Patchable Indistinguishability Obfuscation: $i\mathcal{O}$ for Evolving Software

Prabhanjan Ananth[*] Abhishek Jain[†] Amit Sahai[‡]

Abstract

In this work, we introduce *patchable indistinguishability obfuscation*: our notion adapts the notion of indistinguishability obfuscation ($i\mathcal{O}$) to a very general setting where obfuscated software evolves over time. We model this broadly by considering software patches P as arbitrary Turing Machines that take as input the description of a Turing Machine M, and output a new Turing Machine description $M' = P(M)$. Thus, a short patch P can cause changes everywhere in the description of M and can even cause the description length of the machine to increase by an arbitrary polynomial amount. We further consider*multi-program* patchable indistinguishability obfuscation where a patch is applied not just to a single machine M, but to an unbounded set of machines $M_1, \ldots, M_n$ to yield $P(M_1), \ldots, P(M_n)$.

We consider both single-program and multi-program patchable indistinguishability obfuscation in a setting where there are an unbounded number of patches that can be *adaptively* chosen by an adversary. We show that sub-exponentially secure $i\mathcal{O}$ for circuits and sub-exponentially secure re-randomizable encryption schemes[1] imply single-program patchable indistinguishability obfuscation; and we show that sub-exponentially secure $i\mathcal{O}$ for circuits and sub-exponentially secure DDH imply multi-program patchable indistinguishability obfuscation.

At the our heart of results is a new notion of *splittable* $i\mathcal{O}$ that allows us to transform any $i\mathcal{O}$ scheme into a patchable one. Finally, we exhibit some simple applications of patchable indistinguishability obfuscation, to demonstrate how these concepts can be applied.

[*]University of California Los Angeles and Center for Encrypted Functionalities. Email: `prabhanjan@cs.ucla.edu`. Work done in part while visiting the Simons Institute for Theoretical Computer Science, supported by the Simons Foundation and and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467. 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`. Work done in part while visiting the Simons Institute for Theoretical Computer Science, supported by the Simons Foundation and and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467. Supported in part by a DARPA/ARL Safeware Grant W911NF-15-C-0213 and NSF CNS-1414023.

[‡]University of California Los Angeles and Center for Encrypted Functionalities. Email: `sahai@cs.ucla.edu` Work done in part while visiting the Simons Institute for Theoretical Computer Science, supported by the Simons Foundation and and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467. Research supported in part from a DARPA/ARL SAFEWARE award, NSF Frontier Award 1413955, NSF grants 1228984, 1136174, 1118096, and 1065276, 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 author and do not reflect the official policy or position of the Department of Defense, the National Science Foundation, or the U.S. Government.

[1]Re-randomizable encryption schemes can be instantiated under standard assumptions such as DDH, LWE.

Contents

1 Introduction

Program obfuscation is the process of making a program "unintelligible" to any polynomial-time entity while preserving its functionality. A formal study of program obfuscation was initiated more than a decade ago in the works of [Had00, BGI+12]. In the recent years, this research area has seen renewed activity with the emergence of candidate constructions [GGH+13] for a type of general-purpose program obfuscation called indistinguishability obfuscation. This notion has proven to be both extremely useful and the most plausible of existing notions of program obfuscation.

A major limitation of existing notions of program obfuscation is that they only consider "static" programs that do not change with time. In reality, however, programs are rarely changeless. We typically alter programs over time, with *patches* (a.k.a updates) causing the programs to grow and vary, in response to demands for greater or new functionality. Can program obfuscation be adapted to deal with this reality? Specifically, can we obfuscate programs that *evolve* over time? The central intellectual and theoretical focus of this work is to answer this question.

Obfuscation for Evolving Software. A trivial solution to obfuscating evolving software would be to simply apply the obfuscator afresh to each updated version of a particular program. For example, to modify an obfuscation of a program M, the obfuscator may simply release a fresh obfuscation of M' where M' is the patched version of M. Note, however, that in this solution, the total communication complexity is at least $|M| + |M'|$. In particular, this is the case *even if the difference between the programs M and M' can be described in the form of a small patch P*. In contrast, if M was not obfuscated, then we could modify it by simply communicating the patch P to a user, yielding a total communication complexity of only $|M| + |P|$. Our goal is to develop a mechanism for program obfuscation that approximately preserves this communication complexity.

A bit more precisely, we define a notion of *patchable* obfuscation where, informally, there are four algorithms:

- $\mathsf{Obf}(M; r)$ taking as input a program M, and outputting an obfuscated program $\langle M \rangle$, using randomness r.
- $\mathsf{GenPatch}(P; r, r')$ taking as input a patch P, and outputting an encoded patch $\langle P \rangle$, using a combination of the original randomness r and new randomness r'.
- $\mathsf{AppPatch}\left(\langle M \rangle, \langle P \rangle\right)$ taking as input an obfuscated program $\langle M \rangle$ and a patch encoding $\langle P \rangle$, and outputting an obfuscated patched program $\langle M' = P(M) \rangle$.
- $\mathsf{Eval}\left(\langle M \rangle, x\right)$, taking as input an obfuscated program $\langle M \rangle$ and an input x, and outputting the value $y = M(x)$.

The key efficiency requirement is that the size of a patch encoding should not depend on the size of the original program M. Specifically, we want that $|\langle P \rangle| = \text{poly}(|P|, \lambda)$, where λ is the security parameter.

Beyond this basic efficiency requirement, we also discuss some other important considerations w.r.t. patchable obfuscation.

I. No restriction on patches: An important consideration for patchable obfuscation is the class of patches that we wish to allow. Clearly, the larger the class of patches that we can support, the larger the potential application pool.

To maximize the applicability of our notion, we allow for *arbitrary* patches. Specifically, we model a patch P as a Turing machine that takes as input a program M (also modeled as a TM) and outputs a new program M'. We allow for the unpatched program to *grow* in size after patching. That is, M' may be arbitrarily bigger than M.

II. Multiple patches: Another consideration is the number of patches that we wish to allow. In reality, it may be difficult to anticipate in advance how many times a program may need to be patched. Thus, we allow for an *unlimited* number of patches.

Specifically, we consider two modes of patching:

- Sequential patching: Here, given an obfuscated program $\langle M_0 \rangle$ and a sequence of patch encodings $\langle P_1 \rangle, \ldots, \langle P_n \rangle$, one can apply the patches one-by-one, *in order*, to obtain $\langle M_1 \rangle, \ldots, \langle M_n \rangle$ s.t. $M_i = P_i(M_{i-1})$.
- Parallel patching: Here, given an obfuscated program $\langle M_0 \rangle$ and a sequence of patch encodings $\langle P_1 \rangle, \ldots, \langle P_n \rangle$, one can apply each patch to $\langle M \rangle$, *in parallel*, to obtain $\langle M_1 \rangle, \ldots, \langle M_n \rangle$ s.t. $M_i = P_i(M_0)$.

While sequential patching seems to better capture patching of programs in reality, as we discuss later, parallel patching also enables interesting applications of patchable obfuscation. Thus, we consider both patching modes in this work.

III. Support for multiple programs: So far, we have only discussed patching for a single obfuscated program. Now consider the case where an authority wishes to patch *multiple* obfuscated programs $\langle M_1 \rangle, \ldots, \langle M_n \rangle$. Such a situation often arises in practice where, for example, the programs $M_1, \ldots, M_n$ may correspond to different copies of the same core program M that are individualized to different users.

One approach to address this scenario would be to release a separate patch for every obfuscated program. In this case, however, the communication complexity grows linearly with the number of obfuscated programs and may quickly become prohibitive. Instead, we would like to build patchable obfuscation where the obfuscator can release *one* patch that can be applied to all of the obfuscated programs. We refer to this notion as *multi-program patchable obfuscation.*

How to Define Security? Of course, we must define security for patchable obfuscation. The natural direction is to start with a "base" notion of obfuscation (without patching) and extend it to the setting of patching. Our goal in this work is to obtain general positive results for patchable obfuscation. With this viewpoint, we identify indistinguishability obfuscation ($i\mathcal{O}$) [BGI+12] as a natural choice for the base notion. Indeed, over the last few years, several general-purpose candidate constructions, (for example: [GGH+13, BGK+14, BR14]) for $i\mathcal{O}$ have been proposed, and no impossibility results are known. Furthermore, it was shown by [GR07] that $i\mathcal{O}$ is, in fact, "best-possible" obfuscation. $i\mathcal{O}$ has already enabled a long sequence of exciting applications (see e.g., [SW14, GGH+13, BGJ+16, CHN+16]) and its patchable analogue can be expected to find even more applications. Finally, we stress that while the security of $i\mathcal{O}$ remains an area of intense study, there are several known $i\mathcal{O}$ candidates and even *universal* $i\mathcal{O}$ candidates under well-studied assumptions [AJN+16].

In contrast, powerful (base) notions such as virtual black-box obfuscation [BGI+12] and differing-inputs obfuscation [BGI+12, BCP14, ABG+13] have been shown to be impossible to realize for general functions [BGI+12, GK05, BCC+14, GGHW14, BSW16]. This, in turn, means that patchable analogues of these notions are also impossible, in general. The notion of virtual grey-box obfuscation [BC10, BCKP14] is impossible for general Turing Machines but seems to circumvent general impossibility results for circuits; however, it has found rather limited applicability so far.

In light of the above, in this work, we focus on patching in the context of $i\mathcal{O}$. We do believe that the study of patchable obfuscation for other base obfuscation notions (e.g., obfuscation in weaker adversarial models such as virtual black-box obfuscation in hardware token model [GO96, GKR08, GIS$^+$10] or generic model [BR14, BGK$^+$14]) is interesting, and we leave this study to future work. We remark that many of the ideas that we develop in this work should be more widely applicable to other notions of obfuscation, and are not intrinsically tied to $i\mathcal{O}$. As such, we envision these ideas to be portable to other notions of patchable obfuscation.

Patchable Indistinguishability Obfuscation. We develop a notion of *patchable indistinguishability obfuscation* (*pa-*$i\mathcal{O}$)that naturally extends the standard notion of $i\mathcal{O}$ to the setting of patching. Let us explain our notion for the single-program case, for sequential and parallel patches.

- *Sequential patches*: Recall that $i\mathcal{O}$ security dictates that given two equivalent programs M_0 and M_1, obfuscations of M_0 and M_1 are computationally indistinguishable. In single-program *pa-*$i\mathcal{O}$ for sequential patches, we require that given two equivalent programs M_0^0 and M_1^0 and a sequence of patch pairs $(P_0^1, P_1^1), \ldots, (P_0^n, P_1^n)$ such that for every "level" $i \in [n]$, the patched programs $M_0^i = P_0^i(M_0^{i-1})$ and $M_1^i = P_1^i(M_1^{i-1})$ are also equivalent, it should be hard to distinguish the tuples $(\langle M_0^0\rangle, \{\langle P_0^i\rangle\}_{i=1}^n)$ and $(\langle M_1^0\rangle, \{\langle P_1^i\rangle\}_{i=1}^n)$. Intuitively, the equivalence requirement at every patch level i rules out the trivial attack of using a splitting input for the patched programs M_0^i and M_1^i to distinguish the tuples.
- *Parallel patches*: Single-program *pa-*$i\mathcal{O}$ for parallel patches is defined similarly to above, except that here we require equivalence for the patched programs $M_0^i = P_0^i(M_0^0)$ and $M_1^i = P_1^i(M_1^0)$ at every (parallel) "branch" $i \in [n]$.

A few remarks are in order: **(1)** It is easy to see that these definitions ensure *patch hiding*, which is crucial for some of the applications discussed later. **(2)** Our definitions naturally extend to multi-program *pa-*$i\mathcal{O}$ where we start with multiple pairs of programs and equivalence is required for every pair at every level/branch. **(3)** We, in fact, consider *adaptive* security, where the adversary can make the patch queries in an adaptive fashion. See Section 2 for further details.

Implications of *pa-*$i\mathcal{O}$. We view *pa-*$i\mathcal{O}$ as a powerful primitive that is likely to have several applications in the future. To see the power of *pa-*$i\mathcal{O}$, it is instructive to first compare it with $i\mathcal{O}$. While $i\mathcal{O}$ exists if **P**=**NP**,[2] we show that multi-program *pa-*$i\mathcal{O}$ for parallel patches implies secret-key functional encryption (FE) [SW05, BSW11, O'N10]. The construction is remarkably simple: let $M_{f,x}$ be an input-less machine that simply outputs $f(x)$. We construct an FE scheme as follows:

- A secret key for a function f is computed as $\langle M_{f,\perp}\rangle$, i.e., an obfuscation of $M_{f,x}$ where $x = \perp$.
- Encryption of a message m corresponds to generating an encoding $\langle P_m\rangle$ for a patch P_m that modifies $M_{f,\perp}$ to $M_{f,m}$.
- Decryption simply corresponds to applying the patch encoding $\langle P_m\rangle$ on $\langle M_{f,\perp}\rangle$ to obtain $\langle M_{f,m}\rangle$ and then evaluating it to obtain $f(m)$.

Correctness and security of the construction follow in a straightforward manner from the correctness and security of *pa-*$i\mathcal{O}$.[3] As we discuss later, the above basic idea can, in fact, be easily extended to multi-input functional encryption [GGG$^+$14], yielding new results.

[2]Assuming **NP** $\neq$ **co-RP**, it was shown that $i\mathcal{O}$ implies one-way functions [MR13, KMN$^+$14].

[3]An observant reader may notice that in the above construction, it is not important whether the size of a patch encoding depends on the size of an unpatched machine $M_{f,\perp}$ or not. However, it is important that the size of the patch encoding is independent of the number of obfuscated machines that it can be applied to – a property guaranteed by multi-program *pa-*$i\mathcal{O}$.

Alternate viewpoint: Obfuscation with Private Homomorphism. Another way of looking at our notion of *pa-i$\mathcal{O}$* is as a form of $i\mathcal{O}$ that supports a kind of semi-private *homomorphism*: the generation of the patch encoding is private – requiring secret information that was used to obfuscate the original program – although the application of the patch encoding is public. Note that unlike encryption, for the security of obfuscation it is critical that this homomorphism is semi-private – if an adversary was allowed to use public information to arbitrarily modify the program underlying an obfuscation, this would trivially allow the adversary to break the security of the original obfuscated program. On the other hand, our notion of *pa-i$\mathcal{O}$* and the notion of fully homomorphic encryption [Gen09] share a similarity in that they both require a form of compactness for the notions to be non-trivial.

1.1 Our Results

We state our results below.

I. Patchable Indistinguishability Obfuscation. In this work, we formalize the notion of patchable indistinguishability obfuscation. We focus on the setting where programs to be obfuscated and patched are described as Turing Machines.

Multi-Program pa-i$\mathcal{O}$: Our main result is a construction of a multi-program *pa-i$\mathcal{O}$* scheme from sub-exponentially secure $i\mathcal{O}$ and sub-exponentially secure DDH.

Theorem 1 (Multi-program *pa-i$\mathcal{O}$*: Sequential patches)**.** *Assuming the existence of sub-exponentially secure i$\mathcal{O}$ for circuits, sub-exponentially secure DDH, there exists an adaptively secure multi-program pa-i$\mathcal{O}$ scheme with unbounded sequential patches, for Turing Machines where the running time of the patch generation algorithm for a patch P is bounded by $poly(\lambda, |P|, \ell)$, where λ is a security parameter and ℓ is a bound on the input size to the patched program.*

Note that the runtime efficiency of the patch generation algorithm in the above theorem implies the necessary size efficiency for a patch encoding, namely, the size of the encoding of a patch P is bounded by $poly(\lambda, |P|, \ell)$.

Single-Program pa-i$\mathcal{O}$: We obtain the above result in two steps. Our first, and key step is to construct a single-program *pa-i$\mathcal{O}$* scheme for TMs which achieves the desired size efficiency for patches but requires a large state (proportional to the size of the TM being updated) as well as a large patch generation time.

Theorem 2 (Single-program *pa-i$\mathcal{O}$*: Sequential patches)**.** *Assuming the existence of sub-exponentially secure i$\mathcal{O}$ for circuits and sub-exponentially secure re-randomizable encryption schemes, there exists an adaptively secure single-program pa-i$\mathcal{O}$ scheme with unbounded sequential patches, for Turing Machines where the size of the obfuscation of a patch P is bounded by $poly(\lambda, |P|, \ell)$, where λ is a security parameter and ℓ is a bound on the input size to the patched program.*

Main Tool: Splittable i$\mathcal{O}$. The main tool in our construction of single-program is an intermediate notion between $i\mathcal{O}$ and patchable $i\mathcal{O}$, that we refer to as *splittable i$\mathcal{O}$*. Very roughly, splittable $i\mathcal{O}$ allows us to reduce the problem of building patchable $i\mathcal{O}$ to the problem of building a patchable "encoding" scheme, a seemingly simpler problem. Very roughly, an obfuscation of M w.r.t. splittable $i\mathcal{O}$ consists of two parts: an encoding of M w.r.t. a patchable encoding scheme, and some auxiliary information z computed on the encoding as well as the secret key used to encode M. We place suitable efficiency and security requirements on the auxiliary information so as to allow us to transfer the patching property of the encoding scheme to the setting of $i\mathcal{O}$. We refer the reader to the technical overview section for further details on this notion.

From Single-Program to Multi-Program pa-$i\mathcal{O}$: Next, we devise a generic transformation from any such single-program *pa-$i\mathcal{O}$* scheme to a multi-program *pa-$i\mathcal{O}$* scheme with the aforementioned efficient patch generation property.

Theorem 3 (Single-program to Multi-program *pa-$i\mathcal{O}$*)**.** *Assuming the existence of a succinct garbled TM scheme with persistent memory and a compact secret-key functional encryption scheme for general circuits, there exists a general transformation from any single-program pa-$i\mathcal{O}$ scheme to a multi-program pa-$i\mathcal{O}$ scheme for TMs with efficient patch generation.*

In particular, when the underlying primitives are all adaptively secure, then the resulting multi-program *pa-$i\mathcal{O}$* scheme is also adaptively secure. An adaptively secure succinct garbled TM scheme with persistent memory is known from the works of [CCHR B, ACC+ B] based on sub-exponentially secure $i\mathcal{O}$ and DDH assumption, while a compact secret-key functional encryption scheme is known from $i\mathcal{O}$ for general circuits.

For the theorems above, we stress that we place no restrictions on the patches. A patch P can be an arbitrary Turing Machine that takes the original program description M as input, and outputs an arbitrary Turing Machine description $M' = P(M)$ that can differ in arbitrary ways from M. In particular, the description size of $P(M)$ can be any unbounded polynomial in the security parameter, and thus the program size can grow by arbitrary polynomial factors. Furthermore any unbounded polynomial number of patches can be applied sequentially, and the adversary can specify these patches adaptively given all obfuscated programs and patches constructed earlier.

Parallel Patching: We can obtain a similar result for multi-program *pa-$i\mathcal{O}$* in the context of parallel patches. This result follows the same approach as the case of sequential patches. The first step is to obtain single-program *pa-$i\mathcal{O}$* scheme with unbounded parallel patches and the second step is to obtain multi-program *pa-$i\mathcal{O}$* from single-program *pa-$i\mathcal{O}$*. The construction of single-program *pa-$i\mathcal{O}$* with parallel patches will be identical to the one in the sequential patch setting. The transformation from single-program *pa-$i\mathcal{O}$* to multi-program *pa-$i\mathcal{O}$* is, however, different from the sequential setting to enable this transformation. Instead of using garbled TM scheme with persistent memory, we instead employ functional encryption for TMs [GKP+13b, AS16] scheme. Since the techniques employed in the parallel patch setting are similar to the sequential patch setting, we omit the transformation. We have the following theorem.

Theorem 4 (Multi-program *pa-$i\mathcal{O}$*: Parallel patches)**.** *Assuming the existence of sub-exponentially secure $i\mathcal{O}$ for circuits, sub-exponentially secure DDH, there exists an adaptively secure multi-program pa-$i\mathcal{O}$ scheme with unbounded parallel patches, for Turing Machines where the running time of the patch generation algorithm for a patch P is bounded by poly$(\lambda, |P|, \ell)$, where λ is a security parameter and ℓ is a bound on the input size to the patched program.*

II. Applications of *pa-$i\mathcal{O}$*. We view *pa-$i\mathcal{O}$*, and especially multi-program *pa-$i\mathcal{O}$* as a powerful primitive that is likely to have several applications in the future. As initial evidence of this, we demonstrate implications of *pa-$i\mathcal{O}$* to functional encryption and $i\mathcal{O}$ for TMs. In our eyes, the main appeal of these implications is their remarkable *simplicity* that highlights the potential of *pa-$i\mathcal{O}$* as a replacement for $i\mathcal{O}$ in cryptographic applications.

Multi-Input FE for Unbounded Arity Functions: We first show that multi-program *pa-*$i\mathcal{O}$ for parallel updates implies secret-key multi-input functional encryption (MIFE) [GGG+14, AJ15, BKS16] for unbounded arity functions. This implication follows from a straightforward extension of the *pa-*$i\mathcal{O}$ to (single-input) FE implication discussed earlier.

Theorem 5 (Unbounded-Arity MIFE)**.** *Adaptively secure multi-program pa-*$i\mathcal{O}$ *with unbounded parallel updates implies secret-key MIFE for unbounded arity functions with security against pre-ciphertext key queries.*

Combining the above with Theorem 4, we obtain secret-key MIFE for unbounded arity functions from sub-exponentially secure $i\mathcal{O}$ for circuits, sub-exponentially secure DDH. Previously, this result was only known [BGJS15] from a knowledge assumption, namely public-coin differing-input obfuscation [IPS15] and one-way functions.

FE for TMs with Unbounded Length Inputs: The following implication follows as a simple corollary of Theorem 5.

Theorem 6 (Unbounded-Input FE)**.** *Adaptively secure multi-program pa-*$i\mathcal{O}$ *implies secret-key functional encryption for TMs with unbounded input length with security against pre-ciphertext key queries.*

A construction of FE for TMs with unbounded input was recently given by [AS16] based on $i\mathcal{O}$. We emphasize that our construction from multi-program *pa-*$i\mathcal{O}$ is extremely simple, in contrast to the involved construction of [AS16].

We now discuss implications of *pa-*$i\mathcal{O}$ to $i\mathcal{O}$ for TMs. We first recall that all recent progress on achieving $i\mathcal{O}$ for TMs/RAMs[CHJV15, BGL+15, KLW15, CH16, CCC+16] from $i\mathcal{O}$ for circuits has required a polynomial bound ℓ to be placed on the input length to the obfuscated Turing Machine. We share this need for a polynomial bound ℓ on the input size, and the size of our obfuscated patches do grow with this bound. Indeed, if we could remove this restriction, then we would show how to bootstrap $i\mathcal{O}$ for circuits to $i\mathcal{O}$ for Turing Machines without any input length restriction from $i\mathcal{O}$ for circuits – this remains a major open question. Achieving $i\mathcal{O}$ for Turing Machines without any input length restriction currently requires strong assumption such as output-compressing randomized encodings [LPST16] or knowledge-type assumptions such as public-coin *di*$\mathcal{O}$ [BCP14, ABG+13, IPS15]. We do not know how to achieve these objects using only $i\mathcal{O}$ for circuits.

$i\mathcal{O}$ for TMs with Unbounded Length Inputs. So far, in our definition of *pa-*$i\mathcal{O}$, we have only considered "single-use" patches. More accurately, in our definition of single-program (resp., multi-program) *pa-*$i\mathcal{O}$ for sequential patching, the i^{th} patch P_i can only be applied to the updated machine (resp., machines) at level $i-1$. As we discuss now, such "single-use" patches are, in fact, inherent given the current state of art in $i\mathcal{O}$ for TMs.

In particular, is not difficult to see that single-program *pa-*$i\mathcal{O}$ with *reusable* patches (i.e., where a patch P is not tied to any "level" and can be applied an arbitrary number of times, to any machine) in fact, implies $i\mathcal{O}$ for TMs with unbounded length inputs. The construction is extremely simple: let M_x be a family of (input-less) machines parameterized by strings x of arbitrary length, where every machine simply outputs $M(x)$. Obfuscation of a TM M consists of an obfuscation of a machine $M_\perp$ w.r.t. the *pa-*$i\mathcal{O}$ scheme along with encodings of two reusable patches P_0 and P_1.

Patch P_0 is such that it updates any machine M_x to $M_{x\|0}$ while P_1 updates any machine M_x to $M_{x\|1}$.

To evaluate the above obfuscation on any input $x = x_1, \ldots, x_\ell$ for an arbitrary ℓ, a user can transform obfuscation of $M_\perp$ to M_x by applying the patches $P_{x_1}, \ldots, P_{x_n}$ and then execute M_x to obtain $M(x)$. The correctness of the construction is easy to verify.

While we do not consider security for reusable patches in this work, we view the above as a potential new template for building $i\mathcal{O}$ for TMs with unbounded length inputs.

1.2 Technical Overview

We now give an overview of the main technical ideas in our constructions. We start by building a general template for building *pa-*$i\mathcal{O}$, and then discuss our ideas for implementing this template.

1.2.1 A Template for *pa-*$i\mathcal{O}$

In this section, we devise a general template for building *pa-*$i\mathcal{O}$ starting from any non-patchable obfuscation scheme. We keep the discussion in this section to a high-level, focusing on issues directly related to *patching*, and largely ignoring implementation issues that may arise due to the specific properties of the underlying non-patchable obfuscation scheme. For simplicity, in this section, we advise the reader to think of the non-patchable obfuscation scheme as general-purpose virtual-black-box obfuscation. Later, in Section 1.2, we discuss the additional challenges that arise in implementing our template when the non-patchable obfuscation scheme is $i\mathcal{O}$, and our solutions for the same.

Let us start with the weaker goal of building single-program *pa-*$i\mathcal{O}$ where the authority issues a single obfuscated program that can then be patched multiple times, in a sequential order. Our initial idea towards achieving this goal is to identify an encoding scheme that supports patching and then combine it with a non-patchable obfuscation scheme to build a *pa-*$i\mathcal{O}$ scheme. Intuitively, we say that an encoding scheme is patchable if given an encoding of a machine M and an encoding of a patch P, it is possible to derive an encoding of $M' = P(M)$. The hope here is that the patching property of the encoding scheme can be translated into patching property for obfuscation.

A natural candidate for a patchable encoding scheme is *fully homomorphic encryption* (FHE). Indeed, given an encryption (i.e., encoding) of a machine M and an encryption of a patch P, one can obtain an encryption of the patched machine $M' = P(M)$ by homomorphically evaluating the function $f(M, P) = P(M)$. Starting with FHE and any non-patchable obfuscation scheme, we can build an initial template for *pa-*$i\mathcal{O}$ as follows: to obfuscate M, first encrypt M using FHE and then provide an obfuscation of the FHE decryption circuit that has the FHE decryption key hardcoded into it. Evaluation on an input x can be done as follows: first use FHE evaluation to transform encryption of M into an encryption of $M(x)$, and then use the obfuscated decryption circuit to obtain $M(x)$. To patch the obfuscated program, we can simply patch the encryption of M in the manner as described above.

While this solution seems to offer the functionality of patching, it does not offer any security. Specifically, in the above template, an adversary can choose an arbitrary patch P^* on its own and then use FHE evaluation of the function $f_{P^*}(M) = P^*(M)$ to transform encryption of M into an encryption of $P^*(M)$. If this patch P^* is such that for two equivalent machines M_0 and M_1, $P^*(M_0)$ and $P^*(M_1)$ are not equivalent, then the adversary can easily break the security of *pa-*$i\mathcal{O}$. Indeed, the security of *pa-*$i\mathcal{O}$ prevents an adversary from creating patches on its own, while the above template does not place this restriction in any way. In particular, we need to crucially use the fact that patch generation is a secret key operation.

Towards that end, we modify the above template such that an evaluator can only apply *authenticated* patches. The obfuscation of M consists of an FHE encryption of M as before but the obfuscated FHE decryption circuit now takes as input old encryption $Enc(M)$, updated encryption $Enc(M')$, encrypted patch $Enc(P)$, a signature σ on $Enc(P)$ and an input x. It checks if the signature is valid and also if $Enc(M')$ is obtained by updating $Enc(M)$ using P. If the check passes, then it decrypts $Enc(M')$ and evaluates M' on x. During the patching phase, the authority sends both $Enc(P)$ and the signature σ. This signature now prevents a user from applying "invalid" patches to the obfuscation; however, we note that in the context of $i\mathcal{O}$, this authentication will need to be done in a much more careful manner, as we elaborate below.

Enforcing Ordered Executions of Patches. While the above template does not seem to suffer from any immediate issues when we consider a single patch, unfortunately, its security breaks down when we consider the setting of multiple patches. Indeed, in the above template, given (say) two patch encodings $(Enc(P_1), \sigma_1)$, $(Enc(P_2), \sigma_2)$, an adversary may first apply the second patch and then the first patch, which may break the equivalence requirement on the patched machines in the security definition of *pa-*$i\mathcal{O}$. In fact, an adversary can also repeatedly apply the same patch multiple times in the above template, which may also break the equivalence requirement on the patched machines in the security definition of *pa-*$i\mathcal{O}$. Indeed, the definition of *pa-*$i\mathcal{O}$ requires that the patch encodings can only be applied *in order*, namely, the i^{th} patch encoding can only be applied to the $(i-1)^{th}$ patched obfuscation, *once*.

Towards this, we introduce a mechanism to force a user to apply the patches in order. We begin by observing that instead of authenticating the encrypted patch in the above template, if we instead authenticate the encrypted patched machine, then we can enforce ordered executions of patches. That is, suppose we want to update the machine M using patch P, the authority first computes $Enc(P)$ and then updates $Enc(M)$ using $Enc(P)$ to obtain $Enc(M')$. It then signs $Enc(M')$ and sends the signature[4] σ and the encrypted patch $Enc(P)$ to the user. The user now updates $Enc(M)$ using $Enc(P)$ to obtain $Enc(M')$. To evaluate the patched obfuscation on an input x, it inputs $(Enc(M'), \sigma, x)$ to the obfuscated FHE decryption circuit that first checks for validity of the signature and then decrypts $Enc(M')$ followed by computation of $M'(x)$, as before. Crucially, by shifting the authentication to the updated encrypted machine instead of encrypted patch, we are now able to prevent the "out-of-order patching" attacks (as well as "repeated patching" attacks) by an adversary discussed above.

A disadvantage of the above solution is that it requires the authority to maintain large state. In particular, at any time, the authority must remember the last patched machine M_{i-1} in order to generate a valid encoding for the i^{th} patch P_i. Furthermore, the patch encoding generation time now depends on the size of the machine M_{i-1}. While this loss in efficiency may be acceptable for the setting of single-program *pa-*$i\mathcal{O}$, it unfortunately becomes a significant barrier for the setting of multi-program *pa-*$i\mathcal{O}$. Indeed, in the multi-program setting, the number of obfuscated programs are not a priori bounded; as such, if we were to extend the above template to this case, then the authority's state size becomes unbounded! (This is because the authority would need to maintain a separate state for every obfuscated program.)

Compressing the State of Authority. In order to resolve this issue we introduce the next

[4]For this discussion, let us assume that we have a signature scheme where the size of the signature is independent of the length of the message. We will revisit this later when we discuss implementation issues.

idea: "delegating" the state of the authority to the user. That is, the authority now maintains the state at the user's end. Implementing this idea introduces several issues: not only should the state be encrypted at the user's end but it should also be possible to repeatedly update and also compute on this (updated) encrypted state. To address these issues, we turn to a cryptographic primitive called garbled RAMs with persistent memory. This notion allows for encoding a database and repeatedly update this encoding and compute on the updated encodings. The updating and computation operations are enabled by using encodings of RAM programs which are issued by the authority. Using this primitive, we propose a solution template.

- To obfuscate M, the authority computes: (i) $Enc(M)$ and a signature upon it. (ii) An obfuscation of the FHE decryption circuit (as before) that takes an input x, $Enc(M)$ and a signature σ, and outputs $M(x)$ if the signature is valid. (iii) A database encoding $\widetilde{Enc(M)}$ of $Enc(M)$. It then sends $\widetilde{Enc(M)}$, $Enc(M)$, σ and the obfuscated decryption circuit to the user.

- To evaluate the obfuscation on an input x, the user inputs $(x, Enc(M), \sigma)$ to the obfuscated decryption circuit to recover the output $M(x)$.

- To compute a patch encoding of P, the authority first computes $Enc(P)$ (as before) and then computes a garbled RAM encoding $\widetilde{T}$ of a RAM machine T that has $Enc(P)$ hardcoded in it. The machine T uses FHE evaluation over $Enc(M)$ (in the database encoding) and $Enc(P)$ to compute $Enc(M')$ and additionally computes signature σ' over $Enc(M)'$. It outputs σ' in the clear. The user, upon receiving the patch encoding, first computes $Enc(M')$ using $Enc(P)$. It then updates the database encoding $\widetilde{Enc(M)}$ using $\widetilde{T}$. The result is an updated database encoding $\widetilde{Enc(M')}$ and the signature σ' on $Enc(M')$. The user can now evaluate the updated machine on any input in the same manner as before.

Some remarks are in order: first, from an efficiency viewpoint, we need the garbled RAM scheme to be *succinct* where the size of RAM machine encoding is independent of its running time. This is because we are applying the above idea on a single-program *pa-i$\mathcal{O}$* scheme where the patch generation time depends on the size of the machine being updated. Second, in order to argue security in the setting of adaptively chosen patches, we need the garbled RAM scheme to satisfy adaptive security as well. Such a garbled RAM scheme (with persistent memory) was recently constructed in the independent works of [CCHR B, ACC$^+$ B].

Finally, we note that while the above idea successfully compresses the state size of the authority, it still does not suffice for the multi-program setting. This is because in the above solution, when extended to the multi-program case, the authority would need to maintain some small state, namely, the garbling key, for *every* obfuscated machine, which still leads to a state of unbounded size. We address this problem by developing a generic transformation from any single-program *pa-i$\mathcal{O}$* scheme with small state (or alternatively, a *stateless* scheme) into a multi-program *pa-i$\mathcal{O}$* scheme by using a compact secret-key functional encryption scheme for general circuits. We defer the discussion of this transformation to the next section.

1.2.2 Implementation

Issues related to Indistinguishability Obfuscation. While the above template seems promising, several issues arise when we have to implement it only assuming indistinguishability obfuscation *for circuits*. For starters, the above template requires an obfuscation scheme for Turing machines with unbounded length inputs. This is because, the size of the encrypted machine M can grow arbitrarily over a sequence of updates and thus the input to the obfuscated circuit cannot be a priori bounded. We currently know how to realize this only based on strong knowledge-type assumptions [BCP14, ABG+13, IPS15]. Another technical issue is that standard signature schemes are not "compatible" with iO and more generally, using iO restricts the type of cryptographic primitives that we can use. These challenges were encountered in many recent works [BGL+15, KLW15, CHJV15] whose main goal was reducing the problem of constructing iO for Turing machines, where the length of inputs to be evaluated are a priori bounded, to the problem of constructing iO for circuits. We build upon the primitives and notions introduced in the work of [KLW15] to address these challenges. We recall the Turing machine randomized encodings[5] construction by [KLW15].

The core idea in the randomized encodings construction of Koppula et al. [KLW15] is to leverage an obfuscated circuit to perform step-by-step computation of the machine M that is encoded. In more detail, a randomized encoding of (M, x) consists of: (a) input tape initialized with an encoding of M and, (b) an obfuscated circuit C_x that performs "step-by-step" computation of a machine $U_x(\cdot)$. Here, $U_x(\cdot)$ is a universal TM that takes as input machine M and outputs $M(x)$. By step-by-step computation, we mean that the circuit C_x takes as input time step i, encoded symbol and partial information about the current state in an encrypted form and produces a new encoded symbol and state, again in encrypted form, by executing the transition function of U_x. This enables the size of the circuit C_x to be independent of the length of M.

To see how the randomized encodings construction might be useful to our setting, note that we could potentially encode the machine M using a patchable encoding scheme that will allow us to patch M. Furthermore, we can allow the machine size to arbitrarily grow, over a sequence of updates, since the size of the circuit C_x is independent of the machine size M. However, the main issue is that their approach is tied to just a single computation $M(x)$ whereas we require that M be reused on multiple inputs. They propose an approach to achieve reusability by using another layer of obfuscation, with M hardwired in it, that produces fresh encodings of M for every computation. This is highly problematic for us, since patching M would now correspond to patching the underlying obfuscated circuit.

We need to make the randomized encodings construction of KLW reusable while preserving the underlying encoding of M. A recent work of Ananth et al. [AJS17], proposed in a different context of building iO with constant overhead, achieves this goal. In more detail, they showed how to achieve iO for TMs, with a priori bound in the input length, such that an obfuscation of M proceeds in two phases: (a) M is encoded using a suitable encoding scheme and, (b) an obfuscation of a circuit that takes as input x and produces an encoding of x. The evaluation of the obfuscation on an input x proceeds by first obtaining an encoding of x (using the obfuscated circuit) and then decoding this using the encoding of M to recover $M(x)$.

While their work offers a starting point for building patchable iO, we still need to address several issues that specifically arise in the context of patching. For instance, their work only considers the setting when the adversary is given one obfuscated machine whereas in our setting she also receives additionally, patches that share some common randomness with the obfuscated machine. We need to argue that the security holds even with this additional information. Instead of directly digging

[5]A randomized encoding of (M, x) satisfies two properties: (a) it only reveals $M(x)$ and, (b) the size of the encoding is polynomial only in the length of M, x and security parameter.

into the details of [AJS17] to apply it in the context of patching, we undertake a more modular approach. First, we propose an intermediate primitive called *splittable iO* and show that it suffices for building single-program patchable iO. We then show that splittable iO can be implemented assuming only iO for circuits by using the framework of [AJS17]. We describe this primitive in detail next.

Splittable iO: Intermediate Notion between iO and Patchable iO. A splittable iO scheme is a strengthening of iO and is associated with respect to a patchable encoding scheme. A patchable encoding scheme consists of algorithms: Setup, Encode and Decode. Setup generates a secret key sk that will be used by Encode procedure to obtain an encoding of M, $\mathcal{E}_{sk}(M)$. Decode recovers the Turing machine M from the encoding $\mathcal{E}_{sk}(M)$ using the secret key sk. Additionally, it is associated with two algorithms: patch generation algorithm, used to generate secure patches and patch application algorithm,, that enables applying secure patches on encodings of TMs. The security property requires that the encodings and patches hide the underlying TMs and patches, respectively.

We start with a oversimplified template of splittable iO and make suitable modifications later. An obfuscation of M, with respect to splittable iO, consists of two parts: $(\mathcal{E}_{sk}(M), aux_M)$, where (i) $\mathcal{E}_{sk}(M)$ is a patchable encoding of M computed using secret key sk, (ii) aux_M computed as a function of an additional PPT algorithm AuxGen, on $(sk, \mathcal{E}_{sk}(M))$.

Armed with the notion of splittable iO, we show how to construct single-program patchable iO. At first glance, it seems that splittable iO already allows for patching: indeed, since M is encoded with respect to a patchable encoding scheme, we can use the patching algorithm to update this encoding. However, this does not work because the obfuscation also contains aux_M that is tied to encoding of M. Indeed, this is necessary for the security of obfuscation to hold. So if the encoding of M is updated, it is necessary to also update aux_M. A naive way of achieving this is to issue a fresh aux_M every time the encoding is patched. That is, initially the user is issued an encoding of M, $\mathcal{E}_{sk}(M)$ and auxiliary information aux_M. During the patching phase, a secure version of patch P with respect to the patchable encoding scheme is issued. Along with this, a fresh $aux_{M'}$ is issued, which is generated by first patching $\mathcal{E}_{sk}(M)$ using $\widetilde{P}$, secure patch of P, and then executing AuxGen on input $(sk, \mathcal{E}_{sk}(M'))$.

However this raises the question of efficiency: the patch size now grows with the size of $aux_{M'}$. This can be taken care of imposing an efficiency constraint on splittable iO: we require that the size of aux be a polynomial in security parameter and specifically, independent of the size of the machine obfuscated. The next issue is correctness: why should the patched obfuscated machine be correct? for instance: AuxGen could abort on input patched encodings. To take care of this issue, we impose an additional property on splittable iO: the correctness of the obfuscated machine should hold irrespective of whether fresh encodings or patched encodings of the machine are fed to AuxGen.

Finally, we move on to proving the security of patchable iO. A first attempt is to use the security of the underlying patchable encoding scheme to argue this. However, it is unclear why the security of encoding scheme is guaranteed at all given that aux contains information about the secret key of the encoding scheme. If we additionally impose aux to hide the secret key, we can then hope to invoke the security of patchable encoding scheme to argue the security of patchable iO. A natural approach of formalizing this is to use a simulation-based argument – there exists a simulator that can simulate the aux even without knowing the secret key. But this would mean that aux will not able to decode any information about the encoding of M. In order to maintain correctness of the obfuscation of M, we need to hardwire all possible outputs which is clearly infeasible. Instead we use an indistinguishability-based definition: instead of having one encoding of M, we will consider a pair of encodings of M. That is, obfuscation of M consists of $(\mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M))$, computed with respect to secret keys sk_0, sk_1. In addition, it consists of aux generated using $\mathsf{AuxGen}(sk_0, \mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M))$. Now, we impose a security property that says that aux generated using sk_0 is computationally indistinguishable from aux generated using sk_1.

We summarize the (informal) definition of splittable iO below. The formal definition can be found in Section 4.2. In addition to the properties of any iO scheme, a splittable iO scheme has the following properties.

1. *Splittable Property*: An obfuscation of M can be performed in two steps: the first step is encoding M twice using two secret keys sk_0 and sk_1 of a patchable encoding scheme. The second step is generation of aux by computing AuxGen on input $(sk_0, \mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M))$, where $\mathcal{E}_{sk_0}(M)$ and $\mathcal{E}_{sk_1}(M)$ are two encodings of M and sk_0 is the secret key used to encode $\mathcal{E}_{sk_0}(M)$.

2. *Correctness of* AuxGen: The correctness of obfuscation of M holds irrespective of whether AuxGen is executed on fresh encodings of M or whether it is executed on encodings of M obtained as a result of patching. This will be used to argue the correctness of the resulting patchable iO scheme.

3. *Efficiency of aux*: We require that the size of aux is a polynomial in λ and in particular, independent of the size of the machine obfuscated. This will be used to argue the patch size efficiency of patchable iO.

4. *Indistinguishability of aux*: We require that it is computationally hard to distinguish aux generated using secret key sk_0 from aux generated using sk_1. This property will be helpful to argue security of patchable iO.

Going from Single-Program to Multi-Program Patchable Obfuscation. In the solution sketched above, every time the authority has to generate a patch, she has to spend time proportional to the size of the obfuscated machine. In particular, recall that one of the steps in the generation of secure patch is computing aux_M: this step involves first patching the old encoding $\mathcal{E}_{sk}(M)$ and then executing AuxGen. We will use the trick described earlier to solve the problem: we delegate the state of the authority as well as the computation of the secure patches to the user. This can be implemented by using a suitable garbling scheme that works in the persistent memory setting. Once this mechanism is implemented, the authority is only required to store the garbling key.

While this is a viable solution in the single-program setting, this is undesirable when the authority is issuing multiple obfuscated programs. She has to store the garbling keys corresponding to all the machines in this case. The storage space of the authority thus puts a bound on the number of obfuscated machines it can issue.

To overcome this difficulty, we employ another idea for delegating responsibility to the user! The garbling key of every user is maintained at her own storage space in an encrypted form. The computation of the garbled program encodings are then delegated to every user. This mechanism is implemented by using a functional encryption scheme. Every user along with the obfuscated machine, garbled encoding of state, also contains an FE encryption of the garbling key. During the patching phase, the authority sends a FE key containing patch P, that takes as input a garbling key and produces a garbled encoding of P with respect to this garbling key. To carry this out, we only require a *secret-key* FE scheme for *circuits*.

Putting it Together: A Framework for (Multi-Program) Patchable Obfuscation. Putting all the components together, we construct a multi-program patchable iO in the following steps:

1. The first step involves formalizing the notion of splittable iO. This is shown in Section 4.
2. Next, we show how to obtain single-program patchable iO from splittable iO. This is shown in Section 5. The resulting single-program patchable iO scheme is statefull, i.e., the authority is required to maintain a large state.
3. We show how to overcome this problem by giving a transformation from any statefull to a stateless single-program patchable iO scheme. This is shown in Section 6.
4. In the next step, we give a transformation from single-program to multi-program patchable iO. This is shown in Section 7.
5. In the last step, we instantiate splittable iO using the framework of [AJS17]. This is shown in Section 8.

1.3 Related Work: Incremental Cryptography

The area of incremental cryptography was pioneered by Bellare, Goldreich and Goldwasser [BGG94]. Subsequently, this concept of incremental updates has been studied for various standard primitives such as encryption schemes, signature schemes and so on [BGG95, Mic97, Fis97, BKY01, MPRS12]. We remark that none of these works handled the setting of arbitrary updates.

In a concurrent and independent work, [GP15] consider a related notion called *incremental obfuscation*. In incremental obfuscation, individual bits of an existing obfuscated program can be updated one-by-one. While their work shares much in spirit with our work, there are several important differences that we describe below.

Our work focuses on support for arbitrary, adaptively chosen patches that may potentially increase the size of the program(s) being patched, and we consider both single-program and multi-program setting. In contrast, their work considers the single-program setting where bit-wise, non-adaptively chosen patches can be applied such that the size of the circuit being patched remains unchanged. Our main efficiency requirement is that the size of the secure patches (or more strongly, the time to generate the secure patches) is independent of the size of the program. In contrast, their work considers the stronger runtime efficiency requirement where the time to apply the secure patch is also independent of the size of the circuit.

2 Patchable $i\mathcal{O}$: Definitions and Implications

In this section, we present the formal definitions of patchable indistinguishability obfuscation (*pa-*$i\mathcal{O}$) in the single program and multi program setting.

2.1 Definition: Single-Program *pa-i*$\mathcal{O}$

In this section, we present a formal definition of single-program patchable indistinguishability obfuscation, denoted as *pa-i*$\mathcal{O}_{\mathsf{sp}}$. We start by presenting the syntax, and then proceed to give a security definition for sequential updates.

Syntax. A *pa-i*$\mathcal{O}_{\mathsf{sp}}$ scheme, defined for a class of Turing machines $\mathcal{M}$ with an associated family of patches $\mathcal{P}$ and update algorithm Update, consists of a tuple of probabilistic polynomial-time algorithms *pa-i*$\mathcal{O}_{\mathsf{sp}} = (\mathsf{Setup}, \mathsf{Obf}, \mathsf{GenPatch}, \mathsf{AppPatch}, \mathsf{Eval})$ which are defined below.

- **Setup**, $\mathsf{Setup}(1^\lambda)$: It takes as input the security parameter λ and outputs the secret key SK.
- **Obfuscate**, $\mathsf{Obf}(\mathsf{SK}, M)$: It takes as input the secret key SK and a TM $M \in \mathcal{M}$. It outputs an obfuscated TM $\langle M \rangle$ along with state st.
- **(Stateful) Patch Generation**, $\mathsf{GenPatch}(\mathsf{SK}, P, \mathsf{st})$: It takes as input the secret key SK, a description of a patch $P \in \mathcal{P}$, and state st. It outputs a patch encoding $\langle P \rangle$ along with the updated state st'.
- **Applying Patch**, $\mathsf{AppPatch}\Big(\langle M \rangle, \langle P \rangle\Big)$: It takes as input an obfuscated TM $\langle M \rangle$ and a patch encoding $\langle P \rangle$. It outputs an updated obfuscation $\langle M' \rangle$.
- **Evaluation**, $\mathsf{Eval}\Big(\langle M \rangle, x\Big)$: It takes as input an obfuscated TM $\langle M \rangle$ and an input x. It outputs a value y.

Efficiency. We define two efficiency properties:

- *Patch Size Efficiency*: For every patch $P \in \mathcal{P}$, we require that the size of the patch encoding $|\langle P \rangle|$ is a fixed polynomial in $(|P|, \lambda)$, where $(\langle P \rangle, \mathsf{st}') \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P, \mathsf{st})$.
- *Patch Generation Efficiency*: For every patch $P \in \mathcal{P}$, we require that the running time of $\mathsf{GenPatch}(\mathsf{SK}, P, \mathsf{st})$ to be a fixed polynomial in $(|P|, \lambda)$. The length of st could depend on the size of the obfuscated machine its associated with and we require that the running time of $\mathsf{GenPatch}$ to be independent of $|\mathsf{st}|$.

It is easy to see that the second property implies the first property. Our first construction of *pa-i*$\mathcal{O}_{\mathsf{sp}}$ (see Sections 5 and 8) only satisfies the first property. Later, in Section 6, we describe a modified construction that also achieves the second property.

Correctness for Sequential Patches. At a high level, the correctness property states that executing Update on a TM M and a patch P is equivalent to executing $\mathsf{AppPatch}$ on the obfuscation of M and a secure patch of P. In fact we require that this holds even if there are multiple patches that are applied sequentially.

For any TM $M_0 \in \mathcal{M}$, $L > 0$, sequence of patches $P_1, \ldots, P_L \in \mathcal{P}$, consider two processes:

- **Obfuscate-then-Update**: Compute the following: (a) $\mathsf{SK} \leftarrow \mathsf{Setup}(1^\lambda)$, (b) $\Big(\langle M_0 \rangle, \mathsf{st}_0\Big) \leftarrow \mathsf{Obf}(\mathsf{SK}, M_0)$, (c) $\Big(\langle P_i \rangle, \mathsf{st}_i\Big) \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P_i, \mathsf{st}_{i-1})$, (d) $\langle M_i \rangle \leftarrow \mathsf{AppPatch}\Big(\langle M_{i-1} \rangle, \langle P_i \rangle\Big)$.

- **Update**: $M_i \leftarrow \mathsf{Update}(M_{i-1}, P_i)$.

We require that for all $x \in \{0,1\}^*$, every $i \in [L]$, $\mathsf{Eval}\Big(\langle M_i \rangle, x\Big) = M_i(x)$.

Remark 1. *For the case of parallel patching, we require that $\langle M_i \rangle \leftarrow \mathsf{AppPatch}\Big(\langle M_0 \rangle, \langle P_i \rangle\Big)$ is a valid obfuscation of machine M_i. We emphasize that for the case of parallel patching, the patches are applied only on the original machine.*

Adaptive Security for Sequential Patches. We next give an indistinguishability (IND)-style definition for modeling the security of an *pa-i*$\mathcal{O}_{\mathsf{sp}}$ scheme for the case of sequential patches. In an IND-security definition, we consider a security game between the challenger and the adversary. In this game, the adversary sends two machines (M_0^0, M_1^0) to the challenger and in response receives an obfuscation $\langle M_b^0 \rangle$, where b is the challenge bit chosen randomly by the challenger. Then the adversary submits patch queries, adaptively, to the challenger in a series of phases. In each phase, the adversary chooses a pair of patches (P_0^i, P_1^i) and in return gets the patch encoding $\langle P_b^i \rangle$. The patch queries of the adversary are restricted in the following manner: suppose $\Big((P_0^1, P_1^1), \ldots, (P_0^L, P_1^L)\Big)$ is a sequence of adaptive patch queries made by the adversary. We require that the machine M_0^i is functionally equivalent with M_1^i, for every $i \in [L]$, where $M_0^i \leftarrow \mathsf{Update}(M_0^{i-1}, P_0^i)$ (resp., $M_1^i \leftarrow \mathsf{Update}(M_1^{i-1}, P_1^i)$). At the end of the game, the adversary attempts to guess the bit b. If the adversary's guess is the same as b only with probability negligibly close to 1/2, then we say that the scheme is secure. Henceforth, we use the term *adaptive security* to refer to this notion. We proceed to formally defining this notion.

The experiment for the adaptive security definition is formulated below. Let $\mathcal{A}$ be any PPT adversary.

$\underline{\mathsf{Expt}_{\mathcal{A}}^{pa\text{-}i\mathcal{O}_{\mathsf{sp}}}(1^\lambda, b)}$:

1. $\mathcal{A}$ sends (M_0^0, M_1^0) to the challenger.
2. Challenger executes the setup algorithm to obtain $\mathsf{SK} \leftarrow \mathsf{Setup}(1^\lambda)$. It then sends $\langle M_b^0 \rangle \leftarrow \mathsf{Obf}(\mathsf{SK}, M_b^0)$ to $\mathcal{A}$.
3. Repeat the following steps for $i \in \{1, \ldots, L\}$, where L is chosen by $\mathcal{A}$.
   - $\mathcal{A}$ sends (P_0^i, P_1^i) to the challenger.
   - Challenger checks if $M_0^i \equiv M_1^i$, where $M_0^i \leftarrow \mathsf{Update}(M_0^{i-1}, P_0^i)$ and $M_1^i \leftarrow \mathsf{Update}(M_1^{i-1}, P_1^i)$.
   - Challenger computes $\langle P_b^i \rangle \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P_b^i)$ and sends $\langle P_b^i \rangle$ to $\mathcal{A}$.
4. $\mathcal{A}$ outputs the bit b'.

Definition 1 (Adaptive Security)**.** *A single-program patchable indistinguishability obfuscation scheme pa-i*$\mathcal{O}_{\mathsf{sp}}$ *is said to be adaptively secure against sequential updates if for any PPT adversary $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *s.t.*

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

Remark 2. *For the case of parallel patching, the same security is defined with the only difference being that it is required that the machine M_0^i is functionally equivalent to M_1^i, where M_b^i is obtained by patching M_b^0 (the original machine) using P_i .*

2.2 Definition: Multi-Program *pa-i$\mathcal{O}$*

We now present a formal definition of multi-program *pa-i$\mathcal{O}$*, denoted as *pa-i$\mathcal{O}_{\mathsf{mp}}$*. Informally speaking, *pa-i$\mathcal{O}_{\mathsf{mp}}$* allows an authority to obfuscate an arbitrary number of programs in such a way that it is possible to later issue a patch encoding that can be used to update all the obfuscated programs at once. The authority who issues the obfuscated programs stores just a "short" information about all the obfuscated programs issued that enables it to produce a single patch that can act on all these programs. In particular, the size of the storage space of the authority is independent of the joint size of all these programs.[6] This is in contrast to the single-program setting described above, where the authority maintains state and this state can be as big as the program whose obfuscation is issued. There is another difference between both the settings: in the single-program setting, if we were to relax the size of the secure patch to be proportional to the size of the updated program then achieving a feasibility result is straightforward – the secure patch will just be the obfuscation of the updated program. Hence the primary goal is to reduce the size of the patch. However, in the multi-program setting, even if we relax the size of the secure patch to be proportional to the size of any of the updated programs, achieving a feasibility result is already non-trivial. As mentioned earlier, the authority does not have enough space to store all the updated programs and hence the above naïve solution, of sending a fresh obfuscation of the updated program, does not work. As we will see later we not only give a feasibility result in this setting but we also achieve a solution with optimal efficiency where the size of the secure patches depend only on the size of their original patches and in particular, independent of the size of any obfuscated programs issued.

Syntax. A *pa-i$\mathcal{O}_{\mathsf{mp}}$* scheme, defined for a class of Turing machines $\mathcal{M}$ and a family of patches $\mathcal{P}$, consists of a tuple of probabilistic polynomial-time algorithms *pa-i$\mathcal{O}_{\mathsf{mp}}$* $= (\mathsf{Setup}, \mathsf{Obf}, \mathsf{GenPatch}, \mathsf{AppPatch}, \mathsf{Eval})$ which are defined below. We denote the update algorithm associated with $(\mathcal{M}, \mathcal{P})$ to be Update.

- **Setup**, $\mathsf{Setup}(1^\lambda)$: It takes as input the security parameter λ and outputs the secret key SK.
- **Obfuscate**, $\mathsf{Obf}(\mathsf{SK}, M)$: It takes as input the secret key SK and a TM $M \in \mathcal{M}$ id. It outputs an obfuscated TM $\langle M \rangle$.
- **(Stateless) Patch Generation**, $\mathsf{GenPatch}(\mathsf{SK}, P)$: It takes as input the secret key SK and a description of a patch $P \in \mathcal{P}$. It outputs a patch encoding $\langle P \rangle$.
- **Applying Patch**, $\mathsf{AppPatch}\Big(\langle M \rangle, \langle P \rangle\Big)$: It takes as input an obfuscated TM $\langle M \rangle$ and a patch encoding $\langle P \rangle$. It outputs an updated obfuscation $\langle M' \rangle$.
- **Evaluation**, $\mathsf{Eval}\Big(\langle M \rangle, x\Big)$: It takes as input an obfuscated TM $\langle M \rangle$ and an input x. It outputs a value y.

[6]The reason why the authority can't store all the programs is because it is a machine that has a priori bounded memory and yet has the capability to produce an unbounded number of obfuscated programs.

Efficiency. Similar to $pa\text{-}i\mathcal{O}_{\mathsf{sp}}$, we define two efficiency properties for $pa\text{-}i\mathcal{O}_{\mathsf{mp}}$:

- *Patch Size Efficiency*: For every patch $P \in \mathcal{P}$, we require that the size of the patch encoding $|\langle P\rangle|$ is a fixed polynomial in $(|P|, \lambda)$, where $(\langle P\rangle, \mathsf{st}') \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P, \mathsf{st})$.
- *Patch Generation Efficiency*: For every patch $P \in \mathcal{P}$, we require that the running time of $\mathsf{GenPatch}(\mathsf{SK}, P,)$ to be a fixed polynomial in $(|P|, \lambda)$.

It is easy to see that the second property implies the first property. Our construction of $pa\text{-}i\mathcal{O}_{\mathsf{mp}}$ presented in Section 7 achieves both of the properties.

Correctness for Sequential Patches. For every $Q, L > 0$, any sequence of TMs $M_0^1, \ldots, M_0^Q \in \mathcal{M}$, sequence of patches $P_1, \ldots, P_L \in \mathcal{P}$, consider the following two processes. For every $j \in \{1, \ldots, Q\}, i \in \{1, \ldots, L\}$, we have:

- **Obfuscate-then-Update**: Compute the following: (a) $\mathsf{SK} \leftarrow \mathsf{Setup}(1^\lambda)$, (b) $\langle M_0^j\rangle \leftarrow \mathsf{Obf}(\mathsf{SK}, M_0^j)$, (c) $\langle P_i\rangle \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P_i)$, (d) $\langle M_i^j\rangle \leftarrow \mathsf{AppPatch}\Big(\langle M_{i-1}^j\rangle, \langle P_i\rangle\Big)$.
- **Update**: $M_i^j \leftarrow \mathsf{Update}(M_{i-1}^j, P_i)$.

We require that $\forall x \in \{0,1\}^*$, $\forall j \in [Q]$, $\forall i \in [L]$, we have $\mathsf{Eval}\Big(\langle M_i^j\rangle, x\Big) = M_i^j(x)$.

Adaptive Security for Sequential Patches. We next give indistinguishability (IND)-style definitions for modeling the security of a patchable obfuscation scheme. As in the case of single-program patchable obfuscation, the definition is based on a game between the challenger and the adversary. The adversary makes TM queries and patch queries to the challenger. One important distinction is that in this setting, the adversary can make multiple TM queries whereas in the case of single-program obfuscation, it makes just one TM query. We describe the experiment below.

$\underline{\mathsf{Expt}_{\mathcal{A}}^{pa\text{-}i\mathcal{O}_{\mathsf{mp}}}(1^\lambda, b)}$:

1. $\mathcal{A}$ submits a sequence of TM pairs $\Big((M_{0,0}^1, M_{0,1}^1), \ldots, (M_{0,0}^Q, M_{0,1}^Q)\Big)$.
2. Challenger executes the setup algorithm to obtain $\mathsf{SK} \leftarrow \mathsf{Setup}(1^\lambda)$. For every $j \in [Q]$, it computes $\langle M_{0,b}^j\rangle \leftarrow \mathsf{Obf}(\mathsf{SK}, M_{0,b}^j)$ and sends $\Big\{\langle M_{0,b}^j\rangle\Big\}_{j\in[Q]}$ to the adversary.
3. Repeat the following steps for $i \in \{1, \ldots, L\}$, where $L(\lambda)$ is chosen by $\mathcal{A}$:
   - $\mathcal{A}$ sends (P_0^i, P_1^i) to the challenger.
   - Challenger computes $\langle P_b^i\rangle \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P_b^i)$. It sends $\langle P_b^i\rangle$ to $\mathcal{A}$.
4. For every $i \in \{1, \ldots, L\}$, every $j \in \{1, \ldots, Q\}$, the challenger checks if $M_{i,0}^j \equiv M_{i,1}^j$, where $M_{i,0}^j \leftarrow \mathsf{Update}(M_{i-1,0}^j, P_0^i)$ and $M_{i,1}^j \leftarrow \mathsf{Update}(M_{i-1,1}^j, P_1^i)$. If check fails then the challenger aborts the experiment.
5. $\mathcal{A}$ outputs the bit b'.

Definition 2 (Adaptive security)**.** *A multi-program patchable obfuscation scheme pa-$i\mathcal{O}_{\mathsf{mp}}$ is said to be adaptively secure if for any PPT adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ s.t.*

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

Remark 3. *For the case of parallel patching, the correctness and security can be similarly defined.*

3 Preliminaries

We denote the security parameter by λ. We assume familiarity of the reader with standard cryptographic notions.

3.1 Turing Machines

A Turing machine is a tuple $M = \langle Q, \Sigma_{\text{inp}}, \Sigma_{\text{tape}}, \perp, \delta, q_0, q_{\text{acc}}, q_{\text{rej}} \rangle$, where every element in the tuple is defined as follows: (a) Q is the set of finite states. (b) Σ_{inp} is the set of input symbols. (c) Σ_{tape} is the set of tape symbols. (d) $\perp$ denotes the blank symbol. (e) $\delta : Q \times \Sigma_{\text{tape}} \rightarrow Q \times \Sigma_{\text{tape}} \times \{+1, -1\}$ is the transition function. (f) $q_0 \in Q$ is the start state. (g) $q_{\text{acc}} \in Q$ is the accept state. (h) $q_{\text{rej}} \in Q$ is the reject state, where $q_{\text{acc}} \neq q_{\text{rej}}$.

Transforming Turing machines to Circuits. A Turing machine running in time at most $T(n)$ on inputs of size n, can be transformed into a circuit of input length n and of size $O\big((T(n))^2\big)$. This theorem proved by Pippenger and Fischer [PF79] is stated below.

Theorem 7. *Any Turing machine M running in time at most $T(n)$ for all inputs of size n, can be transformed into a circuit $C_M : \{0,1\}^n \rightarrow \{0,1\}$ such that (i) $C_M(x) = M(x)$ for all $x \in \{0,1\}^n$, and (ii) the size of C_M is $|C_M| = O\big((T(n))^2\big)$. We denote this transformation procedure as* $\mathsf{TMtoCKT}$.

Adopted Conventions. We denote by $\mathsf{RunTime}(M, x)$, the time taken by a Turing machine M to evaluate on input x. We adopt the convention that the Turing machine also additionally outputs the time taken to execute. Thus, if we have two inputs x and y, a Turing machine M, then if $M(x) = M(y)$, by this notation, means that not only does M on x output the same value as M on y but also that the running time of M on both x and y are the same.

In this work, we only consider TMs which run in polynomial time on all its inputs, i.e., there exists a polynomial p such that the running time is at most $p(n)$ for every input of length n.

Equivalence of Programs. Let M_0 and M_1 be two Turing machines. We denote by $M_0 \equiv M_1$ if both M_0 and M_1 are functionally equivalent, i.e., if $M_0(x) = M_1(x)$, for all $x \in \{0,1\}^*$.

3.2 Patching Turing Machines

Throughout this work, we consider various families of Turing machines. We assume that any Turing machine family has an associated family of patches that come equipped with a polynomial-time update algorithm. For example, let $\mathcal{M}$ be any Turing machine family with associated patch family $\mathcal{P}$ and update algorithm $\mathsf{Update}_{\mathcal{M},\mathcal{P}}$. Algorithm $\mathsf{Update}_{\mathcal{M},\mathcal{P}}$ takes as input a Turing machine $M \in \mathcal{M}$ and a patch $P \in \mathcal{P}$ and outputs an updated Turing machine $M_{new} \in \mathcal{M}$. That is:

$$M_{new} \leftarrow \mathsf{Update}_{\mathcal{M},\mathcal{P}}(M, P)$$

A natural way to model patches is to consider them as arbitrary polynomial-time Turing machines. That is, we can model a patch $P \in \mathcal{P}$ as a polynomial-time Turing machine that takes $M \in \mathcal{M}$ as input and outputs a new machine $M_{new} = P(M) \in \mathcal{M}$. In this case, the $\mathsf{Update}_{\mathcal{M},\mathcal{P}}$ algorithm simply executes P with input M. One could also consider an alternative modeling of patches where P is simply a string such that $\mathsf{Update}_{\mathcal{M},\mathcal{P}}$ on input (M, P) makes appropriate changes in M as per the description of P to compute M_{new}.

The primitives we discuss and construct in this work are robust to any such formulation of P and $\mathsf{Update}_{\mathcal{M},\mathcal{P}}$.

3.3 Indistinguishability Obfuscation

The notion of indistinguishability obfuscation (iO), first conceived by Barak et al. [BGI+12], guarantees that the obfuscation of two circuits are computationally indistinguishable as long as they both are equivalent circuits, i.e., the output of both the circuits are the same on every input. Analogous to the case of circuits, we can define indistinguishability obfuscation for Turing machines (TMs). We work in a weaker setting of iO for TMs, as considered by the recent works [CHJV15, BGL+15, KLW15, AJS17], where the inputs to the TM are upper bounded by a pre-determined value. This definition of iO for TMs is referred as *succinct iO*. The security property of this notion states that the obfuscations of two machines M_0 and M_1 are computationally indistinguishable as long as $M_0(x) = M_1(x)$ and the time taken by both the machines on input x are the same, i.e., $\mathsf{RunTime}(M_0, x) = \mathsf{RunTime}(M_1, x)$.

iO for Circuits. We define the notion of indistinguishability obfuscation (iO) for circuits below.

Definition 3 (Indistinguishability Obfuscator (iO) for Circuits). *A uniform PPT algorithm $i\mathcal{O}$ is called an indistinguishability obfuscator for a circuit family $\{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$, where $\mathcal{C}_\lambda$ consists of circuits C of the form $C : \{0,1\}^{\mathsf{inp}} \to \{0,1\}$, if the following holds:*

- **Completeness:** *For every $\lambda \in \mathbb{N}$, every $C \in \mathcal{C}_\lambda$, every input $x \in \{0,1\}^{\mathsf{inp}}$, where $\mathsf{inp} = \mathsf{inp}(\lambda)$ is the input length of C, we have that*

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

- **Indistinguishability:** *For any PPT distinguisher D, there exists a negligible function $\mathsf{negl}(\cdot)$ such that the following holds: for all sufficiently large $\lambda \in \mathbb{N}$, for all pairs of circuits $C_0, C_1 \in \mathcal{C}_\lambda$ such that $C_0(x) = C_1(x)$ for all inputs $x \in \{0,1\}^{\mathsf{inp}}$, where $\mathsf{inp} = \mathsf{inp}(\lambda)$ is the input length of C_0, C_1, we have:*

$$\Big|\mathsf{Pr}\left[D(\lambda, i\mathcal{O}(\lambda, C_0)) = 1\right] - \mathsf{Pr}[D(\lambda, i\mathcal{O}(\lambda, C_1)) = 1]\Big| \leq \mathsf{negl}(\lambda)$$

iO for Turing Machines. Analogous to the case of circuits, we can define indistinguishability obfuscation for Turing machines (TMs). The security property states that the obfuscations of two Turing machines M_0 and M_1 are computationally indistinguishable as long as $M_0(x) = M_1(x)$. Note that by our convention adopted for Turing machines, the condition that $M_0(x) = M_1(x)$ already ensures that the running time of $M_0(x)$ and $M_1(x)$ are the same. The succinctness property states that the running time of the obfuscation algorithm on input M is independent of the worst case running time of machine M. The same guarantee also holds for the evaluation of the obfuscated TM. We note that this definition was adopted in the works of [BGL+15, CHJV15, KLW15, AJS17].

Definition 4 (Succinct iO). *A uniform PPT algorithm* SuccIO *is called an succinct indistinguishability obfuscator for a class of Turing machines* $\{\mathcal{M}_\lambda\}_{\lambda \in \mathbb{N}}$ *with an input bound* L*, if the following holds:*

- **Completeness:** *For every* $\lambda \in \mathbb{N}$*, every* $M \in \mathcal{M}_\lambda$*, every input* $x \in \{0,1\}^{\leq L}$*, we have that:* $\mathsf{Pr}\left[M'(x) = M(x) \ : \ M' \leftarrow \mathsf{SuccIO}(\lambda, M, L)\right] = 1$.

- **Indistinguishability:** *For any PPT distinguisher* D*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that the following holds: for all sufficiently large* $\lambda \in \mathbb{N}$*, for all pairs of Turing machines* $M_0, M_1 \in \mathcal{M}_\lambda$ *such that* $M_0(x) = M_1(x)$ *for all inputs* $x \in \{0,1\}^{\leq L}$*, we have:*

$$\Big| \mathsf{Pr}\left[D(\lambda, \mathsf{SuccIO}(\lambda, M_0, L)) = 1\right] - \mathsf{Pr}[D(\lambda, \mathsf{SuccIO}(\lambda, M_1, L)) = 1] \Big| \leq \mathsf{negl}(\lambda)$$

- **Succinctness:** *For every* $\lambda \in \mathbb{N}$*, every* $M \in \mathcal{M}_\lambda$*, we have the running time of* SuccIO *on input* (λ, M, L) *to be* $\mathrm{poly}(\lambda, |M|, L, \log(T))$ *and the evaluation time of* $\widetilde{M}$ *on input* x*, where* $|x| \leq L$*, to be* $\mathrm{poly}(|M|, L, t)$*, where* $\widetilde{M} \leftarrow \mathsf{SuccIO}(\lambda, M, L)$ *and* $t = \mathsf{RunTime}(M, x)$*.*

3.4 Garbled TMs with Persistent Memory

A garbled Turing machine is a randomized encoding, where the encoding time is independent of the computation time. It consists of two components – an input encoding and a TM encoding. The input encoding is an encoding of the input tape of the TM. We consider the concept of garbled TMs (GTM) with persistent memory. In this setting, there are multiple TM encodings that sequentially operate on the same input encoding. To be more precise, denote the input encoding of x to be $\widetilde{x}$. Now, GTM with persistent memory allows the issue of multiple TM encodings $\widetilde{M_1}, \ldots, \widetilde{M_\ell}$ such that (i) $\widetilde{M_1}$ executes on $\widetilde{x}$ and outputs a value y_1 and also updates the input tape to be $\widetilde{x_1}$, (ii) $\widetilde{M_i}$ operates on encoding $\widetilde{x_{i-1}}$; outputs y_i and updates the input tape to be $\widetilde{x_i}$.

The concept of persistent memory has been studied in the context of RAMs [GHRW14, GLOS15]. For our work, it suffices to consider Turing machines. We describe the primitive formally below.

Suppose $\mathcal{M} = \{\mathcal{M}_\lambda\}_{\lambda \in \mathbb{N}}$ be a class of Turing machines where every $M \in \mathcal{M}_\lambda$ is such that the maximum space taken by M on any input $x \in \{0,1\}^{\mathrm{poly}(\lambda)}$ is 2^λ, for every sufficiently large $\lambda \in \mathbb{N}$. A garbled TM with persistent memory GTM, consists of a tuple of algorithms ($\mathsf{Gen}, \mathsf{GarbDB}, \mathsf{GarbTM}, \mathsf{GarbEval}$).

- **Setup, $\mathbf{k} \leftarrow \mathsf{Gen}(1^\lambda)$:** It takes as input a security parameter and outputs a secret key $\mathbf{k}$.

- **Garbling of Turing Machine, $\widehat{M} \leftarrow \mathsf{GarbTM}(\mathbf{k}, M)$:** It takes as input a secret key $\mathbf{k}$, Turing machine $M \in \mathcal{M}$ and outputs an encoding of M, $\widehat{M}$.

- **Garbling of Input Tape,** $\widehat{DB} \leftarrow \mathsf{GarbDB}(\mathbf{k}, DB)$: It takes as input a secret key $\mathbf{k}$, contents of an input tape DB and outputs an encoding of DB, $\widehat{DB}$.
- **Evaluation,** $(y, \widehat{DB'}) \leftarrow \mathsf{GarbEval}(\widehat{M}, \widehat{DB})$: It takes as input an encoding $\widehat{M}$, encoding $\widehat{DB}$ and outputs a value y and also the updated encoding $\widehat{DB'}$.

Remark 4. *Previous works considered definitions, where the algorithms also take as input a space bound. In our setting, we set the space bound of the computations to be 2^{λ} and hence the space bound does not explicitly feature in the definitions.*

We require that the above scheme satisfy the following properties.

Correctness. Consider a sequence of Turing machines $M_1, \ldots, M_\ell \in \mathcal{M}$, input tape DB initialized with x. Suppose a sequential evaluation of $M_1, \ldots, M_\ell$ on DB leads to outputs $y_1, \ldots, y_\ell$ respectively. By this, we mean that M_i when operated on the input tape updated by M_{i-1} would output value y_i.

We require that for every $i \in [\ell]$, it should hold that $(y_i, \widehat{DB_i}) \leftarrow \mathsf{GarbEval}(\widehat{M_i}, \widehat{DB_{i-1}})$, where

- $\mathbf{k} \leftarrow \mathsf{Gen}(1^{\lambda})$
- $\widehat{M_1} \leftarrow \mathsf{GarbTM}(\mathbf{k}, M_1)$
- $\widehat{DB_0} \leftarrow \mathsf{GarbDB}(\mathbf{k}, DB)$

Efficiency. We require that the generation time of the TM encodings be a polynomial only in the size of the TM and security parameter and in particular, independent of either the input tape size or the computation time. More formally, $|\mathsf{GarbTM}(\mathbf{k}, M)| = \mathrm{poly}(\lambda, |M|)$. Furthermore, the generation time of the input tape encoding is independent of the program size. That is, $|\mathsf{GarbDB}(\mathbf{k}, DB)| = \mathrm{poly}(\lambda, |DB|)$. Finally, we require the running time of the evaluation procedure, on input $\widehat{M}$ and $\widehat{DB}$ (notation as defined above), is polynomial in λ and runtime of M on DB.

Security

We consider a simulation-based definition of GTMs with persistent memory. We first consider the adaptive security notion and provide the definition below. Let $\mathcal{A}$ be a (stateful) PPT adversary. And let $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ be a PPT simulator.

$\underline{\mathsf{Ad.Expt}_{\mathcal{A}}^{\mathsf{GTM},\mathsf{Sim}}(1^{\lambda})}$:
Consider the two processes.

- **Honest Execution**:
  - $DB \leftarrow \mathcal{A}(1^{\lambda})$
  - $\mathbf{k} \leftarrow \mathsf{Gen}(1^{\lambda})$,
  - $\widehat{DB} \leftarrow \mathsf{GarbDB}(\mathbf{k}, DB)$,
  - $\ell(\lambda) \leftarrow \mathcal{A}(\widehat{DB})$
  - $\forall i \in [\ell], \left\{ M_i \leftarrow \mathcal{A}(\widehat{M_{i-1}}); \widehat{M_i} \leftarrow \mathsf{GarbTM}(\mathbf{k}, M_i) \right\}$, where $\widehat{M_0} = \perp$.

– $b_{\mathsf{real}} \leftarrow \mathcal{A}(\widehat{M_\ell})$

- **Simulation**:
  – $DB \leftarrow \mathcal{A}(1^\lambda)$
  – $(\mathsf{st}_{\mathsf{Sim}}, \widehat{DB_{\mathsf{ideal}}}) \leftarrow \mathsf{Sim}_1(1^\lambda, 1^{|DB|})$,
  – $\ell(\lambda) \leftarrow \mathcal{A}(\widehat{DB_{\mathsf{ideal}}})$
  – $\forall i \in [\ell], \left\{ M_i \leftarrow \mathcal{A}(\widehat{M_{i-1}^{\mathsf{ideal}}}); (\widehat{M_i^{\mathsf{ideal}}}, \mathsf{st}_{\mathsf{Sim}}) \leftarrow \mathsf{Sim}_2(\mathsf{st}_{\mathsf{Sim}}, y_i, 1^{|M_i|}) \right\}$,
  where $\widehat{M_0} = \perp$ and $(y_i, DB_i) \leftarrow M_i(DB_{i-1})$ with $DB_0 = DB$.
  – $b_{\mathsf{ideal}} \leftarrow \mathcal{A}(\widehat{M_\ell^{\mathsf{ideal}}})$.

If $b_{\mathsf{real}} \neq b_{\mathsf{ideal}}$ then output 1.

Definition 5 (Adaptive GTM with Persistent Memory)**.** *A GTM with persistent memory* GTM *is said to be adaptively secure if for every PPT adversary $\mathcal{A}$, we have* $|\mathsf{Pr}[1 \leftarrow \mathsf{Ad.Expt}_{\mathcal{A}}^{\mathsf{GTM},\mathsf{Sim}}(1^\lambda)]| \leq \frac{1}{2} + \mathsf{negl}(\lambda)$.

We can similarly consider the selective notion, where the adversary declares all the programs ahead of time. And so, the simulator gets to see all the outputs of the programs at once. We define the corresponding experiment to be $\mathsf{Sel.Expt}_{\mathcal{A}}^{\mathsf{GTM},\mathsf{Sim}}$.

Definition 6 (Selective GTM with Persistent Memory)**.** *A GTM with persistent memory* GTM *is said to be selectively secure if for every PPT adversary $\mathcal{A}$, we have* $|\mathsf{Pr}[1 \leftarrow \mathsf{Sel.Expt}_{\mathcal{A}}^{\mathsf{GTM},\mathsf{Sim}}(1^\lambda)]| \leq \frac{1}{2} + \mathsf{negl}(\lambda)$.

Succinct and Non-Succinct GTM Schemes. In the above definition of garbled TMs with persistent memory, we considered the setting where we require that the size of the program encodings are independent of the execution time. We can formalize this by naming such schemes as *succinct* GTM schemes.

Definition 7 (Succinct Garbled TM)**.** *A garbled TM with persistent memory scheme* GTM *is said to be* **succinct** *if* $|\widehat{M}| = \mathrm{poly}(\lambda, |M|)$, *where* $\widehat{M} \leftarrow \mathsf{GarbTM}(\mathbf{k}, M)$.

We can alternately consider a definition where the size of the program encodings depend on the runtime. Such schemes have been studied in the literature in the context of RAMs [LO13, GHL+14, GLOS15, GLO15] and can be constructed based on one-way functions.

Definition 8 (Non-Succinct Garbled TM)**.** *A garbled TM with persistent memory scheme* GTM *is said to be* **non-succinct** *if* $|\widehat{M}| = \mathrm{poly}(\lambda, |M|, T)$, *where* $\widehat{M} \leftarrow \mathsf{GarbTM}(\mathbf{k}, M)$ *and T is a time bound on the running time of M.*

Feasibility. The existence of (selectively-secure) garbled TMs with persistent memory was explored in the work of [CH16, CCC+16]. Recently, the works of [CCHR B, ACC+ B] show the existence of adaptively secure garbled TMs with persistent memory. Their constructions are based on the existence of $\frac{\epsilon}{2^\lambda}$-secure indistinguishability obfuscation and $\frac{\epsilon'}{2^\lambda}$-secure decisional Diffie-Hellman (DDH) assumption, where λ is the security parameter and $\epsilon, \epsilon' \leq \frac{1}{p(\lambda)}$ for some fixed polynomial p. Here we emphasize that the security loss $\frac{\epsilon'}{2^\lambda}$, $\frac{\epsilon'}{2^\lambda}$ do not depend on either the size of the Turing machines or the input. We note that their construction is designed for the more general RAM model of computation, however it suffices for our work to just consider the Turing machine model of computation.

3.5 Secret-Key Functional Encryption

A secret-key functional encryption (FE) scheme FE over a message space $\mathsf{MSG} = \{\mathsf{MSG}_\lambda\}_{\lambda \in \mathbb{N}}$ and a function space $\mathsf{F} = \{\mathsf{F}_\lambda\}_{\lambda \in \mathbb{N}}$ is a tuple $(\mathsf{FE.Setup}, \mathsf{FE.KeyGen}, \mathsf{FE.Enc}, \mathsf{FE.Dec})$ of PPT algorithms with the following properties:

- $\mathsf{FE.Setup}(1^\lambda)$: The setup algorithm takes as input the unary representation of the security parameter, and outputs a secret key $\mathsf{FE.MSK}$.
- $\mathsf{FE.KeyGen}(\mathsf{FE.MSK}, f)$: The key-generation algorithm takes as input the secret key $\mathsf{FE.MSK}$ and a function $f \in \mathsf{F}_\lambda$, and outputs a functional key $\mathsf{FE.SK}_f$.
- $\mathsf{FE.Enc}(\mathsf{FE.MSK}, m)$: The encryption algorithm takes as input the secret key $\mathsf{FE.MSK}$ and a message $m \in \mathsf{MSG}_\lambda$, and outputs a ciphertext CT.
- $\mathsf{FE.Dec}(\mathsf{FE.SK}_f, \mathsf{CT})$: The decryption algorithm takes as input a functional key $\mathsf{FE.SK}_f$ and a ciphertext CT, and outputs $m \in \mathsf{MSG}_\lambda \cup \{\bot\}$.

Correctness. There exists a negligible function $\mathsf{negl}(\cdot)$ such that for all sufficiently large $\lambda \in \mathbb{N}$, for every message $m \in \mathsf{MSG}_\lambda$, and for every function $f \in \mathsf{F}_\lambda$ it holds that

$$\mathsf{FE.Dec}(\mathsf{FE.KeyGen}(\mathsf{FE.MSK}, f), \mathsf{FE.Enc}(\mathsf{FE.MSK}, m)) = f(m)$$

with probability at least $1 - \mathsf{negl}(\lambda)$, where $\mathsf{FE.MSK} \leftarrow \mathsf{FE.Setup}(1^\lambda)$, and the probability is taken over the random choices of all algorithms.

Function privacy. The notion of function privacy is modeled as a game. In the game, a function query made by the adversary is a pair of functions and in response it receives a functional key corresponding to either of the two functions. As long as both the functions are such that they do not split the challenge message-pairs, the adversary should not be able to tell which function was used to generate the functional key. That is, the output of the left function on the left message should be the same as the output of the right function on the right message.

Definition 9 (Function-private adaptively-secure FE)**.** *A secret-key functional encryption scheme* $\mathsf{FE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *over a function space* $\mathsf{F} = \{\mathsf{F}_\lambda\}_{\lambda \in \mathbb{N}}$ *and a message space* $\mathsf{MSG} = \{\mathsf{MSG}_\lambda\}_{\lambda \in \mathbb{N}}$ *is a* ***function-private adaptively-secure secret-key FE scheme*** *if for any PPT adversary* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that for all sufficiently large* $\lambda \in \mathbb{N}$*, the advantage of* $\mathcal{A}$ *is defined to be*

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

where for each $b \in \{0,1\}$ *and* $\lambda \in \mathbb{N}$ *the experiment* $\mathsf{Expt}_{\mathcal{A}}(1^\lambda, b)$*, modeled as a game between the challenger and the adversary* $\mathcal{A}$*, is defined as follows:*

1. *The challenger first executes* $\mathsf{MSK} \leftarrow \mathsf{Setup}(1^\lambda)$. *The adversary then makes the following message queries and function queries in no particular order.*

   - ***Message queries****: The adversary submits a message-pair* (m_0, m_1) *to the challenger. In return, the challenger sends back* $\mathsf{CT} = \mathsf{Enc}(\mathsf{MSK}, m_b)$.
   - ***Function queries****: The adversary then makes functional key queries. For every function-pair query* (f_0, f_1)*, the challenger sends* SK_{f_b} *to the adversary, where* SK_{f_b} *is the output of* $\mathsf{KeyGen}(\mathsf{MSK}, f_b)$ *only if* $f_0(m_0) = f_1(m_1)$*, for all message-pair queries* (m_0, m_1)*. Otherwise, it aborts.*

2. *The output of the experiment is* b'*, where* b' *is the output of* $\mathcal{A}$.

Compactness. We now recall the notion of *compact* FE from [AJ15, BV15]. In a compact FE scheme, the running time of the encryption algorithm only depends on the security parameter and the input message length. In particular, it is independent of the complexity of the function family supported by the FE scheme. We refer to [AJ15, BV15] for a formal definition of this notion.

4 Splittable iO

We describe the notion of splittable iO next. This notion will be associated with a patchable encoding scheme. We define patchable encoding scheme first.

4.1 Patchable Encoding Scheme

A patchable encoding scheme is an encoding scheme associated with a class of Turing machines. This scheme allows for updating an encoding of a machine M using an encoding of a patch P to obtain an encoding of another machine M', where $M' \leftarrow \mathsf{Update}(M, P)$. The secret key, used in the computation of the encodings, is generated using algorithm Gen. Turing machines are encoded using Encode and the patches are encoded using $\mathsf{GenPatch}$. Algorithm $\mathsf{AppPatch}$ is used to apply update the encoding of machine M using encoding of patch P. Finally, Decode is used to decode an encoding of M using the secret key produced by Gen.
Syntax. A patchable encoding scheme is described by the algorithms $\mathsf{UE} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{GenPatch}, \mathsf{AppPatch}, \mathsf{Decode})$ which are defined below. We denote by $\mathcal{M}$, the class of Turing machines it is associated with. We further denote the update algorithm associated with $\mathcal{M}$ to be Update.

- $sk \leftarrow \mathsf{Gen}(1^\lambda)$: On input λ, it produces the secret key sk.

- $\mathcal{E}_{sk}(M) \leftarrow \mathsf{Encode}(sk, M)$: On input secret key sk, Turing machine M, it produces an encoding of M, namely $\mathcal{E}_{sk}(M)$, with respect to sk.

- $\widetilde{P} \leftarrow \mathsf{GenPatch}(sk, P)$: On input secret key sk, patch P, it produces a secure patch $\widetilde{P}$.

- $\mathcal{E}_{sk}(M') \leftarrow \mathsf{AppPatch}\left(\mathcal{E}_{sk}(M), \widetilde{P}\right)$: On input encoding $\mathcal{E}_{sk}(M)$, secure patch $\widetilde{P}$, it produces the updated encoding $\mathcal{E}_{sk}(M)$.

- $M \leftarrow \mathsf{Decode}(sk, \mathcal{E}_{sk}(M))$: On input secret key sk, machine encoding $\mathcal{E}_{sk}(M)$, it produces the machine M.

Efficiency. We require that the size of the secure patches is a (a priori fixed) polynomial in the security parameter and the size of the underlying patch. That is, $|\widetilde{P}| = \text{poly}(\lambda, |P|)$, where $\widetilde{P} \leftarrow \mathsf{GenPatch}(sk, P)$.

Correctness of Sequential Updating. Consider $M \in \mathcal{M}$ and a sequence of patches $P_1, \ldots, P_L$. We consider the following two processes:

- **Encode-then-Update:** Compute the following: (a) $sk \leftarrow \mathsf{Gen}(1^\lambda)$; (b) $\mathcal{E}_{sk}(M_1) \leftarrow \mathsf{Encode}(sk, M)$; (c) For every $i \in [L]$, $\widetilde{P_i} \leftarrow \mathsf{GenPatch}(sk, P_i)$; (d) $\mathcal{E}_{sk}(M_{i+1}) \leftarrow \mathsf{AppPatch}\left(\mathcal{E}_{sk}(M_i), \widetilde{P_i}\right)$.

- **Update:** For every $i \in [L]$, $M_{i+1} \leftarrow \mathsf{Update}(M_i, P_i)$ with $M_1 = M$.

We require that $\mathsf{Decode}(sk, \mathcal{E}_{sk}(M_L)) = M_L$.

Security. We require any patchable encoding scheme to satisfy the following.

Definition 10. *A patchable encoding scheme,* $\mathsf{UE} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{GenPatch}, \mathsf{AppPatch}, \mathsf{Decode})$ *is said to be* **secure** *if the following holds: Consider the game between a challenger and an adversary. The adversary submits machines* $(M_0^1, M_1^1) \ldots, (M_0^Q, M_1^Q) \in \mathcal{M}$ *to the challenger. In return, the adversary receives* $\{\mathcal{E}_{sk}(M_b^j)\}_{j \in [Q]}$*, where* $b \in \{0, 1\}$ *is picked at random. The adversary can then make patch queries* (P_0^i, P_1^i)*, for every* $i \in [L]$*, adaptively. In return it receives* $\widetilde{P_b^i}$*. The probability that the adversary outputs* b *is negligibly close to* $1/2$.

We can correspondingly define an encoding scheme supporting parallel patches.

Instantiation. We can instantiate patchable encoding scheme using a secret key fully homomorphic encryption (FHE) scheme. Let $\mathsf{FHE} = (\mathsf{FHE.Setup}, \mathsf{FHE.Enc}, \mathsf{FHE.Eval}, \mathsf{FHE.Dec})$ be a secret key FHE scheme. We can construct a patchable encoding scheme UE as follows:

- $\mathsf{Gen}(1^\lambda)$: Execute $\mathsf{FHE.Setup}(1^\lambda)$ to obtain $\mathsf{FHE.sk}$. Set the secret key sk to be $\mathsf{FHE.sk}$.

- $\mathsf{Encode}(sk, M)$: Execute the FHE encryption algorithm $\mathsf{FHE.Enc}(\mathsf{FHE.sk}, M)$ to obtain $\mathsf{FHE.CT}_M$. Set the encoding $\mathcal{E}_{sk}(M) = \mathsf{FHE.CT}_M$.

- $\mathsf{GenPatch}(sk, P)$: Execute the FHE encryption algorithm $\mathsf{FHE.Enc}(\mathsf{FHE.sk}, P)$ to obtain $\mathsf{FHE.CT}_P$. Set the encoding $\widetilde{P} = \mathsf{FHE.CT}_P$.

- $\mathsf{AppPatch}\left(\mathcal{E}_{sk}(M), \widetilde{P}\right)$: Execute the evaluation algorithm of FHE; $\mathsf{FHE.CT}_{M'} \leftarrow \mathsf{FHE.Eval}(U_{\mathsf{Update}}, \mathsf{FHE.CT}_M, \mathsf{FHE.CT}_P)$. Here, U_{Update} is a function that takes as input TM-patch pair (M, P) and produces an updated TM M'. Set $\mathcal{E}_{sk}(M') = \mathsf{FHE.CT}_{M'}$.

- $\mathsf{Decode}(sk, \mathcal{E}_{sk}(M))$: Execute the FHE decryption algorithm $\mathsf{FHE.Dec}(\mathsf{FHE.sk}, \mathsf{FHE.CT})$, where $\mathsf{FHE.sk} = sk$ and $\mathsf{FHE.CT} = \mathcal{E}_{sk}(M)$. The decrypted value is output.

The correctness of sequential updating and efficiency properties of UE follows from the corresponding correctness and compactness properties of FHE. The security of UE follows from the semantic security of FHE.

Parallel Patches. Fully homomorphic encryption can also be used to instantiate encoding schemes supporting parallel patches. The construction is same as before.

4.2 Definition of Splittable iO

We define the notion of splittable iO next. A splittable iO is an indistinguishability obfuscation scheme, satisfying additional properties. The model of computation is Turing machines and we work in succinct iO setting (Definition 4). Although the algorithms associated with succinct iO take the input length bound as input, we omit this in the description below. For simplicity, set the input length bound to be λ. Our results can easily be extended to the case when the input bound is an arbitrary polynomial in λ and our parameter sizes would blow accordingly.

Firstly, we require that the obfuscation of M proceeds in two steps: in the first step, M is encoded (twice) using the underlying patchable encoding scheme UE. This is done by generating the setup of UE twice and encoding M using both these secret keys sk_0 and sk_1. Call the two encodings $\mathcal{E}_{sk_0}(M)$ and $\mathcal{E}_{sk_1}(M)$. The second step involves generation of auxiliary information as a function of the encodings $\mathcal{E}_{sk_0}(M)$ and $\mathcal{E}_{sk_1}(M)$ and one of the secret keys. This is enabled via an additional algorithm AuxGen. This requirement on the structure of the obfuscate algorithm is termed as *splittable property*. The second property we require is *correctness of* AuxGen – this says that the correctness of the obfuscated machine should not be affected by whether the two encodings (part of the obfuscated machine) fed to AuxGen are freshly computed or if they are obtained as a result of patching. The third property, which is *efficiency of aux*, states that the auxiliary information produced by AuxGen should be a fixed polynomial in λ. Finally, we have the *indistinguishability of aux* property that states that the auxiliary information obtained by AuxGen on input two encodings $\mathcal{E}_{sk_0}(M)$ and $\mathcal{E}_{sk_1}(M)$ and secret key sk_0 is indistinguishability the output of AuxGen on input $\mathcal{E}_{sk_0}(M)$, $\mathcal{E}_{sk_1}(M)$ and secret key sk_1.

Definition 11 (Splittable iO). *A splittable iO scheme, denoted by* $\mathsf{siO} = (\mathsf{Obf}, \mathsf{Eval})$ *for a class of Turing machines* $\mathcal{M}$, *is an indistinguishability obfuscation scheme that is associated with a patchable encoding scheme* $\mathsf{UE} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{GenPatch}, \mathsf{AppPatch}, \mathsf{Decode})$ *and satisfies the following properties:*

- **Splittable Property:** Obf *consists of* $\mathsf{Gen}, \mathsf{Encode}$ *and an additional PPT algorithm* AuxGen. *On input* $(1^\lambda, M)$ *it proceeds in the following three phases:*

  1. Encoding of M using UE*: (a)* $sk_0 \leftarrow \mathsf{Gen}(1^\lambda)$*;* $sk_1 \leftarrow \mathsf{Gen}(1^\lambda)$*. (b)* $\mathcal{E}_{sk_0}(M) \leftarrow \mathsf{Encode}(sk_0, M)$*;* $\mathcal{E}_{sk_1}(M) \leftarrow \mathsf{Encode}(sk_1, M)$
  2. Generation of aux*:* $aux \leftarrow \mathsf{AuxGen}\,(sk_0, \mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M))$

  Output $\langle M \rangle = (\mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M), aux)$. *The secret state associated with this execution is set to be* (sk_0, sk_1).

- **Correctness of** AuxGen**:** *Let* $M \in \mathcal{M}$ *and let* $P_1, \ldots, P_L$ *be a sequence of patches. Let* M_i *be the* i^{th} *updated machine,* $M_i \leftarrow \mathsf{Update}(M_{i-1}, P)$, *for every* $i \in [L]$, *where* $M_0 = M$.

  Consider the following process:

  – *Let* sk_0, sk_1 *be such that* $sk_0 \leftarrow \mathsf{UE.Gen}(1^\lambda)$, $sk_1 \leftarrow \mathsf{UE.Gen}(1^\lambda)$.

  – *Let* $\mathcal{E}_{sk_0}(M) \leftarrow \mathsf{UE.Encode}(sk_0, M)$ *and* $\mathcal{E}_{sk_1}(M) \leftarrow \mathsf{UE.Encode}(sk_0, M)$.

– *Consider the i^{th} updated encodings, $\mathcal{E}_{sk_0}(M_i) \leftarrow \mathsf{UE.AppPatch}(\mathcal{E}_{sk_0}(M_{i-1}), \mathsf{UE.GenPatch}(sk_0, P_i))$ and $\mathcal{E}_{sk_1}(M_i) \leftarrow \mathsf{UE.AppPatch}\,(\mathcal{E}_{sk_1}(M_{i-1}), \mathsf{UE.GenPatch}(sk_1, P_i))$.*

– *Let $aux \leftarrow \mathsf{AuxGen}(sk_0, \mathcal{E}_{sk_0}(M_L), \mathcal{E}_{sk_1}(M_L))$ and set $\langle M_L \rangle = (\mathcal{E}_{sk_0}(M_L), \mathcal{E}_{sk_1}(M_L), aux)$.*

For every x, we have $\mathsf{Eval}(\langle M_L \rangle, x) = M_L(x)$.

- **Efficiency of aux:** *There exists a polynomial p such that the following holds. Let $(\mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M), aux) \leftarrow \mathsf{Obf}(1^\lambda, M)$ for $M \in \mathcal{M}$. Then, $|aux| = p(\lambda)$.*

- **Indistinguishability of aux:** *Consider $M_0, M_1 \in \mathcal{M}$ such that $M_0(x) = M_1(x)$ for every $x \in \{0,1\}^*$. Suppose E_0, E_1, sk_0, sk_1 are such that $M_0 \leftarrow \mathsf{Decode}(sk_0, E_0)$ and $M_1 \leftarrow \mathsf{Decode}(sk_1, E_1)$. We have,*

$$\{E_0, E_1, sk_0, sk_1, aux_0\} \approx_c \{E_0, E_1, sk_0, sk_1, aux_1\},$$

where $aux_b \leftarrow \mathsf{AuxGen}(sk_b, E_0, E_1)$ for $b \in \{0,1\}$.

An instantiation of splittable iO is presented in Section 8.

We note that the above definition can be extended to the parallel patches setting if the underlying patchable encoding scheme supports parallel patches.

5 Splittable iO to Single-Program *pa-i$\mathcal{O}$*

We give a generic transformation from splittable iO to single-program patchable iO.

Construction. The main tool we use in our construction is a splittable iO scheme $\mathsf{siO} = (\mathsf{siO.Obf}, \mathsf{siO.Eval})$ associated with the updatable encoding scheme $\mathsf{UE} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{GenPatch}, \mathsf{AppPatch}, \mathsf{Decode})$. We construct a single-program patchable obfuscation scheme *pa-i$\mathcal{O}$* below.

Setup, $\mathsf{Setup}(1^\lambda)$: It outputs $\mathsf{SK} = \perp$.

Obfuscate, $\mathsf{Obf}(\mathsf{SK}, M)$: It takes as input the secret key $\mathsf{SK} = \perp$ and a TM $M \in \mathcal{M}$. The obfuscation of M is essentially the obfuscation of M with respect to siO. That is, it executes the obfuscate algorithm of siO on M; $(\mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M), aux) \leftarrow \mathsf{siO.Obf}(1^\lambda, M)$. Denote $(\mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M), aux)$ by $\langle M \rangle$. Let the state associated with this execution be (sk_0, sk_1) (refer to Splittable Property in Definition 11).

It outputs the obfuscated TM $\langle M \rangle$. The state is set to be $\mathsf{st} = (sk_0, sk_1, \mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M))$. That is, the state consists of the two secret keys and the patchable encodings of M with respect to sk_0 and sk_1.

Secure Patch Generation, $\mathsf{GenPatch}(\mathsf{SK}, P, \mathsf{st})$: It takes as input the secret key $\mathsf{SK} = \perp$, a description of a patch $P \in \mathcal{P}$ and state $\mathsf{st} = (sk_0, sk_1, \mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M))$. Then,

- It computes the secure patches, $\widetilde{P}^0 \leftarrow \mathsf{UE.GenPatch}(sk_0, P)$ and $\widetilde{P}^1 \leftarrow \mathsf{UE.GenPatch}(sk_1, P)$.

- It applies the secure patches on the encodings, $\mathcal{E}_{sk_0}(M') \leftarrow \mathsf{UE.AppPatch}(\mathcal{E}_{sk_0}(M), \widetilde{P}^0)$ and $\mathcal{E}_{sk_1}(M') \leftarrow \mathsf{UE.AppPatch}(\mathcal{E}_{sk_1}(M), \widetilde{P}^1)$.

- It then executes AuxGen algorithm of siO. It computes $aux' \leftarrow \mathsf{AuxGen}(sk_0, \mathcal{E}_{sk_0}(M_0), \mathcal{E}_{sk_1}(M_1))$.

It outputs a secure patch $\langle P \rangle = (\widetilde{P}^0, \widetilde{P}^1, aux')$. It updates the state to be $\mathsf{st}' = (sk_0, sk_1, \mathcal{E}_{sk_0}(M'), \mathcal{E}_{sk_1}(M'))$.

Note: It suffices to just include the encodings $(\widetilde{P}^0, \widetilde{P}^1)$ *(and not the updated encodings* $\mathcal{E}_{sk_0}(M'), \mathcal{E}_{sk_1}(M')$ *) as part of secure patch because anyone having the original pair of encodings* $(\mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M))$ *can now recompute the* $(\mathcal{E}_{sk_0}(M'), \mathcal{E}_{sk_1}(M'))$ *by using just* $(\widetilde{P}^0, \widetilde{P}^1)$.

Applying Patch, $\mathsf{AppPatch}(\langle M \rangle, \langle P \rangle)$: It takes as input an obfuscated TM $\langle M \rangle = (\mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M), aux)$ and a secure patch $\langle P \rangle = (\widetilde{P}^0, \widetilde{P}^1, aux')$.

- It applies the secure patches on the encodings, $\mathcal{E}_{sk_0}(M') \leftarrow \mathsf{UE.AppPatch}(\mathcal{E}_{sk_0}(M), \widetilde{P}^0)$ and $\mathcal{E}_{sk_1}(M') \leftarrow \mathsf{UE.AppPatch}(\mathcal{E}_{sk_1}(M), \widetilde{P}^1)$.
- It replaces aux with aux' which is sent as part of the patch.

It outputs an updated obfuscation $\langle M' \rangle = (\mathcal{E}_{sk_0}(M'), \mathcal{E}_{sk_1}(M'), aux')$.

Evaluation, $\mathsf{Eval}(\langle M \rangle, x)$: It takes as input an obfuscated TM $\langle M \rangle$ and an input x. It executes the evaluation algorithm of siO; $y \leftarrow \mathsf{siO.Eval}(\langle M \rangle, x)$. Output y.

Efficiency. We claim that the size of the secure patch solely depends on the size of the patch and the security parameter. In particular, it is independent of the size of the machine.

Consider a patch P. Let the output of $\mathsf{GenPatch}(\mathsf{SK}, P, \mathsf{st})$ be $\langle P \rangle = (\widetilde{P}^0, \widetilde{P}^1, aux')$. From the efficiency of the underlying patchable encoding scheme, $|(\widetilde{P}^0, \widetilde{P}^1)| = \mathrm{poly}(\lambda, |P|)$. From the efficiency of the underlying spittable iO scheme, $|aux'| = \mathrm{poly}(\lambda)$.

Remark 5. *The secure patch generation time in the above scheme is proportional to the size of the obfuscated machine. This is in general undesirable and we show how to deal with this issue in Section 6.*

Correctness of Sequential Updating. Consider a TM $M_0 \in \mathcal{M}$ and a sequence of patches $P_1, \ldots, P_L \in \mathcal{P}$. Consider the following two processes generated using the above scheme. For every $i \in \{1, \ldots, L\}$, we have:

- **Obfuscate-then-Update**: Compute the following: (a) $\mathsf{SK} \leftarrow \mathsf{Setup}(1^\lambda)$, (b) $(\langle M_0 \rangle, \mathsf{st}_0) \leftarrow \mathsf{Obf}(\mathsf{SK}, M_0)$, (c) $(\langle P_i \rangle, \mathsf{st}_i) \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P_i, \mathsf{st}_{i-1})$, (d) $\langle M_i \rangle \leftarrow \mathsf{AppPatch}\Big(\langle M_{i-1} \rangle, \langle P_i \rangle\Big)$.
- **Update**: $M_i \leftarrow \mathsf{Update}(M_{i-1}, P_i)$.

We have the following claim.

Claim 1. *For every x, we have* $\mathsf{Eval}(\langle M_L \rangle, x) = M_L(x)$.

Proof. Let $\langle M_0 \rangle = (E_0^0, E_1^0, aux^0)$, $\mathsf{st} = (sk_0, sk_1, E_0^0, E_1^0)$ and $\langle M_L \rangle = (E_0^L, E_1^L, aux^L)$. Note that E_0 is the output of an execution of $\mathsf{Encode}(sk_0, M_0)$ and aux^0 is the output of $\mathsf{AuxGen}(sk_0, E_0^0, E_1^0)$. From the correctness of patchable encoding scheme, we have $\mathsf{Decode}(\mathsf{SK}_0, E_0^L) = M_L$. Using this fact along with the correctness of AuxGen property of siO, we get that the output of $\mathsf{Eval}(\langle M_L \rangle, x)$ to be $M_L(x)$. □

Security of Sequential Updating. We prove,

Theorem 8. *pa-i$\mathcal{O}$ satisfies security of sequential updating property.*

Proof. We begin with a high level proof overview and then provide the formal details.

Overview. Consider two functionally equivalent Turing machines M_0 and M_1. The goal is to show that any PPT adversary cannot distinguish the obfuscations of M_0 and M_1 even when he is allowed to patch M_0 or M_1 sequentially as long as the patching does not destroy functional equivalence.

In the first hybrid, the adversary is given the obfuscation $(\mathcal{E}_{sk_0}(M_b), \mathcal{E}_{sk_1}(M_b), aux_{sk_0})$, where aux_{sk_0} is generated as $aux_{sk_0} \leftarrow \mathsf{AuxGen}(sk_0, \mathcal{E}_{sk_0}(M_b), \mathcal{E}_{sk_1}(M_b))$ and bit b is picked at random. Similarly, the b^{th} patch is used to generate secure patch in every patch query. In the next step, we change $(\mathcal{E}_{sk_0}(M_b), \mathcal{E}_{sk_1}(M_b), aux_{sk_0})$ to $(\mathcal{E}_{sk_0}(M_b), \mathcal{E}_{sk_1}(M_1), aux_{sk_0})$. Similarly we switch the patches encoded under sk_1 from $\widetilde{P}_b^i$ to $\widetilde{P}_1^i$ for every $i \in [L]$. This is legal since sk_1 is not present in the encodings and thus we can invoke the security of patchable encoding. Then, we switch from generating aux using sk_0 to generating aux using sk_1. We do this, one at a time, for every aux present as part of the obfuscation as well as the secure patches. We explain shortly how this is done. After we have completely switched all the aux, generated using sk_0 to being generated using sk_1, we then switch from $(\mathcal{E}_{sk_0}(M_1), \mathcal{E}_{sk_1}(M_1), aux_{sk_0})$. Also, every secure patch encodes P_1 under both sk_0 and sk_1. At this point, the challenge bit b is completely hidden.

All is remaining is to show that we can indeed switch from aux from sk_0 to sk_1 at every step. A first thought would be to reduce this to the indistinguishability of aux property of siO. That is, a reduction uses the adversary of patchable iO to break the indistinguishability of aux property. Suppose we are switching aux in the secure patch corresponding to i^{th} patch query. The reduction computes the encoding of i^{th} updated machine pair $(\mathcal{E}_{sk_0}(M_b^i), \mathcal{E}_{sk_1}(M_1^i))$ along with (sk_0, sk_1) and sends it to the challenger of siO. In response, it receives aux_i^* generated using either sk_0 or sk_1. The reduction now is expected to use this to respond to the adversary. The main issue here is that the reduction has to, along with aux_i^*, also send patch encodings of (P_b^i, P_1^i). Moreover, these encodings might not be compatible with aux_i^*. To resolve this issue, the reduction generates $(\mathcal{E}_{sk_0}(M_b^i), \mathcal{E}_{sk_1}(M_1^i))$, not afresh as done before, but instead by updating the $(i-1)^{th}$ encoding using patch encodings of (P_b^i, P_1^i). Refer to the hybrids for more details.

Formal Details. Consider the following hybrids. In some of the hybrids below, we use a box around the text to highlight the difference from the previous hybrids.

Hyb_1: Let $(M_0, M_1) \in \mathcal{M}$ be the TM query made by adversary $\mathcal{A}$. Challenger picks a bit b at random and sends the obfuscation of M_b, namely $\langle M_b \rangle$, to the adversary. Then, adversary adaptively submits the patch queries $\big((P_0^1, P_1^1), \ldots, (P_0^L, P_1^L)\big)$ and in response receives the secure patches $\big(\langle P_b^1 \rangle, \ldots, \langle P_b^L \rangle\big)$. The adversary outputs b'. The output of hybrid is 1 if $b' = b$.

Hyb_2: Let $(M_0, M_1) \in \mathcal{M}$ be the TM query made by adversary $\mathcal{A}$. Challenger picks a bit b at random. It then executes the setup of patchable encoding scheme twice: $sk_0 \leftarrow \mathsf{Gen}(1^\lambda)$; $sk_1 \leftarrow \mathsf{Gen}(1^\lambda)$. It then encodes M_b using sk_0; $\mathcal{E}_{sk_0}(M_b) \leftarrow \mathsf{Encode}(sk_0, M_b)$ and it encodes M_1 using sk_1; $\mathcal{E}_{sk_1}(M_1) \leftarrow \mathsf{Encode}(sk_1, M_1)$. It then generates aux by executing $aux \leftarrow \mathsf{AuxGen}(sk_0, \mathcal{E}_{sk_0}(M_b), \mathcal{E}_{sk_1}(M_1))$. It sets $\langle M_b \rangle = (\mathcal{E}_{sk_0}(M_b), \boxed{\mathcal{E}_{sk_1}(M_1)}, aux)$. The challenger handles patch queries as follows: upon receiving the patch query (P_0^i, P_1^i), for every $i \in [L]$, it first computes the secure patches, $(\widetilde{P_b^i})^0 \leftarrow \mathsf{UE.GenPatch}(sk_0, P_b^i)$ and $(\widetilde{P_1^i})^1 \leftarrow \mathsf{UE.GenPatch}(sk_1, P_1^i)$. It then computes aux_i as a function of sk_0 and the updated machine encodings as given in the description of the scheme. It sets $\langle P_b^i \rangle = ((\widetilde{P_b^i})^0, \boxed{(\widetilde{P_1^i})^1}, aux_i)$. The rest of the hybrid is same as before.

Hyb_3: Let $(M_0, M_1) \in \mathcal{M}$ be the TM query made by adversary $\mathcal{A}$. Challenger picks a bit b at random. It then executes the setup of patchable encoding scheme twice: $sk_0 \leftarrow \mathsf{Gen}(1^\lambda)$; $sk_1 \leftarrow \mathsf{Gen}(1^\lambda)$. It then encodes M_b using sk_0; $\mathcal{E}_{sk_0}(M_b) \leftarrow \mathsf{Encode}(sk_0, M_b)$ and it encodes M_1 using sk_1; $\mathcal{E}_{sk_1}(M_1) \leftarrow \mathsf{Encode}(sk_1, M_1)$. It then generates aux by executing $aux \leftarrow \mathsf{AuxGen}(\boxed{sk_1}, \mathcal{E}_{sk_0}(M_b), \mathcal{E}_{sk_1}(M_1))$. Observe that sk_1 is used to generate aux; this is the only difference between this hybrid and the previous hybrid. It sets $\langle M_b \rangle = (\mathcal{E}_{sk_0}(M_b), \mathcal{E}_{sk_1}(M_1), aux)$. The patch queries are handled as in the previous hybrid.

$\mathsf{Hyb}_{4.i}$ **for** $i \in [L]$: The machine query is handled as in the previous hybrid. Let the initial state be $\mathsf{st}_0 = (sk_0, sk_1, \mathcal{E}_{sk_0}(M_b^0), \mathcal{E}_{sk_1}(M_1^0))$. Upon receiving a patch query (P_0^k, P_1^k), the challenger does the following: It computes the encodings of the patches using UE by computing $(\widetilde{P_b^k})^0 \leftarrow \mathsf{UE.GenPatch}(sk_0, P_b^k)$ and $(\widetilde{P_1^k})^1 \leftarrow \mathsf{UE.GenPatch}(sk_1, P_1^k)$. It applies the secure patches on the updated encodings, $\mathcal{E}_{sk_0}(M_b^k) \leftarrow \mathsf{UE.AppPatch}(\mathcal{E}_{sk_0}(M_b^{k-1}), (\widetilde{P_b^k})^0)$ and $\mathcal{E}_{sk_1}(M_1^k) \leftarrow \mathsf{UE.AppPatch}(\mathcal{E}_{sk_1}(M_1^{k-1}), (\widetilde{P_1^k})^1)$. Here, M_b^j (resp., M_1^j) is the machine obtained as the output of $M_b^j \leftarrow \mathsf{Update}(M_b^{j-1}, P_b^j)$ (resp., $M_1^j \leftarrow \mathsf{Update}(M_1^{j-1}, P_1^j)$) for every $1 \leq j \leq k-1$ and $M_b^0 = M_b$ (resp., $M_1^0 = M_1$). It then executes AuxGen algorithm of siO. There are two cases:

- If $k < i$: It computes $aux_k \leftarrow \mathsf{AuxGen}(sk_1, \mathcal{E}_{sk_0}(M_b^k), \mathcal{E}_{sk_1}(M_1^k))$. That is, sk_1 is the secret key used to compute aux_k. It sets the secure patch as:
$$\langle P_b^k \rangle = \Big((\widetilde{P_b^k})^0, (\widetilde{P_1^k})^1, aux_k \Big)$$
- If $k \geq i$: If $k < i$: It computes $aux_k \leftarrow \mathsf{AuxGen}(sk_0, \mathcal{E}_{sk_0}(M_b^k), \mathcal{E}_{sk_1}(M_1^k))$. That is, sk_0 is the secret key used to compute aux_k. It sets the secure patch as:
$$\langle P_b^k \rangle = \Big((\widetilde{P_b^k})^0, (\widetilde{P_1^k})^1, aux_k \Big)$$

The rest of the hybrid is as before.

Hyb_5: Let $(M_0, M_1) \in \mathcal{M}$ be the TM query made by adversary $\mathcal{A}$. Challenger picks a bit b at random. It then executes the setup of patchable encoding scheme twice: $sk_0 \leftarrow \mathsf{Gen}(1^\lambda)$; $sk_1 \leftarrow \mathsf{Gen}(1^\lambda)$. It then encodes M_1 using sk_0; $\mathcal{E}_{sk_0}(M_1) \leftarrow \mathsf{Encode}(sk_0, M_1)$ and it encodes M_1 using sk_1; $\mathcal{E}_{sk_1}(M_1) \leftarrow \mathsf{Encode}(sk_1, M_1)$. It then generates aux by executing $aux \leftarrow \mathsf{AuxGen}(sk_0, \mathcal{E}_{sk_0}(M_1), \mathcal{E}_{sk_1}(M_1))$. It sets $\langle M_b \rangle = (\boxed{\mathcal{E}_{sk_0}(M_1)}, \mathcal{E}_{sk_1}(M_1), aux)$. The challenger handles patch queries as follows: upon receiving the patch query (P_0^i, P_1^i), for every $i \in [L]$, it first computes the secure patches, $(\widetilde{P_1^i})^0 \leftarrow \mathsf{UE.GenPatch}(sk_0, P_1^i)$ and $(\widetilde{P_1^i})^1 \leftarrow \mathsf{UE.GenPatch}(sk_1, P_1^i)$. It then computes aux_i as a function of sk_1 and the updated machine encodings as in $\mathsf{Hyb}_{4.L+1}$. It sets $\langle P_b^i \rangle = (\boxed{(\widetilde{P_1^i})^0}, (\widetilde{P_1^i})^1, aux_i)$. The rest of the hybrid is same as before.

Indistinguishability of Hybrids. Consider the following claims.

Claim 2. *Assuming the security of* UE*, we have* $|\Pr[1 \leftarrow \mathsf{Hyb}_1] - \Pr[1 \leftarrow \mathsf{Hyb}_2]| \leq \mathsf{negl}(\lambda)$*, for some negligible function* negl.

Proof. We define a reduction $\mathcal{B}$ that uses $\mathcal{A}$ to break the security of UE. We denote the challenger of UE by Ch.

$\mathcal{B}$ receives a pair of Turing machines (M_0, M_1) from $\mathcal{A}$. It picks a bit b at random. It sends (M_b, M_1) to Ch. In response $\mathcal{B}$ gets back E_1^*. It then samples a secret key $sk_0 \leftarrow \mathsf{Gen}(1^\lambda)$ and encodes M_b using sk_0 to get $\mathcal{E}_{sk_0}(M_b) \leftarrow \mathsf{Encode}(sk_0, M_b)$. It then computes aux as $aux \leftarrow \mathsf{AuxGen}(sk_0, \mathcal{E}_{sk_0}(M_b), E_1^*)$. Reduction $\mathcal{B}$ then sends $\langle M_b \rangle = (\mathcal{E}_{sk_0}(M_b), E_1^*, aux)$ to $\mathcal{A}$. Adversary $\mathcal{A}$ then makes patch queries of the form (P_0^i, P_1^i). To compute the secure patch, $\mathcal{B}$ first sends the query (P_b^i, P_1^i) to Ch. In response it receives $(\widetilde{P}_*^i)^1$. $\mathcal{B}$ then computes $(\widetilde{P}_b^i)^0 \leftarrow \mathsf{UE.GenPatch}(sk_0, P_b^i)$. It then computes aux_i as $aux_i \leftarrow \mathsf{AuxGen}(sk_0, \mathcal{E}_{sk_0}(M_b^i), E_1^i)$, where (a) $\mathcal{E}_{sk_0}(M_b^i)$ is obtained by updating $\mathcal{E}_{sk_0}(M_b)$ using the patches $\{(\widetilde{P}_b^j)^0\}_{j \leq i}$ and, (b) E_1^i is obtained by updating E_1^* using the patches $\{(\widetilde{P}_*^j)^1\}_{j \leq i}$. Reduction $\mathcal{B}$ then sends $\langle P_i \rangle = \left((\widetilde{P}_b^i)^0, (\widetilde{P}_*^i)^1, aux_i\right)$ to $\mathcal{A}$. This completes the description of the reduction.

Suppose Ch used the challenge bit 0 then we are in Hyb_1, else if it used bit 1 then we are in Hyb_2. From the security of UE, the claim follows. □

Claim 3. *Assuming the indistinguishability of aux property of* siO*, we have* $|\Pr[1 \leftarrow \mathsf{Hyb}_2] - \Pr[1 \leftarrow \mathsf{Hyb}_3]| \leq \mathsf{negl}(\lambda)$*, for some negligible function* negl.

Proof. We define a reduction $\mathcal{B}$ that uses $\mathcal{A}$ to break the security of siO. We denote the challenger of siO to be Ch.

$\mathcal{B}$ receives a pair of Turing machines (M_0, M_1) from $\mathcal{A}$. It picks a bit b at random. It then encodes M_b using sk_0 to get $\mathcal{E}_{sk_0}(M_b) \leftarrow \mathsf{Encode}(sk_0, M_b)$ and M_1 using sk_1 to get $\mathcal{E}_{sk_1}(M_1) \leftarrow \mathsf{Encode}(sk_1, M_1)$. It sets $(E_0, E_1) = (\mathcal{E}_{sk_0}(M_b), \mathcal{E}_{sk_0}(M_1))$. It sends (E_0, E_1, sk_0, sk_1) to Ch and in response it receives aux^*. It then sets $\langle M_b \rangle = (\mathcal{E}_{sk_0}(M_0), \mathcal{E}_{sk_1}(M_1))$. $\mathcal{B}$ sends $\langle M_b \rangle$ to $\mathcal{A}$. The patch queries are answered by $\mathcal{B}$ as in Hyb_2.

Note that E_0 and E_1 are such that $\mathsf{Decode}(sk_0, E_0) = M_b$ and $\mathsf{Decode}(sk_1, E_1) = M_1$. Furthermore, M_0 and M_1 are functionally equivalent. If Ch uses sk_0 to generate aux^* then we are in Hyb_2. If it uses sk_1 to generate aux^* then we are in Hyb_3. From the indistinguishability of aux property of siO, the claim follows. □

Claim 4. $\Pr[1 \leftarrow \mathsf{Hyb}_3] = \Pr[1 \leftarrow \mathsf{Hyb}_{4.1}]$.

Proof. This follows from the fact that Hyb_3 and $\mathsf{Hyb}_{4.1}$ are identical hybrids. □

Claim 5. *Assuming the indistinguishability of aux property of* siO*, we have* $|\Pr[1 \leftarrow \mathsf{Hyb}_{4.i}] - \Pr[1 \leftarrow \mathsf{Hyb}_{4,i+1}]| \leq \mathsf{negl}(\lambda)$*, for every* $i \in [L]$ *and for some negligible function* negl.

Proof. We define a reduction $\mathcal{B}$ that uses $\mathcal{A}$ to break the security of siO. We denote the challenger of siO to be Ch.

$\mathcal{B}$ receives a pair of Turing machines (M_0, M_1) from $\mathcal{A}$. It picks a bit b at random. It then encodes M_b using sk_0 to get $\mathcal{E}_{sk_0}(M_b) \leftarrow \mathsf{Encode}(sk_0, M_b)$ and M_1 using sk_1 to get $\mathcal{E}_{sk_1}(M_1) \leftarrow \mathsf{Encode}(sk_1, M_1)$. It generates $aux \leftarrow \mathsf{AuxGen}(sk_1, \mathcal{E}_{sk_0}(M_b), \mathcal{E}_{sk_1}(M_1))$. It sends $(\mathcal{E}_{sk_0}(M_b), \mathcal{E}_{sk_1}(M_1), aux)$ to $\mathcal{A}$.

Upon receiving the k^{th} patch query (P_0^k, P_1^k), the challenger first computes $\widetilde{P}_b^k \leftarrow \mathsf{UE.GenPatch}(sk_0, P_b^k)$ and $\widetilde{P}_1^k \leftarrow \mathsf{UE.GenPatch}(sk_1, P_1^k)$. Denote by $\mathcal{E}_{sk_0}(M_b^k)$ and $\mathcal{E}_{sk_1}(M_1^k)$ be the k^{th} update phase encodings obtained by patching $\mathcal{E}_{sk_0}(M_b)$ and $\mathcal{E}_{sk_1}(M_1)$ using $\{\widetilde{P}_b^j\}_{j \le k}$ and $\{\widetilde{P}_1^j\}_{j \le k}$ respectively. The next step is decided depending on the following three cases:

- $k < i$: Execute $aux_k \leftarrow \mathsf{AuxGen}(sk_1, \mathcal{E}_{sk_0}(M_b^k), \mathcal{E}_{sk_1}(M_1^k))$. Set $\langle P_b^k \rangle = (\widetilde{P}_b^k, \widetilde{P}_1^k, aux_k)$.
- $k = i$: $\mathcal{B}$ sends $(\mathcal{E}_{sk_0}(M_b^k), \mathcal{E}_{sk_1}(M_1^k), sk_0, sk_1)$ to Ch. In response it receives aux_k^*. Set $\langle P_b^k \rangle = (\widetilde{P}_b^k, \widetilde{P}_1^k, aux_k^*)$.
- $k > i$: Execute $aux_k \leftarrow \mathsf{AuxGen}(sk_0, \mathcal{E}_{sk_0}(M_b^k), \mathcal{E}_{sk_1}(M_1^k))$. Set $\langle P_b^k \rangle = (\widetilde{P}_b^k, \widetilde{P}_1^k, aux_k)$.

$\mathcal{B}$ sends $\langle P_b^k \rangle$ to $\mathcal{A}$.

Note that $\mathcal{E}_{sk_0}(M_b^i)$ and $\mathcal{E}_{sk_1}(M_1^k)$ are such that $\mathsf{Decode}(sk_0, \mathcal{E}_{sk_0}(M_b^i)) = M_b^i$ and $\mathsf{Decode}(sk_1, \mathcal{E}_{sk_1}(M_1^i)) = M_1^i$. Here M_b^i and M_1^i are obtained by updating M_b and M_1 using the sequence of patches $\{P_b^j\}_{j \le i}$ and $\{P_1^j\}_{j \le i}$ respectively. Since $\mathcal{A}$ is a valid adversary, M_b^i and M_1^i are functionally equivalent. Thus, $\mathcal{B}$ is a valid adversary in the security property of siO. If Ch uses sk_0 in the generation of aux_k^* then we are in $\mathsf{Hyb}_{4.i}$, else if it uses sk_1 then we are in $\mathsf{Hyb}_{4.i+1}$. From the indistinguishability of aux security, the claim follows. □

Claim 6. *Assuming the security of* UE, *we have* $\mathsf{Pr}[1 \leftarrow \mathsf{Hyb}_{4.L+1}] - \mathsf{Pr}[1 \leftarrow \mathsf{Hyb}_5]| \le \mathsf{negl}(\lambda)$, *for some negligible function* negl.

Proof. This is identical to the proof of Claim 2. □

Claim 7. $\mathsf{Pr}[1 \leftarrow \mathsf{Hyb}_5] = 0.5$.

Proof. The bit b is information theoretically hidden from $\mathcal{A}$. Hence, the probability that $\mathcal{A}$ outputs b is 0.5. □

From the above claims, we have that $\mathsf{Pr}[1 \leftarrow \mathsf{Hyb}_1] \le 1/2 + \mathsf{negl}(\lambda)$. This proves the theorem.

□

Parallel Patches. The above transformation yields a parallel patchable iO if the underlying splittable iO support parallel patches. The correctness and the security proof follow along the same lines as the sequential setting.

6 Single-Program *pa-i*$\mathcal{O}$: Stateful to Stateless

We give a generic transformation from any single-program patchable iO that maintains state to one that does not maintain any state. The main idea in our transformation is that the state, instead of being maintained at the authority's end, will be maintained at the user's end. That is, the authority delegates the state to the user. Now, the authority can no longer compute the secure patches since these patches are computed as a function of the state. Hence, the authority also delegates the computation of the patch to the user. However, new issues arise: we need to make sure that the state as well as the patches are hidden from the user and yet the user should be able to compute the secure patches. In order to resolve this issue, we use a garbled TM with persistent memory (refer to Section 3.4). The authority initially encodes the state using the database encoding algorithm. This encoded state will be included as part of the initial obfuscated machine. Whenever the authority wants to generate a secure patch of P, it computes a program encoding of $\mathsf{GenPatch}(sk, P, \cdot)$ and sends this program encoding to the user. The user then evaluates this program encoding on encoding of the state and outputs the secure patch $\langle P \rangle$ along with the encoding of the updated state.

We formally define stateless single-program patchable iO below.

Definition 12 (Stateless Single-Program Patchable iO)**.** *A single-program patchable obfuscation* $pa\text{-}i\mathcal{O} = (\mathsf{Setup}, \mathsf{Obf}, \mathsf{GenPatch}, \mathsf{AppPatch}, \mathsf{Eval})$ *for a class of Turing machines* $\mathcal{M}$ *is said to be* stateless *if the following holds: (a) For every* $M \in \mathcal{M}$*,* $\mathsf{st} = \perp$*; where* $\mathsf{SK} \leftarrow \mathsf{Setup}(1^\lambda)$*,* $(\langle M \rangle, \mathsf{st}) \leftarrow \mathsf{Obf}(\mathsf{SK}, M)$ *and, (b) For every patch* P*,* $\mathsf{st}' = \perp$*; where* $(\langle P \rangle, \mathsf{st}') \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P, \mathsf{st})$*.*

Remark 6. *From now on, we omit the parameter* st *in the description of the algorithms of single-program patchable iO.*

Transformation. The main tool in our transformation is garbled TM with persistent memory (Section 3.4). We denote this by $\mathsf{GTM} = (\mathsf{Gen}, \mathsf{GarbDB}, \mathsf{GarbTM}, \mathsf{GarbEval})$.

Let $pa\text{-}i\mathcal{O}_{\mathsf{ST}} = (\mathsf{Setup}, \mathsf{Obf}, \mathsf{GenPatch}, \mathsf{AppPatch}, \mathsf{Eval})$ be any single-program patchable iO scheme. We construct a stateless single-program patchable iO scheme $pa\text{-}i\mathcal{O}$ below.

<u>$\mathsf{Setup}(1^\lambda)$</u>: On input security parameter λ, execute the setup of GTM; $\mathsf{GTM}.\mathbf{k} \leftarrow \mathsf{GTM}.\mathsf{Gen}(1^\lambda)$. Execute the setup of the underlying patchable obfuscation scheme $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$; $pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{SK} \leftarrow pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{Setup}(1^\lambda)$. Output the secret key $\mathsf{SK} = (\mathsf{GTM}.\mathbf{k},\ pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{SK})$.

<u>$\mathsf{Obf}(\mathsf{SK}, M)$</u>: On input secret key $\mathsf{SK} = (\mathsf{GTM}.\mathbf{k},\ pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{SK})$, Turing machine $M \in \mathcal{M}$, execute the obfuscate algorithm of the underlying patchable obfuscation scheme; $(\langle M \rangle_{\mathsf{ST}}, \mathsf{st}) \leftarrow pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{Obf}(pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{SK}, M)$. Encode the state st using the garbled TM scheme. That is, compute $\widehat{\mathsf{st}} \leftarrow \mathsf{GTM}.\mathsf{GarbDB}(\mathsf{GTM}.\mathbf{k}, \mathsf{st})$.

Output the obfuscation $\langle M \rangle = \big(\langle M \rangle_{\mathsf{ST}}, \widehat{\mathsf{st}}\big)$.

<u>$\mathsf{GenPatch}(\mathsf{SK}, P)$</u>: On input secret key $\mathsf{SK} = (\mathsf{GTM}.\mathbf{k},\ pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{SK})$ and patch P, generate the garbled TM program encoding of the program $\mathsf{GP} = \mathsf{GP}\,[pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{SK}, P, r]$, where (i) r is picked at random and, (ii) GP is defined as follows: GP takes as input st, it first computes $(\langle P \rangle_{\mathsf{ST}}, \mathsf{st}') \leftarrow pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{GenPatch}(pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{SK}, P, \mathsf{st};\ r)$. It updates st to be now st'. It outputs $\langle P \rangle_{\mathsf{ST}}$. That is, GP takes as input the database st and essentially executes the patch generation of $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$. The state output by patch generation will be used to update the current state and the secure patch will be the output of GP.

Once GP is defined, compute the encoding, $\widehat{\mathsf{GP}} \leftarrow \mathsf{GTM.GarbTM}(\mathsf{GTM}.\mathbf{k}, \mathsf{GP})$. Finally, output the secure patch $\langle P \rangle = \widehat{\mathsf{GP}}$.

<u>$\mathsf{AppPatch}(\langle M \rangle, \langle P \rangle)$</u>: On input obfuscation $\langle M \rangle = \big(\langle M \rangle_{\mathsf{ST}}, \widehat{\mathsf{st}}\big)$ and secure patch $\langle P \rangle = \widehat{\mathsf{GP}}$, execute the evaluation algorithm of the garbled TM scheme. That is, compute $\langle P \rangle_{\mathsf{ST}} \leftarrow \mathsf{GarbEval}(\widehat{GP},\ \widehat{\mathsf{st}})$ and denote the updated databased encoding to be $\widehat{\mathsf{st}'}$. After this, compute the $\mathsf{AppPatch}$ algorithm of $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$. That is, compute $\langle M' \rangle_{\mathsf{ST}} \leftarrow pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{AppPatch}(\langle M \rangle_{\mathsf{ST}}, \langle P \rangle_{\mathsf{ST}})$.

Output the updated obfuscated machine, $\langle M' \rangle = \Big(\langle M' \rangle_{\mathsf{ST}}, \widehat{\mathsf{st}'}\Big)$.

<u>$\mathsf{Eval}(\langle M \rangle, x)$</u>: On input $\langle M \rangle = (\langle M \rangle_{\mathsf{ST}}, \widehat{\mathsf{st}})$ and instance x, execute the evaluation algorithm of $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$; $y \leftarrow pa\text{-}i\mathcal{O}_{\mathsf{ST}}.\mathsf{Eval}(\langle M \rangle_{\mathsf{ST}}, x)$. Output y.

Efficiency. The efficiency requirements satisfied by the underlying garbled TMs scheme and the single-program patchable iO scheme, affect the corresponding efficiency requirements on the stateless *pa-i*$\mathcal{O}$ scheme. If we start from a succinct garbled TM scheme then the resulting *pa-i*$\mathcal{O}$ scheme satisfies 'Patch Generation Efficiency' property assuming that $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$ satisfied 'Patch Size Efficiency' property. This is because the running time of patch generation of *pa-i*$\mathcal{O}$ now depends only on the time require to generate the GTM encoding of $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$'s patch generation algorithm. The size of the patch encoding, and hence the secure patch of *pa-i*$\mathcal{O}$, is at least as big as the size of the secure patch of $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$. Note that even if $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$ did not satisfy patch generation efficiency property, the resulting *pa-i*$\mathcal{O}$ will satisfy this property.

However, on the other hand if we start off with a non-succinct garbled TM scheme, then the running time of patch generation of *pa-i*$\mathcal{O}$ will be proportional to the running time of patch generation time of $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$. This is because the running time of GTM encoding now is proportional to the worst-case running time of the program it is encoding. Hence, in order for *pa-i*$\mathcal{O}$ to satisfy 'Patch Generation Efficiency' property, we require that $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$ satisfies 'Patch Generation Efficiency' property.

Summarising, we have

Claim 8. *Suppose* GTM *is a succinct garbled TM scheme (Definition 7) and if* $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$ *satisfied 'Patch Size Efficiency' property then pa-i*$\mathcal{O}$ *satisfies 'Patch Generation Efficiency' property.*

Claim 9. *Suppose* GTM *is a non-succinct garbled TM scheme (Definition 8) and if* $pa\text{-}i\mathcal{O}_{\mathsf{ST}}$ *satisfied 'Patch Generation Efficiency' property then pa-i*$\mathcal{O}$ *satisfies 'Patch Generation Efficiency' property.*

Correctness of Sequential Updating. Consider a TM $M_0 \in \mathcal{M}$ and a sequence of patches $P_1, \ldots, P_L \in \mathcal{P}$. Consider the following two processes generated using the above scheme. For every $i \in \{1, \ldots, L\}$, we have:

- **Obfuscate-then-Update**: Compute the following: (a) $\mathsf{SK} \leftarrow \mathsf{Setup}(1^\lambda)$, (b) $\langle M_0 \rangle \leftarrow \mathsf{Obf}(\mathsf{SK}, M_0)$, (c) $\langle P_i \rangle \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P_i)$, (d) $\langle M_i \rangle \leftarrow \mathsf{AppPatch}\Big(\langle M_{i-1} \rangle, \langle P_i \rangle\Big)$.

- **Update**: $M_i \leftarrow \mathsf{Update}(M_{i-1}, P_i)$.

We need to show that $\mathsf{Eval}(\langle M_L \rangle, x) = M_L(x)$. In order to do this, it suffices to prove the following.

Claim 10. *For every $i \in [L]$, the output of* $\mathsf{AppPatch}(\langle M_{i-1}\rangle, \langle P_i\rangle)$ *is exactly the output of* $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{AppPatch}(\langle M_{i-1}\rangle_{\mathsf{ST}}, \langle P_i\rangle_{\mathsf{ST}})$*, where* $\langle M_{i-1}\rangle_{\mathsf{ST}}$ *is an obfuscation of* M_{i-1} *and* $\langle P_i\rangle_{\mathsf{ST}}$ *is a secure patch of* P_i *computed honestly according to the description of* $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}$.

Suppose we have the above claim, then we invoke the correctness of $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}$. This would show that for every $i \in [L] \cup \{0\}$, for every x, we have $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{Eval}(\langle M_i\rangle_{\mathsf{ST}}, x) = M_i(x)$, where $\langle M_i\rangle_{\mathsf{ST}}$ is the output of $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{AppPatch}\left(\langle M_{i-1}\rangle_{\mathsf{ST}}, \langle P_i\rangle_{\mathsf{ST}}\right)$. We now prove the above claim.

Proof of Claim 10: We prove a stronger statement: Let $\langle M_i\rangle = \left(\langle M_i\rangle_{\mathsf{ST}}, \widehat{\mathsf{st}_i}\right)$ and let $\langle P_i\rangle = \widehat{\mathsf{GP}_i}$; where $\mathsf{GP}_i = \mathsf{GP}_i[\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{SK}, P_i]$. We claim that for every $i \in [L]$, $\langle M_i\rangle_{\mathsf{ST}}$ is an obfuscation of M_i computed honestly under $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}$. We also claim that the output of $\mathsf{GarbEval}(\widehat{\mathsf{GP}_i}, \widehat{\mathsf{st}_i})$ is a secure patch of P_i and $\widehat{\mathsf{st}_i}$ be an encoding of st_i computed honestly under GTM; where st_i is generated as $(\langle P_i\rangle_{\mathsf{ST}}, \mathsf{st}_i) \leftarrow \textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{GenPatch}\left(\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{SK}, P_i, \mathsf{st}_{i-1};\ r_i\right)$ and $(\langle M_0\rangle_{\mathsf{ST}}, \mathsf{st}_0) \leftarrow \textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{Obf}(\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{SK}, M_0)$.

We prove both these claims by induction. The base case corresponds to showing that $\mathsf{Eval}(\langle M_0\rangle, x) = M_0(x)$. This corresponds to showing that $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{Eval}(\langle M_0\rangle_{\mathsf{ST}}, x) = M_0(x)$, where $\langle M_0\rangle_{\mathsf{ST}}$ is an obfuscation of M_0 under $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}$. Thus, the correctness of $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}$ implies the base case.

We move on to the induction hypothesis: Suppose the claim is true for some $k \in [L-1] \cup \{0\}$. We need to prove the claim for $k+1$. From the correctness of garbled TMs and correctness of $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}$, we have that $\widehat{\mathsf{st}_{k+1}}$ is a valid encoding of st_{k+1}, where $\widehat{\mathsf{st}_{k+1}}$ is computed as $(\langle P_{k+1}\rangle_{\mathsf{ST}}, \widehat{\mathsf{st}_{k+1}}) \leftarrow \mathsf{GarbEval}(\widehat{\mathsf{GP}_k}, \widehat{\mathsf{st}_k})$. Furthermore from the correctness of $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}$, we have that the output of $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{AppPatch}(\langle M_k\rangle_{\mathsf{ST}}, \langle P_k\rangle_{\mathsf{ST}})$ is an obfuscation of M_{k+1} with respect to $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}$.

Security. We show,

Theorem 9. *Assuming adaptive (resp., selectively) security of* GTM *and adaptive (resp., selectively)* $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}$*, the scheme* $\textit{pa-i}\mathcal{O}$ *is adaptively (resp., selectively) secure.*

Proof. We prove this is in two steps. In the first step, we simulate the garbled TM encodings. This is done by the reduction computing the secure patches (instead of delegating it to the adversary) and then executing the simulator on these secure patches. In the second step, we invoke the security of $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}$, which allows us to switch from using one branch of computation to another. The formal details are provided below.

$\underline{\mathsf{Hyb}_1}$: Let $(M_0, M_1) \in \mathcal{M}$ be the TM query made by adversary $\mathcal{A}$. Challenger picks a bit b at random and sends the obfuscation of M_b, namely $\langle M_b\rangle$, to the adversary. Then, adversary submits the adaptive patch queries $\left((P_0^1, P_1^1), \ldots, (P_0^L, P_1^L)\right)$ and in response receives the secure patches $\left(\langle P_b^1\rangle, \ldots, \langle P_b^L\rangle\right)$. The adversary outputs b'. The output of hybrid is 1 if $b' = b$.

$\underline{\mathsf{Hyb}_2}$: Let $(M_0, M_1) \in \mathcal{M}$ be the TM query made by adversary $\mathcal{A}$. The challenger executes $(\langle M_b\rangle_{\mathsf{ST}}, \mathsf{st}) \leftarrow \textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{Obf}(\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{SK}, M_b)$. It then executes $(St_{\mathsf{Sim}}, \widehat{\mathsf{st}_{\mathsf{ideal}}}) \leftarrow \mathsf{Sim}_1(1^\lambda, 1^{|\mathsf{st}|})$. It sets $\langle M_b\rangle = (\langle M_b\rangle_{\mathsf{ST}}, \widehat{\mathsf{st}_{\mathsf{ideal}}})$. It sends $\langle M_b\rangle$ to the adversary. Upon receiving the patch query (P_0^i, P_1^i) from the adversary, the challenger executes $(\widetilde{P}_b^i, \mathsf{st}_i) \leftarrow \textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{GenPatch}(\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{SK}, P_b^i, \mathsf{st}_{i-1})$. It then executes $(\widehat{\mathsf{GP}^i_{\mathsf{Sim}}}, \mathsf{st}_{\mathsf{Sim}}) \leftarrow \mathsf{Sim}_2\left(\mathsf{st}_{\mathsf{Sim}}, \widetilde{P}_b^i, 1^{|\mathsf{GP}|}\right)$. It sets $\langle P_b^i\rangle = \widehat{\mathsf{GP}^i_{\mathsf{Sim}}}$. Challenger sends $\langle P_b^i\rangle$ to the adversary. The rest of the hybrid is as before.

From the security of GTM, probability that Hyb_1 outputs 1 is negligibly close to the probability that Hyb_2 outputs 1.

$\underline{\mathsf{Hyb}_3}$: In this hybrid, the challenge bit b will not be used in the computation of either the obfuscated machine or the secure patches. In more detail, the challenger at the beginning of the game picks a bit b at random. Let $(M_0, M_1) \in \mathcal{M}$ be the TM query made by adversary $\mathcal{A}$. The challenger executes $(\langle M_1 \rangle_{\mathsf{ST}}, \mathsf{st}) \leftarrow \textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{Obf}(\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{SK}, M_1)$. The simulated encoding $\widehat{\mathsf{st}_{\mathsf{ideal}}}$ is as computed in the previous hybrid. It sends $\langle M_b \rangle = \left(\langle M_1 \rangle_{\mathsf{ST}}, \widehat{\mathsf{st}_{\mathsf{ideal}}}\right)$ to the adversary. Upon receiving the patch query (P_0^i, P_1^i) from the adversary, the challenger executes $(\widetilde{P}_1^i, \mathsf{st}_i) \leftarrow \textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{GenPatch}(\textit{pa-i}\mathcal{O}_{\mathsf{ST}}.\mathsf{SK}, P_1^i, \mathsf{st}_{i-1})$. It then computes the simulated program encoding $\widehat{\mathsf{GP}^i_{\mathsf{Sim}}}$ as before. It sends $\widehat{\mathsf{GP}^i_{\mathsf{Sim}}}$ to the adversary. The rest of the hybrid is as before.

From the security of $\textit{pa-i}\mathcal{O}_{\mathsf{ST}}$, probability that Hyb_2 outputs 1 is negligibly close to the probability that Hyb_3 outputs 1.

Furthermore, the probability that Hyb_3 outputs 1 is $1/2$ since the bit b is information-theoretically hidden from the adversary. Thus, the probability that Hyb_1 outputs 1 is negligibly close to $1/2$.

□

Extending to Parallel Patches. A similar stateful to stateless transformation can be provided for the parallel patches setting. Instead of using GTM with persistent memory, we require functional encryption to make the above transformation to work in the context of parallel patching. The ideas employed are very similar and we omit the details.

7 Stateless Single-Program *pa-i*$\mathcal{O}$ to Multi-Program *pa-i*$\mathcal{O}$

In this section, we present a general transformation from any stateless single-program *pa-i*$\mathcal{O}$ scheme to a multi-program *pa-i*$\mathcal{O}$ scheme. Our transformation relies on a compact secret-key functional encryption scheme for general circuits.

Notation. Let $\textit{pa-i}\mathcal{O}_{\mathsf{sp}} = (\mathsf{SP.Setup}, \mathsf{SP.Obf}, \mathsf{SP.GenPatch}, \mathsf{SP.AppPatch}, \mathsf{SP.Eval})$ be stateless single-program *pa-i*$\mathcal{O}$ scheme for TMs. Let $\mathsf{FE} = (\mathsf{FE.Setup}, \mathsf{FE.KeyGen}, \mathsf{FE.Enc}, \mathsf{FE.Dec})$ be a compact function-private adaptively secure secret-key FE scheme. Using these two ingredients, we will construct a multi-program *pa-i*$\mathcal{O}$ scheme $\textit{pa-i}\mathcal{O}_{\mathsf{mp}} = (\mathsf{MP.Setup}, \mathsf{MP.Obf}, \mathsf{MP.GenPatch}, \mathsf{MP.AppPatch}, \mathsf{MP.Eval})$ for TMs. Our transformation preserves the patching mode supported by the underlying single-program *pa-i*$\mathcal{O}$ scheme. That is, if $\textit{pa-i}\mathcal{O}_{\mathsf{sp}}$ is secure against sequential (resp., parallel) updates, then so will the resulting $\textit{pa-i}\mathcal{O}_{\mathsf{mp}}$ scheme.

For concreteness, we will present our transformation for the setting of sequential updates, i.e., starting from a $\textit{pa-i}\mathcal{O}_{\mathsf{sp}}$ that is secure against unbounded sequential patches, we will construct $\textit{pa-i}\mathcal{O}_{\mathsf{mp}}$ that is also secure against unbounded sequential patches. Our transformation can be adapted in a straightforward way to the case of parallel updates.

Construction. We now present our construction of $\textit{pa-i}\mathcal{O}_{\mathsf{mp}} = (\mathsf{MP.Setup}, \mathsf{MP.Obf}, \mathsf{MP.GenPatch}, \mathsf{MP.AppPatch}, \mathsf{MP.Eval})$.

$\underline{\mathsf{MP.Setup}(1^\lambda)}$: Compute $\mathsf{MSK} \leftarrow \mathsf{FE.Setup}(1^\lambda)$. Output $\mathsf{SK} = \mathsf{FE.MSK}$.

$\underline{\mathsf{MP.Obf}(\mathsf{SK}, M)}$: Perform the following sequence of steps:

- Compute $\mathsf{SK}_{\mathsf{sp}} \leftarrow \mathsf{SP.Setup}(1^\lambda)$.
- Compute $\langle M \rangle_{\mathsf{sp}} \leftarrow \mathsf{SP.Obf}(\mathsf{SK}_{\mathsf{sp}}, M)$.
- Sample a PRF Key K.
- Compute $\mathsf{FE.CT} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.MSK}, m)$ where $m = (\mathsf{SK}_{\mathsf{sp}}, K)$.

Output $\langle M \rangle = (\langle M \rangle_{\mathsf{sp}}, \mathsf{FE.CT})$.

$\underline{\mathsf{MP.GenPatch}(\mathsf{SK}, P)}$: Perform the following sequence of steps:

- Parse $\mathsf{SK} = \mathsf{FE.MSK}$.
- Compute $\mathsf{FE.SK}_{f_P} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.MSK}, f_P)$ where f_P is described in Figure 1.

f_P

Constants: Patch P.
Inputs: Obfuscation key $\mathsf{SK}_{\mathsf{sp}}$, PRF key K.

1. Compute $r \leftarrow \mathsf{PRF}_K(P)$.
2. Compute $\langle P \rangle_{\mathsf{sp}} \leftarrow \mathsf{SP.GenPatch}(\mathsf{SK}_{\mathsf{sp}}, P; r)$ using randomness r.
3. Output $\langle P \rangle_{\mathsf{sp}}$.

Figure 1: Function f_P

Output $\langle P \rangle = \mathsf{FE.SK}_{f_P}$.

$\underline{\mathsf{MP.AppPatch}(\langle M \rangle, \langle P \rangle)}$: Perform the following sequence of steps:

- Parse $\langle M \rangle = (\langle M \rangle_{\mathsf{sp}}, \mathsf{FE.CT})$ and $\langle P \rangle = \mathsf{SK}_{f_P}$.
- Compute $\langle P \rangle_{\mathsf{sp}} \leftarrow \mathsf{FE.Dec}(\mathsf{FE.CT}, \mathsf{SK}_{f_P})$.
- Compute $\langle M' \rangle_{\mathsf{sp}} \leftarrow \mathsf{SP.AppPatch}(\langle M \rangle_{\mathsf{sp}}, \langle P \rangle_{\mathsf{sp}})$.

Output $\langle M' \rangle = (\langle M' \rangle_{\mathsf{sp}}, \mathsf{FE.CT})$.

$\underline{\mathsf{MP.Eval}(\langle M \rangle, x)}$: Parse $\langle M \rangle = (\langle M \rangle_{\mathsf{sp}}, \mathsf{FE.CT})$ and output $y \leftarrow \mathsf{SP.Eval}(\langle M \rangle_{\mathsf{sp}}, x)$.

This completes the construction of *pa-i*$\mathcal{O}_{\mathsf{mp}}$.

Remark 7. *Some remarks regarding the above construction are in order:*

- *The above description implicitly assumes that each patch P issued by the authority is unique. Note that this is without loss of generality since we can modify the i^{th} patch P_i to (i, P_i).*
- *It is crucial in the above construction that the FE scheme is compact. Indeed, since we allow patch size to be arbitrary, we require that the FE scheme supports general functions of arbitrary size (with an a priori fixed input length).*
- *The FE plaintexts $m = (\mathsf{SK}_{\mathsf{sp}}, K)$ and the functions f_P in above construction are padded with sufficient zeros to match the sizes of the corresponding plaintexts and functions in the security proof.*

The correctness and efficiency properties of the above construction are easy to verify. Below, we provide a proof of security.

7.1 Proof of Security

We prove the security of our construction by a hybrid argument.

Hybrid Hyb_0: Same as real world experiment with challenge bit $b = 0$. That is, for every $j \in Q$, machine $M^j_{0,0}$ is obfuscated, and for every $i \in L$, patch P^i_0 is encoded.

Hybrid Hyb_1: Same as Hyb_0, except that:

1. For every $j \in [Q]$, the j^{th} obfuscated machine $\langle M^j_{0,0}\rangle = (\langle M^j_{0,0}\rangle_{\mathsf{sp}}, \mathsf{FE.CT}^j)$ is modified such that $\mathsf{FE.CT}^j$ is now an encryption of $m^j = (j, \mathsf{SK}^j_{\mathsf{sp}}, K^j)$.
2. for every $i \in [L]$, the i^{th} patch encoding is computed as an FE key $\mathsf{FE.SK}_{f^{i,1}_{[P^i_0,P^i_1]}}$ for function $f^{i,1}_{[P^i_0,P^i_1]}$, where $f^{i,1}_{[P^i_0,P^i_1]}$ is described in Figure 2.

$f^{i,1}_{[P^i_0,P^i_1]}$

Constants: Patches P^i_0, P^i_1.
Inputs: Index ℓ, obfuscation key $\mathsf{SK}^\ell_{\mathsf{sp}}$, PRF key K^ℓ.

(a) Compute $r \leftarrow \mathsf{PRF}_{K^\ell}(P^i_0)$.
(b) Compute $\langle P^i_0\rangle_{\mathsf{sp}} \leftarrow \mathsf{SP.GenPatch}(\mathsf{SK}^\ell_{\mathsf{sp}}, P^i_0; r)$ using randomness r.
(c) Output $\langle P^i_0\rangle_{\mathsf{sp}}$.

Figure 2: Function $f^{i,1}_{[P^i_0,P^i_1]}$

Next, for every $j \in Q$, and $t \in [5]$, we describe hybrids $\mathsf{Hyb}_{2:j:t}$.

Hybrid $\mathsf{Hyb}_{2:j:1}$: Same as the previous hybrid, except that:

- For every $i \in [L]$, the i^{th} patch encoding is computed as an FE key $\mathsf{FE.SK}_{f^{i,j:1}_{[P^i_0,P^i_1,j,\langle P^i_0\rangle^j_{\mathsf{sp}}]}}$ for function $f^{i,j:1}_{[P^i_0,P^i_1,j,\langle P^i_0\rangle^j_{\mathsf{sp}}]}$, where $f^{i,j:1}_{[P^i_0,P^i_1,j,\langle P^i_0\rangle^j_{\mathsf{sp}}]}$ is described in Figure 3. The encoding $\langle P^i_0\rangle^j_{\mathsf{sp}}$ hardwired in $f^{i,j:1}_{[P^i_0,P^i_1,j,\langle P^i_0\rangle^j_{\mathsf{sp}}]}$ is computed as $\langle P^i_0\rangle^j_{\mathsf{sp}} \leftarrow \mathsf{SP.GenPatch}(\mathsf{SK}^j_{\mathsf{sp}}, P^i_0; r)$ using PRF generated randomness $r \leftarrow \mathsf{PRF}_{K^j}(P^i_0)$.
- The j^{th} obfuscated machine $\langle M^j_{0,0}\rangle = (\langle M^j_{0,0}\rangle_{\mathsf{sp}}, \mathsf{FE.CT}^j)$ is modified such that $\mathsf{FE.CT}^j$ is now an encryption of $m^j = (j, \perp, \perp)$.

Hybrid $\mathsf{Hyb}_{2:j:2}$: Same as the previous hybrid, except that for every $i \in [L]$, the encoding $\langle P^i_0\rangle^j_{\mathsf{sp}}$ hardwired in $f^{i,j:2}_{[P^i_0,P^i_1,j,\langle P^i_0\rangle^j_{\mathsf{sp}}]}$ is computed as $\langle P^i_0\rangle^j_{\mathsf{sp}} \leftarrow \mathsf{SP.GenPatch}(\mathsf{SK}^j_{\mathsf{sp}}, P^i_0; r)$ using true randomness r.

$f^{i,j:1}_{[P_0^i, P_1^i, j, \langle P_0^i \rangle^j_{\mathsf{sp}}]}$

Constants: Patches P_0^i, P_1^i, index j, patch encoding $\langle P_0^i \rangle^j_{\mathsf{sp}}$.
Inputs: Index ℓ, obfuscation key $\mathsf{SK}^\ell_{\mathsf{sp}}$, PRF key K^ℓ.

1. If $\ell = j$, output $\langle P_0^i \rangle^j_{\mathsf{sp}}$.
2. If $\ell < j$, let $P = P_1^i$, else let $P = P_0^i$.
3. Compute $r \leftarrow \mathsf{PRF}_{K^\ell}(P)$.
4. Compute $\langle P \rangle_{\mathsf{sp}} \leftarrow \mathsf{SP.GenPatch}(\mathsf{SK}^\ell_{\mathsf{sp}}, P; r)$ using randomness r.
5. Output $\langle P \rangle_{\mathsf{sp}}$.

Figure 3: Function $f^{i,j:1}_{[P_0^i, P_1^i, j, \langle P_0^i \rangle^j_{\mathsf{sp}}]}$

Hybrid $\mathsf{Hyb}_{2:j:3}$: Same as the previous hybrid, except that:

1. For every $i \in [L]$, we now hardwire the encoding $\langle P_1^i \rangle^j_{\mathsf{sp}}$ in $f^{i,j:3}_{[P_0^i, P_1^i, j, \langle P_1^i \rangle^j_{\mathsf{sp}}]}$, where $\langle P_1^i \rangle^j_{\mathsf{sp}}$ is computed as $\langle P_1^i \rangle^j_{\mathsf{sp}} \leftarrow \mathsf{SP.GenPatch}(\mathsf{SK}^j_{\mathsf{sp}}, P_1^i; r)$ using true randomness r.
2. For every $j \in [Q]$, the j^{th} obfuscated machine is now computed as $\langle M^j_{0,1} \rangle = (\langle M^j_{0,1} \rangle_{\mathsf{sp}}, \mathsf{FE.CT}^j)$, where $\langle M^j_{0,1} \rangle_{\mathsf{sp}} \leftarrow \mathsf{SP.Obf}(\mathsf{SK}^j_{\mathsf{sp}}, M^j_{0,1})$.

Hybrid $\mathsf{Hyb}_{2:j:4}$: Same as the previous hybrid, except that for every $i \in [L]$, the encoding $\langle P_1^i \rangle^j_{\mathsf{sp}}$ hardwired in $f^{i,j:4}_{[P_0^i, P_1^i, j, \langle P_1^i \rangle^j_{\mathsf{sp}}]}$ is computed as $\langle P_1^i \rangle^j_{\mathsf{sp}} \leftarrow \mathsf{SP.GenPatch}(\mathsf{SK}^j_{\mathsf{sp}}, P_1^i; r)$ using PRF generated randomness $r \leftarrow \mathsf{PRF}_{K^j}(P_1^i)$.

Hybrid $\mathsf{Hyb}_{2:j:5}$: Same as the previous hybrid, except that the j^{th} obfuscated machine $\langle M^j_{0,1} \rangle = (\langle M^j_{0,1} \rangle_{\mathsf{sp}}, \mathsf{FE.CT}^j)$ is modified such that $\mathsf{FE.CT}^j$ is now an encryption of $m^j = (j, \mathsf{SK}^j_{\mathsf{sp}}, K^j)$.

Hybrid Hyb_3: Same as the previous hybrid, except that:

1. For every $j \in [Q]$, the j^{th} obfuscated machine $\langle M^j_{0,1} \rangle = (\langle M^j_{0,1} \rangle_{\mathsf{sp}}, \mathsf{FE.CT}^j)$ is modified such that $\mathsf{FE.CT}^j$ is now an encryption of $m^j = (\mathsf{SK}^j_{\mathsf{sp}}, K^j)$.
2. for every $i \in [L]$, the i^{th} patch encoding is computed as an FE key $\mathsf{FE.SK}_{f^{i,3}_{[P_0^i, P_1^i]}}$ for function $f^{i,3}_{[P_0^i, P_1^i]}$, where $f^{i,1}_{[P_0^i, P_1^i]}$ is described in Figure 2.

Hybrid Hyb_4: Same as real world experiment with challenge bit $b = 1$. That is, for every $j \in Q$, machine $M^j_{0,1}$ is obfuscated, and for every $i \in L$, patch P_1^i is encoded.

This completes the description of the hybrids. We now argue the following:

- $\mathsf{Hyb}_0 \approx \mathsf{Hyb}_1$
- $\mathsf{Hyb}_1 \approx \mathsf{Hyb}_{2:1:1}$

$f^{i,3}_{[P_0^i,P_1^i]}$

Constants: Patches P_0^i, P_1^i.
Inputs: Obfuscation key $\mathsf{SK}_{\mathsf{sp}}$, PRF key K.

(a) Compute $r \leftarrow \mathsf{PRF}_K(P_1^i)$.

(b) Compute $\langle P_1^i \rangle_{\mathsf{sp}} \leftarrow \mathsf{SP.GenPatch}(\mathsf{SK}_{\mathsf{sp}}, P_1^i; r)$ using randomness r.

(c) Output $\langle P_1^i \rangle_{\mathsf{sp}}$.

Figure 4: Function $f^{i,3}_{[P_0^i,P_1^i]}$

- For every $j \in [Q]$:
  - $\mathsf{Hyb}_{2:j:1} \approx \mathsf{Hyb}_{2:j:2}$
  - $\mathsf{Hyb}_{2:j:2} \approx \mathsf{Hyb}_{2:j:3}$
  - $\mathsf{Hyb}_{2:j:3} \approx \mathsf{Hyb}_{2:j:4}$
  - $\mathsf{Hyb}_{2:j:4} \approx \mathsf{Hyb}_{2:j:5}$
- $\mathsf{Hyb}_{2:j:5} \approx \mathsf{Hyb}_{2:j+1:1}$ for every $j \in [Q-1]$,
- $\mathsf{Hyb}_{2:Q:5} \approx \mathsf{Hyb}_3$
- $\mathsf{Hyb}_3 \approx \mathsf{Hyb}_4$

Combining the above, we obtain that $\mathsf{Hyb}_0 \approx \mathsf{Hyb}_4$, as desired.

Indistinguishability of Hyb_0 and Hyb_1: The only difference between Hyb_0 and Hyb_1 is how the FE ciphertexts and functions keys are generated. Specifically, for every $j \in [Q]$, $i \in [L]$: in Hyb_0, $\mathsf{FE.CT}^j$ is computed as an encryption of $m^j = (\mathsf{SK}^j_{\mathsf{sp}}, K^j)$ (prepended with sufficient zeros) and the i^{th} function key $\mathsf{FE.SK}_{f^{i,0}_{[P_0^i]}}$ is computed for the function $f^{i,0}_{[P_0^i]} = f_{[P_0^i]}$, where $f_{[P_0^i]}$ is as described in Figure 1. In Hyb_1, $m^j = (j, \mathsf{SK}^j_{\mathsf{sp}}, K^j)$ and the i^{th} function key $\mathsf{FE.SK}_{f^{i,1}_{[P_0^i,P_1^i]}}$ is computed for the function $f^{i,1}_{[P_0^i,P_1^i]}$. The indistinguishability of Hyb_0 and Hyb_1 immediately follows from the function privacy of the FE scheme.

Indistinguishability of Hyb_1 and $\mathsf{Hyb}_{2:1:1}$: The only difference between Hyb_1 and $\mathsf{Hyb}_{2:1:1}$ is how the FE ciphertexts and functions keys are generated. Specifically, for every $i \in [L]$, and $j = 1$: in Hyb_1, $\mathsf{FE.CT}^j$ is computed as an encryption of $m^j = (j, \mathsf{SK}^j_{\mathsf{sp}}, K^j)$ and the i^{th} function key $\mathsf{FE.SK}_{f^{i,1}_{[P_0^i,P_1^i]}}$ is computed for the function $f^{i,1}_{[P_0^i,P_1^i]}$. In Hybrid $\mathsf{Hyb}_{2:1:1}$, $\mathsf{FE.CT}^j$ is computed as an encryption of $m^j = (j, \perp, \perp)$ and $\mathsf{FE.SK}_{f^{i,j:1}_{[P_0^i,P_1^i,j,\langle P_0^i \rangle^j_{\mathsf{sp}}]}}$ is computed for the function $f^{i,j:1}_{[P_0^i,P_1^i,j,\langle P_0^i \rangle^j_{\mathsf{sp}}]}$. Further, the only difference between $f^{i,1}_{[P_0^i,P_1^i]}$ and $f^{i,j:1}_{[P_0^i,P_1^i,j,\langle P_0^i \rangle^j_{\mathsf{sp}}]}$ (where $j = 1$) is that on input $(j, \cdot, \cdot)$, $f^{i,1}_{[P_0^i,P_1^i]}$ computes $\langle P_0^i \rangle^j_{\mathsf{sp}}$ "on-the-fly", while $f^{i,j:1}_{[P_0^i,P_1^i,j,\langle P_0^i \rangle^j_{\mathsf{sp}}]}$ outputs the same hardwired value. Thus, they have the same input-output behavior. The indistinguishability of Hyb_0 and Hyb_1 immediately follows from the function privacy of the FE scheme.

Indistinguishability of $\mathsf{Hyb}_{2:j:1}$ and $\mathsf{Hyb}_{2:j:2}$: The only difference between $\mathsf{Hyb}_{2:j:1}$ and $\mathsf{Hyb}_{2:j:2}$ is how the patch encodings $\langle P_0^i \rangle_{\mathsf{sp}}^j$ are computed. Specifically, for every $i \in [L]$: in $\mathsf{Hyb}_{2:j:1}$, $\langle P_0^i \rangle_{\mathsf{sp}}^j$ is computed using PRF generated randomness (using key K^j) while in $\mathsf{Hyb}_{2:j:2}$, it is computed using true randomness. Further, note that the PRF key K^j is not used anywhere else in the experiments. The indistinguishability of $\mathsf{Hyb}_{2:j:1}$ and $\mathsf{Hyb}_{2:j:2}$ follows from the security of the PRF.

Indistinguishability of $\mathsf{Hyb}_{2:j:2}$ and $\mathsf{Hyb}_{2:j:3}$: The only difference between $\mathsf{Hyb}_{2:j:2}$ and $\mathsf{Hyb}_{2:j:3}$ is that in $\mathsf{Hyb}_{2:j:2}$, we use patch encodings $\langle P_0^i \rangle_{\mathsf{sp}}^j$ (for every $i \in [L]$) and obfuscation $\langle M_{0,1}^j \rangle_{\mathsf{sp}}$, while in $\mathsf{Hyb}_{2:j:3}$ we use patch encodings $\langle P_1^i \rangle_{\mathsf{sp}}^j$ (for every $i \in [L]$) and obfuscation $\langle M_{0,1}^j \rangle_{\mathsf{sp}}$. Further, note that the obfuscation key $\mathsf{SK}_{\mathsf{sp}}^j$ is not used anywhere else in the experiments. The indistinguishability of $\mathsf{Hyb}_{2:j:2}$ and $\mathsf{Hyb}_{2:j:3}$ follows from the adaptive security of $pa\text{-}i\mathcal{O}_{\mathsf{sp}}$.

Indistinguishability of $\mathsf{Hyb}_{2:j:3}$ and $\mathsf{Hyb}_{2:j:4}$: This follows in the same manner as the indistinguishability of $\mathsf{Hyb}_{2:j:1}$ and $\mathsf{Hyb}_{2:j:2}$. We omit the details.

Indistinguishability of $\mathsf{Hyb}_{2:j:4}$ and $\mathsf{Hyb}_{2:j:5}$: This follows in a straightforward way from the security of the FE scheme.

Indistinguishability of $\mathsf{Hyb}_{2:j:5}$ and $\mathsf{Hyb}_{2:j+1:1}$: This follows in a similar manner as the indistinguishability of Hyb_1 and $\mathsf{Hyb}_{2:1:1}$. We omit the details.

Indistinguishability of $\mathsf{Hyb}_{2:Q:5}$ and Hyb_3: This follows in a similar manner as the indistinguishability of Hyb_1 and $\mathsf{Hyb}_{2:1:1}$. We omit the details.

Indistinguishability of Hyb_3 and Hyb_4: This follows in a similar manner as the indistinguishability of Hyb_0 and Hyb_1. We omit the details.

8 Instantiation of Splittable iO

We explain how to instantiate splittable iO (Section 4) assuming indistinguishability obfuscation for circuits and re-randomizable encryption schemes[7] secure against sub-exponential adversaries. In order to show this, we employ the framework of Ananth et al. [AJS17] proposed in the context of obtaining succinct iO with constant multiplicative overhead. While their result is not directly applicable to our setting, their approach will be useful in the context of patching. Their framework consists of the following steps:

1. Firstly, they construct attribute based encryption (ABE) schemes where the decryption keys are associated with Turing machines. They consider the weaker security, where the adversary gets only one decryption key. An ABE scheme that satisfies this security notion is termed as 1-key ABE. They show how to construct this based on iO for circuits and one-way functions.

[7]The existence of re-randomizable encryption schemes can be based on assumptions such as decisional Diffie-Helman, learning with errors and so on.

2. Next, they adapt the notion of two-outcome ABE, on the lines of [GKP+12] in the context of Turing machines. They apply the transformation from ABE to two-outcome ABE, proposed by [GKP+12][8], in the single-key setting.

3. Then, they introduce a notion called oblivious evaluation encodings (OEE). It can be thought of as "iO-friendly" version of reusable randomized encodings for Turing machines. They show how to construct OEE starting from any 1-key two-outcome ABE scheme. This transformation additionally assumes the existence of FHE and garbled circuits.

4. Finally, they show how to transform any OEE scheme into a succinct iO scheme assuming sub-exponentially secure iO for circuits.

We extend their framework to our setting. In particular, we employ the following steps:

1. We propose a notion called *splittable* ABE. This is an ABE scheme with an additional property. We show that the ABE construction of [AJS17] already satisfies this additional property and thus is a splittable ABE scheme.

2. In the next step, we propose a notion called *splittable* 1-key two-outcome ABE. If we replace the 1-key ABE with splittable 1-key ABE in the construction of 1-key two-outcome ABE by [AJS17], we obtain a splittable two-outcome ABE scheme.

3. In the next step, we propose a notion called *splittable* OEE. This is an OEE scheme with additional properties. If we replace the 1-key ABE scheme in the OEE construction of [AJS17] with a splittable ABE scheme then we show that the resulting OEE scheme is a splittable OEE scheme.

4. Finally, we show that if we replace the OEE scheme in the succinct iO construction of [AJS17] with a splittable OEE scheme then the resulting iO scheme is a splittable iO scheme.

We first start with the definitions of splittable 1-key ABE, splittable 1-key two-outcome ABE and splittable OEE.

8.1 Definitions

Splittable 1-Key ABE for TMs. We first recall that definition of 1-key ABE from [AJS17].

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

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

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

[8]Unlike [GKP+12], the transformation of [AJS17] is not generic and uses the structure of the underlying 1-key ABE scheme.

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

- **Decryption,** $\mathsf{ABE.Dec}(\mathsf{ABE}.sk_M, \mathsf{ABE.CT}_{(x,\mathsf{msg})})$: On input the ABE key $\mathsf{ABE}.sk_M$ and encryption $\mathsf{ABE.CT}_{(x,\mathsf{msg})}$, 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 $\mathsf{ABE.Dec}(\mathsf{ABE}.sk_M, \mathsf{ABE.CT}_{(x,\mathsf{msg})})$ is (always) msg if $M(x) = 1$, where

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

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 $\mathcal{A}$.

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

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; $\left(\mathsf{ABE.PP}, \mathsf{ABE}.sk_M, \mathsf{ABE.CT}_{(x,\mathsf{msg}_b)}\right)$, where the values are computed as follows:
   - $(\mathsf{ABE.PP}, \mathsf{ABE.SK}) \leftarrow \mathsf{ABE.Setup}(1^\lambda)$,
   - $\mathsf{ABE}.sk_M \leftarrow \mathsf{ABE.KeyGen}(\mathsf{ABE.SK}, M)$
   - $\mathsf{ABE.CT}_{(x,\mathsf{msg}_b)} \leftarrow \mathsf{ABE.Enc}(\mathsf{ABE.PP}, x, \mathsf{msg}_b)$.

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 13. *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* $\mathcal{A}$*,*

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

Remark 8. *Henceforth, we will omit the term "weak-selective" when referring to the security of ABE schemes.*

We propose the definition of splittable 1-key ABE below. It is essentially a 1-key ABE scheme, with additional property that we call *Split Decryption Key* property: it says that the decryption key of a machine M is of the form (M, aux), where $|aux|$ is of size polynomial in λ. Note that any splittable 1-key ABE scheme has decryption keys with additive overhead in the size of the machine its associated with.

Definition 14 (Splittable 1-key ABE). *A 1-key attribute based encryption scheme* $\mathsf{ABE} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$ *for a class of TMs* $\mathcal{M}$ *is said to be* **splittable** *if it satisfies the following property:*

- **Split Decryption Key***: For every* $M \in \mathcal{M}$*, we have* $\mathsf{ABE}.sk_M = (M, aux)$*, where (i)* $(\mathsf{SK}, \mathsf{PP}) \leftarrow \mathsf{Setup}(1^\lambda)$*, (ii)* $sk_M \leftarrow \mathsf{KeyGen}(\mathsf{SK}, M)$*. Furthermore,* $|aux|$ *is a fixed polynomial in* λ*. In particular, it is independent of* $|M|$*.*

<u>**Splittable 1-Key Two-Outcome ABE for TMs.**</u> We first recall that definition of 1-key ABE from [AJS17]. Goldwasser et al. [GKP+13a] proposed the notion of 1-key two-outcome ABE for circuits as a variant of 1-key ABE 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 $(\mathsf{TwoABE}.sk_M, \mathsf{TwoABE}.\mathsf{CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)})$, 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 below.

A 1-key two-outcome ABE for TMs scheme, defined for a class of Turing machines $\mathcal{M}$ and message space MSG, consists of four PPT algorithms, $(\mathsf{TwoABE}.\mathsf{Setup}, \mathsf{TwoABE}.\mathsf{KeyGen}, \mathsf{TwoABE}.\mathsf{Enc}, \mathsf{TwoABE}.\mathsf{Dec})$. The syntax of the algorithms is given below.

- **Setup,** $\mathsf{TwoABE}.\mathsf{Setup}(1^\lambda)$: On input a security parameter λ in unary, it outputs a secret key $\mathsf{TwoABE}.\mathsf{SK}$ and public key $\mathsf{TwoABE}.\mathsf{PP}$.
- **Key Generation,** $\mathsf{TwoABE}.\mathsf{KeyGen}(\mathsf{TwoABE}.\mathsf{SK}, M \in \mathcal{M})$: On input a secret key $\mathsf{TwoABE}.\mathsf{SK}$ and a TM $M \in \mathcal{M}$, it outputs an ABE key $\mathsf{TwoABE}.\mathsf{SK}_M$.
- **Encryption,** $\mathsf{TwoABE}.\mathsf{Enc}(\mathsf{TwoABE}.\mathsf{PP}, x, \mathsf{msg}_0, \mathsf{msg}_1)$: On input the public key $\mathsf{TwoABE}.\mathsf{PP}$, attribute $x \in \{0,1\}^*$ and a pair of messages $(\mathsf{msg}_0 \in \mathsf{MSG}, \mathsf{msg}_1 \in \mathsf{MSG})$, it outputs the ciphertext $\mathsf{TwoABE}.\mathsf{CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)}$.
- **Decryption,** $\mathsf{TwoABE}.\mathsf{Dec}(\mathsf{TwoABE}.\mathsf{SK}_M, \mathsf{TwoABE}.\mathsf{CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)})$: On input the ABE key $\mathsf{ABE}.\mathsf{SK}_M$ and ciphertext $\mathsf{ABE}.\mathsf{CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)}$, it outputs the decrypted value out.

Correctness. The correctness property dictates that the decryption of a ciphertext of $(x, \mathsf{msg}_0, \mathsf{msg}_1)$ using an ABE key for M yields the message msg_0 if $M(x) = 0$, and msg_1 otherwise. Formally, $\mathsf{TwoABE}.\mathsf{Dec}(\mathsf{TwoABE}.\mathsf{SK}_M, \mathsf{TwoABE}.\mathsf{CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)})$ is (always) msg_0 if $M(x) = 0$ or msg_1 if $M(x) = 1$, where

- $(\mathsf{TwoABE.SK}, \mathsf{TwoABE.PP}) \leftarrow \mathsf{TwoABE.Setup}(1^\lambda)$,
- $\mathsf{TwoABE.SK}_M \leftarrow \mathsf{TwoABE.KeyGen}(\mathsf{TwoABE.SK}, M \in \mathcal{M})$ and
- $\mathsf{TwoABE.CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)} \leftarrow \mathsf{TwoABE.Enc}(\mathsf{TwoABE.PP}, x, \mathsf{msg}_0, \mathsf{msg}_1)$.

Security. Similar to 1-key ABE for TMs, we define an indistinguishability based security notion of 1-key two-outcome ABE for TMs. The security notion is formalized in the form of the following security experiment between a challenger and a PPT adversary. We denote the challenger by Ch and the adversary by $\mathcal{A}$.

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

1. $\mathcal{A}$ sends to Ch a key query M, and input comprising of the attribute x and two pairs of messages $\Big((\mathsf{msg}_{0,0}, \mathsf{msg}_{0,1}), (\mathsf{msg}_{1,0}, \mathsf{msg}_{1,1})\Big)$.
2. Ch checks if (i) $M(x) = 0$ and $\mathsf{msg}_{0,0} = \mathsf{msg}_{1,0}$ or if (ii) $M(x) = 1$ and $\mathsf{msg}_{0,1} = \mathsf{msg}_{1,1}$. If both the conditions are not satisfied then Ch aborts the experiment. Otherwise, it replies to $\mathcal{A}$ with $(\mathsf{TwoABE.PP}, \mathsf{TwoABE.SK}_M, \mathsf{TwoABE.CT}^*)$ as defined below.
   - $(\mathsf{TwoABE.PP}, \mathsf{TwoABE.SK}) \leftarrow \mathsf{TwoABE.Setup}(1^\lambda)$,
   - $(\mathsf{TwoABE.SK}_M) \leftarrow \mathsf{ABE.KeyGen}(1^\lambda, M)$
   - $\mathsf{TwoABE.CT}^* \leftarrow \mathsf{TwoABE.Enc}(\mathsf{TwoABE.PP}, x, \mathsf{msg}_{b,0}, \mathsf{msg}_{b,1})$.
3. The experiment terminates when the adversary outputs the bit b'.

We are now ready to define the security of 1-key two-outcome ABE for TMs scheme. We say that a 1-key two-outcome ABE for TMs scheme is secure if any PPT adversary can guess the challenge bit only with negligible probability.

Definition 15. *A 1-key two-outcome ABE for TMs scheme is said to be secure if there exists a negligible function* $\mathsf{negl}(\cdot)$ *s.t. for every PPT adversary* $\mathcal{A}$*:*

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

We can define the notion of splittable 1-Key two-outcome ABE for TMs scheme on the lines of splittable ABE scheme.

Definition 16 (Splittable Two-Outcome 1-key ABE)**.** *A two-outcome 1-key attribute based encryption scheme* $\mathsf{TwoABE} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$ *for a class of TMs* $\mathcal{M}$ *is said to be* **splittable** *if it satisfies the following property:*

- **Split Decryption Key***: For every* $M \in \mathcal{M}$*, we have* $\mathsf{ABE}.sk_M = (M, aux)$*, where (i)* $(\mathsf{SK}, \mathsf{PP}) \leftarrow \mathsf{Setup}(1^\lambda)$*, (ii)* $sk_M \leftarrow \mathsf{KeyGen}(\mathsf{SK}, M)$*. Furthermore,* $|aux|$ *is a fixed polynomial in* λ*. In particular, it is independent of* $|M|$*.*

Splittable Oblivious Evaluation Encodings (OEE). We begin by describing the notion of oblivious evaluation encodings (OEE) scheme as introduced by [AJS17]. Later, we define the notion of splittable OEE scheme.

This primitive was introduced as a strengthening of the notion of machine hiding encodings (MHE) introduced in [KLW15]. 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 $M(x)$.

Several changes are made 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 $x' \neq x$.

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.

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

- **Input Puncturing,** $\mathsf{puncInp}(\mathsf{sk}, x)$: It takes as input a secret key sk, input $x \in \{0,1\}^*$ and outputs a punctured key sk_x.
- **Encoding using Inp. Punctured Key,** $\mathsf{PIEncode}(\mathsf{sk}_x, x', b)$: It takes as input a punctured secret key sk_x, an input $x' \neq x$, a bit b and outputs an input encoding $\widetilde{(x', b)}$.
- **Puncturing at Choice Bit,** $\mathsf{puncBit}(\mathsf{sk}, b)$: It takes as input a secret key sk, an input bit b and outputs a key $\mathsf{OEE}.sk_b$.
- **Encoding using Bit Puncture Key,** $\mathsf{PBEncode}(\mathsf{OEE}.sk_b, x)$: It takes as input a punctured 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 $b \in \{0,1\}$,
$$\mathsf{Decode}\Big(\widetilde{(M_0, M_1)}, \widetilde{(x, b)}\Big) = M_b(x),$$
where (i) $\mathsf{sk} \leftarrow \mathsf{Setup}(1^\lambda)$, (ii) $\widetilde{(M_0, M_1)} \leftarrow \mathsf{TMEncode}(\mathsf{sk}, M_0, M_1)$ and, (iii) $\widetilde{(x, b)} \leftarrow \mathsf{InpEncode}(\mathsf{sk}, x, b)$.
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 $b \in \{0,1\}$,
$$\mathsf{Decode}\left(\widetilde{(M_0, M_1)}, \widetilde{(x', b)}\right) = M_b(x'),$$
where (i) $\mathsf{sk} \leftarrow \mathsf{Setup}\left(1^\lambda\right)$; (ii) $\widetilde{(M_0, M_1)} \leftarrow \mathsf{TMEncode}(\mathsf{sk}, M_0, M_1)$ and, (iii) $\widetilde{(x', b)} \leftarrow \mathsf{PIEncode}\left(\mathsf{puncInp}\left(\mathsf{sk}, x\right), x', b\right)$.
3. *Correctness of Bit Puncturing:* For all $M_0, M_1 \in \mathcal{M}$, $x \in \{0,1\}^*$ and $b \in \{0,1\}$,
$$\mathsf{Decode}\left(\widetilde{(M_0, M_1)}, \widetilde{(x, b)}\right) = M_b(x),$$
where (i) $\mathsf{sk} \leftarrow \mathsf{Setup}\left(1^\lambda\right)$, (ii) $\widetilde{(M_0, M_1)} \leftarrow \mathsf{TMEncode}(\mathsf{sk}, M_0, M_1)$ and, (iii) $\widetilde{(x, b)} \leftarrow \mathsf{PBEncode}(\mathsf{puncBit}\left(\mathsf{sk}, b\right), x)$.

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 $\mathsf{TMEncode}(\mathsf{sk}, M_0 \in \mathcal{M}, M_1 \in \mathcal{M})$ is a polynomial in $(\lambda, |M_0|, |M_1|)$, where $\mathsf{sk} \leftarrow \mathsf{Setup}(1^\lambda)$.
2. The running time of $\mathsf{InpEncode}(\mathsf{sk}, x \in \{0,1\}^*, b)$ is a polynomial in $(\lambda, |x|)$, where $\mathsf{sk} \leftarrow \mathsf{Setup}(1^\lambda)$.

3. The running time of $\mathsf{Decode}(\widetilde{(M_0, M_1)}, \widetilde{(x, b)})$ is a polynomial in $(\lambda, |M_0|, |M_1|, |x|, t)$, where $\mathsf{sk} \leftarrow \mathsf{Setup}(1^\lambda)$, $\widetilde{(M_0, M_1)} \leftarrow \mathsf{TMEncode}(\mathsf{sk}, M_0 \in \mathcal{M}, M_1 \in \mathcal{M})$, $\widetilde{(x, b)} \leftarrow \mathsf{InpEncode}(\mathsf{sk}, x \in \{0,1\}^*, b)$ 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) $\mathsf{sk} \leftarrow \mathsf{Setup}(1^\lambda)$, (b) machine encoding $\widetilde{(M_0, M_1)} \leftarrow \mathsf{TMEncode}(\mathsf{sk}, M_0, M_1)$, (c) input encoding $\widetilde{(x, b)} \leftarrow \mathsf{InpEncode}(\mathsf{sk}, x, b)$, and (d) punctured key $\mathsf{sk}_x \leftarrow \mathsf{puncInp}(\mathsf{sk}, x)$. Finally, it sends the following tuple to $\mathcal{A}$:
$$\left(\widetilde{(M_0, M_1)}, \widetilde{(x, b)}, \mathsf{sk}_x\right).$$

- *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 17 (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) $\mathsf{sk} \leftarrow \mathsf{Setup}(1^\lambda)$, (b) $\widetilde{(\mathsf{TM}_1, \mathsf{TM}_2)} \leftarrow \mathsf{TMEncode}(\mathsf{sk}, \mathsf{TM}_1, \mathsf{TM}_2)$, 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) $\mathsf{sk}_b \leftarrow \mathsf{puncBit}(\mathsf{sk}, c)$. Finally, it sends the following tuple to $\mathcal{A}$:
$$\left(\widetilde{(\mathsf{TM}_1, \mathsf{TM}_2)}, \mathsf{sk}_c\right).$$

- *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 18 (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,* $\mathsf{adv}_{\mathrm{OEE}_2} = \mathsf{negl}(\lambda)$.

We extend the above primitive and define the notion of splittable OEE. A splittable OEE is associated with a patchable encoding scheme and has additionally the following properties associated with it. The first property, being split machine encoding property. This property requires that both the setup and TM encode algorithms of the OEE scheme are divided into two steps. In the case of setup algorithm, the first step involves executing the setup of patchable encoding scheme twice to yield (sk_0, sk_1) and the second step involves generating auxiliary parameters aux_{stp}. In the case of Turing machine encode algorithm, the first step involves encoding the machines (M_0, M_1) using the patchable encoding scheme. The second step involves generating the auxiliary parameters as a function of these encodings. The output of TM encode algorithm are the encodings of (M_0, M_1) along with auxiliary parameters. The second property is special bit punctured key property. This property states that the bit punctured key consists of aux_{stp} and one of (sk_0, sk_1). The third property states that the correctness of OEE is maintained irrespective of whether fresh encodings or updated encodings are input to $\mathsf{AuxGen}_{\mathsf{oee}}$. The final property is termed as efficiency of aux property. This requires that the size of auxiliary parameters output by TM encode algorithm has size polynomial in λ.

Definition 19 (Splittable OEE)**.** *An oblivious evaluation encodings (OEE) scheme* $\mathsf{OEE} = (\mathsf{Setup}, \mathsf{InpEncode}, \mathsf{TMEncode}, \mathsf{Decode})$ *for a class of Turing machines* $\mathcal{M}$ *is said to be* **splittable** *if it is associated with a patchable encoding scheme* $\mathsf{UE} = (\mathsf{Gen}, \mathsf{Encode}, \mathsf{GenPatch}, \mathsf{AppPatch}, \mathsf{Decode})$ *and it satisfies the following properties:*

- **Split Machine Encoding***: Let* $M_0, M_1 \in \mathcal{M}$*.*
  - $\mathsf{Setup}(1^\lambda)$ *runs in two steps: (i) execute the setup of* UE *twice,* $sk_0 \leftarrow \mathsf{Gen}(1^\lambda)$*;* $sk_1 \leftarrow \mathsf{Gen}(1^\lambda)$*, (ii) generate auxiliary parameters* aux_{stp}*;* $aux_{\mathsf{stp}} \leftarrow \mathsf{AuxGen}_{\mathsf{stp}}(1^\lambda)$*. Output* $\mathsf{sk} = (sk_0, sk_1, aux_{\mathsf{stp}})$*.*
  - $\mathsf{TMEncode}(\mathsf{sk}, M_0, M_1)$ *runs in two steps: (i) execute the encode procedure on* M_0 *and* M_1*;* $\mathcal{E}_{sk_0}(M_0) \leftarrow \mathsf{Encode}(sk_0, M_0)$*,* $\mathcal{E}_{sk_1}(M_1) \leftarrow \mathsf{Encode}(sk_1, M_1)$*, (ii) generate auxiliary parameters* aux_{tm}*;* $aux_{\mathsf{tm}} \leftarrow \mathsf{AuxGen}_{\mathsf{oee}}(sk_0, aux_{\mathsf{stp}}, \mathcal{E}_{sk_0}(M_0), \mathcal{E}_{sk_1}(M_1))$*. Output* $(\mathcal{E}_{sk_0}(M_0), \mathcal{E}_{sk_1}(M_1), aux_{\mathsf{tm}})$*.*
- **Special Bit Punctured Key***: Let* $\mathsf{sk} = (sk_0, sk_1, aux_{\mathsf{stp}}) \leftarrow \mathsf{Setup}(1^\lambda)$*. Puncture the secret key at bit b,* $\mathsf{sk}_b \leftarrow \mathsf{puncBit}(\mathsf{sk}, b)$*. We require that* sk_b *is of the form* $(sk_b, aux_{\mathsf{stp}})$*.*
- **Correctness of** $\mathsf{AuxGen}_{\mathsf{oee}}$**:** *Let* $M \in \mathcal{M}$ *and let* $P_1, \ldots, P_L$ *be a sequence of patches. Let* M_i *be the* i^{th} *updated machine,* $M_i \leftarrow \mathsf{Update}(M_{i-1}, P)$*, for every* $i \in [L]$*, where* $M_0 = M$*.*

  Consider the following process:
  - *Let* sk_0, sk_1 *be such that* $sk_0 \leftarrow \mathsf{UE.Gen}(1^\lambda)$*,* $sk_1 \leftarrow \mathsf{UE.Gen}(1^\lambda)$*.*
  - *Let* $\mathcal{E}_{sk_0}(M) \leftarrow \mathsf{UE.Encode}(sk_0, M)$ *and* $\mathcal{E}_{sk_1}(M) \leftarrow \mathsf{UE.Encode}(sk_0, M)$*.*
  - *Consider the* i^{th} *updated encodings,* $\mathcal{E}_{sk_0}(M_i) \leftarrow \mathsf{UE.AppPatch}(\mathcal{E}_{sk_0}(M_{i-1}), \mathsf{UE.GenPatch}(sk_0, P_i))$ *and* $\mathcal{E}_{sk_1}(M_i) \leftarrow \mathsf{UE.AppPatch}\,(\mathcal{E}_{sk_1}(M_{i-1}), \mathsf{UE.GenPatch}(sk_1, P_i))$*.*
  - *Let* $aux_{\mathsf{tm}} \leftarrow \mathsf{AuxGen}_{\mathsf{oee}}(sk_0, \mathcal{E}_{sk_0}(M_L), \mathcal{E}_{sk_1}(M_L))$ *and set* $\langle M_L \rangle = (\mathcal{E}_{sk_0}(M_L), \mathcal{E}_{sk_1}(M_L), aux_{\mathsf{tm}})$*.*

  For every x, we have $\mathsf{Eval}(\langle M_L \rangle, x) = M_L(x)$*.*
- **Efficiency of** aux_{tm}*: We require that* $|aux_{\mathsf{tm}}|$ *is a fixed polynomial in security parameter, where* $(\mathcal{E}_{sk_0}(M_0), \mathcal{E}_{sk_1}(M_1), aux_{\mathsf{tm}}) \leftarrow \mathsf{TMEncode}(\mathsf{sk}, M_0, M_1)$*. In particular, it is independent of* $|M_0|$ *or* $|M_1|$*.*

We now proceed to demonstrate feasibility results for the above definitions. The preliminaries for the tools employed in these constructions are provided in Section A.

8.2 Splittable 1-key ABE for TMs

We first recall the construction of 1-key ABE from [AJS17]. We then prove that this construction is a splittable 1-key ABE scheme.

Construction of 1-Key ABE for TMs [AJS17]. We will use the following primitives in our construction:

1. A puncturable PRF family denoted by F.
2. A storage accumulator scheme based on $i\mathcal{O}$ and one-way functions that was constructed by [KLW15]. We denote it by Acc = ($\mathsf{SetupAcc}$, $\mathsf{EnforceRead}$, $\mathsf{EnforceWrite}$, $\mathsf{PrepRead}$, $\mathsf{PrepWrite}$, $\mathsf{VerifyRead}$, $\mathsf{WriteStore}$, Update). Let $\Sigma_{\rm tape}$ be the associated message space with accumulated value of size ℓ_{Acc} bits.
3. An iterators scheme denoted by 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 ABE = ($\mathsf{ABE.Setup}$, $\mathsf{ABE.KeyGen}, \mathsf{ABE.Enc}, \mathsf{ABE.Dec}$) 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.

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

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

$$\Big(\mathsf{ABE.PP} = (\mathsf{VK}_{\mathsf{tm}}, \mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP}_{\mathsf{Itr}}, v_0), \mathsf{ABE.SK} = (\mathsf{ABE.PP}, \mathsf{SK}_{\mathsf{tm}})\Big)$$

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

1. Parse the public key $\mathsf{ABE.PP}$ as $(\mathsf{VK}_{\mathsf{tm}}, \mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP}_{\mathsf{Itr}}, v_0)$.
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 $\widetilde{store}_j = \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, \widetilde{store}_{j-1}, j-1, M_j)$, $aux_j = \mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}}, \widetilde{store}_{j-1}, j-1)$, $\widetilde{w}_j = \mathsf{Update}(\mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_{j-1}, M_j, j-1, aux_j)$, 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 $\sigma_0 \leftarrow \mathsf{SignSpl}(\mathsf{SK}_{\mathsf{tm}}, \mu = (v_0, q_0, w_0, 0))$, where q_0 is the start state of M.

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

[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 $store_0$.]

$\underline{\mathsf{ABE.Enc}(\mathsf{ABE.PP}, x, \mathsf{msg})}$: On input a public key $\mathsf{ABE.PP} = (\mathsf{VK}_{\mathsf{tm}}, \mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP}_{\mathsf{Itr}}, v_0)$, 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 5, namely $N \leftarrow i\mathcal{O}(\mathsf{NxtMsg}\{U_x(\cdot), \mathsf{msg}, \mathsf{PP}_{\mathsf{Acc}}, \mathsf{PP}_{\mathsf{Itr}}, K_A\})$. 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 6. 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 $\mathsf{ABE.CT} = (N, S)$.

$\underline{\mathsf{ABE.Dec}(\mathsf{ABE}.sk_M, \mathsf{ABE.CT})}$: On input the ABE key $\mathsf{ABE}.sk_M = (M, w_0, \sigma_{\mathsf{tm}}, v_0, \widetilde{store}_0)$ and a ciphertext $\mathsf{ABE.CT} = (N, S)$, 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 $1 \le j \le \ell_{\mathsf{tm}}$, 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 concludes the construction.

```
Program NxtMsg

Constants: Turing machine $U_x = \langle Q, \Sigma_{\text{tape}}, \delta, q_0, q_{\text{acc}}, q_{\text{rej}} \rangle$, 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 $\mathsf{sym}_{\mathsf{in}} \in \Sigma_{\text{tape}}$, 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 $m_{\text{in}} = (v_{\mathsf{in}}, \mathsf{st}_{\mathsf{in}}, w_{\mathsf{in}}, \mathsf{pos}_{\mathsf{in}})$. If $\mathsf{VerSpl}(\mathsf{VK}_A, m_{\text{in}}, \sigma_{\mathsf{in}}) = 0$ 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 $\mathsf{st}_{\mathsf{out}} = q_{\text{rej}}$ output $\perp$.
    • If $\mathsf{st}_{\mathsf{out}} = q_{\text{acc}}$ output $\mathsf{msg}$.
4. Updating the accumulator and the iterator values:
    • Compute $w_{\mathsf{out}} = \mathsf{Accumulate}(\mathsf{PP}_{\mathsf{Acc}}, w_{\mathsf{in}}, \mathsf{sym}_{\mathsf{out}}, \mathsf{pos}_{\mathsf{in}}, aux)$. 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 $m_{\text{out}} = (v_{\mathsf{out}}, \mathsf{st}_{\mathsf{out}}, w_{\mathsf{out}}, \mathsf{pos}_{\mathsf{out}})$ and $\sigma_{\mathsf{out}} = \mathsf{SignSpl}(\mathsf{SK}'_A, m_{\text{out}})$.
6. Output $\mathsf{sym}_{\mathsf{out}}, \mathsf{pos}_{\mathsf{out}}, \mathsf{st}_{\mathsf{out}}, w_{\mathsf{out}}, v_{\mathsf{out}}, \sigma_{\mathsf{out}}$.
```

Figure 5: 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 $\sigma_0$.
```

Figure 6: Program SignProg

Theorem 10. *The scheme* ABE *is a 1-key ABE for TMs scheme.*

The proof of the above theorem can be found in [AJS17].

Proof that ABE is Splittable. We now claim that the above scheme is a splittable 1-key ABE scheme.

Theorem 11. ABE *is a splittable 1-key ABE for TMs scheme.*

Proof. To prove this, we just need to show that ABE satisfies the split decryption key property.

Let $M \in \mathcal{M}$. The decryption key of M is generated as $(M, w_0, \sigma_{\mathsf{tm}}, v_0, \widetilde{store}_0) \leftarrow \mathsf{ABE.KeyGen}(\mathsf{SK}_{\mathsf{tm}}, M)$. Thus, ABE satisfies the split decryption key property since $(w_0, \sigma_{\mathsf{tm}}, v_0, \widetilde{store}_0)$ can be viewed as aux and furthermore, $|aux|$ is indeed a fixed polynomial in λ (independent of $|M|$). $\square$

8.3 Splittable 1-Key Two-Outcome ABE from Splittable 1-Key ABE

We first recall the construction of 1-key two-outcome ABE scheme from [AJS17]. We then demonstrate why this scheme is splittable.

Construction of 1-Key Two-Outcome ABE from 1-Key ABE [AJS17]. The only tool we use in our construction is a 1-key ABE for TMs with additive overhead, $\mathsf{ABE} = (\mathsf{ABE.Setup}, \mathsf{ABE.KeyGen}, \mathsf{ABE.Enc}, \mathsf{ABE.Dec})$. We denote the associated class of TMs to be $\mathcal{M}$ and the associated message space to be MSG.

$\underline{\mathsf{TwoABE.Setup}(1^\lambda)}$: On input a security parameter λ in unary, execute $\mathsf{ABE.Setup}$ twice to obtain $(\mathsf{ABE.PP}_0, \mathsf{ABE.SK}_0) \leftarrow \mathsf{ABE.Setup}(1^\lambda)$ and $(\mathsf{ABE.PP}_1, \mathsf{ABE.SK}_1) \leftarrow \mathsf{ABE.Setup}(1^\lambda)$. Output $\big(\mathsf{TwoABE.PP} = (\mathsf{ABE.PP}_0, \mathsf{ABE.PP}_1), \mathsf{TwoABE.SK} = (\mathsf{ABE.SK}_0, \mathsf{ABE.SK}_1)\big)$.

$\underline{\mathsf{TwoABE.KeyGen}(\mathsf{TwoABE.SK}, M \in \mathcal{M})}$: On input a secret key $\mathsf{TwoABE.SK} = (\mathsf{ABE.SK}_0, \mathsf{ABE.SK}_1)$ and a Turing machine $M \in \mathcal{M}$, first compute two ABE keys: $\mathsf{ABE.SK}^0_M \leftarrow \mathsf{ABE.KeyGen}(\mathsf{ABE.SK}_0, M \in \mathcal{M})$ and $\mathsf{ABE.SK}^1_M \leftarrow \mathsf{ABE.KeyGen}(\mathsf{ABE.SK}_1, \overline{M})$, where $\overline{M}$ (complement of M) on input x outputs $1 - M(x)$.[9] Output the attribute key, $\mathsf{TwoABE.SK}_M = (\mathsf{ABE.SK}^0_M, \mathsf{ABE.SK}^1_M)$.

$\underline{\mathsf{TwoABE.Enc}(\mathsf{TwoABE.PP}, x, \mathsf{msg}_0, \mathsf{msg}_1)}$: On input a public key $\mathsf{TwoABE.PP} = (\mathsf{ABE.PP}_0, \mathsf{ABE.PP}_1)$, attribute $x \in \{0,1\}^*$ and messages $(\mathsf{msg}_0, \mathsf{msg}_1) \in \mathsf{MSG}^2$, compute two ciphertexts: $\mathsf{ABE.CT}_0 \leftarrow \mathsf{ABE.Enc}(\mathsf{ABE.PP}, x, \mathsf{msg}_0)$ and $\mathsf{ABE.CT}_1 \leftarrow \mathsf{ABE.Enc}(\mathsf{ABE.PP}, x, \mathsf{msg}_1)$. Output the ciphertext, $\mathsf{TwoABE.CT} = (\mathsf{ABE.CT}_0, \mathsf{ABE.CT}_1)$.

$\underline{\mathsf{TwoABE.Dec}(\mathsf{TwoABE.SK}_M, \mathsf{TwoABE.CT})}$: On input an attribute key $\mathsf{TwoABE.SK}_M = (\mathsf{TwoABE.SK}^0_M, \mathsf{TwoABE.SK}^1_M)$ and $\mathsf{TwoABE.CT} = (\mathsf{ABE.CT}_0, \mathsf{ABE.CT}_1)$, first compute $\mathsf{out}_0 \leftarrow \mathsf{ABE.Dec}(\mathsf{ABE.SK}^0_M, \mathsf{ABE.CT}_0)$ and then compute $\mathsf{out}_1 \leftarrow \mathsf{ABE.Dec}(\mathsf{ABE.SK}^1_M, \mathsf{ABE.CT}_1)$. Let out_b, for some $b \in \{0,1\}$, be such that $\mathsf{out}_b \neq \perp$. Output $\mathsf{out} = \mathsf{out}_b$.

The following theorem was proved by [AJS17].

Theorem 12. *twoabe is a 1-key two-outcome ABE scheme.*

[9]Here we are only considering Turing machines with boolean output.

Proof that TwoABE is a Splittable 1-Key Two-Outcome ABE scheme. We show that TwoABE is splittable if the scheme ABE used in the above construction is a splittable ABE scheme.

Theorem 13. *Assuming* ABE *is a splittable 1-key ABE scheme, we have* TwoABE *is a splittable 1-key two-outcome ABE scheme.*

Proof. Notice that a TwoABE key of M is a pair of ABE keys of machines M and $\overline{M}$, respectively. Denote the keys to be $\mathsf{ABE}.sk_M, \mathsf{ABE}.sk_{\overline{M}}$. Here, $\overline{M}$ is defined to be: $\overline{M}(x) = 1 - M(x)$. Since ABE is splittable, we have $\mathsf{ABE}.sk_M = (M, aux)$ and $\mathsf{ABE}.sk_{\overline{M}} = (\overline{M}, aux')$. Since, $\overline{M}$ can be derived from M, the TwoABE key of M can be represented by (M, aux, aux'). Since $|aux|$ and $|aux'|$ is a fixed polynomial in λ, TwoABE satisfies split decryption property of 1-key two-outcome ABE scheme. □

8.4 Splittable OEE from Splittable 1-Key Two-Outcome ABE

We start with the construction of OEE from [AJS17] starting from a 1-key two-outcome ABE scheme. We then show how this construction already yields a splittable OEE scheme when the underlying 1-key two-outcome ABE is a splittable scheme.

Construction of OEE from 1-Key Two-Outcome ABE [AJS17]. To construct a patchable oblivious evaluation encoding scheme, we will use the following ingredients.

1. A 1-key two-outcome PABE 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* (Section 3), 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 patchable oblivious evaluation encoding scheme to be $\mathsf{OEE} = (\mathsf{Setup}, \mathsf{InpEncode}, \mathsf{TMEncode}, \mathsf{Decode})$ that is equipped with auxiliary algorithms $(\mathsf{puncInp}, \mathsf{PIEncode}, \mathsf{puncBit}, \mathsf{PBEncode})$. The construction of OEE is presented below.

$\underline{\mathsf{Setup}(1^\lambda)}$: 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 $\mathsf{sk} = (\mathsf{TwoABE.SK}, \mathsf{TwoABE.PP}, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$.

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

- Parse 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 7.

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

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

- Let $U = U_{x,\mathsf{ind}_t}(\cdot)$ be a universal Turing machine that on input a Turing machine M, outputs $M(x)$ if the computation terminates within 2^{ind_t} number of steps, otherwise it outputs $\perp$.
- Transform the universal Turing machine U into a circuit using Theorem 7 (Section 3) 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.

Figure 7: Description of program N.

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

- Parse 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 $\mathsf{ind}_t \in [\lambda]$, compute a garbled circuit along with the wire keys, $\left(\mathcal{GC}_{\mathsf{ind}_t}, \{w_{i,0}^{\mathsf{ind}_t}, w_{i,1}^{\mathsf{ind}_t}\}_{i\in[q]}\right) \leftarrow \mathsf{Garble}(1^\lambda, G)$, 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 $\mathsf{ind}_t \in [\lambda]$, compute a TwoABE ciphertext $\mathsf{TwoABE.CT}_{i,\mathsf{ind}_t} \leftarrow \mathsf{TwoABE.Enc}\Big(\mathsf{TwoABE.PP}, (x, i, \mathsf{ind}_t), w_{i,0}^{\mathsf{ind}_t}, w_{i,1}^{\mathsf{ind}_t}\Big)$ of the message pair $(w_{i,0}^{\mathsf{ind}_t}, w_{i,1}^{\mathsf{ind}_t})$ along with attribute (x, i, ind_t).

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

$\underline{\mathsf{Decode}(\widetilde{(M_0, M_1)}, \widetilde{(x, b)})}$: 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 $\widetilde{(x, b)} = \big(\mathsf{TwoABE.PP}, \{\mathcal{GC}\}_{\mathsf{ind}_t\in[\lambda]}, \{\mathsf{TwoABE.CT}_{i,\mathsf{ind}_t}\}_{i\in[q],\mathsf{ind}_t\in[\lambda]}\big)$.

- For every $\mathsf{ind}_t \in [\lambda]$, do the following:

  1. For every $i \in [q]$, execute the decryption procedure of TwoABE to obtain the wire keys of the garbled circuit, $\widetilde{w}_i^{\mathsf{ind}_t} \leftarrow \mathsf{TwoABE.Dec}(\mathsf{TwoABE.SK}_N, \mathsf{TwoABE.CT}_{i,\mathsf{ind}_t})$.

2. Execute $\mathsf{EvalGC}(\mathcal{GC}_{\mathsf{ind}_t}, \widetilde{w}_1^{\mathsf{ind}_t}, \ldots, \widetilde{w}_q^{\mathsf{ind}_t})$ to obtain $\mathsf{out}_{\mathsf{ind}_t}$.
3. If $\mathsf{out}_{\mathsf{ind}_t} \neq \perp$ then output $\mathsf{out} = \mathsf{out}_{\mathsf{ind}_t}$. Otherwise, continue.

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

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

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

[Note: The algorithm InpEncode *can directly be executed on the punctured key* sk_x *and input* x' *because the master secret key* TwoABE.SK *is never used during its execution.]*

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

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

[Note: The algorithm InpEncode *can directly be executed on the punctured key* 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.]*

Theorem 14. OEE *is a oblivious evaluation encodings scheme.*

The proof of the above theorem can be found in [AJS17].

<u>Proof that OEE is Splittable OEE.</u> We now claim that the above scheme is a splittable OEE scheme.

Theorem 15. OEE *is a splittable OEE scheme assuming that* TwoABE *is a splittable two-outcome ABE scheme.*

Proof. We need to show that OEE satisfies the following properties:

(1) Split Machine Encoding: We first remark about setup ofOEE, Setup. Note that we generate the FHE setup twice to get two pairs of public key-secret key pairs $(\mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0)$ and $(\mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$. This can be viewed as executing the setup of UE twice since a fully homomorphic encryption scheme is an instantiation of UE (refer to Section 4). More specifically, we can treat the UE secret keys as $sk_0 = \mathsf{FHE.sk}_0$ and $sk_1 = \mathsf{FHE.sk}_1$. The computation of the ABE secret key-public key pair can be treated as computation of aux_{stp}. In addition, we include the FHE public keys as part of aux_{stp}. That is, $\mathsf{AuxGen}_{\mathsf{stp}}$ outputs TwoABE.Setup and $(\mathsf{FHE.pk}_0, \mathsf{FHE.pk}_1)$. Thus, Setup satisfies the requirements of the split machine encoding property.

We now move on to $\mathsf{TMEncode}$. Consider the execution of $\mathsf{TMEncode}$ on input (sk, M_0, M_1). Observe that the first step involves FHE encrypting M_0 using $\mathsf{FHE.pk}_0$ and FHE encrypting M_1 using $\mathsf{FHE.pk}_1$. Denote the output of this step by $(\mathsf{FHE.CT}_0, \mathsf{FHE.CT}_1)$. The second step is the generation of a TwoABE decryption key on a program N. Finally, the output of $\mathsf{TMEncode}$ is the decryption key on N. Observe that N can be written as $\mathsf{FHE.CT}_0||\mathsf{FHE.CT}_1||\alpha$, where α is of size polynomial in λ. Furthermore, from the decomposability property of TwoABE, we have the decryption key to be of the form (N, aux_{abe}). Thus, the output of $\mathsf{TMEncode}$ is of the form $(\mathsf{FHE.CT}_0, \mathsf{FHE.CT}_1, aux_{\mathsf{tm}})$. This fact combined with our earlier observation that FHE is an instantiation of UE, it follows that $\mathsf{TMEncode}$ satisfies the requirements of the split machine encoding property.

(2) Special Bit Punctured Key: Observe that sk_b is of the form $(\mathsf{TwoABE.PP}, \mathsf{FHE.pk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_b)$. From the above discussion, this is of the form $\mathsf{sk}_b = (aux_{\mathsf{stp}}, sk_b)$.

(3) Correctness of $\mathsf{AuxGen}_{\mathsf{oee}}$: This property follows along the same lines as the correctness of OEE. Intuitively, the main reason boils down to showing that the correctness is maintained irrespective of whether evaluated FHE ciphertexts or fresh FHE ciphertexts are used.

(4) Efficiency of aux_{tm}: Observe that $aux_{\mathsf{tm}} = (\alpha, aux_{\mathsf{abe}})$. As described above, α is a polynomial in λ. From the efficiency requirement associated with the split decryption property of TwoABE, we have that aux_{abe} has size polynomial in λ. Thus, aux_{tm} has size polynomial in λ.

$\square$

8.5 Splittable iO from Splittable OEE

We recall (a modified version of) the construction of iO from OEE in [AJS17]. We then demonstrate that if the underlying OEE scheme is splittable then the resulting scheme is a splittable iO scheme.

Modified Construction of iO from OEE [AJS17]. Let $\mathsf{OEE} = (\mathsf{Setup}, \mathsf{InpEncode}, \mathsf{TMEncode}, \mathsf{Decode})$ be an OEE scheme with constant multiplicative overhead that is equipped with auxiliary algorithms $(\mathsf{puncInp}, \mathsf{PIEncode}, \mathsf{puncBit}, \mathsf{PBEncode})$. Let $i\mathcal{O}$ 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.

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:

- $\mathsf{sk} \leftarrow \mathsf{Setup}(1^\lambda)$.
- Compute the punctured key, $\mathsf{sk}_0 \leftarrow \mathsf{puncBit}(\mathsf{sk}, 0)$.
  Remark: In the construction of AJS, the secret key is not punctured.
- $\widetilde{(M, M)} \leftarrow \mathsf{TMEncode}(\mathsf{sk}, M, M)$.
- $\widetilde{C} \leftarrow i\mathcal{O}\left(C_{[K,\mathsf{sk}_0]}\right)$, where K is a randomly chosen key for the puncturable PRF family and $C_{[K,\mathsf{sk}_0]}$ is the circuit described in Figure 8.

$C_{[K,\mathsf{sk}_0]}(x)$

1. Compute $r \leftarrow \mathsf{PRF}_K(x)$.
2. Compute $\widetilde{(x,0)} \leftarrow \mathsf{PBEncode}(\mathsf{sk}_0, x)$ using randomness r.
   Remark: In the construction of AJS,
3. Output $\widetilde{(x,0)}$.

Figure 8: Circuit $C_{[K,\mathsf{sk}_0]}$.

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 $y \leftarrow \mathsf{Decode}\left(\widetilde{(M,M)}, \widetilde{(x,0)}\right)$ and outputs y.

Theorem 16. *The above scheme* SuccIO *is an indistinguishability obfuscation scheme.*

While the construction of [AJS17] does not involve puncturing the OEE secret key, their security proof however can be adopted to prove the above theorem.

Proof that SuccIO is Splittable iO. We now claim that the above scheme is a splittable iO scheme.

Theorem 17. SuccIO *is a splittable iO scheme, assuming that* OEE *is a splittable OEE scheme.*

Proof. We prove the properties below.

(1) Splittable Property: Consider a machine $M \in \mathcal{M}$.

- The first step of $\mathsf{SuccIO}(1^\lambda, M)$ involves executing the setup of OEE. From the splittable machine encoding property, $\mathsf{Setup}(1^\lambda)$ runs in two steps: (i) execute the setup of UE twice, $sk_0 \leftarrow \mathsf{Gen}(1^\lambda)$; $sk_1 \leftarrow \mathsf{Gen}(1^\lambda)$, (ii) generate auxiliary parameters aux_{stp}; $aux_{\mathsf{stp}} \leftarrow \mathsf{AuxGen}_{\mathsf{stp}}(1^\lambda)$. The secret key is set to be $\mathsf{sk} = (sk_0, sk_1, aux_{\mathsf{stp}})$.
- From the special bit punctured key property, the second step just sets the punctured key sk_0 to be $(sk_0, aux_{\mathsf{stp}})$.
- From the splittable machine encoding property, we have the following. $\mathsf{TMEncode}(\mathsf{sk}, M_0, M_1)$ runs in two steps: (i) execute the encode procedure on M_0 and M_1; $\mathcal{E}_{sk_0}(M_0) \leftarrow \mathsf{Encode}(sk_0, M_0)$, $\mathcal{E}_{sk_1}(M_1) \leftarrow \mathsf{Encode}(sk_1, M_1)$, (ii) generate auxiliary parameters aux_{tm}; $aux_{\mathsf{tm}} \leftarrow \mathsf{AuxGen}_{\mathsf{oee}}(sk_0, aux_{\mathsf{stp}}, \mathcal{E}_{sk_0}(M_0), \mathcal{E}_{sk_1}(M_1))$. Output $(\mathcal{E}_{sk_0}(M_0), \mathcal{E}_{sk_1}(M_1), aux_{\mathsf{tm}})$.

The above can be re-written as follows. On input $(1^\lambda, M)$, SuccIO computes:

1. Execute the setup of UE twice, $sk_0 \leftarrow \mathsf{Gen}(1^\lambda)$; $sk_1 \leftarrow \mathsf{Gen}(1^\lambda)$
2. Execute the encode procedure on M_0 and M_1; $\mathcal{E}_{sk_0}(M_0) \leftarrow \mathsf{Encode}(sk_0, M_0)$, $\mathcal{E}_{sk_1}(M_1) \leftarrow \mathsf{Encode}(sk_1, M_1)$
3. Generate auxiliary parameters by executing $\mathsf{AuxGen}(sk_0, \mathcal{E}_{sk_0}(M_0), \mathcal{E}_{sk_1}(M_1))$ as defined below:

- Generate auxiliary parameters aux_{stp}; $aux_{\mathsf{stp}} \leftarrow \mathsf{AuxGen}_{\mathsf{stp}}(1^\lambda)$.
- Generate auxiliary parameters aux_{tm}; $aux_{\mathsf{tm}} \leftarrow \mathsf{AuxGen}_{\mathsf{oee}}(sk_0, aux_{\mathsf{stp}}, \mathcal{E}_{sk_0}(M_0), \mathcal{E}_{sk_1}(M_1))$
- Set $aux = aux_{\mathsf{tm}}$.

4. Output the obfuscated machine $\langle M\rangle = (\mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M), aux)$.

Note that the execution of $\mathsf{AuxGen}_{\mathsf{stp}}$ and Encode can be switched because the execution of Encode does not depend on the output of $\mathsf{AuxGen}_{\mathsf{stp}}$.

From the above, we have that SuccIO satisfies splittable property.

(2) Correctness of AuxGen: This follows directly from the correctness of AuxGen.

(3) Efficiency of aux: We have $|aux_{\mathsf{stp}}| = \text{poly}(\lambda)$ since AuxGen just takes the security parameter as input. From the efficiency of OEE, we have $|aux_{\mathsf{tm}}| = \text{poly}(\lambda)$. This proves that SuccIO satisfies efficiency of aux property.

(4) Indistinguishabiliy of aux: We prove the following lemma that shows that SuccIO satisfies indistinguishability of aux property.

Lemma 1. *Consider $M_0, M_1 \in \mathcal{M}$ such that $M_0(x) = M_1(x)$ for every $x \in \{0,1\}^*$. Suppose E_0, E_1, sk_0, sk_1 are such that $M_0 \leftarrow \mathsf{Decode}(sk_0, E_0)$ and $M_1 \leftarrow \mathsf{Decode}(sk_1, E_1)$. Assuming sub-exponential security of OEE, we have:*

$$\{E_0, E_1, sk_0, sk_1, aux_0\} \approx_c \{E_0, E_1, sk_0, sk_1, aux_1\},$$

where $aux_b \leftarrow \mathsf{AuxGen}(sk_b, E_0, E_1)$ for $b \in \{0,1\}$.

Proof. Consider the following hybrids.

Hybrid H_1: Real world experiment where machine M_0 is obfuscated. The adversary is given the obfuscated program $\left(\widetilde{(M_0, M_0)}, \widetilde{C}\right)$.

Hybrid H_2: Same as H_1, except that we replace the machine encoding $\widetilde{(M_0, M_0)}$ with $\widetilde{(M_0, M_1)}$.

The indistinguishability of H_1 and H_2 follows from the indistinguishability of machine encoding property of the OEE scheme.

Hybrid H_3: Same as H_2, except that $\widetilde{C}$ is now computed as $\widetilde{C} \leftarrow i\mathcal{O}\left(C^3_{[K,\mathsf{sk}_1]}\right)$, where $\mathsf{sk}_1 \leftarrow \mathsf{puncBit}(\mathsf{sk}, 1)$ and $C^3_{[K,\mathsf{sk}_1]}$ is the circuit described in Figure 9.

$C^3_{[K,\mathsf{sk}_1]}(x)$

1. Compute $r \leftarrow \mathsf{PRF}_K(x)$.
2. Compute $\widetilde{(x,1)} \leftarrow \mathsf{PBEncode}(\mathsf{sk}_1, x)$ using randomness r.
3. Output $\widetilde{(x,1)}$.

Figure 9: Circuit $C^3_{[K,\mathsf{sk}_1]}$.

Indistinguishability of H_2 and H_3. Let $x_1, \ldots, x_N$ denote the N inputs to machines M_0 and M_1, sorted in lexicographic order. To argue ϵ'-indistinguishability of H_2 and H_3, we will consider $N+1$ internal hybrids $H_{2:0}, \ldots, H_{2:N}$. Below, we describe the hybrids $H_{2:i}$, starting with $i = 0$ and then $0 < i \leq N$.

Hybrid $H_{2:0}$: Same as H_2, except that $\widetilde{C}$ is now computed as $\widetilde{C} \leftarrow i\mathcal{O}\left(C^{2:0}_{[K,\mathsf{sk}]}\right)$, where $C^{2:i}_{[K,\mathsf{sk}]}$ is the same as circuit $C_{[K,\mathsf{sk}]}$ described in Figure 8.

Hybrid $H_{2:i}$: Same as $H_{2:i-1}$, except that $\widetilde{C}$ is now computed as $\widetilde{C} \leftarrow i\mathcal{O}\left(C^{2:i}_{[K,\mathsf{sk}]}\right)$, where $C^{2:i}_{[K,\mathsf{sk}]}$ is the circuit described in Figure 10.

$C^{2:i}_{[K,\mathsf{sk}]}(x)$

1. If $x \leq x_i$, then $b = 1$, else $b = 0$.
2. Compute $r \leftarrow \mathsf{PRF}_K(x)$.
3. Compute $\widetilde{(x,b)} \leftarrow \mathsf{InpEncode}(\mathsf{sk}, x, b)$ using randomness r.
4. Output $\widetilde{(x,b)}$.

Figure 10: Circuit $C^{2:i}_{[K,\mathsf{sk}]}$.

For every $0 \leq i \leq L$, we will argue the indistinguishability of $H_{2:i}$ and $H_{2:i+1}$. To facilitate this, we consider another sequence of intermediate hybrids $H_{2:i:1}, \ldots, H_{2:i:4}$, where $0 \leq i < L$. We describe them below.

Hybrid $H_{2:i:1}$: Same as $H_{2:i}$, except that $\widetilde{C}$ is now computed as $\widetilde{C} \leftarrow i\mathcal{O}\left(C^{2:i:1}_{\left[K_{x_{i+1}}, \mathsf{sk}_{x_{i+1}}, \widetilde{(x_{i+1},0)}\right]}\right)$, where:

- $K_{x_{i+1}} \leftarrow \mathsf{PRFPunc}(K, x_{i+1})$.
- $\mathsf{sk}_{x_{i+1}} \leftarrow \mathsf{puncInp}(\mathsf{sk}, x_{i+1})$.
- $\widetilde{(x_{i+1},0)} \leftarrow \mathsf{InpEncode}(\mathsf{sk}, x_{i+1}, 0)$ using randomness $r \leftarrow \mathsf{PRF}(K, x_{i+1})$.
- Circuit $C^{2:i:1}_{\left[K_{x_{i+1}}, \mathsf{sk}_{x_{i+1}}, \widetilde{(x_{i+1},0)}\right]}$ contains the values $K_{x_{i+1}}$, $\mathsf{sk}_{x_{i+1}}$ and $\widetilde{(x_{i+1},0)}$ hardwired, and is described in Figure 11.

Hybrid $H_{2:i:2}$: Same as $H_{2:i:1}$, except that the hardwired value $\widetilde{(x_{i+1},0)} \leftarrow \mathsf{InpEncode}(\mathsf{sk}, x, 0)$ is now computed using true randomness (as opposed to PRF generated randomness).

Hybrid $H_{2:i:3}$: Same as $H_{2:i:2}$, except that we now replace the hardwired value $\widetilde{(x_{i+1},0)}$ with $\widetilde{(x_{i+1},1)}$, where $\widetilde{(x_{i+1},1)} \leftarrow \mathsf{InpEncode}(\mathsf{sk}, x, 1)$ is computed using true randomness.

Hybrid $H_{2:i:4}$: Same as $H_{2:i:3}$, except that the hardwired value $\widetilde{(x_{i+1},1)} \leftarrow \mathsf{InpEncode}(\mathsf{sk}, x, 1)$ is now computed using randomness $r \leftarrow \mathsf{PRF}_K(x_{i+1}\|1)$.

This completes the description of the intermediate hybrids. We now make the following indistinguishability claims:

$C^{2:i:1}_{\left[K_{x_{i+1}}, \mathsf{sk}_{x_{i+1}}, \widetilde{(x_{i+1},0)}\right]}(x)$

```
1. If $x = x_{i+1}$, output $\widetilde{(x_{i+1}, 0)}$.
2. If $x \le x_i$, then $b = 1$, else $b = 0$.
3. Compute $r \leftarrow \mathsf{PRF}_{K_{x_{i+1}}}(x)$.
4. Compute $\widetilde{(x, b)} \leftarrow \mathsf{InpEncode}(\mathsf{sk}_{x_{i+1}}, x, b)$ using randomness $r$.
5. Output $\widetilde{(x, b)}$.
```

Figure 11: Circuit $C^{2:i:1}_{[K,\mathsf{sk}]}$.

- $H_2 \approx H_{2:0}$.
- For every $0 \le i < N$:
  - $H_{2:i} \approx H_{2:i:1}$.
  - $H_{2:i:1} \approx H_{2:i:2}$.
  - $H_{2:i:2} \approx H_{2:i:3}$.
  - $H_{2:i:3} \approx H_{2:i:4}$.
  - $H_{2:i:4} \approx H_{2:i+1}$.
- $H_{2:N} \approx H_2$.

Finally, we will combine all these claims to argue the indistinguishability of H_2 and H_3.

Indistinguishability of H_2 and $H_{2:0}$. Let $C^{2:0}_{[K,\mathsf{sk}]}$ denote the circuit used in hybrid $H_{2:0}$. We will show that the circuits $C^{2:0}_{[K,\mathsf{sk}]}$ and $C^{2}_{[K,\mathsf{sk}_0]}$ are functionally equivalent. The indistinguishability of $H_{2:0}$ and H_2 then follows from the security of the indistinguishability obfuscator $i\mathcal{O}$.

Circuit $C^{2:0}_{[K,\mathsf{sk}]}$ on input x computes $\mathsf{InpEncode}(\mathsf{sk}, x, 0)$ using randomness $r \leftarrow \mathsf{PRF}_K(x)$ while $C^{2}_{[K,\mathsf{sk}_0]}$ computes $\mathsf{PBEncode}(\mathsf{sk}_0, x)$ using randomness r. From the correctness of bit puncturing property of the OEE scheme, we have that $\mathsf{InpEncode}(\mathsf{sk}, x, 0) = \mathsf{PBEncode}(\mathsf{sk}_0, x)$. Thus, $C^{2:0}_{[K,\mathsf{sk}]}$ and $C^{2}_{[K,\mathsf{sk}_0]}$ are functionally equivalent.

Indistinguishability of $H_{2:i}$ and $H_{2:i:1}$. We show that the two circuits $C^{2:i}_{[K,\mathsf{sk}]}$ and $C^{2:i:1}_{\left[K_{x_{i+1}}, \mathsf{sk}_{x_{i+1}}, \widetilde{(x_{i+1},0)}\right]}$ are functionally equivalent. The indistinguishability of $H_{2:i}$ and $H_{2:i:1}$ then follows from the security of the indistinguishability obfuscator $i\mathcal{O}$.

First observe that since the punctured PRF preserves functionality under puncturing and the OEE scheme satisfies correctness of input puncturing property, it follows that the behavior of circuits $C^{2:i}_{[K,\mathsf{sk}]}$ and $C^{2:i:1}_{\left[K_{x_{i+1}}, \mathsf{sk}_{x_{i+1}}, \widetilde{(x_{i+1},0)}\right]}$ is identical on all inputs $x \ne x_{i+1}$. On input x_{i+1}, circuit $C^{2:i}_{[K,\mathsf{sk}]}$ outputs $\mathsf{InpEncode}(\mathsf{sk}, x_{i+1}, 0)$ that is computed using randomness $r \leftarrow \mathsf{PRF}_K(x_{i+1})$, while circuit $C^{2:i:1}_{\left[K_{x_{i+1}}, \mathsf{sk}_{x_{i+1}}, \widetilde{(x_{i+1},0)}\right]}$ outputs the hardwired value $\widetilde{(x_{i+1}, 0)}$. However, it follows from the description of $H_{2:i:1}$ that $\widetilde{(x_{i+1}, 0)} = \mathsf{InpEncode}(\mathsf{sk}, x_{i+1}, 0)$ (where randomness r as described above is used). Then, combining the above, we have that $C^{2:i}_{[K,\mathsf{sk}]}$ and $C^{2:i:1}_{\left[K_{x_{i+1}},\mathsf{sk}_{x_{i+1}},\widetilde{(x_{i+1},0)}\right]}$ are functionally equivalent.

Indistinguishability of $H_{2:i:1}$ and $H_{2:i:2}$. This follows immediately from the security of the punctured PRF family used in the construction.

Indistinguishability of $H_{2:i:2}$ and $H_{2:i:3}$. Note that in both experiments $H_{2:i:2}$ and $H_{2:i:3}$, only the punctured key $\mathsf{sk}_{x_{i+1}}$ is used. Then, the indistinguishability of $H_{2:i:2}$ and $H_{2:i:3}$ follows from the indistinguishability of encoding bit property of the OEE scheme.

Indistinguishability of $H_{2:i:3}$ and $H_{2:i:4}$. This follows immediately from the security of the punctured PRF family used in the construction.

Indistinguishability of $H_{2:i:4}$ and $H_{2:i+1}$. This follows in the same manner as the proof of the indistinguishability of hybrids $H_{2:i}$ and $H_{2:i:1}$. We omit the details.

Indistinguishability of $H_{2:L}$ and H_3. This follows in the same manner as the proof of the indistinguishability of hybrids H_2 and $H_{2:0}$. We omit the details.

□

□

By instantiating the tools used in [AJS17], we have the following theorem.

Theorem 18. *Assuming the existence of sub-exponentially secure iO for circuits and sub-exponentially secure re-randomizable encryption schemes, there exists a splittable iO scheme for Turing machines (with bounded input length).*

Since we can instantiate re-randomizable encryption schemes from standard assumptions such as DDH, LWE, we have the following corollary.

Corollary 1. *Assuming the existence of sub-exponentially secure iO for circuits and sub-exponentially secure DDH, there exists a splittable iO scheme for Turing machines (with bounded input length).*

9 Implications of *pa-i$\mathcal{O}$*

In this section, we first show how to use multi-program *pa-i$\mathcal{O}$* for parallel updates to construct a secret-key functional encryption (FE) scheme for unbounded-input Turing machines. We then extend our construction in a simple manner to obtain a secret-key multi-input functional encryption (MIFE) scheme for functions of unbounded arity. Starting from adaptively secure multi-program *pa-i$\mathcal{O}$*, both of our constructions achieve indistinguishability security against pre-ciphertext key queries, namely, where the adversary first submits all the function key queries and then issues ciphertext queries in an adaptive fashion. We refer the reader to [AS16] and [BGJS15] for formal definitions of FE for unbounded-input Turing machines and MIFE for functions of unbounded arity, respectively.

9.1 FE for Unbounded-Input Turing Machines

Let $\mathcal{M}$ be any family of Turing machines that supports arbitrary length inputs. We describe a secret-key FE scheme $\mathsf{FE} = (\mathsf{FE.Setup}, \mathsf{FE.KeyGen}, \mathsf{FE.Enc}, \mathsf{FE.Dec})$ for $\mathcal{M}$ that achieves indistinguishability security against pre-ciphertext key queries. The only ingredient in our construction is an adaptively secure multi-program patchable indistinguishability obfuscation scheme $pa\text{-}i\mathcal{O}_{\mathsf{mp}} = (\mathsf{Setup}, \mathsf{Obf}, \mathsf{GenPatch}, \mathsf{AppPatch}, \mathsf{Eval})$ for a Turing machine family $\mathcal{M}_{\mathsf{mp}}$ that supports an unbounded number of parallel updates from an associated patch family $\mathcal{P}_{\mathsf{mp}}$. We denote that associated update algorithm by $\mathsf{Update}_{\mathsf{mp}}$.

- A Turing machine $\mathsf{TM} \in \mathcal{M}_{\mathsf{mp}}$ is of the form $\mathsf{TM} = \mathsf{TM}_{[M,x]}$ where $M \in \mathcal{M}$ and $x \in \{0,1\}^* \cup \emptyset$. On *any* input y, TM outputs $\perp$ if $x = \emptyset$. Otherwise, it computes and outputs $M(x)$.
- A patch $P \in \mathcal{P}_{\mathsf{mp}}$ is of the form $P = P_{[x']}$ where $x' \in \{0,1\}^*$.
- The update algorithm $\mathsf{Update}_{\mathsf{mp}}$ on input $(\mathsf{TM}_{[M,x]}, P_{[x']})$ outputs $\mathsf{TM}' = \mathsf{TM}_{[M,x']}$.

We now proceed to describe FE.

- $\mathsf{FE.Setup}(1^\lambda)$: On input the security parameter λ in unary, compute $\mathsf{SK} \leftarrow \mathsf{Setup}(1^\lambda)$. Output $\mathsf{FE.MSK} = \mathsf{SK}$.
- $\mathsf{FE.KeyGen}(\mathsf{FE.MSK}, M)$: On input $\mathsf{FE.MSK} = \mathsf{SK}$ and a Turing machine $M \in \mathcal{M}$, compute $\langle \mathsf{TM}_M \rangle \leftarrow \mathsf{Obf}(\mathsf{SK}, \mathsf{TM}_{[M,\emptyset]})$ where $\mathsf{TM}_{[M,\emptyset]} \in \mathcal{M}_{\mathsf{mp}}$. Output $\mathsf{FE.SK}_M = \langle \mathsf{TM}_M \rangle$.
- $\mathsf{FE.Enc}(\mathsf{FE.MSK}, x)$: On input $\mathsf{FE.MSK} = \mathsf{SK}$ and a message $x \in \{0,1\}^*$, compute $\langle P_x \rangle \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P_{[x]})$ where $P_{[x]} \in \mathcal{P}_{\mathsf{mp}}$. Output $\mathsf{CT} = \langle P_x \rangle$.
- $\mathsf{FE.Dec}(\mathsf{FE.SK}_M, \mathsf{CT})$: On input a functional key $\mathsf{FE.SK}_M = \langle \mathsf{TM}_M \rangle$ and a ciphertext $\mathsf{CT} = \langle P_x \rangle$, compute $\langle \mathsf{TM}_{M'} \rangle \leftarrow \mathsf{AppPatch}\Big(\langle \mathsf{TM}_M \rangle, \langle P_x \rangle\Big)$. Output $\mathsf{Eval}\Big(\langle \mathsf{TM}_{M'} \rangle, 0\Big)$.

Correctness. Let $\mathsf{FE.SK}_M = \langle \mathsf{TM}_M \rangle$ be a functional key for Turing machine $M \in \mathcal{M}$ where $\langle \mathsf{TM}_M \rangle = \mathsf{Obf}(\mathsf{SK}, \mathsf{TM}_{[M,\emptyset]})$ for $\mathsf{TM}_{[M,\emptyset]} \in \mathcal{M}_{\mathsf{mp}}$. Let $\mathsf{CT} = \langle P_x \rangle$ be a ciphertext where $\langle P_x \rangle \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P_{[x]})$ for $P_{[x]} \in \mathcal{P}_{\mathsf{mp}}$. Now, from the correctness properties of $pa\text{-}i\mathcal{O}_{\mathsf{mp}}$, it follows that $\mathsf{AppPatch}\Big(\langle \mathsf{TM}_M \rangle, \langle P_x \rangle\Big) = \langle \mathsf{TM}_{M'} \rangle$ s.t. $\langle \mathsf{TM}_{M'} \rangle$ is functionally equivalent to $\mathsf{Update}_{\mathsf{mp}}(\mathsf{TM}_{[M,\emptyset]}, P_{[x]}) = \mathsf{TM}_{[M,x]}$. From the definition of $\mathsf{TM}_{[M,x]}$, we have that $\mathsf{TM}_{[M,x]}(0) = M(x)$, as required.

Security. We briefly sketch the proof of security here. Let $M_1, \ldots, M_k$ be the function key queries and let $(x_0^1, x_1^1), \ldots, (x_0^n, x_1^n)$ be the (adaptive) ciphertext queries made by the adversary for polynomials $k = \text{poly}(\lambda)$ and $n = \text{poly}(\lambda)$ in the pre-ciphertext key query security game for secret-key FE. For every $i \in [k]$, let $\mathsf{FE.SK}_{M_i} = \mathsf{Obf}(\mathsf{SK}, \mathsf{TM}_{[M_i,\emptyset]})$ where $\mathsf{TM}_{[M_i,\emptyset]} \in \mathcal{M}_{\mathsf{mp}}$. Further, for every $j \in [n]$, let $\mathsf{CT}_j = \mathsf{GenPatch}(\mathsf{SK}, P_{[x_b^j]})$, where $P_{[x_b^j]} \in \mathcal{P}_{\mathsf{mp}}$ and b is the challenge bit chosen by the adversary.

Now, from the requirement in the security definition of FE, we have that for every $i \in [k]$, $j \in [n]$, $M_i(x_0^j) = M_i(x_1^j)$. This implies that $\mathsf{Update}(\mathsf{TM}_{[M_i,\emptyset]}, P_{[x_0^j]})$ and $\mathsf{Update}(\mathsf{TM}_{[M_i,\emptyset]}, P_{[x_1^j]})$ are functionally equivalent. The security of FE now easily follows from the security of $pa\text{-}i\mathcal{O}_{\mathsf{mp}}$.

9.2 MIFE for Unbounded-Arity Functions

Let $\mathcal{M}$ be any family of Turing machines that supports arbitrary number of arbitrary length inputs. We describe a secret-key MIFE scheme $\mathsf{MIFE} = (\mathsf{MIFE.Setup}, \mathsf{MIFE.KeyGen}, \mathsf{MIFE.Enc}, \mathsf{MIFE.Dec})$ for $\mathcal{M}$ that achieves indistinguishability security against pre-ciphertext key queries. The only ingredient in our construction is an adaptively secure multi-program patchable indistinguishability obfuscation scheme $pa\text{-}i\mathcal{O}_{\mathsf{mp}} = (\mathsf{Setup}, \mathsf{Obf}, \mathsf{GenPatch}, \mathsf{AppPatch}, \mathsf{Eval})$ for a Turing machine family $\mathcal{M}_{\mathsf{mp}}$ that supports an unbounded number of parallel updates from an associated patch family $\mathcal{P}_{\mathsf{mp}}$. We denote that associated update algorithm by $\mathsf{Update}_{\mathsf{mp}}$.

- A Turing machine $\mathsf{TM} \in \mathcal{M}_{\mathsf{mp}}$ is of the form $\mathsf{TM} = \mathsf{TM}_{[M,\ell,x_1,\ldots,x_\ell]}$ where $M \in \mathcal{M}$, $\ell \geq 0$ and $x_i \in \{0,1\}^*$. On *any* input y, TM outputs $\perp$ if $\ell = 0$. Otherwise, it computes and outputs $M(x_1, \ldots, x_\ell)$.
- A patch $P \in \mathcal{P}_{\mathsf{mp}}$ is of the form $P = P_{[x]}$ where $x \in \{0,1\}^*$.
- The update algorithm $\mathsf{Update}_{\mathsf{mp}}$ on input $(\mathsf{TM}_{[M,\ell,x_1,\ldots,x_\ell]}, P_{[x]})$ outputs $\mathsf{TM}' = \mathsf{TM}_{[M,\ell',x_1,\ldots,x_{\ell'}]}$ where $\ell' = \ell + 1$ and $x_{\ell'} = x$.

We now proceed to describe MIFE.

- $\mathsf{MIFE.Setup}(1^\lambda)$: On input the security parameter λ in unary, compute $\mathsf{SK} \leftarrow \mathsf{Setup}(1^\lambda)$. Output $\mathsf{MIFE.MSK} = \mathsf{SK}$.
- $\mathsf{MIFE.KeyGen}(\mathsf{MIFE.MSK}, M)$: On input $\mathsf{MIFE.MSK} = \mathsf{SK}$ and a Turing machine $M \in \mathcal{M}$, compute $\langle \mathsf{TM}_{M_0} \rangle \leftarrow \mathsf{Obf}(\mathsf{SK}, \mathsf{TM}_{[M,0]})$ where $\mathsf{TM}_{[M,0]} \in \mathcal{M}_{\mathsf{mp}}$. Output $\mathsf{MIFE.SK}_M = \langle \mathsf{TM}_{M_0} \rangle$.
- $\mathsf{MIFE.Enc}(\mathsf{MIFE.MSK}, x)$: On input $\mathsf{MIFE.MSK} = \mathsf{SK}$ and a message $x \in \{0,1\}^*$, compute $\langle P_x \rangle \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P_{[x]})$ where $P_{[x]} \in \mathcal{P}_{\mathsf{mp}}$. Output $\mathsf{CT} = \langle P_x \rangle$.
- $\mathsf{MIFE.Dec}(\mathsf{MIFE.SK}_M, \mathsf{CT}_1, \ldots, \mathsf{CT}_\ell)$: On input a functional key $\mathsf{MIFE.SK}_M = \langle \mathsf{TM}_M \rangle$ and an arbitrary number of ciphertexts $\mathsf{CT}_1, \ldots, \mathsf{CT}_\ell$ where $\mathsf{CT}_i = \langle P_{x_i} \rangle$, compute for every $i \in [\ell]$, $\langle \mathsf{TM}_{M_i} \rangle \leftarrow \mathsf{AppPatch}\Big(\langle \mathsf{TM}_{M_{i-1}} \rangle, \langle P_{x_i} \rangle\Big)$. Output $\mathsf{Eval}\Big(\langle \mathsf{TM}_{M_\ell} \rangle, 0\Big)$.

Correctness. Let $\mathsf{FE.SK}_M = \langle \mathsf{TM}_{M_0} \rangle$ be a functional key for Turing machine $M \in \mathcal{M}$ where $\langle \mathsf{TM}_{M_0} \rangle = \mathsf{Obf}(\mathsf{SK}, \mathsf{TM}_{[M,0]})$ for $\mathsf{TM}_{[M,0]} \in \mathcal{M}_{\mathsf{mp}}$. Let $\mathsf{CT}_1, \ldots, \mathsf{CT}_\ell$ be an arbitrary number of ciphertexts s.t. $\mathsf{CT}_i = \langle P_{x_i} \rangle$ where $\langle P_{x_i} \rangle \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P_{[x_i]})$ for $P_{[x_i]} \in \mathcal{P}_{\mathsf{mp}}$. Now, from the correctness properties of $pa\text{-}i\mathcal{O}_{\mathsf{mp}}$, it follows that for every $i \in [L]$, $\mathsf{AppPatch}\Big(\langle \mathsf{TM}_{M_{i-1}} \rangle, \langle P_{x_i} \rangle\Big) = \langle \mathsf{TM}_{M_i} \rangle$ s.t. $\langle \mathsf{TM}_{M_i} \rangle$ is functionally equivalent to $\mathsf{Update}_{\mathsf{mp}}(\mathsf{TM}_{[M,i-1,x_1,\ldots,x_{i-1}]}, P_{[x_i]}) = \mathsf{TM}_{[M,i,x_1,\ldots,x_i]}$. From the definition of $\mathsf{TM}_{[M,i,x_1,\ldots,x_i]}$, we have that $\mathsf{TM}_{[M,\ell,x_1,\ldots,x_\ell]}(0) = M(x_1, \ldots, x_\ell)$, as required.

Security. The security of the above construction can be easily argued by extending the security proof of the single-ary FE construction.

Acknowledgements

We would like to thank Aloni Cohen for his constructive ideas during the early stages of this work. The second author would like to thank representatives from Toyota for motivating the problem of patchable obfuscation during a meeting at Johns Hopkins University.

References

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

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

[AJ15] Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In *CRYPTO*, 2015.

[AJN$^+$16] Prabhanjan Ananth, Aayush Jain, Moni Naor, Amit Sahai, and Eylon Yogev. Universal obfuscation and witness encryption: Boosting correctness and combining security. In *CRYPTO*, 2016.

[AJS17] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Indistinguishability obfuscation with constant size overhead. In *CRYPTO*, 2017.

[AS16] Prabhanjan Ananth and Amit Sahai. Functional encryption for turing machines. In *TCC*, 2016.

[BC10] Nir Bitansky and Ran Canetti. On strong simulation and composable point obfuscation. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 520–537, 2010.

[BCC$^+$14] Nir Bitansky, Ran Canetti, Henry Cohn, Shafi Goldwasser, Yael Tauman Kalai, Omer Paneth, and Alon Rosen. The impossibility of obfuscation with auxiliary input or a universal simulator. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 71–89, 2014.

[BCKP14] Nir Bitansky, Ran Canetti, Yael Tauman Kalai, and Omer Paneth. On virtual grey box obfuscation for general circuits. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 108–125, 2014.

[BCP14] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. In Yehuda Lindell, editor, *Theory of Cryptography - 11th Theory of Cryptography Conference, TCC 2014, San Diego, CA, USA, February 24-26, 2014. Proceedings*, volume 8349 of *Lecture Notes in Computer Science*, pages 52–73. Springer, 2014.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology—CRYPTO'94*, pages 216–233. Springer, 1994.

[BGG95] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography and application to virus protection. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 45–56. ACM, 1995.

[BGI$^+$12] 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.

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

[BGJ$^+$16] Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-lock puzzles from randomized encodings. In *ITCS*, 2016.

[BGJS15] Saikrishna Badrinaraynan, Divya Gupta, Abhishek Jain, and Amit Sahai. Multi-input functional encryption for unbounded arity functions. In *ASIACRYPT*, 2015.

[BGK$^+$14] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pages 221–238, 2014.

[BGL$^+$15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Siddartha Telang. Succinct randomized encodings and their applications. In *STOC*, 2015.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 784–796. ACM, 2012.

[BKS16] Zvika Brakerski, Ilan Komargodski, and Gil Segev. From single-input to multi-input functional encryption in the private-key setting. In *EUROCRYPT*, 2016.

[BKY01] Enrico Buonanno, Jonathan Katz, and Moti Yung. Incremental unforgeable encryption. In *Fast Software Encryption*, pages 109–124. Springer, 2001.

[BR14] Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In *TCC*, pages 1–25, 2014.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *Theory of Cryptography*, pages 253–273. Springer, 2011.

[BSW16] Mihir Bellare, Igors Stepanovs, and Brent Waters. New negative results on differing-inputs obfuscation. *IACR Cryptology ePrint Archive*, 2016:162, 2016.

[BV15] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. In *FOCS*, 2015.

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

[CCC$^+$16] 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.

[CCHR B] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Succinct adaptive garbled ram. In *TCC*, 2016-B.

[CH16] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. In *ITCS*, 2016.

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Indistinguishability obfuscation of iterated circuits and RAM programs. In *STOC*, 2015.

[CHN$^+$16] Aloni Cohen, Justin Holmgren, Ryo Nishimaki, Vinod Vaikuntanathan, and Daniel Wichs. Watermarking cryptographic capabilities. In *STOC*, 2016.

[CLTV15] Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. Obfuscation of probabilistic circuits and applications. In *TCC*. 2015.

[Fis97] Marc Fischlin. Incremental cryptography and memory checkers. In *Advances in Cryptology—EUROCRYPT'97*, pages 393–408. Springer, 1997.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009.

[GGG$^+$14] Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 578–602. Springer, 2014.

[GGH$^+$13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 40–49. IEEE Computer Society, 2013.

[GGHW14] Sanjam Garg, Craig Gentry, Shai Halevi, and Daniel Wichs. On the implausibility of differing-inputs obfuscation and extractable witness encryption with auxiliary input. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, pages 518–535, 2014.

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

[GHL$^+$14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. In *Advances in Cryptology–EUROCRYPT 2014*, pages 405–422. Springer, 2014.

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private ram computation. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 404–413. IEEE, 2014.

[GIS$^+$10] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, pages 308–326, 2010.

[GK05] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 553–562, 2005.

[GKP$^+$12] Shafi Goldwasser, Yael Tauman Kalai, Raluca A Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Succinct functional encryption and applications: Reusable garbled circuits and beyond. *IACR Cryptology ePrint Archive*, 2012:733, 2012.

[GKP$^+$13a] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 555–564. ACM, 2013.

[GKP$^+$13b] Shafi Goldwasser, Yael Tauman Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In *CRYPTO*, 2013.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, pages 39–56, 2008.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled ram. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 210–229. IEEE, 2015.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled ram from one-way functions. In *47th Annual ACM Symposium on Theory of Computing. ACM Press*, 2015.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[GP15] Sanjam Garg and Omkant Pandey. Incremental program obfuscation. Cryptology ePrint Archive, Report 2015/997, 2015. http://eprint.iacr.org/.

[GR07] Shafi Goldwasser and Guy N. Rothblum. On best-possible obfuscation. In *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*, pages 194–213, 2007.

[Had00] Satoshi Hada. Zero-knowledge and code obfuscation. In *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, pages 443–457, 2000.

[IPS15] Yuval Ishai, Omkant Pandey, and Amit Sahai. Public-coin differing-inputs obfuscation and its applications. In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings*, pages 668–697, 2015.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In *STOC*, 2015.

[KMN$^+$14] Ilan Komargodski, Tal Moran, Moni Naor, Rafael Pass, Alon Rosen, and Eylon Yogev. One-way functions and (im)perfect obfuscation. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 374–383, 2014.

[KPTZ13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 669–684. ACM, 2013.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble ram programs? In *Advances in Cryptology–EUROCRYPT 2013*, pages 719–734. Springer, 2013.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of yao?s protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[LPST16] Huijia Lin, Rafael Pass, Karn Seth, and Sidharth Telang. Output-compressing randomized encodings and applications. In *TCC*, 2016.

[Mic97] Daniele Micciancio. Oblivious data structures: applications to cryptography. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 456–464. ACM, 1997.

[MPRS12] Ilya Mironov, Omkant Pandey, Omer Reingold, and Gil Segev. Incremental deterministic public-key encryption. In *Advances in Cryptology–EUROCRYPT 2012*, pages 628–644. Springer, 2012.

[MR13] Tal Moran and Alon Rosen. There is no indistinguishability obfuscation in pessiland. *IACR Cryptology ePrint Archive*, 2013:643, 2013.

[O'N10] Adam O'Neill. Definitional issues in functional encryption. *IACR Cryptology ePrint Archive*, 2010:556, 2010.

[PF79] Nicholas Pippenger and Michael J Fischer. Relations among complexity measures. *Journal of the ACM (JACM)*, 26(2):361–381, 1979.

[RAD78] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[SW05] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, pages 457–473, 2005.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 475–484, 2014.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

A Preliminaries: Instantiation of Splittable iO

A.1 Puncturable Pseudorandom Functions

A pseudorandom function family F consisting of functions of the form $\mathsf{PRF}_K(\cdot)$, that is defined over input space $\{0,1\}^{\eta(\lambda)}$, output space $\{0,1\}^{\chi(\lambda)}$ and key K in the key space $\mathcal{K}$, is said to be a *secure puncturable PRF family* if there exists a PPT algorithm $\mathsf{PRFPunc}$ that satisfies the following properties:

- **Functionality preserved under puncturing.** $\mathsf{PRFPunc}$ takes as input a PRF key K, sampled from $\mathcal{K}$, and an input $x \in \{0,1\}^{\eta(\lambda)}$ and outputs K_x such that for all $x' \neq x$, $\mathsf{PRF}_{K_x}(x') = \mathsf{PRF}_K(x')$.

- **Pseudorandom at punctured points.** For every PPT adversary $(\mathcal{A}_1, \mathcal{A}_2)$ such that $\mathcal{A}_1(1^\lambda)$ outputs an input $x \in \{0,1\}^{\eta(\lambda)}$, consider an experiment where $K \xleftarrow{\$} \mathcal{K}$ and $K_x \leftarrow \mathsf{PRFPunc}(K, x)$. Then for all sufficiently large $\lambda \in \mathbb{N}$, for a negligible function μ,

$$\left|\Pr[\mathcal{A}_2(K_x, x, \mathsf{PRF}_K(x)) = 1] - Pr[\mathcal{A}_2(K_x, x, U_{\chi(\lambda)}) = 1]\right| \leq \mu(\lambda)$$

  where $U_{\chi(\lambda)}$ is a string drawn uniformly at random from $\{0,1\}^{\chi(\lambda)}$.

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

Theorem 19 ([GGM86, BW13, BGI14, KPTZ13])**.** *If μ-secure one-way functions*[10] *exist, then for all polynomials $\eta(\lambda)$ and $\chi(\lambda)$, there exists a μ-secure puncturable PRF family that maps $\eta(\lambda)$ bits to $\chi(\lambda)$ bits.*

A.2 Garbling schemes

Yao in his seminal work [Yao86, LP09] proposed the notion of garbled circuits as a solution to the problem of secure two party computation. Recently, Bellare et al. [BHR12] formalized this in the form of a primitive called *garbling scheme*. We adopt this notion. The syntax of the garbling scheme, defined below, is similar to the definition of Bellare et al.

A garbling scheme GC for a class of circuits $\mathcal{C} = \{\mathcal{C}_n\}_{n\in\mathbb{N}}$ consists of two PPT algorithms namely $(\mathsf{Garble}, \mathsf{EvalGC})$.

- **Garbling algorithm**, $\mathsf{Garble}(1^\lambda, C \in \mathcal{C})$: On input a security parameter λ in unary and a circuit $C \in \mathcal{C}_n$ of input length n, it outputs a garbled circuit along with its wire keys, $\left(\mathcal{GC}, \{w_{i,0}, w_{i,1}\}_{i\in[n]}\right)$.

- **Garbled circuit evaluation algorithm**, $\mathsf{EvalGC}\left(\mathcal{GC}, \{w_{i,x_i}\}_{i\in[n]}\right)$: On input the garbled circuit $\mathcal{GC}$ and wire keys $\{w_{i,x_i}\}_{i\in[n]}$ corresponding to x, it outputs out.

The correctness property of a garbling scheme dictates that for every $C \in \mathcal{C}_n$, the output of the evaluation procedure $\mathsf{EvalGC}(\mathcal{GC}, \{w_{i,x_i}\}_{i\in[n]})$ is $C(x)$, where $(\mathcal{GC}, \{w_{i,0}, w_{i,1}\}_{i\in[n]}) \leftarrow \mathsf{Garble}(1^\lambda, C)$.

[10]We say that a one-way function family is μ-secure if the probability of inverting a one-way function, that is sampled from the family, is at most $\mu(\lambda)$.

Security. A garbling scheme is said to be secure if the joint distribution of garbled circuit along with the wire keys, corresponding to some input, reveals only the output of the circuit and nothing else. This can be formalized in the form of a simulation-based notion as given below.

Definition 20. *A garbling scheme* $\mathsf{GC} = (\mathsf{Garble}, \mathsf{EvalGC})$ *for a class of circuits* $\mathcal{C} = \{\mathcal{C}_n\}_{n\in\mathbb{N}}$ *is said to be secure if there exists a simulator* $\mathsf{SimGarble}$ *such that for any* $C \in \mathcal{C}_n$*, any input* $x \in \{0,1\}^n$*, the following two distributions are computationally distinguishable.*

1. $\left\{ \mathsf{SimGarble}\left(1^\lambda, \phi(C), C(x)\right) \right\}$*, where* $\phi(C)$ *denotes the topology of the circuit* C*.*
2. $\left\{ (\mathcal{GC}, \{w_{i,x_i}\}_{i\in[n]}) \right\}$*, where* $(\mathcal{GC}, \{w_{i,0}, w_{i,1}\}_{i\in[n]}) \leftarrow \mathsf{Garble}(1^\lambda, C)$*.*

A.3 Fully Homomorphic Encryption for circuits

Another main tool that we use in one of our constructions is a fully homomorphic encryption scheme [RAD78] (FHE).

A public key fully homomorphic encryption (FHE) scheme for a class of circuits $\mathcal{C} = \{\mathcal{C}_\lambda\}_\lambda$ and message space $\mathsf{MSG} = \{\mathsf{MSG}_\lambda\}_{\lambda\in\mathbb{N}}$ consists of four PPT algorithms, namely, $(\mathsf{FHE.Setup}, \mathsf{FHE.Enc}, \mathsf{FHE.Eval}, \mathsf{FHE.Dec})$. The syntax of the algorithms are described below.

- **Setup,** $\mathsf{FHE.Setup}(1^\lambda)$: On input a security parameter 1^λ it outputs a public key-secret key pair $(\mathsf{FHE.pk}, \mathsf{FHE.sk})$.

- **Encryption,** $\mathsf{FHE.Enc}(\mathsf{FHE.pk}, m \in \mathsf{MSG}_\lambda)$: On input public key $\mathsf{FHE.pk}$ and message $m \in \mathsf{MSG}_\lambda$, it outputs a ciphertext denoted by $\mathsf{FHE.CT}$.

- **Evaluation,** $\mathsf{FHE.Eval}(\mathsf{FHE.pk}, C \in \mathcal{C}, \mathsf{FHE.CT})$: On input public key $\mathsf{FHE.pk}$, a circuit $C \in \mathcal{C}_\lambda$ and a FHE ciphertext $\mathsf{FHE.CT}$, it outputs the evaluated ciphertext $\mathsf{FHE.CT}'$.

- **Decryption,** $\mathsf{FHE.Dec}(\mathsf{FHE.sk}, \mathsf{FHE.CT})$: On input the secret key $\mathsf{FHE.sk}$ and a ciphertext $\mathsf{FHE.CT}$, it outputs the decrypted value out.

The correctness property guarantees the following, where $(\mathsf{FHE.pk}, \mathsf{FHE.sk}) \leftarrow \mathsf{FHE.Setup}(1^\lambda)$ and $\mathsf{FHE.CT} \leftarrow \mathsf{FHE.Enc}(\mathsf{FHE.pk}, m \in \mathsf{MSG}_\lambda)$:

- $m \leftarrow \mathsf{FHE.Dec}(\mathsf{FHE.sk}, \mathsf{FHE.CT})$
- For any circuit $C \in \mathcal{C}_\lambda$ where the input length of C is $|m|$, we have $C(m) \leftarrow \mathsf{FHE.Dec}(\mathsf{FHE.sk}, \mathsf{FHE.Eval}(\mathsf{FHE.pk}, C, \mathsf{FHE.CT}))$.

The security notion of an FHE scheme is identical to the definition of semantic security of a public key encryption scheme.

An FHE scheme should also satisfy the so called compactness property. At a high level, the compactness property ensures that the length of the ciphertext output by $\mathsf{FHE.Eval}(\mathsf{FHE.pk}, C, \cdot)$ is independent of the size of C.

FHE from iO and Re-Randomizable Encryption. Recently, Canetti et al. [CLTV15] proposed a construction of fully homomorphic encryption from (sub-exponentially secure) iO and re-randomizable encryption schemes. We instantiate the FHE scheme we use in our work using this construction.

B KLW Building Blocks [KLW15]

We recall some notions introduced in the work of Koppula, Lewko, Waters [KLW15]. There are three main building blocks: positional accumulators, splittable signatures and iterators. The following definitions are stated verbatim from [KLW15]. In this section, we only state the essential security properties of these primitives we explicitly use in this work. There are other security properties associated with these primitives - we define them in Appendix B.1.

I. Positional Accumulators. We give a brief overview of this primitive. It consists of the algorithms $(\mathsf{Setup}, \mathsf{EnforceRead}, \mathsf{EnforceWrite}, \mathsf{PrepRead}, \mathsf{PrepWrite}, \mathsf{VerifyRead}, \mathsf{WriteStore}, \mathsf{Update})$ and is associated with message space Msg_λ.

The algorithm Setup generates the accumulator public parameters along with the initial storage value and the initial root value. It helps to think of the storage as being a hash tree and its associated accumulator value being the root and they both are initialized to $\perp$. There are two algorithms that generate "fake" public parameters, namely, $\mathsf{EnforceRead}$ and $\mathsf{EnforceWrite}$. The algorithm $\mathsf{EnforceRead}$ takes as input a special index INDEX^* along with a sequence of k computational steps $(m_1, \mathrm{INDEX}_1), \ldots, (m_k, \mathrm{INDEX}_k)$ (symbol-index pairs) and produces fake public parameters along with initialized storage and root values. Later, we will put forth a requirement that any PPT adversary cannot distinguish "real" public parameters (generated by Setup) and "fake" public parameters (generated by $\mathsf{EnforceRead}$). Also we put forth an information theoretic requirement that any storage generated by the "fake" public parameters is such that the accumulator value associated with the storage *determines a unique value at the location* INDEX^*. The algorithm $\mathsf{EnforceWrite}$ has a similar flavor as $\mathsf{EnforceRead}$ and we describe in the formal definition.

Once the setup algorithm is executed, there are two algorithms that deal with arguing about the correctness of the storage. The first one, $\mathsf{PrepRead}$ takes as input a storage, an index and produces the symbol at the location at that index and an accompanying proof – we later require this proof to be "short" (in particular, independent of the size of storage). $\mathsf{PrepWrite}$ essentially does the same task except that it does not output the symbol at that location – that it only produces the proof (in the formal definition, we call this aux). Another procedure, $\mathsf{VerifyRead}$ then verifies whether the proof produced by $\mathsf{PrepRead}$ is valid or not. The above algorithms help to verify the correctness of storage. But how do we compute the storage? $\mathsf{WriteStore}$ takes as input an old storage along with a new symbol and the location where the new symbol needs to be assigned. It updates the storage appropriately and outputs the new storage. The algorithm Update describes how to "succinctly" update the accumulator by just knowing the public parameters, accumulator value, message symbol, index and auxiliary information aux (produced by $\mathsf{WriteStore}$). Here, "succinctness" refers to the fact that the update time of Update is independent of the size of the storage.

Syntax. A positional accumulator for message space Msg_λ consists of the following algorithms.

- $\mathsf{SetupAcc}(1^\lambda, T) \to (\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0)$: The setup algorithm takes as input a security parameter λ in unary and an integer T in binary representing the maximum number of values that can stored. It outputs public parameters $\mathsf{PP}_{\mathsf{Acc}}$, an initial accumulator value w_0, and an initial storage value $store_0$.

- $\mathsf{EnforceRead}(1^\lambda, T, (m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k), \text{INDEX}^*) \to (\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0)$: The setup enforce read algorithm takes as input a security parameter λ in unary, an integer T in binary representing the maximum number of values that can be stored, and a sequence of symbol, index pairs, where each index is between 0 and $T-1$, and an additional INDEX^* also between 0 and $T-1$. It outputs public parameters $\mathsf{PP}_{\mathsf{Acc}}$, an initial accumulator value w_0, and an initial storage value $store_0$.

- $\mathsf{EnforceWrite}(1^\lambda, T, (m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k)) \to (\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0)$: The setup enforce write algorithm takes as input a security parameter λ in unary, an integer T in binary representing the maximum number of values that can be stored, and a sequence of symbol, index pairs, where each index is between 0 and $T-1$. It outputs public parameters $\mathsf{PP}_{\mathsf{Acc}}$, an initial accumulator value w_0, and an initial storage value $store_0$.

- $\mathsf{PrepRead}(\mathsf{PP}_{\mathsf{Acc}}, store_{in}, \text{INDEX}) \to (m, \pi)$: The prep-read algorithm takes as input the public parameters $\mathsf{PP}_{\mathsf{Acc}}$, a storage value $store_{in}$, and an index between 0 and $T-1$. It outputs a symbol m (that can be ϵ) and a value π.

- $\mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}}, store_{in}, \text{INDEX}) \to aux$: The prep-write algorithm takes as input the public parameters $\mathsf{PP}_{\mathsf{Acc}}$, a storage value $store_{in}$, and an index between 0 and $T-1$. It outputs an auxiliary value aux.

- $\mathsf{VerifyRead}(\mathsf{PP}_{\mathsf{Acc}}, w_{in}, m_{read}, \text{INDEX}, \pi) \to (\{True, False\})$: The verify-read algorithm takes as input the public parameters $\mathsf{PP}_{\mathsf{Acc}}$, an accumulator value w_{in}, a symbol, m_{read}, an index between 0 and $T-1$, and a value π. It outputs $True$ or $False$.

- $\mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, store_{in}, \text{INDEX}, m) \to store_{out}$: The write-store algorithm takes in the public parameters, a storage value $store_{in}$, an index between 0 and $T-1$, and a symbol m. It outputs a storage value $store_{out}$.

- $\mathsf{Update}(\mathsf{PP}_{\mathsf{Acc}}, w_{in}, m_{write}, \text{INDEX}, aux) \to (w_{out} \text{ or } Reject)$: The update algorithm takes in the public parameters $\mathsf{PP}_{\mathsf{Acc}}$, an accumulator value w_{in}, a symbol m_{write}, and index between 0 and $T-1$, and an auxiliary value aux. It outputs an accumulator value w_{out} or $Reject$.

Remark 9. *In our construction, we will set $T = 2^\lambda$ and so T will not be an explicit input to all the algorithms.*

Correctness. We consider any sequence $(m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k)$ of symbols $m_1, \ldots, m_k$ and indices $\text{INDEX}_1, \ldots, \text{INDEX}_k$ each between 0 and $T-1$. Let $(\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0) \leftarrow \mathsf{SetupAcc}(1^\lambda, T)$. For j from 1 to k, we define $store_j$ iteratively as $store_j := \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, store_{j-1}, \text{INDEX}_j, m_j)$. We similarly define aux_j and w_j iteratively as $aux_j := \mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}}, store_{j-1}, \text{INDEX}_j)$ and $w_j := Update(\mathsf{PP}_{\mathsf{Acc}}, w_{j-1}, m_j, \text{INDEX}_j, aux_j)$. Note that the algorithms other than $\mathsf{SetupAcc}$ are deterministic, so these definitions fix precise values, not random values (conditioned on the fixed starting values $\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0$).

We require the following correctness properties:

1. For every INDEX between 0 and $T-1$, $\mathsf{PrepRead}(\mathsf{PP}_{\mathsf{Acc}}, store_k, \text{INDEX})$ returns m_i, π, where i is the largest value in $[k]$ such that $\text{INDEX}_i = \text{INDEX}$. If no such value exists, then $m_i = \epsilon$.

2. For any INDEX, let $(m, \pi) \leftarrow \mathsf{PrepRead}(\mathsf{PP}_{\mathsf{Acc}}, store_k, \text{INDEX})$. Then $\mathsf{VerifyRead}(\mathsf{PP}_{\mathsf{Acc}}, w_k, m, \text{INDEX}, \pi) = True$.

Efficiency. We require that $|\pi|$ is a fixed polynomial in λ, where $(m, \pi) \leftarrow \mathsf{PrepRead}(\mathsf{PP}_{\mathsf{Acc}}, store_{in}, \text{INDEX})$. In particular, $|\pi|$ should be independent of $|store_{in}|$. We similarly, require that $|aux|$ to be a fixed polynomial in λ, where $aux \leftarrow \mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}}, store_{in}, \text{INDEX})$.

The security properties are defined in Section B.1.

II. Splittable Signatures. A splittable signature scheme is a deterministic signature scheme consisting of the basic algorithms ($\mathsf{SetupSpl}, \mathsf{SignSpl}, \mathsf{VerSpl}$) (i.e, setup, signing and verification algorithms) as in a standard signature scheme except that $\mathsf{SetupSpl}$, in addition to the standard signing key-verification key pair (SK, VK), also outputs a rejection-verification key $\mathsf{VK}_{\mathsf{rej}}$. $\mathsf{VK}_{\mathsf{rej}}$ is defined to be such that it rejects every message-signature pair – this will be useful in the security proof. In addition to the above algorithms, the scheme is associated with algorithms $\mathsf{SplitSpl}$ and $\mathsf{SignSplAbo}$. The algorithm $\mathsf{SplitSpl}$ takes as input a signing key-message pair (SK, m^*) and outputs a verification key $\mathsf{VK}_{\mathsf{one}}$, an all-but-one signing key $\mathsf{SK}_{\mathsf{abo}}$ and an all-but-one verification key $\mathsf{VK}_{\mathsf{abo}}$. Using $\mathsf{VK}_{\mathsf{one}}$ we can verify whether a signature associated with m^* is valid or not. For any other message, $\mathsf{VK}_{\mathsf{one}}$ is useless – that is, the output of the verification algorithm is 0. Similarly, $\mathsf{VK}_{\mathsf{abo}}$ is used to verify the signatures on all messages except m^*. The key $\mathsf{SK}_{\mathsf{abo}}$ is used to sign all the messages except m^*. To sign using the key $\mathsf{SK}_{\mathsf{abo}}$ we use a special signing algorithm $\mathsf{SignSplAbo}$.

The security properties associated are described at a high level below:

1. $\mathsf{VK}_{\mathsf{rej}}$ *indistinguishability*: $\mathsf{VK}_{\mathsf{rej}}$ is indistinguishable from the standard verification key VK when no signatures are given to the adversary.

2. $\mathsf{VK}_{\mathsf{one}}$ *indistinguishability*: It is computationally hard to distinguish $\mathsf{VK}_{\mathsf{one}}$ and VK even when the adversary is given a signature on m^* (and no other signatures).

3. $\mathsf{VK}_{\mathsf{abo}}$ *indistinguishability*: It is hard to distinguish $\mathsf{VK}_{\mathsf{abo}}$ and VK even when the adversary is given the signing key $\mathsf{SK}_{\mathsf{abo}}$.

4. *Splitting Indistinguishability*: Suppose ($\mathsf{SK}, \mathsf{VK}, \mathsf{VK}_{\mathsf{rej}}$) be the output of one execution of $\mathsf{SetupSpl}$ of the signature scheme. This security property says that for any message m^* it is hard to distinguish the following: (a) ($\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}}, \mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}}$) with ($\mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}}$) derived from ($\mathsf{SK}, \mathsf{VK}$) and, (b) ($\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}}, \mathsf{SK}^*_{\mathsf{abo}}, \mathsf{VK}^*_{\mathsf{abo}}$) with ($\mathsf{SK}^*_{\mathsf{abo}}, \mathsf{VK}^*_{\mathsf{abo}}$) derived from ($\mathsf{SK}^*, \mathsf{VK}^*$) which in turn is generated independently of (SK, VK).

Syntax. A splittable signature scheme $\mathsf{SplScheme}$ for message space Msg consists of the following algorithms:

- $\mathsf{SetupSpl}(1^\lambda)$ The setup algorithm is a randomized algorithm that takes as input the security parameter λ and outputs a signing key SK, a verification key VK and *reject-verification key* $\mathsf{VK}_{\mathsf{rej}}$.

- $\mathsf{SignSpl}(\mathsf{SK}, m)$ The signing algorithm is a deterministic algorithm that takes as input a signing key SK and a message $m \in \mathsf{Msg}$. It outputs a signature σ.

- $\mathsf{VerSpl}(\mathsf{VK}, m, \sigma)$ The verification algorithm is a deterministic algorithm that takes as input a verification key VK, signature σ and a message m. It outputs either 0 or 1.

- $\mathsf{SplitSpl}(\mathsf{SK}, m^*)$ The splitting algorithm is randomized. It takes as input a secret key SK and a message $m^* \in \mathsf{Msg}$. It outputs a signature $\sigma_{\mathsf{one}} = \mathsf{SignSpl}(\mathsf{SK}, m^*)$, a one-message verification key $\mathsf{VK}_{\mathsf{one}}$, an all-but-one signing key $\mathsf{SK}_{\mathsf{abo}}$ and an all-but-one verification key $\mathsf{VK}_{\mathsf{abo}}$.

- $\mathsf{SignSplAbo}(\mathsf{SK}_{\mathsf{abo}}, m)$ The all-but-one signing algorithm is deterministic. It takes as input an all-but-one signing key $\mathsf{SK}_{\mathsf{abo}}$ and a message m, and outputs a signature σ.

Correctness. Let $m^* \in \mathsf{Msg}$ be any message. Let $(\mathsf{SK}, \mathsf{VK}, \mathsf{VK}_{\mathsf{rej}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$ and $(\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}}, \mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}}) \leftarrow \mathsf{SplitSpl}(\mathsf{SK}, m^*)$. Then, we require the following correctness properties:

1. For all $m \in \mathsf{Msg}$, $\mathsf{VerSpl}(\mathsf{VK}, m, \mathsf{SignSpl}(\mathsf{SK}, m)) = 1$.
2. For all $m \in \mathsf{Msg}, m \neq m^*$, $\mathsf{SignSpl}(\mathsf{SK}, m) = \mathsf{SignSplAbo}(\mathsf{SK}_{\mathsf{abo}}, m)$.
3. For all σ, $\mathsf{VerSpl}(\mathsf{VK}_{\mathsf{one}}, m^*, \sigma) = \mathsf{VerSpl}(\mathsf{VK}, m^*, \sigma)$.
4. For all $m \neq m^*$ and σ, $\mathsf{VerSpl}(\mathsf{VK}, m, \sigma) = \mathsf{VerSpl}(\mathsf{VK}_{\mathsf{abo}}, m, \sigma)$.
5. For all $m \neq m^*$ and σ, $\mathsf{VerSpl}(\mathsf{VK}_{\mathsf{one}}, m, \sigma) = 0$.
6. For all σ, $\mathsf{VerSpl}(\mathsf{VK}_{\mathsf{abo}}, m^*, \sigma) = 0$.
7. For all σ and all $m \in \mathsf{Msg}$, $\mathsf{VerSpl}(\mathsf{VK}_{\mathsf{rej}}, m, \sigma) = 0$.

Security: $\mathsf{VK}_{\mathsf{one}}$ indistinguishability. We describe $\mathsf{VK}_{\mathsf{one}}$ indistinguishability security property at a high level below. The rest of the security properties (that will only implicitly be used in our work) is described in Appendix B.1.

Definition 21 ($\mathsf{VK}_{\mathsf{one}}$ indistinguishability)**.** *A splittable signature scheme* $\mathsf{SplScheme}$ *for a message space* Msg *is said to be* $\mathsf{VK}_{\mathsf{one}}$ *indistinguishable if any PPT adversary* $\mathcal{A}$ *has negligible advantage in the following security game:*

$\mathsf{Expt}(1^\lambda, \mathsf{SplScheme}, \mathcal{A})$*:*

1. $\mathcal{A}$ *sends a message* $m^* \in \mathsf{Msg}$.

2. *Challenger computes* $(\mathsf{SK}, \mathsf{VK}, \mathsf{VK}_{\mathsf{rej}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$*. Next, it computes* $(\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}}, \mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}}) \leftarrow \mathsf{SplitSpl}(\mathsf{SK}, m^*)$*. It chooses* $b \leftarrow \{0,1\}$*. If* $b = 0$*, it sends* $(\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}})$ *to* $\mathcal{A}$*. Else, it sends* $(\sigma_{\mathsf{one}}, \mathsf{VK})$ *to* $\mathcal{A}$*.*
3. $\mathcal{A}$ *sends its guess* b'*.*

$\mathcal{A}$ *wins if* $b = b'$*.*

We note that in the game above, $\mathcal{A}$ only receives the signature σ_{one} on m^*, on which VK and VK_{one} behave identically.

III. Iterators. This notion is similar in spirit to the notion of accumulators described earlier. While accumulators were used to bind the storage, the role of iterators is to bind the state of the Turing machines. An iterator scheme consists of three algorithms ($\mathsf{SetupItr}, \mathsf{ItrEnforce}, \mathsf{Iterate}$). $\mathsf{SetupItr}$ is used to generate the public parameters. $\mathsf{ItrEnforce}$ is used to generate "fake" parameters, that is indistinguishable from the real parameters. These parameters can then be used to generate iterator values using the algorithm $\mathsf{Iterate}$.

Syntax. Let ℓ be any polynomial. An iterator $\mathsf{PP}_{\mathsf{Itr}}$ with message space $\mathsf{Msg}_\lambda = \{0,1\}^{\ell(\lambda)}$ and state space $\mathcal{S}_\lambda$ consists of three algorithms $\mathsf{SetupItr}$, $\mathsf{ItrEnforce}$ and $\mathsf{Iterate}$ as defined below.

- $\mathsf{SetupItr}(1^\lambda, T)$ The setup algorithm takes as input the security parameter λ (in unary), and an integer bound T (in binary) on the number of iterations. It outputs public parameters $\mathsf{PP}_{\mathsf{Itr}}$ and an initial state $v_0 \in \mathcal{S}_\lambda$.

- $\mathsf{ItrEnforce}(1^\lambda, T, \vec{m} = (m_1, \ldots, m_k))$ The enforced setup algorithm takes as input the security parameter λ (in unary), an integer bound T (in binary) and k messages $(m_1, \ldots, m_k)$, where each $m_i \in \{0,1\}^{\ell(\lambda)}$ and k is some polynomial in λ. It outputs public parameters $\mathsf{PP}_{\mathsf{Itr}}$ and a state $v_0 \in \mathcal{S}_\lambda$.

- $\mathsf{Iterate}(\mathsf{PP}_{\mathsf{Itr}}, v_{\mathsf{in}}, m)$ The iterate algorithm takes as input the public parameters $\mathsf{PP}_{\mathsf{Itr}}$, a state v_{in}, and a message $m \in \{0,1\}^{\ell(\lambda)}$. It outputs a state $v_{\mathsf{out}} \in \mathcal{S}_\lambda$.

Remark 10. *As in the case of positional accumulators, we set* T *to be* 2^λ *and not mention* T *as an explicit input to the above algorithms.*

For any integer $k \le T$, we will use the notation $\mathsf{Iterate}^k(\mathsf{PP}_{\mathsf{Itr}}, v_0, (m_1, \ldots, m_k))$ to denote $\mathsf{Iterate}(\mathsf{PP}_{\mathsf{Itr}}, v_{k-1}, m_k)$, where $v_j = \mathsf{Iterate}(\mathsf{PP}_{\mathsf{Itr}}, v_{j-1}, m_j)$ for all $1 \le j \le k-1$.

Remark 11. *Unlike standard cryptographic primitives, positional iterators (as defined by [KLW15]) does not have any correctness notion associated with it.*

B.1 Security Properties of Primitives of [KLW15]

We provide the security properties of the primitives verbatim from [KLW15].

B.2 Security Properties of Positional Accumulators

Let Acc = ($\mathsf{SetupAcc}$, $\mathsf{EnforceRead}$, $\mathsf{EnforceWrite}$, $\mathsf{PrepRead}$, $\mathsf{PrepWrite}$, $\mathsf{VerifyRead}$, $\mathsf{WriteStore}$, Update) be a positional accumulator for symbol set M. We require Acc to satisfy the following notions of security.

Definition 22 (Indistinguishability of Read Setup). *A positional accumulator* Acc *is said to satisfy indistinguishability of read setup if any PPT adversary $\mathcal{A}$'s advantage in the security game* $\mathsf{Expt}_{\mathsf{Acc}}(1^\lambda, \mathcal{A})$ *is at most negligible in λ, where* $\mathsf{Expt}_{\mathsf{Acc}}$ *is defined as follows.*

$\mathsf{Expt}_{\mathsf{Acc}}(1^\lambda, \mathcal{A})$

1. *Adversary chooses a bound $T \in \Theta(2^\lambda)$ and sends it to challenger.*
2. *$\mathcal{A}$ sends k messages $m_1, \ldots, m_k \in \mathsf{M}$ and k indices* $\mathrm{INDEX}_1, \ldots,$ $indexA_k \in \{0, \ldots, T-1\}$ *to the challenger.*
3. *The challenger chooses a bit b. If $b = 0$, the challenger outputs* $(\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0) \leftarrow \mathsf{SetupAcc}(1^\lambda, T)$. *Else, it outputs* $(\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0) \leftarrow \mathsf{EnforceRead}(1^\lambda, T, (m_1, \mathrm{INDEX}_1), \ldots, (m_k, \mathrm{INDEX}_k))$.
4. *$\mathcal{A}$ sends a bit b'.*

$\mathcal{A}$ wins the security game if $b = b'$.

Definition 23 (Indistinguishability of Write Setup). *A positional accumulator* Acc *is said to satisfy indistinguishability of write setup if any PPT adversary $\mathcal{A}$'s advantage in the security game* $\mathsf{Expt}_{\mathsf{Acc}}(1^\lambda, \mathcal{A})$ *is at most negligible in λ, where* $\mathsf{Expt}_{\mathsf{Acc}}$ *is defined as follows.*

$\mathsf{Expt}_{\mathsf{Acc}}(1^\lambda, \mathcal{A})$

1. *Adversary chooses a bound $T \in \Theta(2^\lambda)$ and sends it to challenger.*
2. *$\mathcal{A}$ sends k messages $m_1, \ldots, m_k \in \mathsf{M}$ and k indices* $\mathrm{INDEX}_1, \ldots,$ $indexA_k \in \{0, \ldots, T-1\}$ *to the challenger.*
3. *The challenger chooses a bit b. If $b = 0$, the challenger outputs* $(\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0) \leftarrow \mathsf{SetupAcc}(1^\lambda, T)$. *Else, it outputs* $(\mathsf{PP}_{\mathsf{Acc}}, w_0, store_0) \leftarrow \mathsf{EnforceWrite}(1^\lambda, T, (m_1, \mathrm{INDEX}_1), \ldots, (m_k, \mathrm{INDEX}_k))$.
4. *$\mathcal{A}$ sends a bit b'.*

$\mathcal{A}$ wins the security game if $b = b'$.

Definition 24 (Read Enforcing). *Consider any $\lambda \in \mathbb{N}$, $T \in \Theta(2^\lambda)$, $m_1, \ldots, m_k \in \mathsf{M}$,* $\mathrm{INDEX}_1, \ldots, \mathrm{INDEX}_k \in \{0, \ldots, T-1\}$ *and any* $\mathrm{INDEX}^* \in \{0, \ldots, T-1\}$.

Let $(\mathsf{PP}_{\mathsf{Acc}}, w_0, \mathsf{st}_0) \leftarrow \mathsf{EnforceRead}(1^\lambda, T, (m_1, \mathrm{INDEX}_1), \ldots, (m_k, \mathrm{INDEX}_k), \mathrm{INDEX}^*)$. *For j from 1 to k, we define $store_j$ iteratively as* $store_j := \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, store_{j-1}, \mathrm{INDEX}_j, m_j)$. *We similarly define aux_j and w_j iteratively as* $aux_j := \mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}}, store_{j-1}, \mathrm{INDEX}_j)$ *and* $w_j := Update(\mathsf{PP}_{\mathsf{Acc}}, w_{j-1}, m_j, \mathrm{INDEX}_j, aux_j)$. Acc *is said to be* read enforcing *if* $\mathsf{VerifyRead}(\mathsf{PP}_{\mathsf{Acc}}, w_k, m, \mathrm{INDEX}^*, \pi) = True$, *then either* $\mathrm{INDEX}^* \notin \{\mathrm{INDEX}_1, \ldots, \mathrm{INDEX}_k\}$ *and $m = \epsilon$, or $m = m_i$ for the largest $i \in [k]$ such that* $\mathrm{INDEX}_i = \mathrm{INDEX}^*$. *Note that this is an information-theoretic property: we are requiring that for all other symobls m, values of π that would cause* $\mathsf{VerifyRead}$ *to output $True$ at* INDEX^* *do no exist.*

Definition 25 (Write Enforcing). *Consider any* $\lambda \in \mathbb{N}$, $T \in \Theta(2^\lambda)$, $m_1, \ldots, m_k \in \mathsf{M}$, $\text{INDEX}_1, \ldots, \text{INDEX}_k \in \{0, \ldots, T-1\}$. *Let* $(\mathsf{PP}_{\mathsf{Acc}}, w_0, \mathsf{st}_0) \leftarrow \mathsf{EnforceWrite}(1^\lambda, T, (m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k))$. *For* j *from 1 to* k, *we define* $store_j$ *iteratively as* $store_j := \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, store_{j-1}, \text{INDEX}_j, m_j)$. *We similarly define* aux_j *and* w_j *iteratively as* $aux_j := \mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}}, store_{j-1}, \text{INDEX}_j)$ *and* $w_j := Update(\mathsf{PP}_{\mathsf{Acc}}, w_{j-1}, m_j, \text{INDEX}_j, aux_j)$. Acc *is said to be* write enforcing *if* $\mathsf{Update}(\mathsf{PP}_{\mathsf{Acc}}, w_{k-1}, m_k, \text{INDEX}_k, aux) = w_{out} \neq Reject$, *for any aux, then* $w_{out} = w_k$. *Note that this is an information-theoretic property: we are requiring that an aux value producing an accumulated value other than* w_k *or Reject deos not exist.*

B.3 Security Properties of Splittable Signatures

We will now define the security notions for splittable signature schemes. Each security notion is defined in terms of a security game between a challenger and an adversary $\mathcal{A}$.

Definition 26 ($\mathsf{VK}_{\mathsf{rej}}$ indistinguishability). *A splittable signature scheme § is said to be* $\mathsf{VK}_{\mathsf{rej}}$ *indistinguishable if any PPT adversary* $\mathcal{A}$ *has negligible advantage in the following security game:*

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

1. *Challenger computes* $(\mathsf{SK}, \mathsf{VK}, \mathsf{VK}_{\mathsf{rej}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$ *.Next, it chooses* $b \leftarrow \{0,1\}$. *If* $b = 0$, *it sends* VK *to* $\mathcal{A}$. *Else, it sends* $\mathsf{VK}_{\mathsf{rej}}$.
2. $\mathcal{A}$ *sends its guess* b'.

$\mathcal{A}$ *wins if* $b = b'$.

We note that in the game above, $\mathcal{A}$ never receives any signatures and has no ability to produce them. This is why the difference between VK and $\mathsf{VK}_{\mathsf{rej}}$ cannot be tested.

Definition 27 ($\mathsf{VK}_{\mathsf{one}}$ indistinguishability). *A splittable signature scheme § is said to be* $\mathsf{VK}_{\mathsf{one}}$ *indistinguishable if any PPT adversary* $\mathcal{A}$ *has negligible advantage in the following security game:*

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

1. $\mathcal{A}$ *sends a message* $m^* \in \mathsf{M}$.
2. *Challenger computes* $(\mathsf{SK}, \mathsf{VK}, \mathsf{VK}_{\mathsf{rej}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$. *Next, it computes* $(\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}}, \mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}}) \leftarrow \mathsf{SplitSpl}(\mathsf{SK}, m^*)$. *It chooses* $b \leftarrow \{0,1\}$. *If* $b = 0$, *it sends* $(\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}})$ *to* $\mathcal{A}$. *Else, it sends* $(\sigma_{\mathsf{one}}, \mathsf{VK})$ *to* $\mathcal{A}$.
3. $\mathcal{A}$ *sends its guess* b'.

$\mathcal{A}$ *wins if* $b = b'$.

We note that in the game above, $\mathcal{A}$ only receives the signature σ_{one} on m^*, on which VK and VK_{one} behave identically.

Definition 28 ($\mathsf{VK}_{\mathsf{abo}}$ indistinguishability). *A splittable signature scheme § is said to be* $\mathsf{VK}_{\mathsf{abo}}$ *indistinguishable if any PPT adversary* $\mathcal{A}$ *has negligible advantage in the following security game:*

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

1. $\mathcal{A}$ *sends a message* $m^* \in \mathsf{M}$.

2. *Challenger computes* $(\mathsf{SK}, \mathsf{VK}, \mathsf{VK}_{\mathsf{rej}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$*. Next, it computes* $(\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}}, \mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}}) \leftarrow \mathsf{SplitSpl}(\mathsf{SK}, m^*)$*. It chooses* $b \leftarrow \{0,1\}$*. If* $b = 0$*, it sends* $(\mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}})$ *to* $\mathcal{A}$*. Else, it sends* $(\mathsf{SK}_{\mathsf{abo}}, \mathsf{VK})$ *to* $\mathcal{A}$*.*
3. $\mathcal{A}$ *sends its guess* b'*.*

$\mathcal{A}$ *wins if* $b = b'$*.*

We note that in the game above, $\mathcal{A}$ does not receive or have the ability to create a signature on m^*. For all signatures $\mathcal{A}$ can create by signing with $\mathsf{SK}_{\mathsf{abo}}$, $\mathsf{VK}_{\mathsf{abo}}$ and VK will behave identically.

Definition 29 (Splitting indistinguishability)**.** *A splittable signature scheme § is said to be splitting indistinguishable if any PPT adversary* $\mathcal{A}$ *has negligible advantage in the following security game:*

$\mathsf{Expt}_{\mathsf{Spl}}(1^\lambda, \mathcal{A})$*:*

1. $\mathcal{A}$ *sends a message* $m^* \in \mathsf{M}$*.*
2. *Challenger computes* $(\mathsf{SK}, \mathsf{VK}, \mathsf{VK}_{\mathsf{rej}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$*,* $(\mathsf{SK}', \mathsf{VK}', \mathsf{VK}'_{\mathsf{rej}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$*. Next, it computes* $(\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}}, \mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}}) \leftarrow \mathsf{SplitSpl}(\mathsf{SK}, m^*)$*,* $(\sigma'_{\mathsf{one}}, \mathsf{VK}'_{\mathsf{one}}, \mathsf{SK}'_{\mathsf{abo}}, \mathsf{VK}'_{\mathsf{abo}}) \leftarrow \mathsf{SplitSpl}(\mathsf{SK}', m^*)$*. . It chooses a bit* b*. If* $b = 0$*, it sends* $(\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}}, \mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}})$ *to* $\mathcal{A}$*. Else, it sends* $(\sigma'_{\mathsf{one}}, \mathsf{VK}'_{\mathsf{one}}, \mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}})$ *to* $\mathcal{A}$*.*
3. $\mathcal{A}$ *sends its guess* b'*.*

$\mathcal{A}$ *wins if* $b = b'$*.*

In the game above, $\mathcal{A}$ is either given a system of $\sigma_{\mathsf{one}}, \mathsf{VK}_{\mathsf{one}}, \mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}}$ generated together by one call of $\mathsf{SetupSpl}$ or a "split" system of $(\sigma'_{\mathsf{one}}, \mathsf{VK}'_{\mathsf{one}}, \mathsf{SK}_{\mathsf{abo}}, \mathsf{VK}_{\mathsf{abo}})$ where the all but one keys are generated separately from the signature and key for the one message m^*. Since the correctness conditions do not link the behaviors for the all but one keys and the one message values, this split generation is not detectable by testing verification for the σ_{one} that $\mathcal{A}$ receives or for any signatures that $\mathcal{A}$ creates honestly by signing with $\mathsf{SK}_{\mathsf{abo}}$.

B.4 Security Properties of Iterators

Let $\mathsf{Itr} = (\mathsf{SetupItr}, \mathsf{ItrEnforce}, \mathsf{Iterate})$ be an interator with message space $\{0,1\}^\ell$ and state space $\S_\lambda$. We require the following notions of security.

Definition 30 (Indistinguishability of Setup)**.** *An iterator* Itr *is said to satisfy indistinguishability of Setup phase if any PPT adversary* $\mathcal{A}$*'s advantage in the security game* $\mathsf{Expt}_{\mathsf{Itr}}(1^\lambda, \mathcal{A})$ *at most is negligible in* λ*, where* $\mathsf{Expt}_{\mathsf{Itr}}$ *is defined as follows.*

$\mathsf{Expt}_{\mathsf{Itr}}(1^\lambda, \mathcal{A})$

1. *The adversary* $\mathcal{A}$ *chooses a bound* $T \in \Theta(2^\lambda)$ *and sends it to challenger.*
2. $\mathcal{A}$ *sends* k *messages* $m_1, \ldots, m_k \in \{0,1\}^\ell$ *to the challenger.*
3. *The challenger chooses a bit* b*. If* $b = 0$*, the challenger outputs* $(\mathsf{PP}_{\mathsf{Itr}}, v_0) \leftarrow \mathsf{SetupItr}(1^\lambda, T)$*. Else, it outputs* $(\mathsf{PP}_{\mathsf{Itr}}, v_0) \leftarrow \mathsf{ItrEnforce}(1^\lambda, T, 1^k, \vec{m} = (m_1, \ldots, m_k))$*.*
4. $\mathcal{A}$ *sends a bit* b'*.*

$\mathcal{A}$ *wins the security game if* $b = b'$*.*

Definition 31 (Enforcing)**.** *Consider any* $\lambda \in \mathbb{N}$*,* $T \in \Theta(2^\lambda)$*,* $k < T$ *and* $m_1, \ldots, m_k \in \{0,1\}^\ell$*. Let* $(\mathsf{PP}_{\mathsf{Itr}}, v_0) \leftarrow \mathsf{ItrEnforce}(1^\lambda, T, \vec{m} = (m_1, \ldots, m_k))$ *and* $v_j = \mathsf{Iterate}^j(\mathsf{PP}_{\mathsf{Itr}}, v_0, (m_1, \ldots, m_j))$ *for all* $1 \leq j \leq k$*. Then,* $\mathsf{Itr} = (\mathsf{SetupItr}, \mathsf{ItrEnforce}, \mathsf{Iterate})$ *is said to be* enforcing *if*

$$v_k = \mathsf{Iterate}(\mathsf{PP}_{\mathsf{Itr}}, v', m') \implies (v', m') = (v_{k-1}, m_k).$$

Note that this is an information-theoretic property.