after the t^*-th step.

  $\Pi_0(\vec{x}_0)$ and $\Pi_1(\vec{x}_1)$ both give the same output immediately after the t^*-th step.

- The transition function of Π_0 and the transition function of Π_1, restricted to states with a timestamp of at least t^*, are functionally equivalent.

7.2 Construction

$\mathsf{WkGarble}(\Pi, \vec{x})$ takes a RAM program Π and a memory configuration $\vec{x}$ as input and outputs $\mathsf{Harden}(\Pi', \vec{x}')$, where Π' is another RAM program (to be defined), and $\vec{x}'$ is a memory configuration.

Π' is defined as a RAM program with Π hard-coded. Π' views memory as a complete binary tree (we will say location i stores a node whose children are at $2i$ and $2i+1$).

Π''s states take the form $(t, t_{exp}, i, i_{\text{exp}}, q_\Pi)$. t is the timestamp, t_{exp} is the timestamp that the currently accessed word must have. i is the address that Π is currently accessing, and i_{exp} is the location tag that the currently accessed word must have. q_Π is the current state of the emulated Π.

To execute an access of Π to address a_t, Π' accesses each node on the path to the a_t^{th} leaf. At each access, Π' expects a particular timestamp, and Π' will abort if the node it is given does not have that timestamp. Similarly, Π' knows what location it is accessing, and it will abort if the given node does not have that location tag.

Π' knows what timestamp to expect because in addition to a "self-timestamp", each non-leaf node contains timestamps for both of its children. The invariant is that a parent's timestamp for its child should match the child's self-timestamp. Π' maintains this invariant in the obvious way. In particular, when Π' accesses a node at time t, it writes that node back with two of its timestamps set to t - the self-timestamp, and the timestamp corresponding to whichever child Π' will access next.[1] Our timestamp convention is that a state with timestamp t writes a word with timestamp t, and the first timestamp is 1.

Say that $\vec{x} = (x_1, \ldots, x_N)$. Without loss of generality suppose that N is a power of two. Then we define $\vec{x}'$ as $(x'_1, \ldots, x'_{2N-1})$, where

$$x'_i = \begin{cases} (i, 0, 0, 0, \perp) & \text{if } i \leq N-1 \\ (i, 0, \perp, \perp, x_{i-(N-1)}) & \text{otherwise.} \end{cases}$$

[1] Π' is said to be "at time t" if Π's emulated state has timestamp t. Thus every node accessed in a tree traversal is written with the same self-timestamp.

Generally the components of $\vec{x}'$ take the form (i, t, t_L, t_R, v). We call i the location label, t the self-timestamp, t_L and t_R the timestamps of the left (respectively right) child, and we call v the stored value. Sometimes we say that a word is at leaf j to mean that its location label is $j + (n-1)$ (and therefore the word's value is x_j).

$\mathsf{WkGarble}$ outputs $\tilde{\Pi}, \tilde{\vec{x}} \leftarrow \mathsf{Harden}(\Pi', (\perp), \vec{x}')$.

7.3 Security Proof

It suffices to prove a simpler lemma: informally, that a RAM program performing j dummy steps is indistinguishable from a RAM program performing $j+1$ dummy steps.

Definition 11. Suppose Π is a RAM program on N words, and suppose Γ is a circuit mapping $[T] \to [N]$. For any $j \in [T]$, define $D_{j,q^*,\Gamma,\Pi}$ as a RAM program which executes the following steps:

1. At time $t \in \{1, \ldots, j-1\}$, access but don't modify address $\Gamma(t)$. At these times the state of $D_{j,q^*,\Gamma,\Pi}$ is $(t, \perp)$.

2. At time $t = j$, resume execution as Π, using q^* as the starting state and accessing whatever location q^* accesses.

3. At time $t > j$, just act as Π. At these times the state of $D_{j,q^*,\Gamma,\Pi}$ is (t, q_t) for some q_t which is a state of Π.

Lemma 4. *Let Π be a RAM machine which on state q_j modifies memory $\vec{x}_{j-1}$ into $\vec{x}_j$ in a single step by accessing location $\Gamma(j)$, resulting in state q_{j+1}. Then $\mathsf{WkGarble}(D_{j,q_j,\Gamma,\Pi}, \vec{x}_{j-1})$ is indistinguishable from $\mathsf{WkGarble}(D_{j+1,q_{j+1},\Gamma,\Pi}, \vec{x}_j)$*

Proof. We will let Π_0 denote the circuit that $\mathsf{WkGarble}(D_{j,q_j,\Gamma,\Pi})$ hardens and Π_1 denote the circuit that $\mathsf{WkGarble}(D_{j+1,q_{j+1},\Gamma,\Pi})$ hardens.

The lemma follows from an application of Corollary 2. Suppose that at location $\Gamma(j)$, $\vec{x}_{j-1}$ has value v_{j-1} while $\vec{x}_j$ has value v_j.

Definition of S_W If W'' is the set of words of Π_0, define $S_W \subset W''$ as the set of $w = (i, t, t_L, t_R, v)$ satisfying the following properties:

1. If $i = \Gamma(j)$ and $t < j$, then $v = v_j$.
2. If i is on the path to $\Gamma(j)$ and if $t \geq j$, then $t_L \geq j$ or $t_R \geq j$, whichever corresponds to the next node on the path to $\Gamma(j)$.

Definition of S_Q If Q'' is the set of states of Π_0, define $S_Q \subset Q''$ as the set of $q = (t, t_{exp}, i, i_{exp}, q_\Pi)$ satisfying the following properties:

1. q_Π is of the form (t, q_t) for some q_t which is a state of Π. Also $0 \leq t_{exp} < t$, and i_{exp} is a node on the path to i.
2. If $t < j$ then $q_\Pi = (t, \perp)$ and $i = \Gamma(t)$.
3. If $t = j$ then $q_\Pi = (j, q_j)$ and $i = \Gamma(j)$.
4. If $t \geq j$ and if i_{exp} is a node on the path to $\Gamma(j)$ then $t_{exp} \geq j$.

Definitions of ι_W and ι_Q Define the injection $\iota_W : S_W \hookrightarrow W''$ as

$$\iota_W((i, t, t_L, t_R, v) = \begin{cases} (i, t, t_L, t_R, v_{j+1}) & \text{if } i = \Gamma(j) \text{ and } t < j \\ (i, t, t_L, t_R, v) & \text{otherwise} \end{cases}$$

and $\iota_Q : S_Q \hookrightarrow Q''$ as

$$\iota_Q((t, t_{exp}, i, i_{exp}, q_\Pi) = \begin{cases} (t, t_{exp}, i, i_{exp}, (j, \perp)) & \text{if } t = j \\ (t, t_{exp}, i, i_{exp}, q_\Pi) & \text{otherwise} \end{cases}$$

It's not hard to see that S_W, S_Q are invariant sets of Π_0.

Isomorphism of Π_0 and Π_1: In order to apply Corollary 2, we must observe that for all $q = (t, t_{exp}, i, i_{exp}, q_\Pi) \in S_Q$ and all $w = (i_W, t_W, t_L, t_R, v) \in S_W$,

$$((\iota_Q \times \iota_W \times \mathsf{id} \times \mathsf{id}) \circ \Pi_0)(q, w) = (\Pi_1 \circ (\iota_Q \times \iota_W))(q, w) \tag{7.1}$$

This follows from casework: Without loss of generality $t_{exp} = t_W$, $i_{exp} = i_W$, because otherwise the Verifiable Reads transformation ensures that both $\Pi_0(q, w)$ and $\Pi_1(\iota_Q(q), \iota_W(w))$ output $\perp$. Let t' denote the timestamp of the output state q' of $\Pi_0(q, w)$, and let w' denote the written word.

1. If $t \leq j-1$ and $t' \leq j-1$, then $\Pi_0(q, \cdot)$ and $\Pi_1(q, \cdot)$ are functionally equivalent and $\iota_Q(q) = q$, $\iota_Q(q') = q'$. Furthermore, given that $t_W < j$ and $i_W = i_{exp}$, ι_W is "orthogonal" to both $\Pi_0(q, \cdot)$ and $\Pi_1(q, \cdot)$ in its action on memory words: ι_W acts only on the underlying value and location tag, while Π_0 and Π_1 ignore these attributes. So indeed $\Pi_1(\iota_Q(q), \iota_W(w))$ is equal to $((\iota_Q \times \iota_W \times \mathsf{id} \times \mathsf{id}) \circ \Pi_0)(q, w)$.

2. If $t = j-1$ and $t' = j$, then $\Pi_0(q, \cdot)$ and $\Pi_1(q, \cdot)$ are still functionally equivalent and $\iota_Q(q) = q$. S_Q is defined enough so that we know q' is of the form $(j, j-1, \Gamma(j), \mathsf{Root}, q_j)$, while Π_1 produces a new state of the form $(j, j-1, \Gamma(j), \mathsf{Root}, (j, \perp))$ which is exactly $\iota_Q(q')$.

   Again since both t_W and t'_W are less than j, an orthogonality argument shows that $\Pi_1(q, \iota_W(w))$ outputs a word which is $\iota_W(w')$, so Pi_1 also outputs a word which is $\iota_W(w')$.

3. If $t = j$ and $t' = j$, then $\Pi_0(q, \cdot)$ and $\Pi_1(\iota_Q(q), \cdot)$ are not accessing a leaf node, so they are functionally equivalent and $\iota_W(w) = w$ and $\iota_W(w') = w'$. Since q and $\iota_Q(q)$ aren't accessing a leaf node, Π_0's and Π_1's action on them is orthogonal to the action of ι_Q, so $\Pi_1(\iota_Q(q), \iota_W(w))$ is equal to $((\iota_Q \times \iota_W \times \mathsf{id} \times \mathsf{id}) \circ \Pi_0)(q, w)$.

4. If $t = j$ and $t' = j+1$, then by the constraints of S_Q and S_W, q is of the form $(j, t_{exp}, \Gamma(j), \Gamma(j), (j, q_j))$, and w is of the form $(\Gamma(j), t_{exp}, \perp, \perp, v_j)$ where $t_{exp} < j$.

   So we can compute

   $$\iota_W(w) = (\Gamma(j), t_{exp}, \perp, \perp, v_{j+1})$$

   and

   $$\iota_Q(q) = (j, t_{exp}, \Gamma(j), \Gamma(j), (j, \perp))$$

and verify that $\Pi_1(\iota_Q(q), \iota_W(w))$ is equal to $((\iota_Q \times \iota_W \times \mathsf{id} \times \mathsf{id}) \circ \Pi_0)(q, w)$.

5. If $t > j$, then Property 4 of S_Q ensures that $\iota_W(w) = w$ (otherwise $i_{exp} = i_W = \Gamma(j)$ and $t_W < j$, but $t_{exp} \geq j$). The timestamps of q, q', and w' are all at least $j + 1$, so it is also the case that $\iota_Q(q) = q$, $\iota_Q(q') = q'$ and $\iota_W(w') = w'$. So in Equation 7.1, the ι_Q's and ι_W's vanish, and it is sufficient to show $\Pi_0(q, w) = \Pi_1(q, w)$.

   But $\Pi_0(q, \cdot)$ and $\Pi_1(q, \cdot)$ are functionally equivalent, so this is true.

□

The full security property follows from Lemma 4 by a hybrid argument sketched in Figure 7-1. Lemma 4 states that paths differing by a diamond (or the left-most triangle) are indistinguishable, and it is easy to see that the top path can be transformed into the bottom path by a sequence of diamonds.

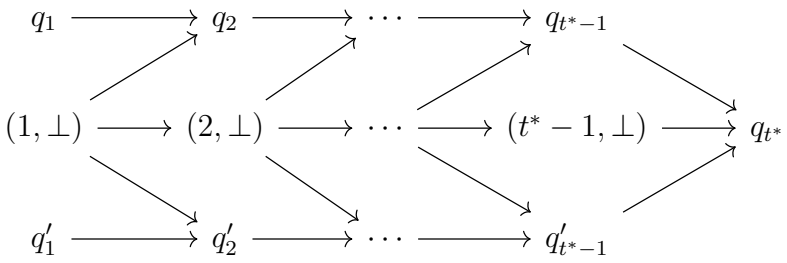

Figure 7-1: Each path of length t^* is a hybrid corresponding to some number of dummies followed by either Π_0 or Π_1. The initial memory configuration for the hybrid corresponding to a path is not shown, but depends on the first non-$\perp$ state.

Chapter 8

Full RAM Garbling

In this chapter, we construct a garbling scheme for RAM programs by combining the primitive of weak garbling from Chapter 7 and the strong oblivious RAM described in Appendix A. In contrast to weak garbling, here we hide the locations accessed by a RAM program, even if they are not locally computable.

A RAM program's input is given as an initial memory configuration $\vec{x} \in W^N$.

8.1 Oblivious RAM (ORAM)

An ORAM is a way of transforming a RAM program to have an oblivious access pattern – one in which the addresses accessed reveal nothing about the underlying program. This is often thought of as an atomic transformation of a program Π and memory configuration $\vec{x}$ into an oblivious program Π' and memory configuration $\vec{x}'$ given by $\mathsf{AddORAM}(\Pi, \vec{x})$, but we will think of it as an online transformation.

That is, at the beginning of the t^{th} underlying access, the ORAM has some state q_t. Given an underlying address a_t, the ORAM emulates an access to a_t by adaptively making η accesses to addresses $a'_{t,1}, \ldots, a'_{t,\eta}$, and then returning a value x_t and a new state q_{t+1}. We implicitly assume the ORAM has a small worst-case overhead η; in particular, η is $\mathsf{poly}(\log N)$. An ORAM also provides an Encode_{ORAM} procedure which encodes the initial memory $\vec{x}$ and produces an initial ORAM state q_0.

Correctness requires that for any initial memory $\vec{x}$, and for any sequence of under-

lying accesses $a_1, \ldots, a_t$, the values $x_1, \ldots, x_t$ returned by the ORAM are with high probability the same as if accesses $a_1, \ldots, a_t$ were executed non-obliviously on the unencoded $\vec{x}$.

In addition to a standard information-theoretic ORAM security property, we will require that the conditional distribution on the ORAM's internal state and external memory after t underlying steps can be efficiently sampled given the following values:

- The addresses that the ORAM accessed up to time t.
- The underlying memory contents at time t.
- The most recent time that each underlying memory address was accessed.

In Appendix A, we show that the ORAM construction of Chung and Pass satisfies our desired properties.

8.2 Construction of Garble:

On input $(\Pi, \vec{x}, 1^\lambda)$, Garble outputs $\mathsf{WkGarble}(\mathsf{AddORAM}(\Pi, \vec{x}, 1^\lambda), 1^\lambda)$. WkGarble and AddORAM are both functionality-preserving and separable transformations, so Garble is also functionality-preserving and separable.

8.3 Security Proof

We now show that our garbling scheme reveals nothing more than $\Pi(\vec{x})$, as well as the running time T, the memory size $|\vec{x}|$, and the program size $|\Pi|$. More precisely:

Theorem 4. *Assuming the existence of an injective one-way function and an* $\mathsf{i}\mathcal{O}$*-obfuscator for circuits, there is a PPT algorithm* Sim *such that for all RAM programs* Π*, all initial memory configurations* $\vec{x}$*, if* Π*'s running time*[1] *is* T*, then for all PPT*

[1] For now we speak only of worst-case running times, but achieving input-specific running times is also possible.

$\mathcal{A}$,

$$\Pr\left[\mathcal{A}(\tilde{\Pi}_b, \tilde{\vec{x}}_b, 1^\lambda) = b \;\middle|\; \begin{array}{l} b \leftarrow \{0,1\}, \tilde{\Pi}_0, \tilde{\vec{x}}_0 \leftarrow \mathsf{Garble}(\Pi, \vec{x}, 1^\lambda), \\ \tilde{\Pi}_1, \tilde{\vec{x}}_1 \leftarrow \mathsf{Sim}(\Pi(\vec{x}), T, |\vec{x}|, |\Pi|, 1^\lambda) \end{array}\right] < \frac{1}{2} + \mathsf{negl}(\lambda).$$

Proof Overview We give $2(T+1)$ hybrid distributions $H_0, H'_1, H_1, \ldots, H'_T, H_T$, followed by a qualitatively different hybrid H_{T+1}. H_0 will be exactly the distribution $\mathsf{Garble}(\Pi, \vec{x}, 1^\lambda)$ and H_{T+1} will be sampleable given only $\Pi(\vec{x})$, T, $|\vec{x}|$, and $|\Pi|$, and so we will define Sim as the algorithm which samples H_{T+1}.

Proof. Our hybrids are as follows.

Hybrid H_i $(0 \leq i \leq T)$. H_i is $\mathsf{WkGarble}(\Pi', \vec{x}'_i)$, where Π' and $\vec{x}'_i$ are sampled as follows:

1. Sample a puncturable PRF $F \leftarrow \mathsf{Gen}_{\mathsf{PRF}}(1^\lambda)$.
2. For $j = 1, \ldots, i$, define $I_j = \mathsf{OSample}(j; F(j))$.
3. Sample $q_{ORAM,i}, \vec{x}'_i \leftarrow \mathsf{Sim}_{ORAM}(\vec{t_i}, \vec{x}_i, I_1, \ldots, I_i)$. Here after i steps of executing Π on $\vec{x}$, $\vec{x}_i$ is the external memory contents and $\vec{t_i}$ is the vector of last-written times for each external memory locations.
4. Define Π' as a RAM program which accesses but does not modify locations $I_1, \ldots, I_i$ and then resumes emulation of Π as usual from the $i + 1^{th}$ step. Π' has $q_{ORAM,i}$ and F hard-coded. If Π on $\vec{x}$ terminates after i steps and produces an output $\Pi(\vec{x})$, then Π' has $\Pi(\vec{x})$ hard-coded. Otherwise, if Π's internal state after i execution steps is given by $q_{\Pi,i}$, then Π' has $q_{\Pi,i}$ and a_i hard-coded, where a_i is the address accessed by $q_{\Pi,i}$.

Hybrid H'_i $(0 \leq i \leq T)$ H'_i is $\mathsf{WkGarble}(\Pi', \vec{x}'_{i+1})$, where Π' and $\vec{x}'_{i+1}$ are sampled as follows:

1. Sample a puncturable PRF $F \leftarrow \mathsf{Gen}_{\mathsf{PRF}}(1^\lambda)$.

2. For $j = 1, \ldots, i$, define $I_j = \mathsf{OSample}(j; F(j))$.

3. Sample $q_{ORAM,i}, \vec{x}'_i \leftarrow \mathsf{Sim}_{ORAM}(\vec{t_i}, \vec{x}_i, I_1, \ldots, I_i)$. Here after i steps of executing Π on $\vec{x}$, $\vec{x}_i$ is the external memory contents and $\vec{t_i}$ is the vector of last-written times for each external memory locations.

4. Let I_{i+1} denote the (physical) memory addresses accessed by an ORAM on state $q_{ORAM,i}$ for underlying access a_i using random bits $F(i+1)$. Here a_i denotes the $i+1^{th}$ address accessed by the underlying Π when executed on $\vec{x}$. Let $q_{ORAM,i+1}$ and $\vec{x}'_{i+1}$ denote the resulting ORAM state and external memory configuration.

5. Define Π' as a RAM program which performs dummy accesses to locations $I_1, \ldots, I_{i+1}$ and then resumes emulation of Π as usual from the $i + 2^{th}$ step. Π' has $q_{ORAM,i+1}$ and F hard-coded. If Π on $\vec{x}$ terminates after $i + 1$ steps and produces an output $\Pi(\vec{x})$, then Π' has $\Pi(\vec{x})$ hard-coded. Otherwise, if Π's internal state after $i + 1$ execution steps is given by $q_{\Pi,i+1}$, then Π' has $q_{\Pi,i+1}$ and a_{i+1} hard-coded, where a_{i+1} is the address accessed by $q_{\Pi,i+1}$.

Hybrid H_{T+1}. H_{T+1} is $\mathsf{WkGarble}(\Pi'_{T+1}, \vec{x}'_{T+1})$, where Π'_{T+1} and $\vec{x}'_{T+1}$ are sampled as follows:

1. Sample a PPRF $F \leftarrow \mathsf{Gen}_{\mathsf{PRF}}(1^\lambda)$

2. Define Π'_{T+1} to make dummy accesses to $I_1, \ldots, I_T$, where $I_j = \mathsf{OSample}(j; F(j))$, and then output $\Pi(\vec{x})$. Π'_{T+1} has F, T, and $\Pi(\vec{x})$ hard-coded.

3. Define $\vec{x}'_{T+1}$ as the length-N' vector $(\perp, \ldots, \perp)$, where N' is the size of an ORAM-encoded length-N memory.

Claim 4. $H_i \approx H'_i$

Proof. Π'_i executed on $\vec{x}'_i$ and $\Pi'_{i,0}$ executed on $\vec{x}'_{i+1}$ have the same state, memory configuration, and functionality after the $i + 1^{th}$ underlying step, and they access the same memory locations (succinctly described by $\mathsf{OSample}$ and the PRF key K. So the security of $\mathsf{WkGarble}$ implies that $H_i \approx H'_i$. □

Claim 5. $H'_i \approx H_{i+1}$.

Proof. H'_i and H_{i+1} differ only in how $q_{ORAM,i+1}$, I_{i+1}, and $\vec{x}'_{i+1}$ are generated: In H_{i+1} they are the result of one honest step executed on a simulated $q_{ORAM,i}$, $\vec{x}_i$, while in H'_i they are just simulated. But because Sim_{ORAM} samples the correct conditional distribution on $q_{ORAM,i+1}, \vec{x}'_{i+1}$ given $I_1, \ldots, I_{i+1}$, these two distributions are the same. This is proved formally in Lemma 5 in Appendix A, which gives us the desired indistinguishability.[2]

□

Claim 6. $H_T \approx H_{T+1}$

Proof. This is applying the security of $\mathsf{WkGarble}$ again, since H_T and H_{T+1} access the same succinctly described set of locations. Specifically, at time i, they access the $i \pmod{\eta}$'th address of $\mathsf{OSample}(\lfloor \frac{i}{\eta} \rfloor; F(\lfloor \frac{i}{\eta} \rfloor))$. H_T and H_{T+1} also both give the same output $(\Pi(\vec{x}))$ after ηT execution steps, and for higher timestamps their transition functions are functionally equivalent (they always output $\perp$).[3] □

Hybrid H_{T+1} can be sampled given $\Pi(\vec{x})$, T, $|\vec{x}|$, and $|\Pi|$, which allows us to define Sim as the algorithm which samples H_{T+1}. This concludes the proof of Theorem 4. □

[2] H'_i actually hard-codes I_{i+1} while H_{i+1} computes I_{i+1} as $\mathsf{OSample}(i+1; F(i+1))$, but this difference is indistinguishable by a standard technique using the puncturability of F.

[3] Here we are simplifying the proof by assuming the execution time is given at garble-time. We can also achieve an input-specific running time T^* if we assume a prior polynomial bound T_{max} on the worst-case running time. We go through an intermediate hybrid, where we use Theorem 2 to indistinguishably change the transition function to be $\perp$ on all timestamps between T^* and T_{max}.

Bibliography

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

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

[AGVW13] Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In Ran Canetti and JuanA. Garay, editors, *Advances in Cryptology âĂŞ CRYPTO 2013*, volume 8043 of *Lecture Notes in Computer Science*, pages 500–518. Springer Berlin Heidelberg, 2013.

[AL11] Gilad Asharov and Yehuda Lindell. A full proof of the bgw protocol for perfectly-secure multiparty computation. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 18, page 36, 2011.

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

[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.

[BGT13] Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. Communication locality in secure multi-party computation. In Amit Sahai, editor, *Theory of Cryptography*, volume 7785 of *Lecture Notes in Computer Science*, pages 356–376. Springer Berlin Heidelberg, 2013.

[BGT14] Nir Bitansky, Sanjam Garg, and Sidharth Telang. Succinct randomized encodings and their applications. Cryptology ePrint Archive, Report 2014/771, 2014. http://eprint.iacr.org/.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10. ACM, 1988.

[BPR14] Nir Bitansky, Omer Paneth, and Alon Rosen. On the cryptographic hardness of finding a nash equilibrium. Cryptology ePrint Archive, Report 2014/1029, 2014. http://eprint.iacr.org/.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *Proceedings of the 2011 IEEE 52Nd Annual Symposium on Foundations of Computer Science*, FOCS '11, pages 97–106, Washington, DC, USA, 2011. IEEE Computer Society.

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

[CGP14] Ran Canetti, Shafi Goldwasser, and Oxana Poburinnaya. Adaptively secure two-party computation from indistinguishability obfuscation. Cryptology ePrint Archive, Report 2014/845, 2014. http://eprint.iacr.org/.

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

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

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[GGG+14] Shafi Goldwasser, S.Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In PhongQ. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology âĂŞ EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 578–602. Springer Berlin Heidelberg, 2014.

[GGH+13a] 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.

[GGH+13b] Sanjam Garg, Craig Gentry, Shai Halevi, Amit Sahai, and Brent Waters. Attribute-based encryption for circuits from multilinear maps. In *Advances in Cryptology - CRYPTO 2013*, volume 8043 of *Lecture Notes in Computer Science*, pages 479–499. Springer Berlin Heidelberg, 2013.

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

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

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

[GL89] Oded Goldreich and Leonid A Levin. A hard-core predicate for all one-way functions. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 25–32. ACM, 1989.

[GLOS14] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled ram from one-way functions. Cryptology ePrint Archive, Report 2014/941, 2014. http://eprint.iacr.org/.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[GPSW06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 89–98, New York, NY, USA, 2006. ACM.

[GVW13] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Attribute-based encryption for circuits. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 545–554, New York, NY, USA, 2013. ACM.

[KLW14] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. Cryptology ePrint Archive, Report 2014/925, 2014. http://eprint.iacr.org/.

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

[LP14] Huijia Lin and Rafael Pass. Succinct garbling schemes and applications. Cryptology ePrint Archive, Report 2014/766, 2014. http://eprint.iacr.org/.

[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.

[SS10] Amit Sahai and Hakan Seyalioglu. Worry-free encryption: Functional encryption with public keys. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 463–472, New York, NY, USA, 2010. ACM.

[SW05] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Advances in Cryptology âĂŞ EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 457–473. Springer Berlin Heidelberg, 2005.

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

[vDGHV09] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. Cryptology ePrint Archive, Report 2009/616, 2009. http://eprint.iacr.org/.

[Wat12] Brent Waters. Functional encryption for regular languages. Cryptology ePrint Archive, Report 2012/384, 2012. http://eprint.iacr.org/.

[Wat14] Brent Waters. A punctured programming approach to adaptively secure functional encryption. Cryptology ePrint Archive, Report 2014/588, 2014. http://eprint.iacr.org/.

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

Appendix A

Oblivious RAM

We describe the oblivious RAM of Chung and Pass [CP13], which is a simplification of [SCSL11], and we highlight the security property needed for our garbled RAM construction.

A.1 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 $O(1)$ words in the local state of Π', Chung and Pass start with a "basic" construction in which the local state of Π' consists of a "position map" $\mathsf{pos} : [N/\alpha] \to [N/\alpha]$ for some constant $\alpha > 1$.

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, or we erase a block once it is found, and write back the rest of 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. (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 $\gamma =$ 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:

Observation 1. Each oblivious READ and WRITE operation traverses the the tree along two randomly chosen paths, independent of the history of operations so far.

This 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, this position is not used in determining what nodes to traverse), and (2) the flushing, by definition, traverses a random path, independent of the history.

A.2 Security Property

Suppose an underlying access pattern is given; we will consider the randomized procedure of executing this sequence of accesses via this ORAM. We want a randomized "dummy" access algorithm $\mathsf{OSample}$ which on input j outputs a list of locations. This list should be distributed according to the real distribution of accesses corresponding to the j^{th} underlying access.

We can now describe our desired security property. Fix some underlying access pattern $a_1, \ldots, a_t$, including both the addresses and values written, and fix an initial memory configurtaion $\vec{x}$. Let Q_s be a random variable for the entire ORAM state (both private registers and memory configuration) after the s^{th} underlying access, and let I_j be a random variable for the addresses accessed by the ORAM on the j^{th} underlying access.

Lemma 5. *There exists a PPT algorithms* Sim *such that for any* s*, and any possible (non-zero probability) values* $i_1, \ldots, i_{s-1}$ *of* $I_1, \ldots, I_{s-1}$*, for all PPT adversaries* $\mathcal{A}$*,*

$$\Pr\left[\mathcal{A}(q_s^b, i_s^b) = b \;\middle|\; \begin{array}{l} q_{s-1}^0 \leftarrow \mathsf{Sim}(i_1, \ldots, i_{s-1}), q_s^0, i_s^0 \leftarrow \mathsf{OAccess}(a_s; q_{s-1}) \\ i_s^1 \leftarrow \mathsf{OSample}(s), q_s^1 \leftarrow \mathsf{Sim}(i_1, \ldots, i_s) \end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

This is a consequence of the following two claims.

Claim 7. I_s *is independent of* $I_1, \ldots, I_{s-1}$ *and* $a_1, \ldots, a_s$*, and is efficiently sampleable.*

Proof. This follows immediately from Observation 1. □

This lets us define $\mathsf{OSample}(s)$ as the efficient algorithm which samples I_s. Specifically, in the CP/SCSL ORAM, $\mathsf{OSample}$ samples and outputs two uniformly random paths in each tree.

Claim 8. *The conditional distribution $Q_s|I_1, \ldots, I_s$ is efficiently sampleable for all values of $I_1, \ldots, I_s$ (that jointly occur with non-zero probability).*

Proof. Recall that Q_s has two parts: a position map Pos and memory contents $\tilde{D}_s$, which are structured as a tree. We first give the sampling procedure for the basic construction, and then extend it to the recursive case. It is easy to verify that this procedure produces the correct distribution.

To sample Q_s given a sequence of memory accesses $(i_1, \ldots, i_s)$, do the following. For every memory block $b \in [N/\alpha]$, let $\tau_b \leq s$ be the last time when block b was accessed. 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 $p_b \leftarrow \lfloor N/\alpha \rfloor$. Compute the unique internal node γ_b such that γ_b is the largest common prefix between p_b and any of $I_{\tau_b}^{\mathsf{flsh}}, \ldots, I_s^{\mathsf{flsh}}$.
- Construct Pos by letting $\mathsf{Pos}(b) = p_b$.
- Construct $\tilde{D}_s$ by writing each memory block b together with its value at time s to the internal node γ_b.

To sample Q_s for the recursive construction, we note that this basic sampler doesn't need to $\vec{x}$ and the entire access pattern; it only needs to know each τ_b described above, as well as the memory contents at time s. This information $(\{\tau'_{b'}\}, \vec{x}'_s)$ for the next smaller recursive case is readily computable. $\vec{x}'_s$ for the next level ORAM is just Pos, which we have already computed. $\tau'_{b'}$ is the maximum of τ_b over all b corresponding to b'. So we can run the basic sampler repeatedly until we have sampled Q_s for the whole recursive construction. □

This allows us to define Sim as the above procedure for efficiently sampling Q_s conditioned on $I_1 = i_1, \ldots, I_s = i_s$.

We can now prove a stronger (statistical) version of Lemma 5, knowing that Sim and OSample output the correct conditional distributions.

Claim 9. $\Pr[q_s^0, i_s^0 = q_s, i_s] \approx \Pr[q_s^1, i_s^1 = q_s, i_s]$.

Proof.

$$\begin{aligned}
\Pr[q_s^0, i_s^0 = q_s, i_s] &\approx \mathbf{E}_{q_{s-1}}[\Pr[q_s, i_s | q_{s-1}, i_1, \ldots, i_{s-1}] | i_1, \ldots, i_{s-1}] \\
&= \Pr[q_s, i_s | i_1, \ldots, i_{s-1}] \\
&= \Pr[q_s | i_1, \ldots, i_s] \Pr[i_s | i_1, \ldots, i_{s-1}] \\
&= \Pr[q_s | i_1, \ldots, i_s] \Pr[i_s] \\
&\approx \Pr[q_s^1, i_s^1 = q_s, i_s].
\end{aligned}$$

1. The first approximate equality follows from Sim approximately sampling q_{s-1} given $i_1, \ldots, i_{s-1}$ and from OAccess (exactly) sampling q_s, i_s given q_{s-1}.

2. The second equality is just marginalization over q_{s-1}.

3. The third equality is the chain rule for probabilities.

4. The fourth equality is Claim 7 - namely that the locations accessed at time s are independent of previously accessed locations.

5. The fifth approximate equality follows from OSample approximately sampling i_s.

□