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

Prabhanjan Ananth[1(✉)], Abhishek Jain[2], and Amit Sahai[1]

[1] Center for Encrypted Functionalities and Department of Computer Science, UCLA, Los Angeles, USA
{prabhanjan,sahai}@cs.ucla.edu
[2] Johns Hopkins University, Baltimore, USA
abhishek@cs.jhu.edu

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 (Re-randomizable encryption schemes can be instantiated under standard assumptions such as

The full version of this paper can be found in [6].

Work done in part while visiting the Simons Institute for Theoretical Computer Science, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467.

P. Ananth—This work was partially supported by grant #360584 from the Simons Foundation and the grants listed under Amit Sahai.

A. Jain—Supported in part by a DARPA/ARL Safeware Grant W911NF-15-C-0213 and NSF CNS-1414023.

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

J.-S. Coron and J.B. Nielsen (Eds.): EUROCRYPT 2017, Part III, LNCS 10212, pp. 127–155, 2017.
DOI: 10.1007/978-3-319-56617-7_5

DDH, LWE.) 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.

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 [10,41]. In the recent years, this research area has seen renewed activity with the emergence of candidate constructions [30] 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'.

- $\mathsf{AppPatch}\,(\langle M\rangle, \langle P\rangle)$ taking as input an obfuscated program $\langle M\rangle$ and a patch encoding $\langle P\rangle$, and outputting an obfuscated patched program $\langle M' = P(M)\rangle$.
- $\mathsf{Eval}\,(\langle M\rangle, x)$, 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 $|\langle P\rangle| = \mathrm{poly}(|P|, \lambda)$, 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 ($i\mathcal{O}$) [10] as a natural choice for the base notion. Indeed, over the last few years, several general-purpose candidate constructions, (for example: [9,22,30]) for $i\mathcal{O}$ have been proposed, and no impossibility results are known. Furthermore, it was shown by [39] 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., [18,28,30,51]) 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 [3].

In contrast, powerful (base) notions such as virtual black-box obfuscation [10] and differing-inputs obfuscation [1,10,20] have been shown to be impossible to realize for general functions [10,13,15,31,36]. This, in turn, means that patchable analogues of these notions are also impossible, in general. The notion of virtual grey-box obfuscation [14,16] 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 [34,38,40] or generic model [9,22]) 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-iO* 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 Sect. 2 for further details.

Implications of *pa-iO*. We view *pa-iO* as a powerful primitive that is likely to have several applications in the future. To see the power of *pa-iO*, it is instructive to first compare it with $i\mathcal{O}$. While $i\mathcal{O}$ exists if **P**=**NP**,[1] we show that multi-program *pa-iO* for parallel patches implies secret-key functional encryption (FE) [19,49,50]. 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-iO*.[2] As we discuss later, the above basic idea can, in fact, be easily extended to multi-input functional encryption [35], yielding new results.

Alternate Viewpoint: Obfuscation with Private Homomorphism. Another way of looking at our notion of *pa-iO* 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-iO* and the notion of fully homomorphic encryption [33] share a similarity in that they both require a form of compactness for the notions to be non-trivial.

[1] Assuming **NP** $\neq$ **co-RP**, it was shown that $i\mathcal{O}$ implies one-way functions [43,48].

[2] 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-iO*.

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 [2,24] 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 [7,37] 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-iO* as a replacement for *iO* in cryptographic applications.

Multi-input FE for Unbounded Arity Functions: We first show that multi-program *pa-iO* for parallel updates implies secret-key multi-input functional encryption (MIFE) [4,21,35] for unbounded arity functions. This implication follows from a straightforward extension of the *pa-iO* to (single-input) FE implication discussed earlier.

Theorem 5 (Unbounded-Arity MIFE). *Adaptively secure multi-program pa-iO 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 *iO* for circuits, sub-exponentially secure DDH. Previously, this result was only known [8] from a knowledge assumption, namely public-coin differing-input obfuscation [42] 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-iO 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 [7] based on *iO*. We emphasize that our construction from multi-program *pa-iO* is extremely simple, in contrast to the involved construction of [7].

We now discuss implications of *pa-iO* to *iO* for TMs. We first recall that all recent progress on achieving *iO* for TMs/RAMs [17,25–27,44] from *iO* 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 *iO* for circuits to *iO* for Turing Machines without any input length restriction from *iO* for circuits – this remains a major open question. Achieving *iO* for Turing Machines without any input length restriction currently requires strong assumption such as output-compressing randomized encodings [45] or knowledge-type assumptions such as public-coin *diO* [1,20,42]. We do not know how to achieve these objects using only *iO* for circuits.

iO for TMs with Unbounded Length Inputs: So far, in our definition of *pa-iO*, we have only considered "single-use" patches. More accurately, in our definition of single-program (resp., multi-program) *pa-iO* 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 *iO* 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 Sect. 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[3] 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 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 $M(x)$.

[3] 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.

– 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 [2,24].

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 [1,20,42]. 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 [17,26,44] 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 [44] to address these challenges. We recall the Turing machine randomized encodings[4] construction by [44].

The core idea in the randomized encodings construction of Koppula et al. [44] 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. [5], 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 into the details of [5] to apply it in the context of patching, we undertake a more modular approach. First, we propose an intermediate primitive

[4] 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.

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 [5]. 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 Sect. 3.2. 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 Sect. 3.
2. Next, we show how to obtain single-program patchable iO from splittable iO. This is shown in Sect. 4. 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 presented in the full version.
4. In the next step, we give a transformation from single-program to multi-program patchable iO. This is presented in the full version.
5. In the last step, we instantiate splittable iO using the framework of [5]. This is presented in the full version.

1.3 Related Work: Incremental Cryptography

The area of incremental cryptography was pioneered by Bellare et al. [11]. Subsequently, this concept of incremental updates has been studied for various standard primitives such as encryption schemes, signature schemes and so on [12,23,29,46,47]. We remark that none of these works handled the setting of arbitrary updates.

In a concurrent and independent work, [32] 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-*$i\mathcal{O}_{\mathsf{sp}}$ only satisfies the first property. In the full version, 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) $\left(\langle M_0 \rangle, \mathsf{st}_0\right) \leftarrow \mathsf{Obf}(\mathsf{SK}, M_0)$, (c) $\left(\langle P_i \rangle, \mathsf{st}_i\right) \leftarrow \mathsf{GenPatch}(\mathsf{SK}, P_i, \mathsf{st}_{i-1})$, (d) $\langle M_i \rangle \leftarrow \mathsf{AppPatch}\left(\langle M_{i-1} \rangle, \langle P_i \rangle\right)$.
- **Update**: $M_i \leftarrow \mathsf{Update}(M_{i-1}, P_i)$.

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

Remark 1. For the case of parallel patching, we require that $\langle M_i \rangle \leftarrow \mathsf{AppPatch}\left(\langle M_0 \rangle, \langle P_i \rangle\right)$ 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 $\left((P_0^1, P_1^1), \ldots, (P_0^L, P_1^L)\right)$ 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*-$\mathsf{i}\mathcal{O}_{\mathsf{mp}}$. Informally speaking, *pa*-$\mathsf{i}\mathcal{O}_{\mathsf{mp}}$ 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.[5] 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

[5] 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.

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.

Efficiency. Similar to *pa-i*$\mathcal{O}_{\mathsf{sp}}$, we define two efficiency properties for *pa*-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*-i$\mathcal{O}_{\mathsf{mp}}$ presented in the full version 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 $\left((M_{0,0}^1, M_{0,1}^1), \ldots, (M_{0,0}^Q, M_{0,1}^Q)\right)$.
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 $\left\{\langle M_{0,b}^j \rangle\right\}_{j\in[Q]}$ 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.*

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

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

3 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.

3.1 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 GenPatch. Algorithm 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, $|\widetilde{P}| = \mathrm{poly}(\lambda, |P|)$, 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 3. *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.

In the full version, we present an instantiation of the above primitive using fully homomorphic encryption.

3.2 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 [17,26,44]. 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 4 (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*:* $aux \leftarrow \mathsf{AuxGen}\left(sk_0, \mathcal{E}_{sk_0}(M), \mathcal{E}_{sk_1}(M)\right)$

  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 the full version.

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

4 Splittable iO to Single-Program *pa-iO*

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-iO* 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 4).

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 4. 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 the full version.

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

A formal proof for the above theorem can be found in the full version.

References

1. Ananth, P., Boneh, D., Garg, S., Sahai, A., Zhandry, M.: Differing-inputs obfuscation and applications. IACR Cryptology ePrint Archive 2013:689 (2013)
2. Ananth, P., Chen, Y.-C., Chung, K.-M., Lin, H., Lin, W.-K.: Delegating RAM computations with adaptive soundness and privacy. In: Hirt, M., Smith, A. (eds.) TCC 2016. LNCS, vol. 9986, pp. 3–30. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53644-5_1
3. Ananth, P., Jain, A., Naor, M., Sahai, A., Yogev, E.: Universal obfuscation and witness encryption: boosting correctness and combining security. In: CRYPTO (2016)
4. Ananth, P., Jain, A.: Indistinguishability obfuscation from compact functional encryption. In: CRYPTO (2015)
5. Ananth, P., Jain, A., Sahai, A.: Indistinguishability obfuscation with constant size overhead. Cryptology ePrint Archive, report 2015/1023 (2015)
6. Ananth, P., Jain, A., Sahai, A.: Patchable obfuscation. Cryptology ePrint Archive, report 2015/1084 (2015). http://eprint.iacr.org/2015/1084
7. Ananth, P., Sahai, A.: Functional encryption for turing machines. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9562, pp. 125–153. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49096-9_6
8. Badrinarayanan, S., Gupta, D., Jain, A., Sahai, A.: Multi-input functional encryption for unbounded arity functions. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 27–51. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48797-6_2
9. Barak, B., Garg, S., Kalai, Y.T., Paneth, O., Sahai, A.: Protecting obfuscation against algebraic attacks. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 221–238. Springer, Heidelberg (2014). doi:10.1007/978-3-642-55220-5_13
10. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S.P., Yang, K.: On the (im)possibility of obfuscating programs. J. ACM **59**(2), 6 (2012)
11. Bellare, M., Goldreich, O., Goldwasser, S.: Incremental cryptography: the case of hashing and signing. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 216–233. Springer, Heidelberg (1994). doi:10.1007/3-540-48658-5_22

12. Bellare, M., Goldreich, O., Goldwasser, S.: Incremental cryptography and application to virus protection. In: Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, pp. 45–56. ACM (1995)
13. Bellare, M., Stepanovs, I., Waters, B.: New negative results on differing-inputs obfuscation. IACR Cryptology ePrint Archive 2016:162 (2016)
14. Bitansky, N., Canetti, R.: On strong simulation and composable point obfuscation. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 520–537. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14623-7_28
15. Bitansky, N., Canetti, R., Cohn, H., Goldwasser, S., Kalai, Y.T., Paneth, O., Rosen, A.: The impossibility of obfuscation with auxiliary input or a universal simulator. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 71–89. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44381-1_5
16. Bitansky, N., Canetti, R., Kalai, Y.T., Paneth, O.: On virtual grey box obfuscation for general circuits. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 108–125. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44381-1_7
17. Bitansky, N., Garg, S., Lin, H., Pass, R., Telang, S.: Succinct randomized encodings and their applications. In: STOC (2015)
18. Bitansky, N., Goldwasser, S., Jain, A., Paneth, O., Vaikuntanathan, V., Waters, B.: Time-lock puzzles from randomized encodings. In: ITCS (2016)
19. Boneh, D., Sahai, A., Waters, B.: Functional encryption: definitions and challenges. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 253–273. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19571-6_16
20. Boyle, E., Chung, K.-M., Pass, R.: On extractability obfuscation. In: Lindell, Y. (ed.) TCC 2014. LNCS, vol. 8349, pp. 52–73. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54242-8_3
21. Brakerski, Z., Komargodski, I., Segev, G.: From single-input to multi-input functional encryption in the private-key setting. In: EUROCRYPT (2016)
22. Brakerski, Z., Rothblum, G.N.: Virtual black-box obfuscation for all circuits via generic graded encoding. In: Lindell, Y. (ed.) TCC 2014. LNCS, vol. 8349, pp. 1–25. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54242-8_1
23. Buonanno, E., Katz, J., Yung, M.: Incremental unforgeable encryption. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 109–124. Springer, Heidelberg (2002). doi:10.1007/3-540-45473-X_9
24. Canetti, R., Chen, Y., Holmgren, J., Raykova, M.: Succinct adaptive garbled RAM. In: TCC (2016-B)
25. Canetti, R., Holmgren, J.: Fully succinct garbled RAM. In: ITCS (2016)
26. Canetti, R., Holmgren, J., Jain, A., Vaikuntanathan, V.: Indistinguishability obfuscation of iterated circuits and RAM programs. In: STOC (2015)
27. Chen, Y.-C., Chow, S.S.M., Chung, K.-M., Lai, R.W.F., Lin, W.-K., Zhou, H.-S.: Computation-trace indistinguishability obfuscation and its applications. In: ITCS (2016)
28. Cohen, A., Holmgren, J., Nishimaki, R., Vaikuntanathan, V., Wichs, D.: Watermarking cryptographic capabilities. In: STOC (2016)
29. Fischlin, M.: Incremental cryptography and memory checkers. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 393–408. Springer, Heidelberg (1997). doi:10.1007/3-540-69053-0_27
30. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: 54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, Berkeley, CA, USA, 26–29 October 2013, pp. 40–49. IEEE Computer Society (2013)

31. Garg, S., Gentry, C., Halevi, S., Wichs, D.: On the implausibility of differing-inputs obfuscation and extractable witness encryption with auxiliary input. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8616, pp. 518–535. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44371-2_29
32. Garg, S., Pandey, O.: Incremental program obfuscation. Cryptology ePrint Archive, report 2015/997 (2015). http://eprint.iacr.org/
33. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC, Bethesda, MD, USA, 31 May–2 June 2009, pp. 169–178. ACM (2009)
34. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996)
35. Goldwasser, S., Gordon, S.D., Goyal, V., Jain, A., Katz, J., Liu, F.-H., Sahai, A., Shi, E., Zhou, H.-S.: Multi-input functional encryption. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 578–602. Springer, Heidelberg (2014). doi:10.1007/978-3-642-55220-5_32
36. Goldwasser, S., Kalai, Y.T.: On the impossibility of obfuscation with auxiliary input. In: Proceedings of 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), Pittsburgh, PA, USA, 23–25 October 2005, pp. 553–562 (2005)
37. Goldwasser, S., Kalai, Y.T., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: How to run turing machines on encrypted data. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 536–553. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40084-1_30
38. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: One-time programs. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 39–56. Springer, Heidelberg (2008). doi:10.1007/978-3-540-85174-5_3
39. Goldwasser, S., Rothblum, G.N.: On best-possible obfuscation. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 194–213. Springer, Heidelberg (2007). doi:10.1007/978-3-540-70936-7_11
40. Goyal, V., Ishai, Y., Sahai, A., Venkatesan, R., Wadia, A.: Founding cryptography on tamper-proof hardware tokens. In: Micciancio, D. (ed.) TCC 2010. LNCS, vol. 5978, pp. 308–326. Springer, Heidelberg (2010). doi:10.1007/978-3-642-11799-2_19
41. Hada, S.: Zero-knowledge and code obfuscation. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 443–457. Springer, Heidelberg (2000). doi:10.1007/3-540-44448-3_34
42. Ishai, Y., Pandey, O., Sahai, A.: Public-coin differing-inputs obfuscation and its applications. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. LNCS, vol. 9015, pp. 668–697. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46497-7_26
43. Komargodski, I., Moran, T., Naor, M., Pass, R., Rosen, A., Yogev, E.: One-way functions and (im)perfect obfuscation. In: 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS, Philadelphia, PA, USA, 18–21 October 2014, pp. 374–383 (2014)
44. Koppula, V., Lewko, A.B., Waters, B.: Indistinguishability obfuscation for turing machines with unbounded memory. In: STOC (2015)
45. Lin, H., Pass, R., Seth, K., Telang, S.: Output-compressing randomized encodings and applications. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9562, pp. 96–124. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49096-9_5
46. Micciancio, D.: Oblivious data structures: applications to cryptography. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pp. 456–464. ACM (1997)

47. Mironov, I., Pandey, O., Reingold, O., Segev, G.: Incremental deterministic public-key encryption. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 628–644. Springer, Heidelberg (2012). doi:10.1007/978-3-642-29011-4_37
48. Moran, T., Rosen, A.: There is no indistinguishability obfuscation in pessiland. IACR Cryptology ePrint Archive 2013:643 (2013)
49. O'Neill, A.: Definitional issues in functional encryption. IACR Cryptology ePrint Archive 2010:556 (2010)
50. Sahai, A., Waters, B.: Fuzzy identity-based encryption. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 457–473. Springer, Heidelberg (2005). doi:10.1007/11426639_27
51. Sahai, A., Waters, B.: How to use indistinguishability obfuscation: deniable encryption, and more. In: Symposium on Theory of Computing, STOC 2014, New York, NY, USA, 31 May–03 June 2014, pp. 475–484 (2014)