Indistinguishability Obfuscation for Turing Machines: Constant Overhead and Amortization

Prabhanjan Ananth[⋆] Abhishek Jain[⋆⋆] Amit Sahai[⋆⋆⋆]

Abstract. We study the asymptotic efficiency of indistinguishability obfuscation (i$\mathcal{O}$) on two fronts:

– **Obfuscation size:** Present constructions of indistinguishability obfuscation (i$\mathcal{O}$) create obfuscated programs where the size of the obfuscated program is at least a multiplicative factor of security parameter larger than the size of the original program.
  In this work, we construct the first i$\mathcal{O}$ scheme for (bounded-input) Turing machines that achieves only a *constant* multiplicative overhead in size. The constant in our scheme is, in fact, 2.
– **Amortization:** Suppose we want to obfuscate an arbitrary polynomial number of (bounded-input) Turing machines $M_1, \ldots, M_n$. We ask whether it is possible to obfuscate $M_1, \ldots, M_n$ using a *single* application of an i$\mathcal{O}$ scheme for a circuit family where the size of any circuit is *independent* of n as well the size of any Turing machine M_i.
  In this work, we resolve this question in the affirmative, obtaining a new bootstrapping theorem for obfuscating arbitrarily many Turing machines.

In order to obtain both of these results, we develop a new template for obfuscating Turing machines that is of independent interest and likely to find applications in future. The security of our results rely on the

[⋆]University of California Los Angeles. Email: `prabhanjan@cs.ucla.edu`. This work was partially supported by grant #360584 from the Simons Foundation and the grants listed under Amit Sahai.

[⋆⋆]Johns Hopkins University. Email: `abhishek@cs.jhu.edu`. Supported in part by DARPA/ARL Safeware Grant W911NF-15-C-0213.

[⋆⋆⋆]University of California Los Angeles. Email: `sahai@cs.ucla.edu` Research supported in part from a DARPA/ARL SAFEWARE award, NSF Frontier Award 1413955, NSF grants 1619348, 1228984, 1136174, and 1065276, BSF grant 2012378, a Xerox Faculty Research Award, a Google Faculty Research Award, an equipment grant from Intel, and an Okawa Foundation Research Grant. This material is based upon work supported by the Defense Advanced Research Projects Agency through the ARL under Contract W911NF-15-C-0205. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense, the National Science Foundation, or the U.S. Government.

existence of sub-exponentially secure $\mathsf{i}\mathcal{O}$ for circuits and re-randomizable encryption schemes.

1 Introduction

The notion of indistinguishability obfuscation ($\mathsf{i}\mathcal{O}$) [11] guarantees that given two equivalent programs M_0 and M_1, their obfuscations are computationally indistinguishable. The first candidate for general-purpose $\mathsf{i}\mathcal{O}$ was given by Garg et al. [32]. Since their work, $\mathsf{i}\mathcal{O}$ has been used to realize numerous advanced cryptographic tasks, such as functional encryption [32], deniable encryption [55], software watermarking [28] and PPAD hardness [14], that previously seemed beyond our reach.

Over the last few years, research on $\mathsf{i}\mathcal{O}$ constructions has evolved in two directions. The first line of research concerns with developing $\mathsf{i}\mathcal{O}$ candidates with stronger security guarantees and progressively weaker reliance on multilinear maps [31], with the goal of eventually building it from standard cryptographic assumptions. By now, a large sequence of works (see, e.g., [32,10,21,53,39,3,56,9,4,15,5,49,51,34]) have investigated this line of research. The works of [51,34] constitute the state of the art in this direction, where [51] give a construction of $\mathsf{i}\mathcal{O}$ for circuits from a concrete assumption on constant-degree multilinear maps, while [34] gives an $\mathsf{i}\mathcal{O}$ candidate in a weak multilinear map model [52] that resists all known attacks on multilinear maps [27,36,18,30,29,52].

Another line of research concerns with building $\mathsf{i}\mathcal{O}$ candidates with improved efficiency in a *generic* manner. The goal here is to develop bootstrapping theorems for $\mathsf{i}\mathcal{O}$ that achieve greater efficiency when obfuscating different classes of programs. This started with the work of Garg et al. [32], which showed, roughly speaking, that $\mathsf{i}\mathcal{O}$ for functions computed by branching programs implies $\mathsf{i}\mathcal{O}$ for functions computed by general boolean circuits. While this first bootstrapping theorem achieved $\mathsf{i}\mathcal{O}$ for all polynomial-time circuits, it still left open the question of obfuscating natural representations of the original program (for example, Turing machines).

Last year, this question was addressed in multiple works [13,24,48] showing that $\mathsf{i}\mathcal{O}$ for circuits implies $\mathsf{i}\mathcal{O}$ for Turing Machines with bounded-length inputs (see also [23,26,22,2] for extensions). Moving to the Turing Machine model yields significant efficiency improvements over the circuit model since the size of a Turing Machine may be much smaller than the corresponding circuit size. Importantly, it also achieves per-input running time, as opposed to incurring worst-case running time that is inherent to the circuit model of computation.

Our Work. In this work, we continue the study of bootstrapping mechanisms for $\mathsf{i}\mathcal{O}$ to achieve further qualitative and quantitative gains in the asymptotic efficiency of obfuscation. We note that despite the recent advances, existing mechanisms for general-purpose $\mathsf{i}\mathcal{O}$ remain highly inefficient and incur large polynomial overhead in the size of the program being obfuscated. We seek to improve the state of affairs on two fronts:

- *Size Efficiency:* First, we seek to develop obfuscation mechanisms where the size of an obfuscated program incurs only a small overhead in the size of the program being obfuscated.
- *Amortization:* Second, we seek to develop $\mathsf{i}\mathcal{O}$ amortization techniques, where a single expensive call to an obfuscation oracle (that obfuscates programs of a priori fixed size) can be used to obfuscate arbitrarily many programs.

We expand on each of our goals below. Below, we restrict our discussion to Turing machine obfuscation, which is the main focus of this work.

I. Size Efficiency of $\mathsf{i}\mathcal{O}$. All known mechanisms for $\mathsf{i}\mathcal{O}$ yield obfuscated programs of size polynomial in the size of the underlying program and the security parameter, thus incurring a multiplicative overhead of at least the security parameter in the size of the underlying program.[1] The works of [19,1,46] achieve these parameters by relying on (public-coin) differing inputs obfuscation [11,46]. In contrast, [13,24,48,23,26,22,2] only rely upon $\mathsf{i}\mathcal{O}$ for circuits; however, these works are restricted to programs with bounded-length inputs, and as such incur overhead of $\mathrm{poly}(\lambda, |M|, L)$, where L is the bound on the input length.

In this work, we ask the question:

Is it possible to realize general-purpose $\mathsf{i}\mathcal{O}$ with
constant multiplicative overhead in program size?

More precisely, we ask whether it is possible to obfuscate bounded-input Turing Machines such that the resulting machine is of size $c \cdot |M| + \mathrm{poly}(\lambda, L)$, where c is a universal constant and L is the input length bound.

Achieving constant multiplicative overhead has been a major goal in many areas of computer science, from constructing asymptotically good error correcting codes, to encryption schemes where the size of the ciphertext is linear in the size of the plaintext. To the best of our knowledge, however, this question in the context of program obfuscation has appeared to be far out of reach in the context of basing security on $\mathsf{i}\mathcal{O}$ itself.[2]

II. $\mathsf{i}\mathcal{O}$ Amortization. We next consider the case of obfuscating *multiple* Turing machines. Since known circuit obfuscation mechanisms are inefficient not just in terms of obfuscation size but also the obfuscation time, we would like to minimize the use of circuit obfuscation for obfuscating multiple Turing machines. We ask the following question:

[1] For the case of circuits, the recent work of [15] gives an $\mathsf{i}\mathcal{O}$ construction where the obfuscated circuit incurs only a constant overhead in the size of the underlying circuit but polynomial dependence on the depth of the circuit. While our focus is on Turing machine obfuscation, our results also yield improvements for circuit obfuscation. We refer the reader to Section 1.4 for a comparison of our results with [15].

[2] We observe that using (public-coin) differing input obfuscation, a variant of the construction given by [19,1,46] where FHE is combined with hybrid encryption, can yield constant multiplicative overhead. However, the plausibility of differing input obfuscation has come under scrutiny [33,12]. Thus, in this work, we focus only on achieving $\mathsf{i}\mathcal{O}$ with constant multiplicative overhead from the existence of $\mathsf{i}\mathcal{O}$ (without constant multiplicative overhead) itself.

Is it possible to obfuscate arbitrarily many Turing machines by using a *single* invocation to an $\mathsf{i}\mathcal{O}$ obfuscator for circuits of *a priori fixed* polynomial size?

More precisely, let O be an $\mathsf{i}\mathcal{O}$ obfuscator for circuits of a fixed polynomial size. Then, we want the ability to obfuscate multiple Turing machines $M_1, \ldots, M_n$ for an unbounded polynomial n, by making a single invocation to O. As above, we study this question for Turing machines with an a priori fixed input length bound L, and allow O to depend on L.

Note that a successful resolution of this question will yield an *amortization* phenomenon where arbitrarily many Turing machines can be obfuscated using (relatively) less expensive cryptographic primitives and only a single expensive invocation to a circuit obfuscator.

Bounded-input vs Unbounded-input Turing machines. We note that if we could build $\mathsf{i}\mathcal{O}$ for Turing machines with *unbounded* input length, then both of our aforementioned questions become moot. This is because one could simply obfuscate a universal Turing machine and pass on the actual machine that one wishes to obfuscate as an (encrypted) input. The state of the art in $\mathsf{i}\mathcal{O}$ research, however, is still limited to Turing machines with *bounded* input length. In this case, the above approach does not work since the size of the obfuscation for bounded-input TMs grows polynomially in the input length bound.

In a recent work, [50] provide a transformation from output compressing randomized encodings for TMs to $\mathsf{i}\mathcal{O}$ for unbounded-input TMs. However, no construction (with a security reduction) is presently known for such randomized encodings. In particular, in the same work, [50] show that such randomized encodings, in general, do not exist.

1.1 Our Results

I. $\mathsf{i}\mathcal{O}$ with Constant Multiplicative Overhead. Our first result is a construction of $\mathsf{i}\mathcal{O}$ for Turing machines with bounded input length where the size of obfuscation of a machine M is only $2|M| + \text{poly}(\lambda, L)$, where L is the input length bound. Our construction is based on sub-exponentially secure $\mathsf{i}\mathcal{O}$ for general circuits and one-way functions.

Theorem 1 (Informal). *Assuming sub-exponentially secure $\mathsf{i}\mathcal{O}$ for general circuits and sub-exponentially secure re-randomizable encryption schemes, there exists an $\mathsf{i}\mathcal{O}$ for Turing Machines with bounded input length such that the size of the obfuscation of a Turing machine M is $2 \cdot |M| + \text{poly}(\lambda, L)$, where L is an input length bound.*

Re-randomizable encryption schemes can be based on standard assumptions such as DDH and learning with errors.

In order to obtain this result, we develop a new template for obfuscating Turing machines starting from indistinguishability obfuscation for circuits. An obfuscation of Turing machine M in our template comprises of:

– A reusable encoding of M.
– An obfuscated input encoder circuit, that takes as input x and produces an encoding of x. This encoding is then decoded, together with the (reusable) encoding of M, to recover $M(x)$.

Our template exhibits two salient properties: (a) the reusable encoding of M is constructed from standard cryptographic primitives, *without any use of* $\mathsf{i}\mathcal{O}$. (b) The size of the input encoder circuit is *independent* of M. In contrast, prior templates for $\mathsf{i}\mathcal{O}$ for Turing machines comprised of a single obfuscated encoder circuit that contains M hardwired in its description, and therefore depends on the size of M.

We use the above template to reduce the problem of construction of iO for Turing machines with constant multiplicative overhead in size to the problem of constructing reusable TM encodings with constant multiplicative overhead in size. We defer discussion on the security properties associated with the reusable encoding scheme to the technical overview (Section 1.2). As we discuss next, our template enables some new applications.

II. A Bootstrapping Theorem for Obfuscating Multiple Programs. We now state our second result. Using our new template for Turing machine obfuscation, we show how to obfuscate $N = \mathrm{poly}(\lambda)$ Turing machines $M_1, \ldots, M_N$, for any polynomial N, using just a single invocation to an obfuscated circuit where the circuit size is independent of N and the size of each M_i and only depends on the security parameter and an input length bound L for every TM M_i. At a high level, this can be achieved by combining the input encoder circuits corresponding to the N machines into a *single* circuit whose size is independent of N.

Theorem 2 (Informal). *Let $\mathsf{i}\mathcal{O}_{\mathsf{ckt}}$ be an indistinguishability obfuscation for circuits scheme for a class of circuits $\mathcal{C}_{\lambda,L}$. There exists an indistinguishability obfuscation scheme $\mathsf{i}\mathcal{O}_{\mathsf{tm}}$ for Turing machines with input length bound L, where any polynomial number of Turing machines can be simultaneously obfuscated by making a single call to $\mathsf{i}\mathcal{O}_{\mathsf{ckt}}$.*

We emphasize that in order to obtain the above result, we crucially rely on the two aforementioned salient properties of our template. Indeed, it is unclear how to use the prior works [13,24,48] to obtain the above result.

We remark that the above bootstrapping theorem, combined with the fact that the reusable TM encodings in our template for TM obfuscation achieve constant overhead in size, implies the following useful corollary:

Corollary 1. *Assuming sub-exponentially secure $\mathsf{i}\mathcal{O}$ for general circuits and sub-exponentially secure re-randomizable encryption schemes, there exists an $\mathsf{i}\mathcal{O}$ scheme for Turing Machines with bounded input length such that the total size of the obfuscations of N Turing machines $M_1, \ldots, M_N$ is $2\Sigma_i|M_i| + \mathrm{poly}(\lambda, L)$, where L is an input length bound.*[3]

[3] Note that, in contrast, a naive (direct) use of Theorem 1 would yield a result where the the total size is $2\Sigma_i|M_i| + N \cdot \mathrm{poly}(\lambda, L)$.

We refer the reader to Section 1.3 for a brief discussion on how we obtain Theorem 2.

III. Subsequent work: Patchable $\mathsf{i}\mathcal{O}$. In a subsequent work [7], the same authors show how to use our template to construct patchable $\mathsf{i}\mathcal{O}$. This notion allows for updating obfuscation of a Turing machine M to an obfuscation of another machine M' with the help of patches that are privately-generated. Apart from being a natural extension to $\mathsf{i}\mathcal{O}$, many applications of $\mathsf{i}\mathcal{O}$ can be obtained from this primitive using extremely simple transformations. At a high level, the reason why our template finds use in their work is because they effectively reduce the problem of patchable $\mathsf{i}\mathcal{O}$ to building a patchable reusable encoding scheme. However, their work has to deal with several conceptual issues that arise specifically in the context of patching. We refer the reader to [7] for more details.

IV. Other Applications. Our result on $\mathsf{i}\mathcal{O}$ for TMs with constant overhead in size can be applied in many applications of $\mathsf{i}\mathcal{O}$ to achieve commensurate efficiency gains. Below we highlight some of these applications.

Functional Encryption with Constant Overhead. Plugging in our $\mathsf{i}\mathcal{O}$ in the functional encryption (FE) scheme of Waters [**?**],[4] we obtain an FE scheme for Turing machines where the size of a function key for a Turing machine M with input length bound L is only $c \cdot |M| + \mathrm{poly}(\lambda, L)$ for some constant c. Further, the size of a ciphertext for any message x is only $c' \cdot |x| + \mathrm{poly}(\lambda)$ for some constant c'.[5]

The size of the function keys can be further reduced by leveraging the recent result of [8] who construct adaptively secure FE for TMs with *unbounded* length inputs, based on $\mathsf{i}\mathcal{O}$ and one-way functions. Instantiating their FE construction with our $\mathsf{i}\mathcal{O}$ and the above discussed FE scheme, we obtain the *first* construction of an (adaptively secure) FE scheme where the size of a function key for an unbounded length input TM M is only $c \cdot |M| + \mathrm{poly}(\lambda)$ for some constant c.

Reusable Garbled Turing Machines with Constant Overhead. By applying the transformations of De Caro et al. [25] and Goldwasser et al. [40] on the above FE scheme, we obtain the *first* construction of Reusable Garbled TM scheme where both the machine encodings and input encodings incur only constant multiplicative overhead in the size of the machine and input, respectively. Specifically, the encoding size of a machine M is $c \cdot |M| + \mathrm{poly}(\lambda)$, while the encoding size of an input x is $c_1 \cdot |x| + c_2|M(x)| + \mathrm{poly}(\lambda)$ for some constants c, c_1, c_2.

Previously, Boneh et al. [16] constructed reusable garbled *circuits* with additive overhead in either the circuit encoding size, or the input encoding size (but not both simultaneously).

[4] [**?**] presents two FE schemes: the first one only handles post-challenge key queries, while the second one allows for both pre-challenge and post-challenge key queries. We only consider the instantiation of the first scheme with our $\mathsf{i}\mathcal{O}$.

[5] The construction of [**?**] already achieves the second property.

1.2 Technical Overview: New Template for Succinct $\mathsf{i}\mathcal{O}$

We now provide an overview of the main ideas underlying our results. We start by motivating and explaining our new template for succinct $\mathsf{i}\mathcal{O}$ and then explain how we build succinct $\mathsf{i}\mathcal{O}$ with constant overhead in size. Next, in Section 1.3, we explain how we obtain our bootstrapping theorem for obfuscating arbitrarily many Turing machine.

We start by recalling the common template for constructing $\mathsf{i}\mathcal{O}$ for Turing machines (TM) used in all the prior works in the literature [13,24,48,23,26]. Similar to these works, we focus on the restricted setting of TMs with inputs of a priori bounded length. For simplicity of discussion, however, we will ignore this restriction in this section.

Prior template for succinct $\mathsf{i}\mathcal{O}$. [13,24,48] reduce the problem of obfuscating Turing machines to the problem of obfuscating circuits. This is achieved in the following two steps:

1. *Randomized encoding for TMs.* The first step is to construct a randomized encoding (RE) [44] for Turing machines using $\mathsf{i}\mathcal{O}$ for circuits.
2. *From RE to* $\mathsf{i}\mathcal{O}$*.* The second step consists of obfuscating the encoding procedure of RE (constructed in the first step). Very roughly, to obfuscate a machine M, we simply obfuscate a circuit $C_{M,K}$ that has the machine M and a PRF key K hardwired. On any input x, circuit C_M outputs a "fresh" RE of $M(x)$ using randomness $\mathsf{PRF}_K(x)$. To recover $M(x)$, the evaluator simply executes the decoding algorithm of RE.

Following [13,24,48], all of the subsequent works on succinct $\mathsf{i}\mathcal{O}$ [23,26] follow the above template.[6]

Shortcomings of the prior template. However, as we discuss now, this template is highly problematic for achieving our goal of $\mathsf{i}\mathcal{O}$ with constant multiplicative overhead in size.

– First, note that since the obfuscation of machine M corresponds to obfuscating a circuit that has machine M hardwired, in order to achieve constant overhead in the size of M, we would require the underlying circuit obfuscator to already satisfy the constant overhead property!
– Furthermore, since the description of circuit C_M includes the encoding procedure of the RE, we would require the RE scheme constructed in the first step to not only achieve constant overhead in size, but also in *encoding time*. In particular, we would require that the *running time* of the RE encode procedure on input (M, x) has only a constant multiplicative overhead in $|M| + |x|$.
  We stress that this is a much more serious issue. Indeed, ensuring that the running time has only a constant multiplicative overhead in the input size is in general a hard problem for many cryptographic primitives (see [45] for discussion).

[6] We note that the above template also works for obfuscating RAM programs if we start with an RE for RAM in the first step.

Towards that end, we devise a new template for constructing $\mathsf{i}\mathcal{O}$ for TMs which is more amenable to our goal of $\mathsf{i}\mathcal{O}$ with constant overhead in size.

A new template for succinct $\mathsf{i}\mathcal{O}$: Starting ideas. Our first idea is to modify the above template in such a manner that the obfuscated circuit does not contain machine M anymore. Instead, the machine M is encoded separately. Specifically, in our modified template, obfuscation of a machine M consists of two components:

– An encoding $\widetilde{M}$ of the machine M using an encoding key sk.
– Obfuscation of the "input encoder", i.e., a circuit $C'_{sk,K}$ that has hardwired in its description the encoding key sk and a PRF key K. On any input x, $C'_{sk,K}$ computes an input encoding $\tilde{x}$ using sk and randomness $\mathsf{PRF}_K(x)$.

To evaluate the obfuscation of M on an input x, the evaluator first executes the obfuscated circuit to obtain an encoding $\tilde{x}$. It then decodes $(\widetilde{M}, \tilde{x})$ to obtain $M(x)$.

A few remarks are in order: first, note that the above template requires a *decomposable* RE where a machine M and an input x can be encoded separately using an encoding key sk. Second, the RE scheme must be *reusable*, i.e., given an encoding $\widetilde{M}$ of M and multiple input encodings $\tilde{x}_1, \ldots, \tilde{x}_n$ (computed using the same encoding key sk) for any n, it should be possible to decode $(\widetilde{M}, \tilde{x}_i)$ to obtain $M(x_i)$ for every $i \in [n]$.

It is easy to verify the correctness of the above construction. Now, let us see why this template is more suitable for our goal. Observe that if the (reusable) encoding of M has constant overhead in size, then the obfuscation scheme also achieves the same property. Crucially, we do not need the RE scheme to achieve constant overhead in encoding time.

At this point, it seems that we have reduced the problem of $\mathsf{i}\mathcal{O}$ for TMs with constant size overhead to the problem of reusable RE with constant size overhead. This intuition, unfortunately, turns out to be misleading. The main challenge arises in proving security of the above template. Very briefly, we note that following [38,53,39], prior works on succinct $\mathsf{i}\mathcal{O}$ use a common "input-by-input" proof strategy to argue the security of their construction. Recall that the obfuscation of a TM M in these works corresponds to obfuscation of the circuit $C_{M,K}$ described earlier. Then, in order to implement the "input-by-input" proof strategy, it is necessary that the PRF key K supports *puncturing* [55,17,20,47]. Note, however, that in our new template, obfuscation of M consists of a reusable encoding of M and obfuscation of circuit $C'_{sk,K}$ described above. Then, implementing a similar proof strategy for our template would require the encoding key sk of the reusable RE (that is embedded in the circuit $C'_{sk,K}$) to also support puncturing. However, the standard notion of reusable RE [40] does not support key puncturing, and therefore, does not suffice for arguing security of the above construction.

Oblivious Evaluation Encodings. Towards that end, our next idea is to develop an "iO-friendly" notion of reusable RE that we refer to as *oblivious* *evaluation encodings* (OEE). Our definition of OEE is specifically tailored to facilitate a security proof of the construction discussed above.

In an OEE scheme, instead of encoding a single machine, we allow encoding of two machines M_0 and M_1 together using an encoding key sk. Further, an input x is encoded together with a "choice" bit b using sk. The decode algorithm, on input encodings of (x, b) and (M_0, M_1), outputs $M_b(x)$.[7]

An OEE scheme also comes equipped with two *key puncturing* algorithms:

- *Input puncturing*: On input an encoding key sk and input x, it outputs a punctured encoding key sk_x^{inp}. This punctured key allows for computation of encodings of $(x', 0)$ and $(x', 1)$ for all inputs $x' \neq x$. The security property associated with it is as follows: for any input x, given a machine encoding of (M_0, M_1) s.t. $M_0(x) = M_1(x)$ and a punctured key sk_x^{inp}, no PPT adversary should be able to distinguish between encodings of $(x, 0)$ and $(x, 1)$.
- *(Choice) Bit puncturing*: On input an encoding key sk and bit b, it outputs a punctured encoding key sk_b^{bit}. This punctured key allows for computation of encodings of (x, b) for all x. The security property associated with it is as follows: for any machine pair (M_0, M_1), given a punctured key sk_0^{bit}, no PPT adversary should be able to distinguish encoding of (M_0, M_0) from (M_0, M_1). (The security for punctured key sk_1^{bit} can be defined analogously.)

Finally, we say that an OEE scheme achieves constant multiplicative overhead in size if the size of machine encoding of any pair (M_0, M_1) is $|M_0| + |M_1| + \text{poly}(\lambda)$. Further, similar to reusable RE, we require that the size of the input encoding of x is $\text{poly}(\lambda, |x|)$ and in particular, independent of the size of $|M_0|$ and $|M_1|$.

$\mathsf{i}\mathcal{O}$ for TMs from OEE. We now describe our modified template for constructing $\mathsf{i}\mathcal{O}$ for TMs where reusable RE is replaced with OEE. An obfuscation of a machine M consists of two components: (a) An OEE TM encoding of (M, M) generated using an OEE secret key sk. (b) Obfuscation of the OEE input encoder, i.e., a circuit $C_{sk,K}$ that on input x outputs an OEE input encoding of $(x, 0)$ using the OEE key sk and randomness generated using the PRF key K. To evaluate the obfuscated machine on an input x, an evaluator first computes encoding of $(x, 0)$ using the obfuscated $C_{sk,K}$ and then decodes $(x, 0)$ and (M, M), using the OEE decode algorithm, to obtain $M(x)$.

To prove security, we need to argue that obfuscations of two equivalent machines M_0 and M_1 are computationally indistinguishable. For the above construction, this effectively boils down to transitioning from a hybrid where we give out a machine encoding of (M_0, M_0) to one where we give out a machine encoding of (M_1, M_1). We achieve this by crucially relying on the security of the key puncturing algorithms. Very roughly, we first use the punctured key sk_0^{bit} to transition from (M_0, M_0) to (M_0, M_1) and then later, we use sk_1^{bit} to transition from (M_0, M_1) to (M_1, M_1). In between these steps, we rely on punctured keys

[7] An informed reader might find some similarities between OEE and oblivious transfer. Indeed, the name for our primitive is inspired by oblivious transfer.

sk_x^{inp}, for every input x (one at a time), to transition from a hybrid where the (obfuscated) input encoder produces encodings corresponding to bit $b = 0$ to one where $b = 1$.

Now, note that if we instantiate the above construction with an OEE scheme that achieves constant overhead in size, then the resulting obfuscation scheme also satisfies the same property *even* if the obfuscation of circuit $C_{sk,K}$ has polynomial overhead in size. Here, note that it is crucial that we require that the size of the OEE input encodings to be independent of $|M_0|$ and $|M_1|$. Thus, in order to achieve our goal of i$\mathcal{O}$ for TMs with constant size overhead, the remaining puzzle piece is a construction of OEE with constant overhead in size. Our main technical contribution is to provide such a construction.

Construction of OEE: Initial Challenges. A natural approach towards constructing OEE is to start with known constructions of reusable RE for TMs. We note that the only known approach in the literature for constructing reusable RE for TMs is due to [37]. In their approach, for every input, a "fresh" instance of a single-use RE for TMs [13,24,48,23,26,8] is computed on the fly. However, as discussed at the beginning of this section, in order to achieve constant overhead in size, such an approach would require that the single-use RE achieves constant overhead in *encoding time*, which is a significantly harder problem. Therefore, this approach is ill suited to our goal.

In light of the above, we start from the (single-use) RE construction of Koppula et al. [48] and use the ingredients developed in their work to build all the properties necessary for an OEE scheme with constant overhead in size. Indeed, the work of KLW forms the basis of all known subsequent constructions of randomized encodings for TMs/ RAMs [23,26,22,2] that do not suffer from space bound restrictions (unlike [13,24]); therefore, it is a natural starting point for our goal.

We start by recalling the RE construction of KLW. We only provide a simplified description, omitting several technical details.

An RE encoding of (M, x) has two components:

- Authenticated Hash Tree: the first component consists of a verifiable hash tree[8] computed on an encryption of the input tape initialized with TM M. The root of the hash tree is authenticated using a special signature.
- Obfuscated Next Step Function: The second component is an obfuscated circuit of the next step function of $U_x(\cdot)$, where U_x is a universal TM that takes as input a machine M and produces $M(x)$. The hash key, signing and verification keys and decryption key are hardwired inside this obfuscated circuit. It takes as input an encrypted state, an encrypted symbol on the tape along with a proof of validity that consists of authentication path in the hash tree and the signature on the root. Upon receiving such an input, it first checks input validity using the hash key and the signature verification

[8] [48] uses a special hash tree called positional accumulator. For this high-level overview, one can think of it as a "iO-friendly" Merkle hash tree. We refer the reader to the technical sections for further details.

key. It then decrypts the state and the symbol using the decryption key. Next, it executes the next step of the transition function. It then re-encrypts the new state and the new symbol. Using the old authentication path, it recomputes the new authentication path and a fresh signature on the root. Finally, it outputs the new signed root.

A reader familiar with [48] will notice that in the above discussion, we have flipped the roles of the machine and the input. First, it is easy to see that these two presentations are equivalent. More importantly, this specific presentation is crucial to our construction of OEE because by flipping the roles of the machine and the input, we are able to leverage the inherent asymmetry in the machine encoding and input encoding in KLW. This point will become more clear later in this section.

The security proof of KLW, at a high level, works by authenticating one step of the computation at a time. In particular, this idea is implemented using a recurring hybrid where for any execution step i of $U_x(M)$, the obfuscated circuit *only accepts a unique input and all other inputs are rejected*. This unique input is a function of the parameters associated with the hashing scheme, signature scheme and the encryption scheme. We call such hybrids *unique-input accepting hybrids*. Such hybrids have, in fact, been used in other $\mathsf{i}\mathcal{O}$-based constructions as well.[9]

Using the above template, we discuss initial ideas towards constructing an OEE scheme. In the beginning, we restrict our attention to achieving reusability with constant overhead. Later, we will discuss how to achieve the key puncturing properties later.

Challenge #1: Reusability. A natural first idea to achieve reusability is to have the first component in the above construction to be the machine encoding and the second component to be the input encoding. To argue security, lets consider a simple case when the adversary is given a machine encoding of M and two input encodings of x_1 and x_2. A natural first approach is to argue the security of M on x_1 first and then argue the security of M on x_2. The hope here is that we can reuse the (single-use) security proof of KLW separately for x_1 and x_2. This, however, doesn't work because the unique-accepting hybrids corresponding to input x_1 would be incompatible with the computation of M on x_2 (and vice versa). An alternate idea would be to employ *multi-input accepting hybrids* instead of unique-input accepting hybrids, where the obfuscated next step function, for a given i, accepts a fixed set of multiple inputs and rejects all other inputs. However, this would mean that we can only hardwire an a priori fixed number of values in the obfuscated circuit which would then put a bound on the number of input encodings that can be issued. Hence this direction is also not feasible.

[9] For example, Hubacek and Wichs [43] consider a scenario where the obfuscated circuit accepts pre-images of the hash function as input. In order to use $\mathsf{i}\mathcal{O}$ security, they use a special hash function (SSB hash) that is programmed to accept only one input on a special index.

In order to resolve the above difficulty, we modify the above template. For any input x, to generate an input encoding, we generate fresh parameters of the hashing, signature and the encryption schemes. We then generate the obfuscated circuit of the next step function of $U_x(\cdot)$ with all the parameters (including the freshly generated ones) hardwired inside it. The machine encoding of M, however, is computed with respect to parameters that are decided at the OEE setup time. At this point it is not clear why correctness should hold: the parameters associated with encodings of x and M are independently generated.

To address this, we introduce a *translation* mechanism that translates machine encoding of M with respect to one set of parameters into another machine encoding of M w.r.t to a different set of parameters. In more detail, every input encoding will be equipped with a freshly generated translator. A translator, associated with an encoding of x, takes as input a machine encoding of M, computed using the parameters part of OEE setup, checks for validity and outputs a new encoding of M corresponding to the fresh parameters associated with the encoding of x. For now, the translator can be thought of as an obfuscated circuit that has hardwired inside it the old and the new parameters. Later we discuss its actual implementation.

Finally, a word about security. Roughly speaking, due to the use of fresh parameters for every input, we are able to reduce security to the one-input security of KLW.

Challenge #2: Constant Overhead. While the above high level approach tackles reusability, it does not yet suffice for achieving constant multiplicative overhead in the size of the machine encodings. Recall that the machine encoding consists of an encryption of the machine along with the hash tree and a signature on the root. We first observe that the hashing algorithm is public and hence, it is not necessary to include the hash tree as part of the input encoding; instead the input encoding can just consist of an encryption of the machine, root of the hash tree and a signature on it. The decoder can reconstruct the hash tree and proceed as before. We can then use an encryption scheme with constant overhead in size to ensure that the encryption of M only incurs constant overhead in size. Note, however, that such an encryption scheme should also be compatible with hash tree computation over it.

While one might envision constructing such a scheme, a bigger issue is the size of the translator. In fact, the size of the translator, as described above, is polynomial in the input length, which corresponds to the size of the machine encoding. It therefore invalidates the efficiency requirement of OEE on the size of input encodings.

One plausible approach to overcome this problem might be to not refresh the encryption of M and in fact just translate the signature associated with the root of the hash tree into a different signature. This would mean that the decryption key associated with the encryption of M would be common among all the input encodings. However, in the security proof, this conflicts with the unique-input accepting hybrids as discussed earlier.

Construction of OEE: Our Approach. The main reason why the above solution does not work is because the machine M is in an encrypted form. Suppose we instead focus on the weaker goal of achieving *authenticity* without privacy. That is, we guarantee the correctness of computation against dishonest evaluators but not hide the machine. Our crucial observation is that the above high level template sans encryption of the machine is already a candidate solution for achieving this weaker security goal. An astute reader would observe that this setting resembles attribute based encryption [54,42] (ABE). Indeed in order to build OEE, our first step is to build an ABE scheme for TMs where the key size incurs a constant multiplicative overhead in the original machine length. We achieve this goal by using the ideas developed above. We then provide a *generic* reduction to transform such an ABE scheme into an OEE scheme satisfying constant multiplicative overhead in size. We now explain our steps in more detail.

ABE for TMs. Recall that in an ABE scheme, an encryption of an attribute, message pair (x, m) can be decrypted using a secret key corresponding to a machine M to recover m only if $M(x) = 1$. An ABE scheme is said to have a constant multiplicative overhead in size if the size of key of M is $c|M| + \mathrm{poly}(\lambda)$ for a constant c. Here, note that neither x nor M are required to be hidden.

The starting point of our construction of ABE is the *message hiding encoding* (MHE) scheme of KLW. An MHE scheme is effectively a "privacy-free" RE scheme, and therefore, perfectly suited for our goal of constructing an ABE scheme. More concretely, an MHE encoding of a tuple (M, x, msg), where M is a TM and x is an input to M, allows one to recover msg iff $M(x) = 1$. On the efficiency side, computing the encoding of (M, x, msg) should take time independent of the time to compute M on x. The important point to note here is that only msg needs to be hidden from the adversary and in particular, it is not necessary to hide the computation of M on x.

The construction of MHE follows along the same lines as the RE construction of KLW with the crucial difference that the machine M (unlike the RE construction) is not encrypted. Following the above discussion, this has the right template from which we can build our ABE scheme. Using KLW's template and incorporating the ideas we developed earlier, we sketch the construction of ABE below. This is an oversimplified version and several intricate technical details are omitted.

- Generate secret key sk, verification key vk of a special signature scheme (termed as splittable signature scheme in [48]). Generate a hash key hk of a verifiable hash tree. The public key consists of (vk, hk) and the secret key is sk.
- ABE key of a machine M is computed by first computing a hash tree on M using hk. The root of the hash tree rt is then signed using sk to obtain σ. Output (M, σ). Note that $|\sigma| = \mathrm{poly}(\lambda)$ and thus, the constant multiplicative overhead property is satisfied.
- ABE encryption of (x, msg) is computed by first computing an obfuscated circuit of the next step function of $U_{x,\mathsf{msg}}(\cdot)$. We have $U_{x,\mathsf{msg}}$ to be a circuit

that takes as input circuit C and outputs msg if $C(x) = 1$. The parameters hardwired in this obfuscated circuit contains (among other parameters) (sk', vk') of a splittable signature scheme where (sk', vk') is sampled afresh. In addition, it consists of a *signature translator* SignProg, that we introduced earlier. This signature translator takes as input a pair (rt, σ). Upon receiving such an input, it first verifies the validity of the signature w.r.t vk and then outputs a signature on rt w.r.t sk' if the verification succeeds. Otherwise it outputs $\perp$.

The final output of the encryption algorithm is the obfuscated circuit along with the signature translator.

To argue security, we need to rely on the underlying security of message hiding encodings. Unlike several recent constructions that use KLW, thanks to the modularization of our approach, we are able to reduce the security of our construction to the security of MHE construction of KLW. We view this as a positive step towards reducing the "page complexity" of research works in this area.

Construction of OEE from ABE for TMs. One of the main differences between OEE and ABE is that OEE guarantees privacy of computation while ABE only offers authenticity. Therefore, we need to employ a privacy mechanism in order to transform ABE into an OEE scheme. A similar scenario was encountered by Goldwasser et al. [40] in a different context. Their main goal was to obtain single-key FE for circuits from ABE for circuits while we are interested in constructing OEE, which has seemingly stronger requirements than FE, from ABE for Turing machines. Nevertheless, we show how their techniques will be useful to develop a basic template of our construction of OEE.

As a starting point, we encode the pair of machines (M_0, M_1) by first encrypting them together. Since we perform computation on the machines, the encryption scheme we use is fully homomorphic [35]. In the input encoding of (x, b), we encrypt the choice bit b using the same public key. To evaluate (M_0, M_1) on (x, b), we execute the homomorphic evaluation function. Notice, however, that the output is in encrypted form. We need to provide additional capability to the evaluator to decrypt the output (and nothing else). One way around is that the input encoding algorithm publishes a garbling of the FHE decryption algorithm. But the input encoder must somehow convey the garbled circuit wire keys, corresponding to the output of the FHE evaluation, to the evaluator.

This is where ABE for TMs comes to the rescue. Using ABE, we can ensure that the evaluator gets *only* the wire keys corresponding to the output of the FHE evaluation. Once this is achieved, the garbled circuit that is provided as part of the input encoding can then be evaluated to obtain the decrypted output. We can then show that the resulting OEE scheme has constant multiplicative overhead if the underlying ABE scheme also satisfies this property.

While the above high level idea is promising, there are still some serious issues. The first issue is that we need to homomorphically evaluate on Turing machines as against circuits. This can be resolved by using the powers-of-two evaluation technique from the work of [41]. The second and the more important question is: what are the punctured keys? The input puncturing key could simply be the ABE public key and the FHE public key-secret key pair. The choice bit puncturing key, however, is more tricky. Note that setting the FHE secret key to be the punctured key will ensure correctness but completely destroy the security. To resolve this issue, we use the classic two-key technique: we encrypt machines M_0 and M_1 using two different FHE public keys. The choice bit puncturing key is set to be one of the FHE secret keys depending on which bit needs to be punctured.

1.3 Technical Overview: Boostrapping Theorem

We now explain how our template for Turing machine obfuscation can be used to obtain Theorem 2. Suppose that we wish to obfuscate N Turing machines $M_1, \ldots, M_N$ for $N = \mathrm{poly}(\lambda)$.

Using our template discussed above, a starting idea towards obtaining Theorem 2 is as follows. Let K_1 and K_2 be keys for two puncturable PRF families. The obfuscation of $M_1, \ldots, M_N$ consists of the following parts:

- N different OEE TM encodings $(\widetilde{M_1, M_1}), \ldots, (\widetilde{M_N, M_N})$ where each $(\widetilde{M_i, M_i})$ is computed using an encoding key sk_i that is generated using randomness $PRF_{K_1}(i)$.
- Obfuscation of a "joint" input encoder circuit C_{K_1,K_2} that contains K_1 and K_2 hardwired in its description. It takes as input a pair (x, i) and performs the following steps: (a) Compute an OEE encoding key sk_i "on-the-fly" by running the OEE setup algorithm using randomness $\mathsf{PRF}_{K_1}(i)$. (b) Compute and output an OEE input encoding $\widetilde{(x, 0)}_i$ for the ith machine using the key sk_i and fresh randomness $\mathsf{PRF}_{K_2}(i, x)$.

Then security of the above construction can be argued using a straightforward hybrid argument using the puncturing properties of the PRF.

The above idea, however, does not immediately yield Theorem 2. The problem is that the OEE input encoding $\widetilde{(x, 0)}_i$ itself contains obfuscated programs. Therefore, the circuit C_{K_1,K_2} (described above) itself needs to make queries to a circuit obfuscation scheme.

We resolve the above problem in the following manner. Recall from above that an OEE input encoding in our scheme consists of two components: a garbled circuit and an ABE ciphertext. An ABE ciphertext, in turn, consists of obfuscations of two circuits. Lets refer to these circuits as C_1^{sub} and C_2^{sub}. Then, our idea is to simply "absorb" the functionality of C_1^{sub} and C_2^{sub} within C_{K_1,K_2}. In more detail, we consider a modified input encoder circuit C'_{K_1,K_2} that works in three modes: (a) In mode 1, it takes as input (x, i) and simply outputs the garbled circuit component of the input encoding $\widetilde{(x, 0)}_i$. (b) In mode 2, it takes an input for circuit C_1^{sub} and produces its output. (c) In mode 3, it takes an input for circuit C_2^{sub} and produces its output.

With the above modification, obfuscation of $M_1, \ldots, M_N$ now consists of N different OEE TM encodings $(\widetilde{M_1, M_1}), \ldots, (\widetilde{M_N, M_N})$ and obfuscation of the modified input encoder circuit C'_{K_1,K_2}. Crucially, this process only involves a *single* invocation of the circuit obfuscation scheme for the circuit family $\{C'_{K_1,K_2}\}$, where the size of C'_{K_1,K_2} is independent of N as well as the size of any M_i. This gives us Theorem 2.

1.4 Related Work

In a recent work, [15] give a construction of $\mathsf{i}\mathcal{O}$ for circuits where the size of obfuscation of a circuit C with depth d and inputs of length L is $2 \cdot C + \text{poly}(\lambda, d, L)$. Their construction relies on (sub-exponentially secure) $\mathsf{i}\mathcal{O}$ for circuits with polynomial overhead and the learning with errors assumption.

While we focus on the Turing machine model in this work, we note that our construction can be easily downgraded to the case of circuits to obtain an $\mathsf{i}\mathcal{O}$ scheme where the size of obfuscation of a circuit C with inputs of length L is $2 \cdot C + \text{poly}(\lambda, L)$. In particular, it does not grow with the circuit depth beyond the dependence on the circuit size. Our construction requires (sub-exponentially secure) $\mathsf{i}\mathcal{O}$ for circuits with poly overhead and re-randomizable encryption schemes.

1.5 Full Version

Due to space constraints, much of the details of our constructions and the corresponding security proofs are omitted from this manuscript. The full version of the paper is available at [6].

2 Attribute-based Encryption for TMs with Additive Overhead

In an attribute-based encryption (ABE) scheme, a message m can be encrypted together with an attribute x such that an evaluator holding a decryption key corresponding to a predicate P can recover m if and only if $P(x) = 1$. Unlike most prior works on ABE that model predicates as circuits, in this work, following [41], we model predicates as Turing machines with inputs of arbitrary length. We only consider the setting where the adversary can receive only one decryption key. We refer to this as 1-key ABE.

Below, we start by providing definition of 1-key ABE for TMs. In Section 2.2, we present our construction. The proof of security of this construction can be found in the full version. Finally, in Section 2.3, we extend our 1-key ABE construction to build two-outcome ABE for TMs.

2.1 Definition

A 1-key ABE for Turing machines scheme, defined for a class of Turing machines $\mathcal{M}$, consists of four PPT algorithms, $\mathsf{1ABE} = (\mathsf{1ABE.Setup}, \mathsf{1ABE.KeyGen}, \mathsf{1ABE.Enc}, \mathsf{1ABE.Dec})$. We denote the associated message space to be MSG. The syntax of the algorithms is given below.

- **Setup,** $\mathsf{1ABE.Setup}(1^\lambda)$: On input a security parameter λ in unary, it outputs a public key-secret key pair $(\mathsf{1ABE.PP}, \mathsf{1ABE.SK})$.

- **Key Generation,** $\mathsf{1ABE.KeyGen}(\mathsf{1ABE.SK}, M)$: On input a secret key $\mathsf{1ABE.SK}$ and a TM $M \in \mathcal{M}$, it outputs an ABE key $\mathsf{1ABE}.sk_M$.

- **Encryption,** $\mathsf{1ABE.Enc}(\mathsf{1ABE.PP}, x, \mathsf{msg})$: On input the public parameters $\mathsf{1ABE.PP}$, attribute $x \in \{0,1\}^*$ and message $\mathsf{msg} \in \mathsf{MSG}$, it outputs the ciphertext $\mathsf{1ABE.CT}_{(x,\mathsf{msg})}$.

- **Decryption,** $\mathsf{1ABE.Dec}(\mathsf{1ABE}.sk_M, \mathsf{1ABE.CT}_{(x,\mathsf{msg})})$: On input the ABE key $\mathsf{1ABE}.sk_M$ and encryption $\mathsf{1ABE.CT}_{(x,\mathsf{msg})}$, it outputs the decrypted result out.

Correctness. The correctness property dictates that the decryption of a ciphertext of (x, msg) using an ABE key for M yields the message msg if $M(x) = 1$. In formal terms, the output of the decryption procedure $\mathsf{1ABE.Dec}(\mathsf{1ABE}.sk_M, \mathsf{1ABE.CT}_{(x,\mathsf{msg})})$ is (always) msg if $M(x) = 1$, where

- $(\mathsf{1ABE.SK}, \mathsf{1ABE.PP}) \leftarrow \mathsf{1ABE.Setup}(1^\lambda)$,
- $\mathsf{1ABE}.sk_M \leftarrow \mathsf{1ABE.KeyGen}(\mathsf{1ABE.SK}, M \in \mathcal{M})$ and,
- $\mathsf{1ABE.CT}_{(x,\mathsf{msg})} \leftarrow \mathsf{1ABE.Enc}(\mathsf{1ABE.PP}, x, \mathsf{msg})$.

Security. The security framework we consider is identical to the indistinguishability based security notion of ABE for circuits except that (i) the key queries correspond to Turing machines instead of circuits and (ii) the adversary is only allowed to make a single key query. Furthermore, we only consider the setting when the adversary submits both the challenge message pair as well as the key query at the beginning of the game itself. We term this *weak selective security*. We formally define this below.

The security is defined in terms of the following security experiment between a challenger and a PPT adversary. We denote the challenger by Ch and the adversary by $\mathcal{A}$.

$\underline{\mathsf{Expt}^{\mathsf{1ABE}}_{\mathcal{A}}(1^\lambda, b)}$:

1. $\mathcal{A}$ sends to Ch a tuple consisting of a Turing machine M, an attribute x and two messages $(\mathsf{msg}_0, \mathsf{msg}_1)$. If $M(x) = 1$ then the experiment is aborted.

2. The challenger Ch replies to $\mathcal{A}$ with the public key, decryption key of M, the challenge ciphertext; $\left(\mathsf{1ABE.PP}, \mathsf{1ABE}.sk_M, \mathsf{1ABE.CT}_{(x,\mathsf{msg}_b)}\right)$, where the values are computed as follows:
   - $(\mathsf{1ABE.PP}, \mathsf{1ABE.SK}) \leftarrow \mathsf{1ABE.Setup}(1^\lambda)$,
   - $\mathsf{1ABE}.sk_M \leftarrow \mathsf{1ABE.KeyGen}(\mathsf{1ABE.SK}, M)$
   - $\mathsf{1ABE.CT}_{(x,\mathsf{msg}_b)} \leftarrow \mathsf{1ABE.Enc}(\mathsf{1ABE.PP}, x, \mathsf{msg}_b)$.

3. The experiment terminates when the adversary outputs the bit b'.

We say that a 1-key ABE for TMs scheme is weak-selectively secure if any PPT adversary can guess the challenge bit only with negligible probability.

Definition 1. *A 1-key attribute based encryption for TMs scheme is said to be* **weak-selectively secure** *if there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that for every PPT adversary* $\mathcal{A}$*,*

$$\left|\mathsf{Pr}[0 \leftarrow \mathsf{Expt}_{\mathcal{A}}^{\mathsf{1ABE}}(1^\lambda, 0)] - \mathsf{Pr}[0 \leftarrow \mathsf{Expt}_{\mathcal{A}}^{\mathsf{1ABE}}(1^\lambda, 1)]\right| \leq \mathsf{negl}(\lambda)$$

Remark 1. Henceforth, we will omit the term "weak-selective" when referring to the security of ABE schemes.

1-Key Attribute Based Encryption for TMs with Additive Overhead. We say that a 1-key attribute based encryption for TMs scheme achieves additive overhead property if the size of an ABE key for a TM M is only $|M| + \text{poly}(\lambda)$. More formally,

Definition 2. *A 1-key attribute based encryption for TMs scheme,* $\mathsf{1ABE}$*, defined for a class of Turing machines* $\mathcal{M}$*, satisfies additive overhead property if* $|\mathsf{1ABE}.sk_M| = |M| + \text{poly}(\lambda)$*, where* $(\mathsf{1ABE.SK}, \mathsf{1ABE.PP}) \leftarrow \mathsf{1ABE.Setup}(1^\lambda)$ *and* $\mathsf{1ABE}.sk_M \leftarrow \mathsf{1ABE.KeyGen}(\mathsf{1ABE.SK}, M \in \mathcal{M})$*.*

2.2 Construction of 1-Key ABE

We now present our construction of 1-key ABE for TMs. We begin with a brief overview.

Overview. Our construction uses three main primitives imported from [48] – namely, positional accumulators, splittable signatures and iterators.

The setup first generates the setup of the splittable signatures scheme to yield $(\mathsf{SK_{tm}}, \mathsf{VK_{tm}}, \mathsf{VK_{rej}})$. The rejection-verification key $\mathsf{VK_{rej}}$ will be discarded. $(\mathsf{SK_{tm}}, \mathsf{VK_{tm}})$ will be the master signing key-verification key pair. The setup also generates accumulator and iterator parameters.

The signing key $\mathsf{SK_{tm}}$ will be the ABE secret key and the rest of the parameters form the public key. To generate an ABE key of M, first compute the accumulator storage of M. Then sign the accumulator value of M using $\mathsf{SK_{tm}}$ to obtain σ. Output the values M, σ and accumulator value[10].

An ABE encryption of (x, msg) is an obfuscation of the next step function that computes $U_x(\cdot)$ (universal circuit with x hardcoded in it) one step at a time. Call this obfuscated circuit N. Embedded into this obfuscated circuit are accumulator and iterator parameters, part of the public parameters. In addition, it has a PRF key K_A used to generate fresh splittable signature instantiations. In order for this to be compatible with the master signing key, a signature translator is provided as part of the ciphertext. This translator circuit, which will be obfuscated, takes as input message, a signature verifiable using $\mathsf{VK_{tm}}$ and

[10] In this construction, the key generation also outputs an iterator value.

produces a new signature with respect to parameters generated using K_A. Call this obfuscated circuit S. The ciphertext consists of (N, S).

Construction. We will use the following primitives in our construction:

1. A puncturable PRF family denoted by F.
2. A storage accumulator scheme based on $\mathsf{i}\mathcal{O}$ and one-way functions that was constructed by [48]. We denote it by $\mathsf{Acc} = (\mathsf{SetupAcc}, \mathsf{EnforceRead}, \mathsf{EnforceWrite}, \mathsf{PrepRead}, \mathsf{PrepWrite}, \mathsf{VerifyRead}, \mathsf{WriteStore}, \mathsf{Update})$. Let Σ_{tape} be the associated message space with accumulated value of size ℓ_{Acc} bits.
3. An iterators scheme denoted by $\mathsf{Itr} = (\mathsf{SetupItr}, \mathsf{ItrEnforce}, \mathsf{Iterate})$. Let $\{0,1\}^{2\lambda+\ell_{\mathsf{Acc}}}$ be the associated message space with iterated value of size ℓ_{Itr} bits.
4. A splittable signatures scheme denoted by $\mathsf{SplScheme} = (\mathsf{SetupSpl}, \mathsf{SignSpl}, \mathsf{VerSpl}, \mathsf{SplitSpl}, \mathsf{SignSplAbo})$. Let $\{0,1\}^{\ell_{\mathsf{Itr}}+\ell_{\mathsf{Acc}}+2\lambda}$ be the associated message space.

Our Scheme. We now describe our construction of a 1-key ABE scheme $\mathsf{1ABE} = (\mathsf{1ABE.Setup}, \mathsf{1ABE.KeyGen}, \mathsf{1ABE.Enc}, \mathsf{1ABE.Dec})$ for the Turing machine family $\mathcal{M}$. Without loss of generality, the start state of every Turing machine in $\mathcal{M}$ is denoted by q_0. We denote the message space for the ABE scheme as MSG.

$\underline{\mathsf{1ABE.Setup}(1^\lambda)}$: On input a security parameter λ, it first executes the setup of splittable signatures scheme to compute $(\mathsf{SK_{tm}}, \mathsf{VK_{tm}}, \mathsf{VK_{rej}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$. Next, it executes the setup of the accumulator scheme to obtain the values $(\mathsf{PP_{Acc}}, \widetilde{w}_0, \widetilde{store}_0) \leftarrow \mathsf{SetupAcc}(1^\lambda)$. It then executes the setup of the iterator scheme to obtain the public parameters $(\mathsf{PP_{Itr}}, v_0) \leftarrow \mathsf{SetupItr}(1^\lambda)$.

It finally outputs the following public key-secret key pair,

$$\Big(\mathsf{1ABE.PP} = (\mathsf{VK_{tm}}, \mathsf{PP_{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP_{Itr}}, v_0), \mathsf{1ABE.SK} = (\mathsf{1ABE.PP}, \mathsf{SK_{tm}})\Big)$$

$\underline{\mathsf{1ABE.KeyGen}(\mathsf{SK_{tm}}, M \in \mathcal{M})}$: On input a master secret key $\mathsf{1ABE.SK} = (\mathsf{1ABE.PP}, \mathsf{SK_{tm}})$ and a Turing machine $M \in \mathcal{M}$, it executes the following steps:

1. Parse the public key $\mathsf{1ABE.PP}$ as $(\mathsf{VK_{tm}}, \mathsf{PP_{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP_{Itr}}, v_0)$.

2. **Initialization of the storage tree**: Let $\ell_{\mathsf{tm}} = |M|$ be the length of the machine M. For $1 \leq j \leq \ell_{\mathsf{tm}}$, compute $\widetilde{store}_j = \mathsf{WriteStore}(\mathsf{PP_{Acc}}, \widetilde{store}_{j-1}, j-1, M_j)$, $aux_j = \mathsf{PrepWrite}(\mathsf{PP_{Acc}}, \widetilde{store}_{j-1}, j-1)$, $\widetilde{w}_j = \mathsf{Update}(\mathsf{PP_{Acc}}, \widetilde{w}_{j-1}, M_j, j-1, aux_j)$, where M_j denotes the j^{th} bit of M. Set the root $w_0 = \widetilde{w}_{\ell_{\mathsf{tm}}}$.

3. **Signing the accumulator value**: Generate a signature on the message $(v_0, q_0, w_0, 0)$ by computing $\sigma_0 \leftarrow \mathsf{SignSpl}(\mathsf{SK_{tm}}, \mu = (v_0, q_0, w_0, 0))$, where q_0 is the start state of M.

It outputs the ABE key $\mathsf{1ABE}.sk_M = (M, w_0, \sigma_{\mathsf{tm}}, v_0, \widetilde{store}_0)$.

[Note: The key generation does not output the storage tree $store_0$ but instead it just outputs the initial store value $\widetilde{store}_0$. As we see later, the evaluator in

possession of M, $\widetilde{store}_0$ and $\mathsf{PP}_{\mathsf{Acc}}$ can reconstruct the tree $store_0$.]

<u>$\mathsf{1ABE.Enc}(\mathsf{1ABE.PP}, x, \mathsf{msg})$</u>: On input a public key $\mathsf{1ABE.PP} = (\mathsf{VK}_{\mathsf{tm}}, \mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP}_{\mathsf{Itr}}, v_0)$, attribute $x \in \{0,1\}^*$ and message $\mathsf{msg} \in \mathsf{MSG}$, it executes the following steps:

1. Sample a PRF key K_A at random from the family F.

2. **Obfuscating the next step function**: Consider a universal Turing machine $U_x(\cdot)$ that on input M executes M on x for at most 2^λ steps and outputs $M(x)$ if M terminates, otherwise it outputs $\perp$. Compute an obfuscation of the program NxtMsg described in Figure 1, namely $N \leftarrow \mathsf{i}\mathcal{O}(\mathsf{NxtMsg}\{U_x(\cdot), \mathsf{msg}, \mathsf{PP}_{\mathsf{Acc}}, \mathsf{PP}_{\mathsf{Itr}}, K_A\})$. NxtMsg is essentially the next message function of the Turing machine $U_x(\cdot)$ – it takes as input a TM M and outputs $M(x)$ if it halts within 2^λ else it outputs $\perp$. In addition, it performs checks to validate whether the previous step was correctly computed. It also generates authentication values for the current step.

3. Compute an obfuscation of the program $S \leftarrow (\mathsf{SignProg}\{K_A, \mathsf{VK}_{\mathsf{tm}}\})$ where $\mathsf{SignProg}$ is defined in Figure 2. The program $\mathsf{SignProg}$ takes as input a message-signature pair and outputs a signature with respect to a different key on the same message.

It outputs the ciphertext $\mathsf{1ABE.CT} = (N, S)$.

<u>$\mathsf{1ABE.Dec}(\mathsf{1ABE}.sk_M, \mathsf{1ABE.CT})$</u>: On input the ABE key $\mathsf{1ABE}.sk_M = (M, w_0, \sigma_{\mathsf{tm}}, v_0, \widetilde{store}_0)$ and a ciphertext $\mathsf{1ABE.CT} = (N, S)$, it first executes the obfuscated program $S\big(y = (v_0, q_0, w_0, 0), \sigma_{\mathsf{tm}}\big)$ to obtain σ_0. It then executes the following steps.

1. **Reconstructing the storage tree**: Let $\ell_{\mathsf{tm}} = |M|$ be the length of the TM M. For $1 \leq j \leq \ell_{\mathsf{tm}}$, update the storage tree by computing, $\widetilde{store}_j = \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, \widetilde{store}_{j-1}, j-1, M_j)$. Set $store_0 = \widetilde{store}_{\ell_{\mathsf{tm}}}$.
2. **Executing N one step at a time**: For $i = 1$ to 2^λ,
   (a) Compute the proof that validates the storage value $store_{i-1}$ (storage value at $(i-1)^{th}$ time step) at position pos_{i-1}. Let $(\mathsf{sym}_{i-1}, \pi_{i-1}) \leftarrow \mathsf{PrepRead}(\mathsf{PP}_{\mathsf{Acc}}, store_{i-1}, \mathsf{pos}_{i-1})$.

   (b) Compute the auxiliary value, $aux_{i-1} \leftarrow \mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}}, store_{-1}, \mathsf{pos}_{i-1})$.

   (c) Run the obfuscated next message function. Compute $\mathsf{out} \leftarrow N(i, \mathsf{sym}_{i-1}, \mathsf{pos}_{i-1}, \mathsf{st}_{i-1}, w_{i-1}, v_{i-1}, \sigma_{i-1}, \pi_{i-1}, aux_{i-1})$. If $\mathsf{out} \in \mathsf{MSG} \cup \{\perp\}$, output out. Else parse out as $(\mathsf{sym}_{w,i}, \mathsf{pos}_i, \mathsf{st}_i, w_i, v_i, \sigma_i)$.

   (d) Compute the storage value, $store_i \leftarrow \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, store_{i-1}, \mathsf{pos}_{i-1}, \mathsf{sym}_{w,i})$.

This completes the description of the scheme. The correctness of the above scheme follows along the same lines as the message hiding encoding scheme of Koppula et al. For completeness, we give a proof sketch below.

```
Program NxtMsg

Constants: Turing machine $U_x = \langle Q, \Sigma_{\mathrm{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}} \rangle$, message $\mathsf{msg}$, Public parameters for accumulator $\mathsf{PP}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{PP}_{\mathsf{Itr}}$, Puncturable PRF key $K_A \in \mathcal{K}$.

Input: Time $t \in [T]$, symbol $\mathsf{sym}_{\mathsf{in}} \in \Sigma_{\mathrm{tape}}$, position $\mathsf{pos}_{\mathsf{in}} \in [T]$, state $\mathsf{st}_{\mathsf{in}} \in Q$, accumulator value $w_{\mathsf{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value $v_{\mathsf{in}}$, signature $\sigma_{\mathsf{in}}$, accumulator proof $\pi$, auxiliary value $aux$.

1. Verification of the accumulator proof:
   – If $\mathsf{VerifyRead}(\mathsf{PP}_{\mathsf{Acc}}, w_{\mathsf{in}}, \mathsf{sym}_{\mathsf{in}}, \mathsf{pos}_{\mathsf{in}}, \pi) = 0$ output $\perp$.
2. Verification of signature on the input state, position, accumulator and iterator values:
   – Let $F(K_A, t-1) = r_A$. Compute $(\mathsf{SK}_A, \mathsf{VK}_A, \mathsf{VK}_{A,\mathsf{rej}}) = \mathsf{SetupSpl}(1^\lambda; r_A)$.
   – Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{st}_{\mathsf{in}}, w_{\mathsf{in}}, \mathsf{pos}_{\mathsf{in}})$. If $\mathsf{VerSpl}(\mathsf{VK}_A, m_{\mathsf{in}}, \sigma_{\mathsf{in}}) = 0$ output $\perp$.
3. Executing the transition function:
   – Let $(\mathsf{st}_{\mathsf{out}}, \mathsf{sym}_{\mathsf{out}}, \beta) = \delta(\mathsf{st}_{\mathsf{in}}, \mathsf{sym}_{\mathsf{in}})$ and $\mathsf{pos}_{\mathsf{out}} = \mathsf{pos}_{\mathsf{in}} + \beta$.
   – If $\mathsf{st}_{\mathsf{out}} = q_{\mathrm{rej}}$ output $\perp$.
   – If $\mathsf{st}_{\mathsf{out}} = q_{\mathrm{acc}}$ output $\mathsf{msg}$.
4. Updating the accumulator and the iterator values:
   – Compute $w_{\mathsf{out}} = \mathsf{Update}(\mathsf{PP}_{\mathsf{Acc}}, w_{\mathsf{in}}, \mathsf{sym}_{\mathsf{out}}, \mathsf{pos}_{\mathsf{in}}, aux)$. If $w_{\mathsf{out}} = Reject$, output $\perp$.
   – Compute $v_{\mathsf{out}} = \mathsf{Iterate}(\mathsf{PP}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{st}_{\mathsf{in}}, w_{\mathsf{in}}, \mathsf{pos}_{\mathsf{in}}))$.
5. Generating the signature on the new state, position, accumulator and iterator values:
   – Let $F(K_A, t) = r'_A$. Compute $(\mathsf{SK}'_A, \mathsf{VK}'_A, \mathsf{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{SetupSpl}(1^\lambda; r'_A)$.
   – Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{st}_{\mathsf{out}}, w_{\mathsf{out}}, \mathsf{pos}_{\mathsf{out}})$ and $\sigma_{\mathsf{out}} = \mathsf{SignSpl}(\mathsf{SK}'_A, m_{\mathrm{out}})$.
6. Output $\mathsf{sym}_{\mathsf{out}}, \mathsf{pos}_{\mathsf{out}}, \mathsf{st}_{\mathsf{out}}, w_{\mathsf{out}}, v_{\mathsf{out}}, \sigma_{\mathsf{out}}$.
```

Fig. 1: Program NxtMsg

```
Program SignProg

Constants: PRF key $K_A$ and verification key $\mathsf{VK}_{\mathsf{tm}}$.
Input: Message $y$ and a signature $\sigma_{\mathsf{tm}}$.

1. If $\mathsf{VerSpl}(\mathsf{VK}_{\mathsf{tm}}, y, \sigma_{\mathsf{tm}}) = 0$ then output $\perp$.
2. Execute the pseudorandom function on input 0 to obtain $r_A \leftarrow F(K, 0)$.
   Execute the setup of splittable signatures scheme to compute $(\mathsf{SK}_0, \mathsf{VK}_0) \leftarrow \mathsf{SetupSpl}(1^\lambda; r_A)$.
3. Compute the signature $\sigma_0 \leftarrow \mathsf{SignSpl}(\mathsf{SK}_0, y)$.
4. Output $\sigma_0$.
```

Fig. 2: Program SignProg

Lemma 1. 1ABE *satisfies the correctness property of an ABE scheme.*

Proof sketch. Suppose $\mathsf{1ABE.CT}$ is a ciphertext of message msg w.r.t an attribute x and $\mathsf{1ABE}.sk_M$ is an ABE key for a machine M. We claim that in the i^{th} iteration of the decryption of $\mathsf{1ABE.CT}$ using $\mathsf{1ABE}.sk_M$, the storage corresponds to the work tape of the execution of $M(x)$ at the i^{th} time step, denoted by $W_{t=i}$.[11] Once we show this, the lemma follows.

We prove this claim by induction on the total number of steps in the TM execution. The base case corresponds to 0^{th} time step when the iterations haven't begun. At this point, the storage corresponds to the description of the machine M which is exactly $W_{t=0}$ (work tape at time step 0). In the induction hypothesis, we assume that at time step $i-1$, the storage contains the work tape $W_{t=i-1}$. We need to argue for the case when $t=i$. To take care of this case, we just need to argue that the obfuscated next step function computes the i^{th} step of the execution of $M(x)$ correctly. The correctness of obfuscated next step function in turn follows from the correctness of iO and other underlying primitives.

Remark 2. In the description of Koppula et al., the accumulator and the iterator algorithms also take the time bound T as input. Here, we set $T = 2^\lambda$ since we are only concerned with Turing machines that run in time polynomial in λ.

Additive overhead. Let $\mathsf{1ABE}.sk_M = (M, w_0, \sigma_{\mathsf{tm}}, v_0, \widetilde{store}_0)$ be an ABE key generated as the output of $\mathsf{1ABE.KeyGen}(\mathsf{1ABE.SK}, M \in \mathcal{M})$. From the efficiency property of accumulators, we have that $|w_0|$ and $|\widetilde{store}_0|$ simply polynomials in the security parameter λ. The signature σ_{tm} on the message w_0 is also of length polynomial in the security parameter. Lastly, the iterator parameter v_0 is also only polynomial in the security parameter. Thus, the size of $\mathsf{1ABE}.sk_M$ is $|M| + \text{poly}(\lambda)$.
The proof of security can be found in the full version.

2.3 1-Key Two-Outcome ABE for TMs

Goldwasser et al. [40] proposed the notion of 1-key two-outcome ABE for circuits as a variant of 1-key attribute based encryption for circuits where a pair of secret messages are encoded as opposed to a single secret message. Depending on the output of the predicate, exactly one of the messages is revealed and the other message remains hidden. That is, given an encryption of a single attribute x and two messages $(\mathsf{msg}_0, \mathsf{msg}_1)$, the decryption algorithm on input an ABE key $\mathsf{TwoABE}.sk_M$, outputs msg_0 if $M(x) = 0$ and msg_1 otherwise. The security guarantee then says that if $M(x) = 0$ (resp., $M(x) = 1$) then the pair $(\mathsf{TwoABE}.sk_M, \mathsf{TwoABE.CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)})$, reveal no information about msg_1 (resp., msg_0).

We adapt their definition to the case when the predicates are implemented as Turing machines instead of circuits. We give a formal definition and a simple construction of this primitive in the full version.

[11] To be more precise, the storage in the KLW construction is a tree with the j^{th} leaf containing the value of the j^{th} location in the work tape $W_{t=i}$.

3 Oblivious Evaluation Encodings

In this section, we define and construct *oblivious evaluation encodings* (OEE). This is a strengthening of the notion of machine hiding encodings (MHE) introduced in [48]. Very briefly, machine hiding encodings are essentially randomized encodings (RE), except that in MHE, the machine needs to be hidden whereas in RE, the input needs to be hidden. More concretely, an MHE scheme for Turing machines has an encoding procedure that encodes the output of a Turing machine M and an input x. This is coupled with a decode procedure that decodes the output $M(x)$. The main efficiency requirement is that the encoding procedure should be much "simpler" than actually computing M on x. The security guarantee states that the encoding does not reveal anything more than $M(x)$.

We make several changes to the notion of MHE to obtain our definition of OEE. First, we require that the machine and the input can be encoded *separately*. Secondly, the machine encoding takes as input two Turing machines (M_0, M_1) and outputs a joint encoding. Correspondingly, the input encoding now also takes as input a bit b in addition to the actual input x, where b indicates which of the two machines M_0 or M_1 needs to be used. The decode algorithm on input an encoding of (M_0, M_1) and (x, b), outputs $M_b(x)$. In terms of security, we require the following two properties to be satisfied:

- Any PPT adversary should not be able to distinguish encodings of (M_0, M_0) and (M_0, M_1) (resp., (M_1, M_1) and (M_0, M_1)) even if the adversary is given a *punctured* input encoding key that allows him to encode inputs of the form $(x, 0)$ (resp., $(x, 1)$).
- Any PPT adversary is unable to distinguish the encodings of $(x, 0)$ and $(x, 1)$ even given an oblivious evaluation encoding (M_0, M_1), where $M_0(x) = M_1(x)$ and another type of punctured input encoding key that allows him to generate input encodings of $(x', 0)$ and $(x', 1)$ for all $x' \neq x$.

3.1 Definition

Syntax. We describe the syntax of a oblivious evaluation encoding scheme OEE below. The class of Turing machines associated with the scheme is $\mathcal{M}$ and the input space is $\{0,1\}^*$. Although we consider inputs of arbitrary lengths, during the generation of the parameters we place an upper bound on the running time of the machines which automatically puts an upper bound on the length of the inputs.

- $\mathsf{OEE.Setup}(1^\lambda)$: It takes as input a security parameter λ and outputs a secret key $\mathsf{OEE.sk}$.
- $\mathsf{OEE.TMEncode}(\mathsf{OEE.sk}, M_0, M_1)$: It takes as input a secret key $\mathsf{OEE.sk}$, a pair of Turing machines $M_0, M_1 \in \mathcal{M}$ and outputs a joint encoding $\widetilde{(M_0, M_1)}$.
- $\mathsf{OEE.InpEncode}(\mathsf{OEE.sk}, x, b)$: It takes as input a secret key $\mathsf{OEE.sk}$, an input $x \in \{0,1\}^*$, a choice bit b and outputs an input encoding $\widetilde{(x, b)}$.

– $\mathsf{OEE.Decode}(\widetilde{(M_0, M_1)}, \widetilde{(x, b)})$: It takes as input a joint Turing machine encoding $\widetilde{(M_0, M_1)}$, an input encoding $\widetilde{(x, b)}$, and outputs a value z.

In addition to the above main algorithms, there are four helper algorithms.

– $\mathsf{OEE.puncInp}(\mathsf{OEE.sk}, x)$: It takes as input a secret key $\mathsf{OEE.sk}$, input $x \in \{0,1\}^*$ and outputs a punctured key $\mathsf{OEE.sk}_x$.
– $\mathsf{OEE.pIEncode}(\mathsf{OEE.sk}_x, x', b)$: It takes as input a punctured secret key $\mathsf{OEE.sk}_x$, an input $x' \neq x$, a bit b and outputs an input encoding $\widetilde{(x', b)}$.
– $\mathsf{OEE.puncBit}(\mathsf{OEE.sk}, b)$: It takes as input a secret key $\mathsf{OEE.sk}$, an input bit b and outputs a key $\mathsf{OEE}.sk_b$.
– $\mathsf{OEE.pBEncode}(\mathsf{OEE}.sk_b, x)$: It takes as input a key $\mathsf{OEE}.sk_b$, an input x and outputs an input encoding $\widetilde{(x, b)}$.

Correctness. We say that an OEE scheme is correct if it satisfies the following three properties:

1. *Correctness of Encode and Decode:* For all $M_0, M_1 \in \mathcal{M}$, $x \in \{0,1\}^*$ and $b \in \{0,1\}$,
$$\mathsf{OEE.Decode}\Big(\widetilde{(M_0, M_1)}, \widetilde{(x, b)}\Big) = M_b(x),$$
where (i) $\mathsf{OEE.sk} \leftarrow \mathsf{OEE.Setup}(1^\lambda)$, (ii) $\widetilde{(M_0, M_1)} \leftarrow \mathsf{OEE.TMEncode}(\mathsf{OEE.sk}, M_0, M_1)$ and, (iii) $\widetilde{(x, b)} \leftarrow \mathsf{OEE.InpEncode}(\mathsf{OEE.sk}, x, b)$.
2. *Correctness of Input Puncturing:* For all $M_0, M_1 \in \mathcal{M}$, $x, x' \in \{0,1\}^*$ such that $x' \neq x$ and $b \in \{0,1\}$,
$$\mathsf{OEE.Decode}\left(\widetilde{(M_0, M_1)}, \widetilde{(x', b)}\right) = M_b(x'),$$
where (i) $\mathsf{OEE.sk} \leftarrow \mathsf{OEE.Setup}\left(1^\lambda\right)$; (ii) $\widetilde{(M_0, M_1)} \leftarrow \mathsf{OEE.TMEncode}(\mathsf{OEE.sk}, M_0, M_1)$ and, (iii) $\widetilde{(x', b)} \leftarrow \mathsf{OEE.pIEncode}\left(\mathsf{OEE.puncInp}\left(\mathsf{OEE.sk}, x\right), x', b\right)$.
3. *Correctness of Bit Puncturing:* For all $M_0, M_1 \in \mathcal{M}$, $x \in \{0,1\}^*$ and $b \in \{0,1\}$,
$$\mathsf{OEE.Decode}\left(\widetilde{(M_0, M_1)}, \widetilde{(x, b)}\right) = M_b(x),$$
where (i) $\mathsf{OEE.sk} \leftarrow \mathsf{OEE.Setup}\left(1^\lambda\right)$, (ii) $\widetilde{(M_0, M_1)} \leftarrow \mathsf{OEE.TMEncode}(\mathsf{OEE.sk}, M_0, M_1)$ and, (iii) $\widetilde{(x, b)} \leftarrow \mathsf{OEE.pBEncode}\left(\mathsf{OEE.puncBit}\left(\mathsf{OEE.sk}, b\right), x\right)$.

Efficiency. We require that an OEE scheme satisfies the following efficiency conditions. Informally, we require that the Turing machine encoding (resp., input encoding) algorithm only has a logarithmic dependence on the time bound. Furthermore, the running time of the decode algorithm should take time proportional to the computation time of the encoded Turing machine on the encoded input.

1. The running time of $\mathsf{OEE.TMEncode}(\mathsf{OEE.sk}, M_0 \in \mathcal{M}, M_1 \in \mathcal{M})$ is a polynomial in $(\lambda, |M_0|, |M_1|)$, where $\mathsf{OEE.sk} \leftarrow \mathsf{OEE.Setup}(1^\lambda)$.
2. The running time of $\mathsf{OEE.InpEncode}(\mathsf{OEE.sk}, x \in \{0,1\}^*, b)$ is a polynomial in $(\lambda, |x|)$, where $\mathsf{OEE.sk} \leftarrow \mathsf{OEE.Setup}(1^\lambda)$.
3. The running time of $\mathsf{OEE.Decode}((\widetilde{M_0, M_1}), \widetilde{(x, b)})$ is a polynomial in $(\lambda, |M_0|, |M_1|, |x|, t)$, where $\mathsf{OEE.sk} \leftarrow \mathsf{OEE.Setup}(1^\lambda)$, $(\widetilde{M_0, M_1}) \leftarrow \mathsf{OEE.TMEncode}(\mathsf{OEE.sk}, M_0 \in \mathcal{M}, M_1 \in \mathcal{M})$, $\widetilde{(x, b)} \leftarrow \mathsf{OEE.InpEncode}(\mathsf{OEE.sk}, x \in \{0,1\}^*, b)$ and t is the running time of the Turing machine M_b on x.

Indistinguishability of Encoding Bit. We describe security of encoding bit as a multi-stage game between an adversary $\mathcal{A}$ and a challenger.

– *Setup*: $\mathcal{A}$ chooses two Turing machines $M_0, M_1 \in \mathcal{M}$ and an input x such that $|M_0| = |M_1|$ and $M_0(x) = M_1(x)$. $\mathcal{A}$ sends the tuple (M_0, M_1, x) to the challenger.
  The challenger chooses a bit $b \in \{0,1\}$ and computes the following: (a) $\mathsf{OEE.sk} \leftarrow \mathsf{OEE.Setup}(1^\lambda)$, (b) machine encoding $(\widetilde{M_0, M_1}) \leftarrow \mathsf{OEE.TMEncode}(\mathsf{OEE.sk}, M_0, M_1)$, (c) input encoding $\widetilde{(x, b)} \leftarrow \mathsf{OEE.InpEncode}(\mathsf{OEE.sk}, x, b)$, and (d) punctured key $\mathsf{OEE.sk}_x \leftarrow \mathsf{OEE.puncInp}(\mathsf{OEE.sk}, x)$. Finally, it sends the following tuple to $\mathcal{A}$:
$$\left((\widetilde{M_0, M_1}), \widetilde{(x, b)}, \mathsf{OEE.sk}_x\right).$$
– *Guess*: $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$.

The advantage of $\mathcal{A}$ in this game is defined as $\mathsf{adv}_{\mathrm{OEE}_1} = \Pr[b' = b] - \frac{1}{2}$.

Definition 3 (Indistinguishability of encoding bit). *An OEE scheme satisfies indistinguishability of encoding bit if there exists a neglible function* $\mathsf{negl}(\cdot)$ *such that for every PPT adversary* $\mathcal{A}$ *in the above security game,* $\mathsf{adv}_{\mathrm{OEE}_1} = \mathsf{negl}(\lambda)$.

Indistinguishability of Machine Encoding. We describe security of machine encoding as a multi-stage game between an adversary $\mathcal{A}$ and a challenger.

– *Setup*: $\mathcal{A}$ chooses two Turing machines $M_0, M_1 \in \mathcal{M}$ and a bit $c \in \{0,1\}$ such that $|M_0| = |M_1|$. $\mathcal{A}$ sends the tuple (M_0, M_1, c) to the challenger.
  The challenger chooses a bit $b \in \{0,1\}$ and computes the following: (a) $\mathsf{OEE.sk} \leftarrow \mathsf{OEE.Setup}(1^\lambda)$, (b) $(\widetilde{\mathsf{TM}_1, \mathsf{TM}_2}) \leftarrow \mathsf{OEE.TMEncode}(\mathsf{OEE.sk}, \mathsf{TM}_1, \mathsf{TM}_2)$, where $\mathsf{TM}_1 = M_0, \mathsf{TM}_2 = M_{1\oplus b}$ if $c = 0$ and $\mathsf{TM}_1 = M_{0\oplus b}, \mathsf{TM}_2 = M_1$ otherwise, and (c) $\mathsf{OEE.sk}_c \leftarrow \mathsf{OEE.puncBit}(\mathsf{OEE.sk}, c)$. Finally, it sends the following tuple to $\mathcal{A}$:
$$\left((\widetilde{\mathsf{TM}_1, \mathsf{TM}_2}), \mathsf{OEE.sk}_c\right).$$
– *Guess*: $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$.

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

Definition 4 (Indistinguishability of machine encoding). *An OEE scheme satisfies indistinguishability of machine encoding if there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every PPT adversary* $\mathcal{A}$ *in the above security game,* $\mathsf{adv}_{\mathrm{OEE}_2} = \mathsf{negl}(\lambda)$.

OEE with Constant Multiplicative Overhead. The efficiency property in OEE dictates that the output length of the Turing machine encoding algorithm is a polynomial in the size of the Turing machine. We can restrict this condition further by requiring that the Turing machine encoding is only *linear* in the Turing machine size. We term the notion of OEE that satisfies this property as *OEE with constant multiplicative overhead.*

Definition 5 (OEE with constant multiplicative overhead). *An oblivious evaluation encoding scheme for a class of Turing machines* $\mathcal{M}$ *is said to have constant multiplicative overhead if its Turing machine encoding algorithm* $\mathsf{OEE.TMEncode}$ *on input* $(\mathsf{OEE.sk}, M_0, M_1)$ *outputs an encoding* $\widetilde{(M_0, M_1)}$ *such that* $|\widetilde{(M_0, M_1)}| = c \cdot (|M_0| + |M_1|) + \mathrm{poly}(\lambda)$*, where* c *is a constant* > 0.

3.2 Construction of OEE

Notation. We denote the class of Turing machines associated with oblivious evaluation encoding to be $\mathcal{M}$. For simplicity of notation, we assume that $\mathcal{M}$ consists of only single-bit output Turing machines. In every machine M in $\mathcal{M}$, there is a special location on the worktape in which the output of the Turing machine (0 or 1) is written. Until the termination of the Turing machine, this location contains the symbol $\perp$. We use the notation $M(x)$ to denote the value contained in this special location.

To construct a oblivious evaluation encoding scheme, we will use the following ingredients.

1. A 1-key two-outcome ABE for TMs scheme defined for a class of Turing machines $\mathcal{M}$, represented by $\mathsf{TwoABE} = (\mathsf{TwoABE.Setup}, \mathsf{TwoABE.TMEncode}, \mathsf{TwoABE.InpEncode}, \mathsf{TwoABE.Decode})$.
2. A fully homomorphic encryption scheme for circuits with *additive overhead*, represented by $\mathsf{FHE} = (\mathsf{FHE.Setup}, \mathsf{FHE.Enc}, \mathsf{FHE.Eval}, \mathsf{FHE.Dec})$.
3. A garbling scheme $\mathsf{GC} = (\mathsf{Garble}, \mathsf{EvalGC})$.

Construction. We denote the oblivious evaluation encoding scheme to be $\mathsf{OEE} = (\mathsf{OEE.Setup}, \mathsf{OEE.InpEncode}, \mathsf{OEE.TMEncode}, \mathsf{OEE.Decode})$ that is equipped with auxiliary algorithms $(\mathsf{OEE.puncInp}, \mathsf{OEE.pIEncode}, \mathsf{OEE.puncBit}, \mathsf{OEE.pBEncode})$. The construction of OEE is presented below.

$\underline{\mathsf{OEE.Setup}(1^\lambda)}$: On input a security parameter λ in unary, it executes the following steps.

– Run $\mathsf{TwoABE.Setup}(1^\lambda)$ to obtain a secret key-public parameters pair, $(\mathsf{TwoABE.SK}, \mathsf{TwoABE.PP})$.
– Run $\mathsf{FHE.Setup}(1^\lambda)$ twice to obtain FHE public key-secret key pairs $(\mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0)$ and $(\mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$.

It finally outputs $\mathsf{OEE.sk} = (\mathsf{TwoABE.SK}, \mathsf{TwoABE.PP}, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$.

$\underline{\mathsf{OEE.TMEncode}(\mathsf{OEE.sk}, M_0, M_1)}$: On input a secret key $\mathsf{OEE.sk}$ and a pair of Turing machines $M_0, M_1 \in \mathcal{M}$, it does the following.

– Parse $\mathsf{OEE.sk}$ as $(\mathsf{TwoABE.SK}, \mathsf{TwoABE.PP}, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$.
– Compute FHE encryptions of TMs M_0 and M_1 w.r.t public keys $\mathsf{FHE.pk}_0$ and $\mathsf{FHE.pk}_1$, respectively. That is, compute $\mathsf{FHE.CT}_{M_0} \leftarrow \mathsf{FHE.Enc}(\mathsf{FHE.pk}_0, M_0)$ and $\mathsf{FHE.CT}_{M_1} \leftarrow \mathsf{FHE.Enc}(\mathsf{FHE.pk}_1, M_1)$.
– Compute a TwoABE decryption key $\mathsf{TwoABE.SK}_N \leftarrow \mathsf{TwoABE.KeyGen}(\mathsf{TwoABE.SK}, N)$ for the machine $N = N_{\left(\{\mathsf{FHE.pk}_c, \mathsf{FHE.CT}_{M_c}\}_{c\in\{0,1\}}\right)}$ described in Figure 3.

It outputs the TM encoding $\widetilde{(M_0, M_1)} = \mathsf{TwoABE.SK}_N$.

$N_{\left(\{\mathsf{FHE.pk}_c, \mathsf{FHE.CT}_{M_c}\}_{c\in\{0,1\}}\right)}(x, i, \mathsf{ind})$

– Let $U = U_{x,\mathsf{ind}}(\cdot)$ be a universal Turing machine that on input a Turing machine M, outputs $M(x)$ if the computation terminates within 2^{ind} number of steps, otherwise it outputs $\perp$.
– Transform the universal Turing machine U into a circuit (see full version) by computing $C \leftarrow \mathsf{TMtoCKT}(U)$.
– Execute $\mathsf{FHE.Eval}(\mathsf{FHE.pk}_0, C, \mathsf{FHE.CT}_{M_0})$ to obtain z_1. Similarly execute $\mathsf{FHE.Eval}(\mathsf{FHE.pk}_1, C, \mathsf{FHE.CT}_{M_1})$ to obtain z_2.
– Set $z = (z_1||z_2)$. Output the i^{th} bit of z.

Fig. 3: Description of program N.

$\underline{\mathsf{OEE.InpEncode}(\mathsf{OEE.sk}, x, b)}$: On input the secret key $\mathsf{OEE.sk}$, input x and bit b, it executes the following steps.

– Parse $\mathsf{OEE.sk}$ as $(\mathsf{TwoABE.SK}, \mathsf{TwoABE.PP}, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$.
– For $\mathsf{ind} \in [\lambda]$, compute a garbled circuit along with the wire keys, $\left(\mathsf{gckt}_{\mathsf{ind}}, \{w_{i,0}^{\mathsf{ind}}, w_{i,1}^{\mathsf{ind}}\}_{i\in[q]}\right) \leftarrow \mathsf{Garble}(1^\lambda, G)$, where $G = G_{(\mathsf{FHE.sk}_b, b)}(\cdot)$ is a circuit that takes as input FHE ciphertexts $(\mathsf{FHE.CT}_0, \mathsf{FHE.CT}_1)$ and outputs a_b, where $a_b \leftarrow \mathsf{FHE.Dec}(\mathsf{FHE.sk}_b, \mathsf{FHE.CT}_b)$. Here, q denotes the total length of two FHE ciphertexts $(\mathsf{FHE.CT}_0, \mathsf{FHE.CT}_1)$.
– For every $i \in [q]$ and $\mathsf{ind} \in [\lambda]$, compute a TwoABE ciphertext $\mathsf{TwoABE.CT}_{i,\mathsf{ind}} \leftarrow \mathsf{TwoABE.Enc}\Big(\mathsf{TwoABE.PP}, (x, i, \mathsf{ind}), w_{i,0}^{\mathsf{ind}}, w_{i,1}^{\mathsf{ind}}\Big)$ of the message pair $(w_{i,0}^{\mathsf{ind}}, w_{i,1}^{\mathsf{ind}})$ along with attribute (x, i, ind).

Finally, it outputs the encoding $\widetilde{(x, b)} = \Big(\mathsf{TwoABE.PP}, \{\mathsf{gckt}\}_{\mathsf{ind} \in [\lambda]}, \{\mathsf{TwoABE.CT}_{i,\mathsf{ind}}\}_{i \in [q], \mathsf{ind} \in [\lambda]}\Big)$.

$\underline{\mathsf{OEE.Decode}(\widetilde{(M_0, M_1)}, \widetilde{(x, b)})}$: On input a TM encoding $\widetilde{(M_0, M_1)}$ and an input encoding $\widetilde{(x, b)}$, it executes the following steps.

- Parse the TM encoding $\widetilde{(M_0, M_1)} = \mathsf{TwoABE.SK}_N$ and the input encoding $\widetilde{(x, b)} = \big(\mathsf{TwoABE.PP}, \{\mathsf{gckt}\}_{\mathsf{ind} \in [\lambda]}, \{\mathsf{TwoABE.CT}_{i,\mathsf{ind}}\}_{i \in [q], \mathsf{ind} \in [\lambda]}\big)$.
- For every $\mathsf{ind} \in [\lambda]$, do the following:
  1. For every $i \in [q]$, execute the decryption procedure of TwoABE to obtain the wire keys of the garbled circuit, $\widetilde{w}_i^{\mathsf{ind}} \leftarrow \mathsf{TwoABE.Dec}(\mathsf{TwoABE.SK}_N, \mathsf{TwoABE.CT}_{i,\mathsf{ind}})$.
  2. Execute $\mathsf{EvalGC}(\mathsf{gckt}_{\mathsf{ind}}, \widetilde{w}_1^{\mathsf{ind}}, \ldots, \widetilde{w}_q^{\mathsf{ind}})$ to obtain $\mathsf{out}_{\mathsf{ind}}$.
  3. If $\mathsf{out}_{\mathsf{ind}} \neq \perp$ then output $\mathsf{out} = \mathsf{out}_{\mathsf{ind}}$. Otherwise, continue.

This completes the description of the main algorithms. We now describe the auxiliary algorithms.

$\underline{\mathsf{OEE.puncInp}(\mathsf{OEE.sk}, x)}$: The secret key $\mathsf{OEE.sk} = (\mathsf{TwoABE.SK}, \mathsf{TwoABE.PP}, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$ punctured at point x is $\mathsf{OEE.sk}_x = (\mathsf{TwoABE.PP}, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$. That is, the punctured key is same as the original secret key except that the master secret key of TwoABE is removed. Output $\mathsf{OEE.sk}_x$.

$\underline{\mathsf{OEE.pIEncode}(\mathsf{OEE.sk}_x, x')}$: On input a punctured key $\mathsf{OEE.sk}_x$ and input $x' \neq x$, it executes $\mathsf{OEE.InpEncode}(\mathsf{OEE.sk}_x, x', b)$ to obtain the result $\widetilde{(x', b)}$ which is set to be the output.

[Note: The algorithm $\mathsf{OEE.InpEncode}$ *can directly be executed on the punctured key* $\mathsf{OEE.sk}_x$ *and input* x' *because the master secret key* $\mathsf{TwoABE.SK}$ *is never used during its execution.]*

$\underline{\mathsf{OEE.puncBit}(\mathsf{OEE.sk}, b)}$: On input a secret key $\mathsf{OEE.sk}$ and a bit $b \in \{0, 1\}$, it first interprets $\mathsf{OEE.sk}$ as $(\mathsf{TwoABE.SK}, \mathsf{TwoABE.PP}, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$. It then outputs a punctured key $\mathsf{OEE.sk}_b = (\mathsf{TwoABE.PP}, \mathsf{FHE.pk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_b)$.

$\underline{\mathsf{OEE.pBEncode}(\mathsf{OEE.sk}_b, x)}$: On input the punctured key $\mathsf{OEE.sk}_b$, it computes $\widetilde{(x, b)} \leftarrow \mathsf{OEE.InpEncode}(\mathsf{OEE.sk}_b, x, b)$. The result $\widetilde{(x, b)}$ is then output.

[Note: The algorithm $\mathsf{OEE.InpEncode}$ *can directly be executed on the punctured key* $\mathsf{OEE.sk}_b$ *and input* x *because the FHE secret key associated to* $\overline{b}$*, namely* $\mathsf{FHE.sk}_{\overline{b}}$*, is never used during the execution.]*

This completes the description of the auxiliary algorithms. In the full version, we prove that our construction satisfies the desired correctness, efficiency and security properties.

4 Succinct $\mathsf{i}\mathcal{O}$ with Constant Multiplicative Overhead

Let $\mathsf{OEE} = (\mathsf{OEE.Setup}, \mathsf{OEE.InpEncode}, \mathsf{OEE.TMEncode}, \mathsf{OEE.Decode})$ be an OEE scheme with constant multiplicative overhead that is equipped with auxiliary algorithms $(\mathsf{OEE.puncInp}, \mathsf{OEE.pIEncode}, \mathsf{OEE.puncBit}, \mathsf{OEE.pBEncode})$. Let $\mathsf{i}\mathcal{O}$ be an indistinguishability obfuscator for general circuits. Let PRF be a puncturable PRF family. Using these primitives, we now give a construction of a succinct indistinguishability obfuscator with constant multiplicative overhead. We denote it by SuccIO.

Construction. Let $\mathcal{M}$ denote the family of turing machines. On input the security parameter and a turing machine $M \in \mathcal{M}$, $\mathsf{SuccIO}(1^\lambda, M)$ computes the following:

- $\mathsf{OEE.sk} \leftarrow \mathsf{OEE.Setup}(1^\lambda)$.
- $\widetilde{(M, M)} \leftarrow \mathsf{OEE.TMEncode}(\mathsf{OEE.sk}, M, M)$.
- $\widetilde{C} \leftarrow \mathsf{i}\mathcal{O}\left(C_{[K,\mathsf{OEE.sk}]}\right)$, where K is a randomly chosen key for the puncturable PRF family and $C_{[K,\mathsf{OEE.sk}]}$ is the circuit described in Figure 4.

$C_{[K,\mathsf{OEE.sk}]}(x)$

1. Compute $r \leftarrow \mathsf{PRF}_K(x)$.
2. Compute $\widetilde{(x, 0)} \leftarrow \mathsf{OEE.InpEncode}(\mathsf{OEE.sk}, x, 0)$ using randomness r.
3. Output $\widetilde{(x, 0)}$.

Fig. 4: Circuit $C_{[K,\mathsf{OEE.sk}]}$.

The output of the obfuscator is $\left(\widetilde{(M, M)}, \widetilde{C}\right)$.

To evaluate the obfuscated machine on an input x, the evaluator first computes $\widetilde{C}(x)$ to obtain $\widetilde{(x, 0)}$. Next, it computes $y \leftarrow \mathsf{OEE.Decode}\left(\widetilde{(M, M)}, \widetilde{(x, 0)}\right)$ and outputs y.

The proof of correctness and security can be found in the full version.

References

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

[2] Prabhanjan Ananth, Yu-Chi Chen, Kai-Min Chung, Huijia Lin, and Wei-Kai Lin. Delegating ram computations with adaptive soundness and privacy. In *TCC-II*, 2016.

[3] Prabhanjan Ananth, Divya Gupta, Yuval Ishai, and Amit Sahai. Optimizing obfuscation: Avoiding barrington's theorem. In *ACM CCS*, 2014.

[4] Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In *CRYPTO*, 2015.

[5] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Achieving compactness generically: Indistinguishability obfuscation from non-compact functional encryption. *IACR Cryptology ePrint Archive*, 2015:730, 2015.

[6] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Indistinguishability obfuscation for turing machines: Constant overhead and amortization. *IACR Cryptology ePrint Archive*, 2015:1023, 2015.

[7] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Patchable indistinguishability obfuscation: io for evolving software. In *EUROCRYPT*, 2017.

[8] Prabhanjan Ananth and Amit Sahai. Functional encryption for turing machines. In *TCC 2016-A*, 2016.

[9] Benny Applebaum and Zvika Brakerski. Obfuscating circuits via composite-order graded encoding. In *TCC*, 2015.

[10] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In *EUROCRYPT*, 2014.

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

[12] Mihir Bellare, Igors Stepanovs, and Brent Waters. New negative results on differing-inputs obfuscation. In *EUROCRYPT*, 2016.

[13] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Siddartha Telang. Succinct randomized encodings and their applications. In *STOC*, 2015.

[14] Nir Bitansky, Omer Paneth, and Alon Rosen. On the cryptographic hardness of finding a nash equilibrium. In *FOCS*, pages 1480–1498, 2015.

[15] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. In *FOCS*, 2015.

[16] Dan Boneh, Craig Gentry, Sergey Gorbunov, Shai Halevi, Valeria Nikolaenko, Gil Segev, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. Fully key-homomorphic encryption, arithmetic circuit ABE and compact garbled circuits. In *EUROCRYPT*, 2014.

[17] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *ASIACRYPT*, 2013.

[18] Dan Boneh, David J. Wu, and Joe Zimmerman. Immunizing multilinear maps against zeroizing attacks. *IACR Cryptology ePrint Archive*, 2014:930, 2014.

[19] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. In *TCC*, 2014.

[20] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *PKC*, 2014.

[21] Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In *TCC*, pages 1–25, 2014.

[22] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled ram or: How to delegate your database. In *TCC-II*, 2016.

[23] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. In *ITCS*, 2016.

[24] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Indistinguishability obfuscation of iterated circuits and RAM programs. In *STOC*, 2015.

[25] Angelo De Caro, Vincenzo Iovino, Abhishek Jain, Adam O'Neill, Omer Paneth, and Giuseppe Persiano. On the achievability of simulation-based security for functional encryption. In *CRYPTO*, pages 519–535, 2013.

[26] 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. In *ITCS*, 2016.

[27] Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, and Damien Stehlé. Cryptanalysis of the multilinear map over the integers. In *EUROCRYPT*, 2015.

[28] Aloni Cohen, Justin Holmgren, Ryo Nishimaki, Vinod Vaikuntanathan, and Daniel Wichs. Watermarking cryptographic capabilities. In *STOC*, pages 1115–1127, 2016.

[29] Jean-Sébastien Coron, Craig Gentry, Shai Halevi, Tancrède Lepoint, Hemanta K. Maji, Eric Miles, Mariana Raykova, Amit Sahai, and Mehdi Tibouchi. Zeroizing without low-level zeroes: New MMAP attacks and their limitations. In *CRYPTO*, 2015.

[30] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Cryptanalysis of two candidate fixes of multilinear maps over the integers. *IACR Cryptology ePrint Archive*, 2014:975, 2014.

[31] Sanjam Garg, Craig Gentry, and Shai Halevi. Candidate multilinear maps from ideal lattices. In *EUROCRYPT*, pages 1–17, 2013.

[32] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS*, 2013.

[33] Sanjam Garg, Craig Gentry, Shai Halevi, and Daniel Wichs. On the implausibility of differing-inputs obfuscation and extractable witness encryption with auxiliary input. In *CRYPTO*, 2014.

[34] Sanjam Garg, Eric Miles, Pratyay Mukherjee, Amit Sahai, Akshay Srinivasan, and Mark Zhandry. Secure obfuscation in a weak multilinear map model: A simple construction secure against all known attacks. In *TCC-B*, 2016.

[35] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[36] Craig Gentry, Shai Halevi, Hemanta K. Maji, and Amit Sahai. Zeroizing without zeroes: Cryptanalyzing multilinear maps without encodings of zero. *IACR Cryptology ePrint Archive*, 2014:929, 2014.

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

[38] Craig Gentry, Allison Lewko, and Brent Waters. Witness encryption from instance independent assumptions. In *Advances in Cryptology–CRYPTO 2014*, pages 426–443, 2014.

[39] Craig Gentry, Allison B. Lewko, Amit Sahai, and Brent Waters. Indistinguishability obfuscation from the multilinear subgroup elimination assumption. In *FOCS*, 2015.

[40] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *STOC*, 2013.

[41] Shafi Goldwasser, Yael Tauman Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In *CRYPTO*, 2013.

[42] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM CCS*, pages 89–98, 2006.

[43] Pavel Hubácek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In *ITCS*, pages 163–172, 2015.

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

[45] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *ACM STOC*, 2008.

[46] Yuval Ishai, Omkant Pandey, and Amit Sahai. Public-coin differing-inputs obfuscation and its applications. In *TCC*, 2015.

[47] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *ACM CCS*, 2013.

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

[49] Huijia Lin. Indistinguishability obfuscation from constant-degree graded encoding schemes. In *EUROCRYPT*, pages 28–57, 2016.

[50] Huijia Lin, Rafael Pass, Karn Seth, and Sidharth Telang. Output-compressing randomized encodings and applications. In *TCC-I*, 2016.

[51] Huijia Lin and Vinod Vaikuntanathan. Indistinguishability obfuscation from ddh-like assumptions on constant-degree graded encodings. In *FOCS*, 2016.

[52] Eric Miles, Amit Sahai, and Mark Zhandry. Annihilation attacks for multilinear maps: Cryptanalysis of indistinguishability obfuscation over GGH13. In *CRYPTO*, pages 629–658, 2016.

[53] Rafael Pass, Karn Seth, and Sidharth Telang. Indistinguishability obfuscation from semantically-secure multilinear encodings. In *CRYPTO*, 2014.

[54] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In *EUROCRYPT*, pages 457–473, 2005.

[55] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *ACM STOC*, 2014.

[56] Joe Zimmerman. How to obfuscate programs directly. In *EUROCRYPT*, 2015.