Fully Succinct Garbled RAM

Ran Canetti
Tel-Aviv University and Boston University
canetti@bu.edu

Justin Holmgren
MIT
holmgren@csail.mit.edu

ABSTRACT

We construct the first fully succinct garbling scheme for RAM programs, assuming the existence of indistinguishability obfuscation for circuits and one-way functions. That is, the size, space requirements, and runtime of the garbled program are the same as those of the input program, up to poly-logarithmic factors and a polynomial in the security parameter. The scheme can be used to construct indistinguishability obfuscators for RAM programs with comparable efficiency, at the price of requiring sub-exponential security of the underlying primitives.

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

The scheme builds on the recent schemes of Koppula-Lewko-Waters and Canetti-Holmgren-Jain-Vaikuntanathan [STOC 15]. A key technical challenge here is how to combine the fixed-prefix technique of KLV, which was developed for deterministic programs, with randomized Oblivious RAM techniques. To overcome that, we develop a method for arguing about the indistinguishability of two obfuscated randomized programs that use correlated randomness. Along the way, we also define and construct garbling schemes that offer only partial protection. These may be of independent interest.

Keywords

Obfuscation, Randomized Encodings, Garbling, Delegation

1. INTRODUCTION

A garbling scheme $\mathcal{G}$ converts programs and input values into “opaque” constructs that reveal nothing but the corresponding output values. That is, $\mathcal{G}$ turns a program M into a garbled program $\tilde{M}$ and, separately, turns a value x into a garbled input $\tilde{x}$, with the guarantee that $\tilde{M}(\tilde{x}) = M(x)$ and in addition the pair $(\tilde{M}, \tilde{x})$ reveals nothing but $M(x)$. Originally conceived by Yao [Yao86], garbling schemes are a pillar of cryptographic protocol design, with numerous ap-

plications such as secure two-party and multiparty computation protocols, verifiable delegation schemes, randomized encoding schemes, one time programs, and functional encryption.

A drawback of Yao’s original construction is that the size and runtime of the garbled program are proportional to the circuit representation of the input program. This holds even if the plaintext program is represented more succinctly, say as a Turing machine or a RAM program. (Essentially, one has to first translate the plaintext program to a circuit, and then apply Yao’s garbling method in a gate by gate manner.) This drawback becomes especially significant in situations where the input x is much larger than the program’s size or runtime — as in, say, keyword search in a large-but-sorted database — or when the runtime of the plaintext program varies from input to input.

Noticing this drawback, Goldwasser Kalai et al. [GKP⁺13] construct a garbling scheme for *Turing machines*, namely a scheme where the size, runtime and space requirements of the garbled program are proportional to those of the Turing machine representation of the plaintext program. To do that, they make strong *extractability* assumptions. Namely, they postulate existence of an efficient algorithm for extracting secrets from a certain class of adversaries.

Noticing the same drawback, Lu and Ostrovsky, and later Gentry Halevi et al. and Garg Lu et al. [LO13, GHL⁺14, GLOS15], construct garbling schemes for RAM programs, where the runtime of the garbled program is proportional only to the runtime of the plaintext program on that input. In [GLOS15] this is done assuming only one way functions. Still, the *size* of the garbled program is proportional to the *runtime* of the plaintext program.

Bitansky Garg et al. and Canetti Holmgren et al. construct a *semi-succinct* garbling scheme for RAM programs, assuming non-succinct Indistinguishability Obfuscation (IO) and injective one way functions [BGL⁺15, CHJV15]. That is, they construct garbling schemes where the space and runtime of the garbled program are proportional to the space and runtime of the plaintext program, and where the size of the garbled program is proportional to the *space* complexity of the plaintext program. For this they assume existence of non-succinct IO schemes, i.e. schemes where the complexity of the obfuscated program is polynomial in the size of the circuit representation of the plaintext program. (Indeed, current candidate indistinguishability obfuscators are such [GGH⁺13, BGK⁺14, Zim14, AB15].) We note that, although the overall parameters of these two schemes are roughly comparable, the underlying techniques are different.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ITCS’16, January 14–16, 2016, Cambridge, MA, USA.

© 2016 ACM. ISBN 978-1-4503-4057-1/16/01 ...\$15.00.

DOI: <http://dx.doi.org/10.1145/2840728.2840765>.

Koppula, Lewko and Waters [KLW15] devise a *fully succinct* garbling scheme for Turing machines from non-succinct IO and one way functions, using techniques that extend those of [CHJV15]. That is, in their garbling scheme the runtime, space and description size of the garbled program are proportional to those of the Turing machine representation of the plaintext program. This leaves open the following natural question:

Do there exist fully succinct garbling schemes for RAM programs? If so, under what assumptions?

Any advancement on this question directly applies to the many applications of succinct garbling mentioned in these works, including delegation of computation, functional encryption and others.

From succinct garbling to succinct obfuscation.

In [BGL⁺15, CHJV15] it is also shown how to turn a garbling scheme into a full-fledged program obfuscation scheme with comparable efficiency and succinctness properties, at the price of making stronger assumptions on the underlying cryptographic building blocks. That is, given non-succinct IO (namely IO for circuits), one-way functions, and a garbling scheme $\mathcal{G}$, they construct an IO scheme $\mathcal{O}$ with similar efficiency and size overhead as that for $\mathcal{G}$. The security of $\mathcal{O}$ loses a factor of D , where D is the size of the domain of inputs to the plaintext program. Using this transformation, and assuming sub-exponential one-way functions and IO for circuits, [BGL⁺15, CHJV15] show a semi-succinct IO scheme for RAM machines and [KLW15] show a fully succinct IO scheme for Turing machines. However:

Is there a fully succinct IO scheme for RAM programs? If so, under what assumptions?

We note that, due to the exponential degradation in security in that transform, the security parameter needs to grow linearly with $\log D$. The size of the obfuscated program thus grows polynomially in the length of input to the plaintext program. We only know how to get below this bound under significantly stronger assumptions on the underlying obfuscation scheme [BCP14, IPS15].

1.1 Our contribution

We answer both questions. Given an IO scheme for circuits and one way functions we construct a fully succinct garbling scheme for RAM programs. That is, the runtime, space, and size of the garbled program are the same as those of the plaintext program, up to polylogarithmic factors and a polynomial in the security parameter. The security of the scheme degrades polynomially with the runtime of the plaintext program. Assuming quasipolynomial security of the underlying primitives, the scheme guarantees full security even for programs with arbitrary polynomial runtime. Using the transformation of [BGL⁺15, CHJV15], and assuming sub-exponential security of the underlying primitives, we obtain a fully succinct IO scheme for RAM programs.

Furthermore, similarly to the schemes of [CHJV15, BGL⁺15, KLW15], our garbling scheme extends easily to support persistent data: Multiple machines $M_1, \dots, M_\ell$ can be garbled, along with some (potentially very large) data, such that machine M_i acts on the data configuration left by M_{i-1} , and

such that having access to the garbled data and garbled machines gives no information other than $y_1, \dots, y_\ell$, where y_i is the output of M_i when executed in sequence on the data after $M_1, \dots, M_{i-1}$. Importantly, in our case of RAM machines, each machine can run in time that is *sublinear* in the entire data; for example, each machine may execute a database query, modifying the database and returning some small result. Our transformation preserves the sub-linear complexity of the machines.

The preservation of sublinear complexity is powerful also in the context of delegation of computation. Indeed, consider the task of delegating the computation of sublinear-time RAM programs over large delegated databases. Indeed, when instantiated with our scheme, the delegation of computation scheme for RAM-IO (described in [CHJV15]) is the *first* to guarantee both correctness and *privacy* of the computation, while preserving full succinctness and *sublinear* complexity for the prover.¹

We note that in this work we achieve security only when the RAM machine M and database x are chosen ahead of time. In a follow-up work [CCHR15], it is shown how to achieve security against adaptively chosen RAM machines.

Our Techniques.

While our result may come across as natural and expected given the results of [KLW15] and [CHJV15], obtaining it does require new ideas and significant work. Indeed, naive attempts to extend the techniques of [KLW15] to RAM programs encounter the following problem: The [KLW15] technique applies when the plaintext machine is deterministic and its memory access pattern is fixed and independent of the inputs. When the plaintext program is a Turing machine, making sure that the memory access pattern is fixed incurs only small overhead in complexity. In contrast, hiding the memory access pattern in a RAM program in an efficiency-preserving way requires the memory access pattern to be randomized. Indeed, this is the case for Oblivious RAM schemes [GO96]. Furthermore, the security guarantees provided by Oblivious RAM (ORAM) schemes hold only when the internal random choices of the scheme are hidden from the adversary. However, in our case these internal random choices are encapsulated in a succinct program that is only protected by indistinguishability obfuscation.

A second look reveals the following basic discrepancy between the [CHJV15] technique (which is ORAM-friendly) and the [KLW15] technique (which is not). In both works, security of the garbled program is demonstrated by gradually moving, in a way that's indistinguishable to the adversary, from the real garbled program to a dummy garbled program, where the dummy program has just the result hardwired and is running a fake computation in all steps but the last one. In [CHJV15], the intermediate, hybrid programs start with some number, i , of dummy steps, and then continue the computation from the i th intermediate configuration all the way to the end. To make this technique work with ORAM, [CHJV15] use an ORAM scheme with a strong *forward security* property: Essentially, the addresses accessed before time i must appear independent of

¹In particular, the “generic” method of guaranteeing privacy in delegation schemes by encrypting the data using fully homomorphic encryption incurs a super-linear complexity overhead. We thank Yael Kalai for this observation.

the underlying access pattern, even given the scheme’s internal state at time $i + 1$.

In contrast, [KLW15] move from the real garbled program to the dummy one via intermediate programs that perform the computation from the beginning until some step, i . From then on, the intermediate program performs the dummy computation and in the end it outputs its hardwired value. This reversal of the order of steps in the intermediate programs is the key idea that allows the size of their garbled program to not depend on the space requirements of the plaintext program. However, this new structure of the hybrid programs seems incompatible with ORAM techniques: Indeed, the natural way to extend the [KLW15] argument to this case would be to argue that the program’s memory access pattern at steps i and up is random even given the program’s state at steps 1 through $i - 1$. But this does not hold, since all the steps of the computation up to the transition point i are executed, including the internal random choices of whatever ORAM scheme is in use.

Our first step towards getting around this difficulty is to identify the following property of ORAM schemes. Recall that an ORAM scheme translates the memory access requests made by the underlying program to randomized locations in the actual external memory. We say that an ORAM scheme has **localized randomness** if the random variable describing the physical location of the memory cell accessed by the plaintext program at a certain step of the computation depends only on a relatively small portion of the entire random input of the ORAM scheme. Furthermore, we require that the location of this portion depends only on the last step in which this memory cell was accessed, which in of itself is a deterministic function of the underlying program. To the best of our knowledge, this property of Path ORAMs has not been utilized in previous work, but we observe that the ORAM of [CP13] has localized randomness. (In fact, it seems likely that other schemes do as well, or can be slightly modified to be so.) Now, given an ORAM scheme with localized randomness, we “puncture” the scheme at exactly the points that are necessary for making the external memory access locations at step i appear random even given the punctured program state at step $i - 1$. Furthermore, we can perform this puncturing with minimal overhead in terms of the size of the obfuscated program.

More concretely, we proceed in two main steps. (The actual construction goes through a number of smaller steps, for sake of modularity and clarity.) We first build a “fixed-address” garbler which guarantees that the garbled versions of two machines M_0 and M_1 with inputs x_0 and x_1 are indistinguishable as long they access the same sequence of addresses. We believe that this property is of independent interest. In the second step we use an ORAM scheme with localized randomness to obtain full garbling. Below we provide more detail on these two steps.

1.1.1 Fixed Address Garbling

As an intermediary step towards a fully succinct garbling scheme for RAM programs, we define and obtain the following weaker security property for garbling schemes. We say that a garbling scheme is a *fixed-address garbler* if for any two same-size deterministic programs M_0 and M_1 , and any same-length input values x_0 and x_1 , it holds that $(\tilde{M}_0, \tilde{x}_0) \approx (\tilde{M}_1, \tilde{x}_1)$ as long as (a) $M_0(x_0) = M_1(x_1)$ and (b) The sequence of memory addresses accessed by M_0 when run on

x_0 is identical to the sequence of memory addresses accessed by M_1 when run on x_1 . (Here $\tilde{M}$ and $\tilde{x}$ are the garbled versions of M and x , respectively.) Furthermore, the sequence of addresses accessed by $\tilde{M}$ on input $\tilde{x}$ is identical to the sequence of addresses accessed by M on input x .

The fact that $\tilde{M}$ preserves the access pattern of M provides potential efficiency and practical applicability gains that are not possible in the context of fully secure and succinct garbling of RAM programs, since in the latter the access pattern is inherently randomized. For instance, the garbled machine necessarily has the same fine-grain cache performance as the original one. In contrast, ORAM-based techniques need to resort to coarse-grain cache or other workarounds which impact cache performance.

We construct a fully succinct fixed-address garbling scheme. As a preliminary step, we construct a garbling scheme that is fixed-address, except that $(\tilde{M}_0, \tilde{x}_0) \approx (\tilde{M}_1, \tilde{x}_1)$ only when the two computations have the exact same memory access pattern, including the *contents* of the memory cells accessed. (We call such schemes *fixed-memory* garbling schemes.) Here our technique follows the steps of the [KLW15] machine-hiding encoding scheme. In particular we use the same underlying primitives, namely positional accumulators, cryptographic iterators, and splittable signatures. (We somewhat simplify their interfaces.) We note however that the [KLW15] construction cannot be used in a “black box” way and needs to be redone in the RAM model.

We then move from fixed-memory garbling to fixed-address garbling. Similarly to the move in [KLW15] from machine-hiding encoding to garbling, this step requires encrypting the memory contents in an IO-friendly scheme. We stress however that our situation is different: In their oblivious Turing machine model the memory access pattern contains no information. In contrast, as argued in more detail below, in our case the access pattern can in of itself contain information that is hard to compress in a security-preserving manner. The way we argue about the security of the scheme must change accordingly.

Concretely, to garble M we transform it to a program M' which interleaves *two* executions of M , on two parallel tracks ‘A’ and ‘B’ of memory. Whenever M would access a memory address, M' accesses the corresponding address in both tracks ‘A’ and ‘B’. At each point in time, tracks ‘A’ and ‘B’ both store memory contents corresponding to an execution of M . We then apply the fixed-memory garbling scheme to M' . Let $\tilde{M}'$ denote the resulting program.

To argue fixed-address security, consider two programs M_0 and M_1 and input values x_0 and x_1 that satisfy the fixed-address requirements. To show that $(\tilde{M}_0', \tilde{x}_0) \approx (\tilde{M}_1', \tilde{x}_1)$, we consider an intermediate hybrid in which M_0' is replaced by a new machine M_{01} which now executes M_0 on track ‘A’ but M_1 on track ‘B’. Indistinguishability of the intermediate hybrid from either end is shown by demonstrating how to indistinguishably switch from a machine that outputs the result of track ‘A’ to a machine that outputs the result of track ‘B’.

1.1.2 Full Garbling

Our final and main step is a construction of a succinct fully secure garbler for RAM machines from a succinct fixed-address garbler. Our construction is fully general; it does not use any special properties of the fixed-address garbler,

not even the address-preserving property which we explicitly highlighted above.

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

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

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

For this purpose, we use the *localized randomness* property sketched above and described in more detail here. Localized randomness requires a particularly structured relationship between the random tape R of an ORAM and the addresses $\mathbf{a}_1, \dots, \mathbf{a}_t$ that it accesses. Here each $\mathbf{a}_i$ is itself a sequence of addresses $a_{i,1}, \dots, a_{i,\eta}$, accessed in the emulation of the underlying RAM machine’s i^{th} step. Specifically, we require that (for given underlying memory operations $\text{op}_1, \dots, \text{op}_t$), each $\mathbf{a}_i$ is influenced only by a small subset D_i of the bits of R , and each bit of R influences at most one of $\mathbf{a}_i$. The ORAM must also come with a p.p.t. algorithm OSample such that $\text{OSample}(i)$ has the same distribution as $\mathbf{a}_i$, independently of $\text{op}_1, \dots, \text{op}_t$. A simple analysis in the supporting materials (Appendix A) shows that the ORAM of Chung and Pass [CP13, SCSL11] satisfies this locality property.

To analyze the composition of a fixed-address garbler with a localized-randomness ORAM, we adapt the punctured programming technique of [SW14]. To simulate a garbled program whose output is y and runs in time T , apply a fixed-address garbler to the program that for each i from 1 to T , simulates addresses $\mathbf{a}_i$ to access using $\text{Sim}(F(i))$ for some puncturable PRF F , and output the resulting garbled program. We need to prove that this simulation is indistinguishable from the real garbled machine M , in a sequence of hybrids which changes each $\mathbf{a}_i$ to $\text{Sim}(F(i))$.

This argument is reminiscent of the proof of security for the [CLTV15] construction of a probabilistic iO (PIO) obfuscator, with the complication that $\mathbf{a}_1$ through $\mathbf{a}_t$ are generated adaptively. This complication is handled by switching the $\mathbf{a}_i$ ’s in reverse order – starting with $\mathbf{a}_t$ and ending with $\mathbf{a}_1$. Here it is crucial to note that, despite the adaptivity, $\mathbf{a}_1$ through $\mathbf{a}_t$ are mutually independent random variables by the localized randomness property of the ORAM scheme.

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

1.2 Related Work

In an independent and concurrent work, Chen et al. [CCC⁺15] also show how to garble RAM and even develop new tools to garble *parallel* RAM programs. In the RAM setting, our constructions are essentially the same. Our proof of security is significantly more modular however, while their analysis achieves a slightly better concrete security parameter.

1.3 Roadmap

As mentioned, we build up our main construction in four stages, at each stage strengthening the security properties. In the first two stages, we directly apply the techniques of [KLW15] to produce a very weak garbling scheme for RAM machines. For ease of exposition, we separate this into two parts: In Section 3, we give a garbler which only guarantees indistinguishability of the garbled programs as long as the entire execution transcripts of the two plaintext machines look identical; that is, if they specify the same sequence of internal local states, same addresses accessed, and same values written to memory. We call such schemes *fixed transcript garblers*. In Section 4, we upgrade this garbling scheme to a *fixed-memory garbler*, which no longer needs the machines to have the same internal local states.

Our main technical contributions are the construction of a *fixed-address garbler* in Section 5, and its combination with a local ORAM in Section 6 to build a full RAM garbler. Section 7 presents the application to persistent data.

2. PRELIMINARIES

2.1 Notation

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

2.2 Indistinguishability Obfuscation

We assume the existence of an indistinguishability obfuscator [BGI⁺01, GGH⁺13].

Syntax.

An indistinguishability obfuscator for circuits is a p.p.t. algorithm $i\mathcal{O}$ which takes as input a security parameter 1^λ , a circuit C , and outputs a circuit $\tilde{C}$.

Correctness.

For all x , $\Pr[i\mathcal{O}(1^\lambda, C)(x) = C(x)] = 1$.

Security.

If $|C_0| = |C_1|$ and $C_0(x) = C_1(x)$ for every x , then

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

2.3 The RAM Model

2.3.1 RAM Machines

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

- Σ is a finite set, which is the possible contents of a memory cell. For example, $\Sigma = \{0, 1\}$.
- Q is the set of all possible “local states” of M , containing some initial state q_0 . (We think of Q as a set that grows polynomially as a function of the security parameter. That is, a state $q \in Q$ can encode cryptographic keys, as well as “local memory” of size that is bounded by some fixed polynomial in the security parameter.)
- Y is the output space of M .
- C is a circuit implementing a transition function which maps $Q \times (\Sigma \cup \{\epsilon\}) \rightarrow (Q \times O_\Sigma) \cup Y$. Here O_Σ denotes the set of memory operations with Σ as the alphabet of possible memory symbols. Precisely, $O_\Sigma = (\mathbb{N} \times \Sigma)$. That is, C takes the current state and the value returned by the memory access function, and returns a new state, a memory address, a read/write instruction, and a value to be written in case of a write.

We write $|M|$ to denote the tuple $(\ell_\Sigma, \ell_Q, \ell_Y, |C|)$, where ℓ_Σ is the length of a binary encoding of Σ , and similarly for ℓ_Q and ℓ_Y .

2.3.2 Memory Configurations

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

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

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

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

2.3.3 Execution

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

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

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

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

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

2.4 Garbling

Syntax.

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

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

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

Correctness.

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

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

Efficiency.

A garbling scheme is said to be efficient if:

1. KeyGen , GbPrg , and GbMem are all p.p.t. algorithms. In particular, we emphasize that:

- The bounds T and S are encoded in binary, so the time to garble does not significantly depend on either of these quantities.
- The running time of **GbMem** is polynomial in $\|s\|_0$, the number of non-empty addresses in s . In fact in our scheme the running time is linear in $\|s\|_0$.

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

Security.

A garbling scheme is said to be *secure* if there is an efficient algorithm **Sim** such that for all RAM machines M and memory configurations s with $\text{Time}(M, s) \leq T$ and $\text{Space}(M, s) \leq S$, no p.p.t. algorithm can distinguish

$$\tilde{M}, \tilde{s}$$

from

$$\text{Sim}(1^\lambda, M(s), \text{Time}(M, s), T, S, |M|, \|s\|_0).$$

in the probability space defined by sampling

$$\begin{aligned} SK &\leftarrow \text{KeyGen}(1^\lambda, S, T) \\ \tilde{M} &\leftarrow \text{GbPrg}(SK, M) \\ \tilde{s} &\leftarrow \text{GbMem}(SK, s) \end{aligned}$$

3. FIXED-TRANSCRIPT GARBLING

We first construct a garbling scheme with a very weak security definition. Both the construction and the security proof closely follow the techniques of [KLW15], adapting them to RAM machines.

DEFINITION 1. A *garbling scheme* $(\text{KeyGen}, \text{GbPrg}, \text{GbMem})$ is said to be *fixed-transcript secure* if for all RAM machines M_0 and M_1 and all memory configurations s such that:

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

for all p.p.t. algorithms $\mathcal{A}$,

$$\Pr \left[\mathcal{A}(1^\lambda, \tilde{M}_b, \tilde{s}) = b \right] \leq \frac{1}{2} + \text{negl}(\lambda)$$

in the probability space defined by sampling

$$\begin{aligned} SK &\leftarrow \text{KeyGen}(1^\lambda, S, T) \\ b &\leftarrow \{0, 1\} \\ \tilde{M}_b &\leftarrow \text{GbPrg}(SK, M_b) \\ \tilde{s} &\leftarrow \text{GbMem}(SK, s). \end{aligned}$$

THEOREM 1. If one-way functions and an indistinguishability obfuscator for circuits exist, then there is a fixed-transcript secure garbling scheme for RAM machines.

In the full version [CH15] we include this theorem's proof, which closely follows the machine-hiding encoding construction of [KLW15].

4. FIXED MEMORY GARBLING

We now use a fixed transcript garbling scheme to satisfy a slightly stronger notion which we call fixed-memory garbling. In fixed-memory garbling, the garblings of two different machines are indistinguishable as long as the memory accesses are the same. Notably, it is possible for the two machines to have differing local states.

DEFINITION 2 (FIXED MEMORY SECURITY). A *garbling scheme* $(\text{KeyGen}, \text{GbPrg}, \text{GbMem})$ is said to be *fixed-memory secure* if for all RAM machines M_0 and M_1 , memory configurations s , all time bounds T and space bounds S satisfying:

- $\text{Time}(M_0, s) \leq T$
- $\text{Space}(M_0, s) \leq S$
- $M_0(s) = M_1(s)$
- $|M_0| = |M_1|$
- *Writing* $\mathcal{T}(M_0, s) = ((q_0, a_0, v_0), \dots, (q_{t-1}, a_{t-1}, v_{t-1}))$ and $\mathcal{T}(M_1, s) = ((q'_0, a'_0, v'_0), \dots, (q'_{t'-1}, a'_{t'-1}, v'_{t'-1}))$, it holds that $t = t'$ and for each $i \in [t]$, $a_i = a'_i$ and $v_i = v'_i$.

it holds that for all p.p.t. adversaries $\mathcal{A}$

$$\Pr \left[\mathcal{A}(1^\lambda, \tilde{M}_b, \tilde{s}) = b \right] \leq \frac{1}{2} + \text{negl}(\lambda)$$

in the probability space defined by sampling

$$\begin{aligned} SK &\leftarrow \text{KeyGen}(1^\lambda, T, S) \\ b &\leftarrow \{0, 1\} \\ \tilde{M}_b &\leftarrow \text{GbPrg}(SK, M) \\ \tilde{s} &\leftarrow \text{GbMem}(SK, s) \end{aligned}$$

4.1 Construction

Given a garbling scheme $(\text{KeyGen}', \text{GbPrg}', \text{GbMem}')$ satisfying fixed transcript security, we build a garbling scheme $(\text{KeyGen}, \text{GbPrg}, \text{GbMem})$ satisfying fixed-memory security. All we need to do is mask the internal state for each time-step with a different pseudorandom value.

CONSTRUCTION 1. We define $(\text{KeyGen}, \text{GbPrg}, \text{GbMem})$:

- **KeyGen** $(1^\lambda, T, S)$ samples $K' \leftarrow \text{KeyGen}'(1^\lambda, T, S)$ and a puncturable PRF F , and outputs $K = (K', F, T, S)$.
- **GbPrg** $(K, M = (\Sigma, Q, Y, C))$ samples and outputs

$$\tilde{M} \leftarrow \text{GbPrg}'(K', M' = (\Sigma, Q', Y, C')),$$

where $Q' = [T] \times \{0, 1\}^{\ell_Q}$, and C' is defined in Algorithm 1. q'_0 , the initial state for M' , is defined as $(0, F(0) \oplus q_0)$,

- **GbMem** (K, s) outputs $\text{GbMem}'(K', s)$.

4.2 Proof of Security

THEOREM 2. If $(\text{KeyGen}', \text{GbPrg}', \text{GbMem}')$ is a fixed transcript secure garbling scheme, then Construction 1 defines a fully succinct, efficient, fixed memory secure garbling scheme for RAM machines.

A proof is given in the full version of this paper [CH15].

Input: state q , memory symbol σ
Data: Puncturable PRF F , underlying transition function C

```

1 Parse  $q$  as  $(t, c_q)$ ;
2  $q_{in} := F(t) \oplus c_q$ ;
3  $out := C(q_{in}, \sigma)$ ;
4 if  $out \in Y$  then return  $out$ ;
5 else
6   Parse  $out$  as  $(q_{out}, op)$ ;
7   return  $((t + 1, F(t + 1) \oplus q_{out}), op)$ ;
8 end

```

Algorithm 1: Transition function C'

5. FIXED ADDRESS GARBLING

We now use a fixed memory garbling scheme to construct a slightly stronger notion of garbling. Namely, we will now hide the *data* in memory, but not yet the addresses which are accessed. As discussed in the introduction, in applications where the memory access pattern is known not to leak sensitive information, this notion of garbling may be significantly more efficient. In particular, it preserves the efficacy of cache, for which real-world RAM programs are extensively optimized.

DEFINITION 3. A garbling scheme $(\text{KeyGen}, \text{GbPrg}, \text{GbMem})$ is said to be fixed-address secure if for all RAM machines M , memory configurations s , time bounds T , and space bounds S satisfying the following conditions:

- $\text{Space}(M_0, s_0) \leq S$ and $\text{Time}(M_0, s_0) \leq T$.
- $\{a : s_0(a) \neq \epsilon\} = \{a : s_1(a) \neq \epsilon\}$
- If $\mathcal{T}(M_0, s) = ((q_0, a_0, v_0), \dots, (q_{t-1}, a_{t-1}, v_{t-1}))$ and $\mathcal{T}(M_1, s) = ((q'_0, a'_0, v'_0), \dots, (q'_{t'-1}, a'_{t'-1}, v'_{t'-1}))$, then $t = t'$ and for each $i \in [t]$, $a_i = a'_i$.
- $M_0(x_0) = M_1(x_1)$
- $|M_0| = |M_1|$

for all p.p.t. adversaries $\mathcal{A}$, it holds that

$$\Pr \left[\mathcal{A}(1^\lambda, \tilde{M}_b, \tilde{s}) = b \right] \leq \frac{1}{2} + \text{negl}(\lambda)$$

in the probability space defined by sampling

$SK \leftarrow \text{KeyGen}(1^\lambda, T, S)$
 $b \leftarrow \{0, 1\}$
 $\tilde{M}_b \leftarrow \text{GbPrg}(SK, M)$
 $\tilde{s} \leftarrow \text{GbMem}(SK, s)$

5.1 Construction

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

Overview.

Our construction of $\text{Garble}(M, x, T, S)$ applies Garble' to a transformed version of the machine M and a correspondingly transformed of the input x . The transformed machine, which we will denote by M' , differs from M in three ways:

- M' executes two copies of M in parallel (thereby using twice as much memory). We think of these as an ‘A’ execution and a ‘B’ execution. We think of the external storage of M' as correspondingly consisting of an ‘A’ track and a ‘B’ track. We implement the ‘A’ and ‘B’ tracks by modifying the memory alphabet Σ to hold two symbols.
- M' writes metadata alongside each value to indicate the time and address at which it is written.
- M' authenticates each value it writes: instead of writing (t, a, v, v) to an address a , it writes $(t, a, F((t, a)) \oplus v, G((t, a)) \oplus v)$, where F and G are puncturable pseudorandom functions.

CONSTRUCTION 2. We define $(\text{KeyGen}, \text{GbPrg}, \text{GbMem})$:

- $\text{KeyGen}(1^\lambda, T, S)$ samples $K' \leftarrow \text{KeyGen}'(1^\lambda, T, S)$, as well as puncturable PRFs F and G and outputs

$$K = (K', F, G, T, S).$$

- $\text{GbPrg}(K, M = (\Sigma, Q, Y, C))$ samples and outputs

$$\tilde{M} \leftarrow \text{GbPrg}'(K', M' = (\Sigma', Q', Y, C')),$$

where

$$\Sigma' = [T] \times [S] \times \{0, 1\}^{\ell_\Sigma} \times \{0, 1\}^{\ell_\Sigma},$$

$$Q' = [T] \times Q \times Q,$$

and C' is defined in Algorithm 2. The initial state $q'_0 \in Q'$ of M' is defined as $(0, q_0, q_0)$.

- $\text{GbMem}((K, F, G, T, S), s)$ samples $\tilde{s} \leftarrow \text{GbMem}(K, s')$, where s' is defined such that $s'(a) = \epsilon$ if $s(a) = \epsilon$, and otherwise

$$s'(a) = (0, a, F((0, a)) \oplus s(a), G((0, a)) \oplus s(a))$$

Data: Underlying transition function C , puncturable PRFs F and G
Input: State q , symbol σ

```

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

```

Algorithm 2: C'

THEOREM 3. If $(\text{KeyGen}', \text{GbPrg}', \text{GbMem}')$ is a fixed memory secure garbling scheme, and if one-way functions (and hence puncturable PRFs) exist, then Construction 2 defines a fully succinct, fixed address secure garbling scheme for RAM machines.

A proof is given in the full version of this paper [CH15].

6. FULL SECURITY

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

6.1 Oblivious RAM

6.1.1 Syntax

An oblivious RAM is a tuple of probabilistic polynomial-time algorithms (Setup, OMem, OProg)

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

6.1.2 Correctness

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

$$\Pr [M'(s') = M(s)] \geq 1 - \text{negl}(\lambda)$$

in the probability space defined by sampling

$$\begin{aligned} SK &\leftarrow \text{Setup}(1^\lambda, S) \\ M' &\leftarrow \text{OProg}(SK, M) \\ s' &\leftarrow \text{OMem}(SK, s) \end{aligned}$$

6.1.3 Efficiency

There is a function

$$\begin{aligned} \eta &: \mathbb{N} \rightarrow \mathbb{N} \\ \eta(S) &= \Theta(\text{polylog}(S)) \end{aligned}$$

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

$$\Pr [\text{Time}(M', s') = \eta(S) \cdot \text{Time}(M, s)] = 1,$$

in the same probability space as above.

6.1.4 Localized Randomness

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

$$\begin{aligned} \text{addr}'_{M,s,S,\lambda} : \{0, 1\}^{\mathbb{N}} &\rightarrow \mathbb{N}^{\eta T} \\ \text{addr}'_{M,s,S,\lambda}(\vec{r}) &= \vec{a} \end{aligned}$$

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

$$\begin{aligned} SK &\leftarrow \text{Setup}(1^\lambda, S) \\ M' &\leftarrow \text{OProg}(SK, M) \\ s' &\leftarrow \text{OMem}(SK, s) \\ \vec{a} &\leftarrow \text{addr}(M', s') \end{aligned}$$

DEFINITION 4. An ORAM (Setup, OProg, OMem) is said to have localized randomness if there is a deterministic algorithm **Sim** such that for all RAM machines M , memory configurations s , and space bounds $S \geq \text{Space}(M, s)$ and running times $t = \text{Time}(M, s)$, there exist sets $R_1, \dots, R_t \subseteq \mathbb{N}$ such that,

- For each i , $|R_i| \leq \text{poly}(\log S, \lambda)$.
- For each $i \neq j$, $R_i \cap R_j = \emptyset$.
- For each i ,

$$\text{addr}'_{M,s,S,\lambda}(\vec{r})_{\{\eta(i-1), \dots, \eta i-1\}} = \text{Sim}(\vec{r}_{R_i})$$

with high probability over a uniformly random $\vec{r} \in \{0, 1\}^{\mathbb{N}}$.

In the full version of this paper [CH15], we show that the Chung-Pass ORAM [CP13] satisfies these properties.

6.2 Construction

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

CONSTRUCTION 3. Let $(\text{KeyGen}', \text{GbPrg}', \text{GbMem}')$ be a fixed-address garbling scheme, and let (Setup, OProg, OMem) be an ORAM scheme with localized randomness. We define a garbling scheme $(\text{KeyGen}, \text{GbPrg}, \text{GbMem})$:

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

THEOREM 4. If $(\text{KeyGen}', \text{GbPrg}', \text{GbMem}')$ is a fixed address secure garbling scheme for RAM machines, then Construction 3 defines a (fully secure) garbling scheme for RAM machines.

A proof is given in the full version of this paper [CH15].

7. PERSISTENT DATA

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

DEFINITION 5. A RAM garbling scheme with persistent data is a tuple of p.p.t. algorithms $(\text{KeyGen}, \text{GbPrg}, \text{GbMem})$:

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

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

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

DEFINITION 6. A RAM garbling scheme with persistent data is said to be correct if for every memory configuration s_0 , for every $\ell = \text{poly}(\lambda)$, and every tuple of RAM machines $(M_1, \dots, M_\ell)$, it holds that the outputs of the garbled machines, when run in sequence on the garbled data, equal the outputs of the plaintext machines when run in sequence on the plaintext data. That is:

$$\Pr [y_1 = y'_1 \wedge \dots \wedge y_\ell = y'_\ell] \geq 1 - \text{negl}(\lambda)$$

in the probability space defined by sampling

$SK \leftarrow \text{KeyGen}(1^\lambda)$
 $\tilde{s}_0 \leftarrow \text{GbMem}(SK, s_0)$
 For $i = 1, \dots, \ell$
 $\tilde{M}_i \leftarrow \text{GbPrg}(SK, M_i, i)$
 $y_i \leftarrow M_i(s_{i-1})$
 $y'_i \leftarrow \tilde{M}_i(\tilde{s}_{i-1})$
 $s_i \leftarrow \text{NextMem}(M_i, s_{i-1})$
 $\tilde{s}_i \leftarrow \text{NextMem}(\tilde{M}_i, \tilde{s}_{i-1})$

DEFINITION 7. A RAM garbling scheme with persistent data is said to be secure if there is a p.p.t. algorithm Sim such that for all memory configurations s_0 , all $\ell = \text{poly}(\lambda)$, and all RAM machines $M_1, \dots, M_\ell$, no p.p.t. algorithm $\mathcal{A}$ can correctly distinguish between

$$(\tilde{s}_0, \tilde{M}_1, \dots, \tilde{M}_\ell)$$

and

$$\text{Sim}(y_1, \dots, y_\ell, |s_0|, |M_0|, \dots, |M_\ell|, 1^\lambda)$$

in the probability space defined by sampling

$SK \leftarrow \text{KeyGen}(1^\lambda)$
 $\tilde{s}_0 \leftarrow \text{GbMem}(SK, s_0)$
 For $i = 1, \dots, \ell$
 $\tilde{M}_i \leftarrow \text{GbPrg}(SK, M_i, i)$

and defining

For $i = 1, \dots, \ell$
 $y_i \leftarrow M_i(s_{i-1})$
 $s_i \leftarrow \text{NextMem}(M_i, s_{i-1})$

THEOREM 5. If there is an indistinguishability obfuscator for circuits and there exist one-way functions, then there is a correct, secure RAM garbling scheme with persistent data.

PROOF. (Sketch.) The scheme and the analysis are straightforward extensions of the single-machine case. That is, the memory garbling is identical to the single-machine case, except that the string “step 0” is attached to the root of the merkle tree before signing; The i th machine is garbled by applying the machine garbling algorithm of the single-machine case, with the exception that now the signed root of the Merkle tree is expected to contain also “step $i - 1$ ”, and the root of the Merkle-tree-hash of the final memory configuration is signed together with “step i ”. (All machines are

garbled with the same accumulator, iterator, and splittable signature parameters.)

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

We note that, by garbling one machine per bit of an output, we can garble machines with long outputs. Full compactness with simulation security is easily seen to be impossible in this setting, and recent work strengthens this impossibility result to hold even for a weaker notion of security [LPST15]. Our work does not contradict these results, because the combined size of the garbled machines grows proportionally to the total output length.

Acknowledgements

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

Ran Canetti is supported by the Check Point Institute for Information Security, ISF grant 1523/14, and NSF Frontier CNS-1413920 and 1218461 grants.

Justin Holmgren is supported by the NSF Frontier CNS-1413920 grant.

8. REFERENCES

- [AB15] Benny Applebaum and Zvika Brakerski. Obfuscating circuits via composite-order graded encoding. In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, pages 528–556, 2015.
- [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.
- [BGK⁺14] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In

- Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 221–238. Springer Berlin Heidelberg, 2014.
- [BGL⁺15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. Succinct randomized encodings and their applications. In Ronitt Rubinfeld, editor, *Symposium on the Theory of Computing (STOC)*, 2015.
- [CCC⁺15] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Computation-trace indistinguishability obfuscation and its applications. Cryptology ePrint Archive, Report 2015/406, 2015. <http://eprint.iacr.org/>.
- [CCHR15] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Succinct adaptive garbled ram. Cryptology ePrint Archive, Report 2015/1074, 2015. <http://eprint.iacr.org/>.
- [CH15] Ran Canetti and Justin Holmgren. Succinct garbled ram. Cryptology ePrint Archive, Report 2015/388, 2015. <http://eprint.iacr.org/>.
- [CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for ram programs. In Ronitt Rubinfeld, editor, *Symposium on the Theory of Computing (STOC)*, 2015.
- [CLTV15] Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. Probabilistic indistinguishability obfuscation. In *TCC*, 2015.
- [CP13] Kai-Min Chung and Rafael Pass. A simple ORAM. *IACR Cryptology ePrint Archive*, 2013:243, 2013.
- [GGH⁺13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability and functional encryption for all circuits. In *FOCS*, pages 40–49, 2013.
- [GHL⁺14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT*, pages 405–422, 2014.
- [GKP⁺13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Symposium on Theory of Computing Conference, STOC’13, Palo Alto, CA, USA, June 1-4, 2013*, pages 555–564, 2013.
- [GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled ram from one-way functions. In Ronitt Rubinfeld, editor, *Symposium on the Theory of Computing (STOC)*, 2015.
- [GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.
- [IPS15] Yuval Ishai, Omkant Pandey, and Amit Sahai. Public-coin differing-inputs obfuscation and its applications. In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, pages 668–697, 2015.
- [KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In Ronitt Rubinfeld, editor, *Symposium on the Theory of Computing (STOC)*, 2015.
- [LO13] Steve Lu and Rafail Ostrovsky. How to garble ram programs? In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734. Springer Berlin Heidelberg, 2013.
- [LPST15] Huijia Lin, Rafael Pass, Karn Seth, and Sidharth Telang. Output-compressing randomized encodings and applications. Cryptology ePrint Archive, Report 2015/720, 2015. <http://eprint.iacr.org/>.
- [SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $o((\log n)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.
- [SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *STOC*, pages 475–484, 2014.
- [Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science, SFCS ’86*, pages 162–167, Washington, DC, USA, 1986. IEEE Computer Society.
- [Zim14] Joe Zimmerman. How to obfuscate programs directly. *IACR Cryptology ePrint Archive*, 2014:776, 2014.