Patchable Obfuscation

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

Abstract

In this work, we introduce *patchable 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 the setting where a patch is applied not just to a single machine M, but to an unbounded set of machines $(M_1, \ldots, M_t)$ to yield $(P(M_1), \ldots, P(M_t))$. We call this *multi-program patchable obfuscation*.

We consider both patchable obfuscation and multi-program patchable 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 one-way functions imply patchable obfuscation; and we show that sub-exponentially secure $i\mathcal{O}$ for circuits, sub-exponentially secure one-way functions, and sub-exponentially secure DDH imply multi-program patchable obfuscation.

Finally, we exhibit some simple applications of multi-program patchable obfuscation, to demonstrate how these concepts can be applied.

[*]University of California Los Angeles and Center for Encrypted Functionalities. Email: `prabhanjan@cs.ucla.edu`. Work done in part while visiting the Simons Institute for Theoretical Computer Science, supported by the Simons Foundation and and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467. This work was partially supported by grant #360584 from the Simons Foundation and the grants listed under Amit Sahai.

[†]Johns Hopkins University. Email: `abhishek@cs.jhu.edu`. Work done in part while visiting the Simons Institute for Theoretical Computer Science, supported by the Simons Foundation and and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467.. Supported in part by a DARPA/ARL Safeware Grant W911NF-15-C-0213 and NSF CNS-1414023.

[‡]University of California Los Angeles and Center for Encrypted Functionalities. Email: `sahai@cs.ucla.edu` Work done in part while visiting the Simons Institute for Theoretical Computer Science, supported by the Simons Foundation and and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467. Research supported in part from a DARPA/ARL SAFEWARE award, NSF Frontier Award 1413955, NSF grants 1228984, 1136174, 1118096, and 1065276, a Xerox Faculty Research Award, a Google Faculty Research Award, an equipment grant from Intel, and an Okawa Foundation Research Grant. This material is based upon work supported by the Defense Advanced Research Projects Agency through the ARL under Contract W911NF-15-C-0205. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense, the National Science Foundation, or the U.S. Government.

Contents

1 Introduction

Informally, the notion of indistinguishability obfuscation ($i\mathcal{O}$) requires that given two equivalent programs M_0 and M_1, it should be hard to distinguish $i\mathcal{O}(M_0)$ from $i\mathcal{O}(M_1)$. $i\mathcal{O}$ was introduced by [BGI$^+$12], and the first candidate construction was given by [GGH$^+$13]. Since then, $i\mathcal{O}$ has been used to achieve a large number of exciting applications, from deniable encryption [SW14] to functional encryption [GGH$^+$13, Wat15, ABSV15, AS16], from software watermarking [NW15, CHV15] to time-lock puzzles [BGG$^+$14], and much more.

At its core, $i\mathcal{O}$ allows for the hiding of secrets within software (that we will model as Turing Machines), while approximately preserving the *description length* of the software. The fact that the description length of an obfuscated program does not grow is critical to the non-triviality of $i\mathcal{O}$. In fact, replacing a software implementation of a function with an exponential-size lookup table would yield perfect obfuscation, but blow up the description length. In contrast, with $i\mathcal{O}$, it is possible to obfuscate a program where the description complexity of the underlying program only increases by a small constant multiplicative factor [AJS15] (see also [BV15] for the case of circuits). The central intellectual and theoretical focus of this paper is exploring this connection between obfuscation and the preservation of description complexity, in the context of software that *evolves* over time.

Indeed, software is rarely changeless. We typically alter software over time, with updates and patches causing the software to grow and vary, in response to both demands for greater or new functionality as well as the discovery of bugs that need to be fixed. Can the notion of $i\mathcal{O}$ adapt to deal with this reality? In this paper, we give an affirmative answer to this question by showing how to transform any $i\mathcal{O}$ scheme for circuits into one that approximately preserves the description complexity of software that changes over time.

Patchable Obfuscation. A trivial solution to obfuscating evolving software would be to simply apply $i\mathcal{O}$ afresh to each updated version of a particular program. But this conflicts with the central goal of maintaining the description complexity of the software. (Later, we will consider the case where multiple programs are simultaneously updated, where this problem will be even worse.) For example, suppose we start with a program M, and then we develop a small patch P such that $P(M)$ outputs our updated software M'. This general modeling, where a patch P can arbitrarily modify the original software M follows the same definitional principles underlying the chain rule for Kolmogorov complexity (see e.g. [LV13]). Thus, we stress that even if P is much smaller than M, the patched software $P(M)$ may modify the description of M almost everywhere, and in particular can also cause the software to grow in size[1]. Nevertheless, even if M' and M differ substantially, given that we already have an obfuscated version of M, the obfuscation of M' should only require roughly $|P|$ more bits to describe than the obfuscation of M. Simply re-applying a fresh $i\mathcal{O}$ to each patched version of the software would not achieve this goal, since the total size of all obfuscated software that the user has would be at least $|M| + |M'|$. Instead, we aim to obfuscate the patch P with an obfuscated encoding that grows only with the size of P, and not with the size of the original machine M.

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

[1]We contrast our modeling with the very recent independent work of [GP15] that considers a related problem called incremental obfuscation. We elaborate further in our related work section below.

- $\mathsf{Obfuscate}(M; r)$ 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 program P, and outputting an obfuscated patch $\langle P \rangle$, using a combination of the original randomness r and new randomness r'. The size of the obfuscated patch should not depend on the size of the original program M.
- $\mathsf{ApplyPatch}\Big(\langle M \rangle, \langle P \rangle\Big)$ taking as input an obfuscated program $\langle M \rangle$ and an obfuscated patch $\langle P \rangle$, and outputting an obfuscated patched program $\langle M' = P(M) \rangle$.
- $\mathsf{Evaluate}\Big(\langle M \rangle, x\Big)$, taking as input an obfuscated program $\langle M \rangle$ and an input x, and outputting the value $y = M(x)$.

From a security standpoint, informally, the essential requirement we want is that given two equivalent programs M_0 and M_1, and two patches P_0 and P_1 such that $P_0(M_0)$ and $P_1(M_1)$ are also equivalent, it should be the case that it is hard to distinguish the tuple of obfuscations $(\langle M_0 \rangle, \langle P_0 \rangle)$ from the tuple of obfuscations $(\langle M_1 \rangle, \langle P_1 \rangle)$. In other words, as long as patches yield equivalent programs, an adversary should not be able to distinguish among these patches. Beyond this basic requirement, the actual notion of security that we achieve goes beyond this modeling in two essential respects: **(1)** We consider adaptive security, where the adversary can posit pairs of patches adaptively; and **(2)** We consider an unbounded number of patches, that can cause the software to grow and change in an arbitrary and unbounded manner.

Multi-Program Patchable Obfuscation. We next consider the case where multiple programs are to be updated simultaneously, using a common patch. This situation arises quite commonly: Applications of obfuscation typically involve programs that have customized keys created individually for each user. In unobfuscated programs, such per-user customization of programs is typically achieved by having a common base program, but with per-user customized data files that the program accesses to obtain customized keys or other customized data. With obfuscated programs, however, this simple approach does not work because obfuscation requires that secret keys embedded within the program also be protected. Thus, obfuscated versions of customized programs must differ radically from one another. In this multi-program setting, we would like a *single* obfuscated patch to be able to modify all obfuscated customized programs simultaneously.

In other words, suppose we have a vector of obfuscated programs $(\langle M_1 \rangle, \langle M_2 \rangle, \ldots, \langle M_k \rangle)$. We would like to be able to release single obfuscated patch $\langle P \rangle$, such that the owner of each individual program M_i in the vector can update its obfuscated program $\langle M_i \rangle$ using the obfuscated patch $\langle P \rangle$ to yield a new obfuscated program $\langle P(M_i) \rangle$. And again, critically, the size of the obfuscated patch $\langle P \rangle$ should not depend on the size or number of originally obfuscated programs. We want to achieve this notion in the setting of adaptive security, with an unbounded number of patches that can cause to the software to grow and change in an arbitrary and unbounded manner.

Cryptography from Patchable Obfuscation. We see patchable obfuscation, and especially multi-program patchable obfuscation, as powerful primitives that are likely to have several applications in the future. Indeed, as initial evidence of this, we show that multi-program patchable obfuscation can be used in a simple way to build secret-key multi-input functional encryption [GGG+14, AJ15, BKS15] with *unbounded arity functions* – previously, this result was only known to be achievable [BGJS15] using stronger knowledge assumptions, namely public-coin $di\mathcal{O}$ [BCP14, ABG+13, IPS15] and one-way functions. We also show that multi-program patchable obfuscation implies secret-key functional encryption for Turing Machines with unbounded input [AS16] in a simple and intuitive manner.

Alternative viewpoint: Obfuscation with a private homomorphism. Another way of looking at our notion of patchable obfuscation is as a form of obfuscation that supports a kind of semi-private *homomorphism*: the production of the obfuscated patch is private – requiring secret information that was used to obfuscate the original program – although the application of the obfuscated patch 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 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 patchable obfuscation and the notion of fully homomorphic encryption [Gen09] share a similarity in that they both require a form of compactness for the notions to be non-trivial.

1.1 Our Results

In this work, we formalize the notions of patchable obfuscation and multi-program patchable obfuscation. We focus on the setting where programs to be obfuscated and patched are described as Turing Machines. In this setting, we obtain the following two main theorems:

Theorem 1 (Informal)**.** *Assuming the existence of sub-exponentially secure $i\mathcal{O}$ for circuits and sub-exponentially secure one-way functions, there exists an adaptively secure patchable obfuscation scheme with unbounded updates, for Turing Machines where the size of the obfuscation of a patch P is bounded by $poly(|P|, k, \ell)$, where k is a security parameter and ℓ is a bound on the input size to the patched program.*

Theorem 2 (Informal)**.** *Assuming the existence of sub-exponentially secure $i\mathcal{O}$ for circuits, sub-exponentially secure one-way functions, and sub-exponentially secure DDH, there exists an adaptively secure multi-program patchable obfuscation scheme with unbounded updates, for Turing Machines where the size of the obfuscation of a patch P is bounded by $poly(|P|, k, \ell)$, where k is a security parameter and ℓ is a bound on the input size to the patched program.*

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, and the adversary can specify these patches adaptively given all obfuscated programs and patches constructed earlier.

Upgrading Input Size. We note that all recent progress on achieving $i\mathcal{O}$ for Turing Machines [CHJV15, BGL$^+$15, KLW15] from $i\mathcal{O}$ for circuits has required a polynomial bound ℓ to be placed on the input 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. (If we could remove this restriction, then we would show how to bootstrap $i\mathcal{O}$ for circuits to $i\mathcal{O}$ for Turing Machines without any input length restriction from $i\mathcal{O}$ for circuits – this remains a major open question. Achieving $i\mathcal{O}$ for Turing Machines without any input length restriction currently requires apparently stronger assumptions, such as output-compressing randomized encodings [LPST16] or knowledge-type assumptions such as public-coin $di\mathcal{O}$ [BCP14, ABG$^+$13, IPS15]. We do not know how to achieve these objects using only $i\mathcal{O}$ for circuits.)

However, our result allows for the input bound ℓ to be *upgraded* by patches. That is, for example, if we obfuscate a Turing Machine M and limit the input length to ℓ, but then later we decide that we need to increase this limit to $L > \ell$, then we can issue a patch of size $poly(k, L)$, independent of $|M|$, to upgrade the input size restriction on the obfuscation of M.

Implications of Patchable Obfuscation. It is not difficult to see that patchable obfuscation significantly extends $i\mathcal{O}$. Indeed, while $i\mathcal{O}$ exists if **P**=**NP**, patchable obfuscation implies one-way functions: Intuitively, this is true because patchable obfuscation is representation-dependent. A patch P takes as input a concrete representation of the original machine M in order to update this representation. Thus, the initial patchable obfuscation of M must actually maintain an encoded form of the original representation of M. This can be seen more precisely as follows: Consider a program $M_{b,x}$ parametrized by two secret values (b, x). This program $M_{b,x}$ ignores its input, and simply outputs $\perp$ if $b = 0$, and outputs x if $b = 1$. Note that if $b = 0$, then $M_{b,x}$ is just the all-$\perp$ function, and therefore the initial obfuscation of $M_{b,x}$ must hide x. However, if a future patch P sets the bit b to 1, then the patched program must output x. Therefore the initial obfuscation of $M_{b,x}$ cannot "forget" x. Thus, we have that the function that maps (b, x, r) to $\mathsf{Obfuscate}(M_{b,x}; r)$ must be a one-way function.

The simple intuition behind the implication above gives a glimpse of the power of our notions of patchable obfuscation and multi-program patchable obfuscation. We elaborate on this by giving two example applications of multi-program patchable obfuscation, that follow the same simple intuition described above.

Theorem 3 (Informal)**.** *Adaptively secure multi-program patchable obfuscation implies secret-key functional encryption for Turing Machines with unbounded input with indistinguishability security against adaptive post-ciphertext key queries.*

A construction of functional encryption for Turing machines with unbounded input was recently given by [AS16] based on $i\mathcal{O}$. We note, however, that our construction from multi-program patchable obfuscation is quite simple, in contrast to the involved construction of [AS16].

Theorem 4 (Informal)**.** *Adaptively secure multi-program patchable obfuscation implies secret-key multi-input functional encryption for unbounded arity functions with indistinguishability security adaptive post-ciphertext key queries.*

Combining the above with Theorem 2, we obtain secret-key multi-input functional encryption [GGG$^+$14, AJ15, BKS15] for unbounded arity functions from sub-exponentially secure $i\mathcal{O}$ for circuits, sub-exponentially secure one-way functions, and sub-exponentially secure DDH. This result was not previously known to be achievable from sub-exponentially secure $i\mathcal{O}$ for circuits combined with standard cryptographic assumptions. In particular, this result was only known [BGJS15] using stronger knowledge assumptions, namely public-coin $di\mathcal{O}$ [BCP14, ABG$^+$13, IPS15] and one-way functions.

Both these implications follow in very simple and intuitive ways. We expect several more applications of patchable obfuscation and multi-program patchable obfuscation in the future.

Related Independent Work. In a very recent independent work, [GP15] consider a related notion called *incremental obfuscation*. In incremental obfuscation, individual bits of an existing obfuscated program can be updated one-by-one. Like our setting, these incremental updates should be themselves small. However, our work and theirs differ in several essential ways: **(1)** The patches P that we consider can be themselves small, but they can modify the original program M almost everywhere. For example, a patch P could execute a "Find-And-Replace" fixing all instances of a bug within the original program. In the setting of [GP15], incremental changes can only modify individual bits or words in each update, and so the type of patches that we envision would require multiple incremental updates, which would destroy the description length preservation property that is our central aim. On the other hand, this incremental nature of [GP15] allow them to achieve a level of runtime efficiency that is impossible in our setting, since the processing performed by a patch P in our setting could itself take a long time. **(2)** We consider patches that can *grow* the size of the underlying program. [GP15] do not address this case and mention it as a topic for future work. Indeed, this feature of program growth is central to our applications of patchable obfuscation discussed above. **(3)** Our techniques achieve the notion of *multi-program* patchable obfuscation, a setting that is not considered in [GP15]. **(4)** Our work allows for the upgrading of input sizes for obfuscated programs through patching, something not considered in [GP15]. **(5)** Finally, our work achieves security in the *adaptive* setting, where the adversary can specify patches based on the obfuscated programs and patches he receives. The work of [GP15] considers non-adaptive setting.

Given these basic differences, it is unsurprising that the techniques in our work and that of [GP15] diverge at a basic level. In our work, the idea that patchable obfuscation should hide what a patch does is central to our construction from the start (see Our Techniques below for further elaboration). In contrast, the work of [GP15] first constructs incremental obfuscation that does *not* hide what individual updates do, and then uses ORAM techniques to bootstrap from non-hiding incremental obfuscation to hiding incremental obfuscation. We do not make use of ORAM techniques in any way.

1.2 Our Techniques

Our constructions and proofs of security for patchable obfuscation and multi-program patchable obfuscation are quite involved and include multiple technical layers. In this section, we focus on describing some of the main technical barriers we encounter, and some of our key conceptual ideas for overcoming these technical barriers. For a more technical guide to the different components of our construction and proof, please see Section 3.

The input-size barrier. Let us jump straight into the question of how we can obfuscate a patching program P so that this program can then act on and modify the description of a machine M that underlies an initial obfuscated program $\langle M \rangle$. An immediate natural approach would be to build a program P' that first "decrypts" the obfuscated program $\langle M \rangle$ to obtain M, then executes $P(M)$ to obtain the new description M', and then re-obfuscates it to produce $\langle M' \rangle$. Then, we could obfuscate this new program P' and release it as our patch. This approach fails, and the reason it fails highlights one of the major technical barriers we face: All known methods for bootstrapping $i\mathcal{O}$ for circuits to $i\mathcal{O}$ for Turing Machines imposes a limit on the input size for the obfuscated Turing Machine [CHJV15, BGL$^+$15, KLW15]. Indeed, this is for a fundamental reason: the only way we know how to argue that the obfuscation is secure is by applying complexity leveraging, to argue security on an input-by-input basis [GLSW15]. But the program P' that we considered here takes an enormously long input $\langle M \rangle$, and we cannot afford to pay the cost associated with obfuscating a machine that accepts such a long input. This problem grows even worse in the multi-program patchable obfuscation setting.

Decoupling machine and input. Our first conceptual idea to tackle this barrier is to decouple the complexity leveraging needed for arguing input-by-input security from the actual underlying encoding of the machine M. Taking a step back, if we look at the work of [KLW15] on $i\mathcal{O}$ for Turing Machines, their construction works by obfuscating a program that takes an input x, and outputs a (succinct) randomized encoding of the pair (M, x). From our perspective, the disadvantage of this approach is that it ties together the encoding of the machine M and the input x. In contrast, we would like to split apart these components: We would like to have one fixed encoding of the machine M, and an obfuscated program that takes an input x, and outputs an encoding of x that is compatible with the fixed encoding of M that we already have. Moreover, the encoding of M must be amenable to processing by patches. Specifically, what we build as a first step is new notion of *patchable attribute-based encryption* that allows for this decoupling.

Patching at the machine encoding level. Once we have achieved this decoupling, our next main conceptual idea is to move the application of the patch P down one level of abstraction, such that an encoding of the patch P can operate directly on the encoding of the machine M, minimizing (but not completely eliminating) its interaction with the obfuscated program that encodes inputs x. Technically, making this possible is the most involved and technically complex aspect of our work.

Applying patches. The first difficulty is just to achieve correctness – to encode the patch P in a way that is compatible with the encoding of the machine M, so that P can be executed on the machine M to apply the patch. This difficulty is more severe in the case of multi-program patchable obfuscation, where the encoding of P must be compatible with a potentially unbounded number of unknown machines M_i. For example, following [CHJV15, BGL$^+$15, KLW15], during the execution of the patch $P(M)$, a signature on the resulting intermediate values will need to be updated. In the single-program patchable obfuscation setting, the patch generator can anticipate what the final signature will be, since it is aware of the machine M being modified and therefore can anticipate what $P(M)$ will be. In the multi-program setting, however, the patch generator has no idea what are the underlying machines M_i on which the patch will need to be executed. Thus, in this setting we need a secure method for updating signatures that can work across an unbounded number of patches. The key conceptual step for achieving this is to build a *stateless* procedure for applying patches, where any and all information needed for applying patches is found in the encoded version of the machine being patched.

Proving Security. Beyond merely achieving correctness, we also need our method of applying patches to enable an actual proof of security. Our proofs of security for both patchable obfuscation and multi-program patchable obfuscation will reduce the security of our constructions to the security of the message-hiding encodings from [KLW15]. We stress that we do not make any vague claims about what [KLW15] achieves. Instead, following [AJS15], we only use the theorem from [KLW15] guaranteeing the security of message-hiding encodings – we do not need the more advanced theorem from [KLW15] about secure machine-hiding encodings. Our proofs of security span several layers of abstractions. But at their heart, the central challenge we overcome in our proofs is to allow for sufficient "programmability" to remain while patching is taking place, so that security can be established. To illustrate the challenge, we note that we can start with two programs (M_0, M_1) and then have the adversary specify two sequence of patches $(P_0^1, P_0^2, \ldots, P_0^t)$ and $(P_1^1, P_1^2, \ldots, P_1^t)$. Now we know that for all i, we have that $P_0^i(P_0^{i-1}(\cdots(P_0^1(M_0))\cdots))$ is equivalent to $P_1^i(P_1^{i-1}(\cdots(P_1^1(M_1))\cdots))$. However, we cannot mix and match the patches P_0^j and P_1^j. What this means is that we cannot have hybrids where we switch the patches one-at-a-time from P_0^j to P_1^j. Thus, we need a proof strategy that work across *all* patches – and this technique has to work with an adaptively chosen set of an *unbounded* number of patches. Achieving this involves several technical ideas across our proof. For example, we show that at the abstraction layer of patchable attribute-based encryption, we can actually reduce security to the one-time setting without any patches, because of the specific structure of our construction that allows us to reinterpret patched obfuscations as obfuscated machines. Another example occurs at another abstraction layer, where the key technical challenge is to pass forward secret information that is correlated with public values across a sequence of patches while proving that the adversary cannot derive any advantage despite the correlation that exists with public values. These ideas are presented against the backdrop of suitably modified abstractions recently introduced in [AJS15]. However, we note that every construction we present across these abstractions involves new ideas that are critical because no notion of patching was envisioned or anticipated in [AJS15].

2 Preliminaries

We assume familiarity of the reader with standard cryptographic notions.

2.1 Turing Machines

We describe syntax and terminology related to Turing machines that is used throughout the paper.

Turing machines. A Turing machine is a 7-tuple $M = \langle Q, \Sigma_{\text{tape}}, \Sigma_{\text{inp}}, \delta, q_0, \perp, q_{\text{acc}}, q_{\text{rej}} \rangle$ where Q and Σ_{tape} are finite sets with the following properties:

1. Q is the set of finite states.
2. Σ_{inp} is the set of input symbols.
3. Σ_{tape} is the set of tape symbols.
4. $\perp$ denotes the blank symbol.
5. $\delta : Q \times \Sigma_{\text{tape}} \to Q \times \Sigma_{\text{tape}} \times \{+1, -1\}$ is the transition function.
6. $q_0 \in Q$ is the start state.
7. $q_{\text{acc}} \in Q$ is the accept state.
8. $q_{\text{rej}} \in Q$ is the reject state, where $q_{\text{acc}} \neq q_{\text{rej}}$.

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

Theorem 5. *Any Turing machine M running in time at most $T(n)$ for all inputs of size n, can be transformed into a circuit $C_M : \{0,1\}^n \rightarrow \{0,1\}$ such that (i) $C_M(x) = M(x)$ for all $x \in \{0,1\}^n$, and (ii) the size of C_M is $|C_M| = O\big((T(n))^2\big)$. We denote this transformation procedure as* $\mathsf{TMtoCKT}$.

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

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

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

2.2 Patching Turing Machines

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

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

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

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

2.3 Puncturable Pseudorandom Functions

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

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

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

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

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

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

Theorem 6 ([GGM86, BW13, BGI14, KPTZ13])**.** *If one-way functions exist, then for all polynomials $\eta(\lambda)$ and $\chi(\lambda)$, there exists a puncturable PRF family that maps $\eta(\lambda)$ bits to $\chi(\lambda)$ bits.*

2.4 Indistinguishability Obfuscation

The notion of indistinguishability obfuscation (iO), first conceived by Barak et al. [BGI+12], guarantees that the obfuscation of two circuits are computationally indistinguishable as long as they both are equivalent circuits, i.e., the output of both the circuits are the same on every input. Analogous to the case of circuits, we can define indistinguishability obfuscation for Turing machines (TMs). We work in a weaker setting of iO for TMs, as considered by the recent works [CHJV15, BGL+15, KLW15, AJS15], where the inputs to the TM are upper bounded by a pre-determined value. This definition of iO for TMs is referred as *succinct iO*. The security property of this notion states that the obfuscations of two machines M_0 and M_1 are computationally indistinguishable as long as $M_0(x) = M_1(x)$ and the time taken by both the machines on input x are the same, i.e., $\mathsf{RunTime}(M_0, x) = \mathsf{RunTime}(M_1, x)$.

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

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

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

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

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

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

iO for Turing Machines. Analogous to the case of circuits, we can define indistinguishability obfuscation for Turing machines (TMs). The security property states that the obfuscations of two Turing machines M_0 and M_1 are computationally indistinguishable as long as $M_0(x) = M_1(x)$. Note that by our convention adopted for Turing machines, the condition that $M_0(x) = M_1(x)$ already ensures that the running time of $M_0(x)$ and $M_1(x)$ are the same. The succinctness property states that the running time of the obfuscation algorithm on input M is independent of the worst case running time of machine M. The same guarantee also holds for the evaluation of the obfuscated TM. We note that this definition was adopted in the works of [BGL$^+$15, CHJV15, KLW15, AJS15].

Definition 2 (Succinct iO). *A uniform PPT algorithm* SuccIO *is called an succinct indistinguishability obfuscator for a class of Turing machines* $\{\mathcal{M}_\lambda\}_{\lambda\in\mathbb{N}}$ *with an input bound* L*, if the following holds:*

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

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

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

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

2.5 Garbled TMs with Persistent Memory

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

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

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

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

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

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

Remark 1. *Previous works considered definitions, where the algorithms also take as input a space bound. In our setting, we set the space bound of the computations to be 2^λ and hence the space bound does not explicitly feature in the definitions.*

We require that the above scheme satisfy the following properties.

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

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

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

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

2.5.1 Security

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

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

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

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

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

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

Definition 3 (Adaptive GTM with Persistent Memory)**.** *A GTM with persistent memory* GTM *is said to be adaptively secure if for every PPT adversary* $\mathcal{A}$*, we have* $|\mathsf{Pr}[1 \leftarrow \mathsf{Ad.Expt}_{\mathcal{A}}^{\mathsf{GTM},\mathsf{Sim}}(1^\lambda)]| \leq \frac{1}{2} + \mathsf{negl}(\lambda)$.

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

Definition 4 (Selective GTM with Persistent Memory)**.** *A GTM with persistent memory* GTM *is said to be selectively secure if for every PPT adversary* $\mathcal{A}$*, we have* $|\mathsf{Pr}[1 \leftarrow \mathsf{Sel.Expt}_{\mathcal{A}}^{\mathsf{GTM},\mathsf{Sim}}(1^\lambda)]| \leq \frac{1}{2} + \mathsf{negl}(\lambda)$.

Feasibility. The existence of (selectively-secure) garbled TMs with persistent memory was explored in the work of [CH15, CCC+15]. Recently, the works of [CCHR15, ACC+15] show the existence of adaptively secure garbled TMs with persistent memory. Their constructions are based on the existence of $\frac{\epsilon}{2^\lambda}$-secure indistinguishability obfuscation and $\frac{\epsilon'}{2^\lambda}$-secure decisional Diffie-Hellman (DDH) assumption, where λ is the security parameter and $\epsilon, \epsilon' \leq \frac{1}{p(\lambda)}$ for some fixed polynomial p. Here we emphasize that the security loss $\frac{\epsilon'}{2^\lambda}$, $\frac{\epsilon'}{2^\lambda}$ do not depend on either the size of the Turing machines or the input. We note that their construction is designed for the more general RAM model of computation, however it suffices for our work to just consider the Turing machine model of computation.

3 Technical Guide

We approach the problem of patchable obfuscation in a modular fashion. While almost every aspect of our construction uses new ideas, we build our constructions within the general framework of [AJS15] for building $i\mathcal{O}$ for Turing machines. While the primary focus of [AJS15] is on building $i\mathcal{O}$ with constant multiplicative overhead, we do not seek this efficiency goal in this work. Nevertheless, we find the framework in [AJS15] to be a very useful set of abstractions within which we can apply our ideas to achieve our goal of patchable obfuscation.

Below, we describe the main modular steps involved in our constructions of patchable obfuscation and multi-program patchable obfuscation. The details of all the steps are give in later sections.

Patchable Obfuscation. We construct patchable obfuscation (PO) in three main steps:

- **Step I: Patchable Attribute-based Encryption.** First, in Section 5, we consider a notion, termed as *patchable attribute-based encryption* (PABE). Roughly speaking, a PABE scheme is a 1-key ABE that additionally supports the ability to "patch" ABE keys generated by the scheme. That is, for any patch P, it should be possible to generate an encoding $\widetilde{P}$ s.t. an ABE key sk_M corresponding to a Turing machine M can be "updated" to obtain an ABE key $sk_{M'}$ where $M' = \mathsf{Update}(M, P)$.

  To achieve this primitive, we give a construction that allows us to reduce the security of PABE to a security game in which no patching takes place. Our construction of PABE achieves adaptive security.

- **Step II: PABE to Patchable Oblivious Evaluation Encodings.** Next, in Section 6, we consider the notion of oblivious evaluation encodings (OEE) that was recently introduced by [AJS15]. In an OEE scheme, it is possible to generate a joint encoding of two machines M_0, M_1 s.t. given an encoding of an input x with a bit b, the decoding algorithm returns $M_b(x)$. We extend the notion of OEE to patchable OEE (POEE) that allows for patching of the machine encodings. That is, we allow for computing encoding of patches P_0, P_1 that can be applied over the encoding of M_0, M_1 to obtain an encoding of M_0', M_1', where $M_b' = \mathsf{Update}(M_b, P_b)$ for every $b \in \{0, 1\}$.

  We build a POEE scheme (Section 6.1) from a PABE scheme. A key difficulty in building the POEE scheme is tracking secret information that must be available for patching. To achieve security, we need to use some specific properties of our PABE scheme in order to achieve our goal of POEE.

- **Step III: POEE to PO.** Finally, in Section 7, we present a generic transformation from a POEE scheme to a PO scheme. Our construction resembles the transformation from OEE to $i\mathcal{O}$ in [AJS15] with the crucial difference that we release a fresh input encoder along with every patch encoding.

Multi-program Patchable Obfuscation. We construct multi-program patchable obfuscation (MPO) in three main steps:

- **Step I: Stateless PABE.** First, in Section 9, we consider the notion of *stateless* PABE. This is a special class of PABE, where as the name suggests, no state is maintained during the patch generation process and in particular, the patch generation algorithm only takes as input the secret key and the patch. This is unlike our PABE scheme discussed above, where a private state is maintained during the patch generation process.

  To construct this primitive, we build upon the construction of PABE in Section 5.1 and along the way using adaptive garbled TMs with persistent memory (Definition 3). Our construction of stateless PABE also achieves adaptive security.

- **Step II: Stateless PABE to Multi-Program POEE.** Next, in Section 10, we generalize the notion of POEE to multi-program POEE (MPOEE). In an MPOEE scheme, the secret key can be used to produce TM encodings of multiple pairs of machines. The key requirement is that the patches issued should be applicable on *all* the machines. As an added feature, we also achieve a patch generation mechanism that does not maintain state.

We build upon the construction of POEE (Section 6.1), using additional layers, to achieve our goal of multi-program POEE (Section 10.1).

- **Step III: MPOEE to MPO.** Finally, in Section 11, we present a generic transformation from MPOEE to MPO. Our transformation is, in fact, identical to the one in the single program case. The main novelty here is in the security analysis.

4 Single-Program Patchable Obfuscation

In this section, we present a formal definition of (single-program) patchable obfuscation. As discussed in Section 3, we construct patchable obfuscation in three steps. We refer the reader to Sections 5, 6 and 7 for the details of these steps.

4.1 Syntax

A patchable obfuscation 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 $\mathsf{pO} = (\mathsf{Setup}, \mathsf{Obfuscate}, \mathsf{GenPatch}, \mathsf{ApplyPatch}, \mathsf{Evaluate})$ which are defined below.

- **Setup**, $\mathsf{Setup}(1^\lambda)$: It takes as input the security parameter λ and outputs the secret key $\mathsf{Obf.SK}$.

- **Obfuscate**, $\mathsf{Obfuscate}(\mathsf{Obf.SK}, M)$: It takes as input the secret key $\mathsf{Obf.SK}$ and a TM $M \in \mathcal{M}$. It outputs an obfuscated TM $\langle M \rangle$ along with state st.

- **Secure Patch Generation**, $\mathsf{GenPatch}(\mathsf{Obf.SK}, P, \mathsf{st})$: It takes as input the secret key $\mathsf{Obf.SK}$, a description of a patch $P \in \mathcal{P}$, and state st. It outputs a secure patch $\langle P \rangle$ along with the updated state st'.

- **Applying Patch**, $\mathsf{ApplyPatch}\Big(\langle M \rangle, \langle P \rangle\Big)$: It takes as input an obfuscated TM $\langle M \rangle$ and a secure patch $\langle P \rangle$. It outputs an updated obfuscation $\langle M_{new} \rangle$.

- **Evaluation**, $\mathsf{Evaluate}\Big(\langle M \rangle, x\Big)$: It takes as input an obfuscated TM $\langle M \rangle$ and an input x. It outputs a value y.

Correctness. At a high level, the correctness property states that executing Update on a TM M and a patch P is equivalent to executing $\mathsf{ApplyPatch}$ 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) $\mathsf{Obf.SK} \leftarrow \mathsf{Setup}(1^\lambda)$, (b) $\Big(\langle M_0 \rangle, \mathsf{st}_0\Big) \leftarrow \mathsf{Obfuscate}(\mathsf{Obf.SK}, M_0)$, (c) $\Big(\langle P_i \rangle, \mathsf{st}_i\Big) \leftarrow \mathsf{GenPatch}(\mathsf{Obf.SK}, P_i, \mathsf{st}_{i-1})$, (d) $\langle M_i \rangle \leftarrow \mathsf{ApplyPatch}\Big(\langle M_{i-1} \rangle, \langle P_i \rangle\Big)$.

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

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

Efficiency. We require that all the algorithms in pO are polynomial time in the security parameter. Further, we require that the size of each secure patche is just a polynomial in the size of the (insecure) patch and security parameter. That is, for any $\langle P \rangle \leftarrow \mathsf{GenPatch}(\mathsf{Obf.SK}, P, \mathsf{st})|$, we require that $|\langle P \rangle| = \text{poly}(\lambda, |P|)$ (and in particular independent of $|\mathsf{st}|$).

4.2 Indistinguishability-Based Security

We next give an indistinguishability (IND)-style definition for modeling the security of a patchable obfuscation scheme. In an IND-security definition, we consider a security game between the challenger and the adversary. We give a high level description of the game below.

In this game, the adversary sends two machines (M_0, M_1) to the challenger and in response receives an obfuscation $\langle M_b \rangle$, 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, P_1) and in turn gets the secure patch $\langle P_b \rangle$. The patch queries of the adversary are restricted in the following manner: suppose $\Big((P_0^1, P_1^1), \ldots, (P_0^L, P_1^L)\Big)$ is a sequence of adaptive patch queries made by the adversary. We require that the machine M_0^i is functionally equivalent with M_1^i, for every $i \in [L]$, where (a) $M_0^0 = M_0$, $M_1^0 = M_1$ and, (b) $M_0^i \leftarrow \mathsf{Update}(M_0^{i-1}, P_i)$ (resp., $M_1^i \leftarrow \mathsf{Update}(M_1^{i-1}, P_i)$). 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 the above notion. We proceed to formally defining this notion.

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

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

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

Definition 5 (Adaptive Security). *A patchable obfuscation scheme* pO *is said to be adaptively secure if for any PPT adversary* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *s.t.*

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

Selective security. We note that one could also define selective security for patchable obfuscation where the adversary makes all the patch queries at the beginning of the game. The formal definition follows in a similar manner as above. We omit the details.

5 Patchable Attribute-based Encryption

We start by describing the syntax for a patchable attribute-based encryption (PABE) scheme. We focus on the Turing machine model of computation.

Similar to a standard ABE scheme, a PABE scheme comes equipped with setup, key generation, encryption and decryption algorithms. However, unlike a standard ABE scheme, a PABE scheme also supports a "patching mechanism" for keys generated by the key generation algorithm. We consider two additional algorithms to capture his idea: (a) an algorithm for generating patches that takes as input an "insecure" patch P and generates a "secure" patch $\widetilde{P}$, (b) an algorithm for applying patches that takes as input a key sk_M for Turing machine M and outputs a key $sk_{M_{new}}$ for an updated Turing machine M_{new}, where $M_{new} \leftarrow \mathsf{Update}(M, P)$.

We refer the reader to Section 2.2 for a discussion on the Update algorithm. We proceed to formally define the syntax of the patchable ABE scheme below. We will focus on the single-key setting.

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

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

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

3. **Secure Patch Generation,** $\mathsf{PABE.GenPatch}(\mathsf{PABE.SK}, P, \mathsf{st})$: On input the secret key $\mathsf{PABE.SK}$, a description of a patch $P \in \mathcal{P}$, and state st, it outputs a secure patch $\widetilde{P}$ along with the updated public key $\mathsf{PABE.PP}'$ and the updated state st'.

4. **Applying Patch,** $\mathsf{PABE.ApplyPatch}(\mathsf{PABE}.sk_M, \widetilde{P})$: On input an ABE key $\mathsf{PABE}.sk_M$ and a secure patch $\widetilde{P}$, it outputs an updated ABE key $\mathsf{PABE}.sk_{M_{new}}$.

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

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

Correctness. We say that a PABE scheme is correct if for any Turing machine $M_0 \in \mathcal{M}$, every $L \geq 0$, patch sequence $(P_1, \ldots, P_L) \in \mathcal{P}^L$, every $x \in \{0,1\}^*$, and $\mathsf{msg} \in \mathsf{MSG}$,

$$\Pr\left[\mathsf{PABE.Dec}\left(\mathsf{PABE}.sk_{M_L}, \mathsf{PABE.CT}_{(x,\mathsf{msg})}\right) = \mathsf{msg} : M_L(x) = 1\right] = 1$$

where:

- $(\mathsf{PABE.PP}_0, \mathsf{PABE.SK}) \leftarrow \mathsf{PABE.Setup}(1^\lambda)$,
- $(\mathsf{PABE}.sk_{M_0}, \mathsf{st}_0) \leftarrow \mathsf{PABE.KeyGen}(\mathsf{PABE.SK}, M_0)$,
- $(\widetilde{P_i}, \mathsf{PABE.PP}_i, \mathsf{st}_i) \leftarrow \mathsf{PABE.GenPatch}(\mathsf{PABE.SK}, P_i, \mathsf{st}_{i-1})$,
- $\mathsf{PABE}.sk_{M_i} \leftarrow \mathsf{PABE.ApplyPatch}(\mathsf{PABE}.sk_{M_{i-1}}, \widetilde{P_i})$,
- $\mathsf{PABE.CT}_{(x,\mathsf{msg})} \leftarrow \mathsf{PABE.Enc}(\mathsf{PABE.PP}_L, x, \mathsf{msg})$,
- $M_j = \mathsf{Update}(M_{j-1}, P_j)$.

Remark 2. *Note that in the above definition, an updated key* $\mathsf{PABE}.sk_{M_i}$ *is only required to correctly decrypt ciphertexts* $\mathsf{PABE.CT}_{(x,\mathsf{msg})}$ *that are computed using the updated public-key* $\mathsf{PABE.PP}_i$.

Efficiency. We say that a PABE scheme satisfies efficiency property if $|\widetilde{P}| = \text{poly}(\lambda, |P|)$, where $(\widetilde{P}, \mathsf{PABE.PP}', \mathsf{st}') \leftarrow \mathsf{PABE.GenPatch}(\mathsf{PABE.SK}, P, \mathsf{st})$ and in particular independent of the size of st.

Security. We extend the security framework for ABE to our setting of patchable ABE. Since we only consider the single-key setting, the adversary is restricted to making one key query. However, we allow the adversary to submit a polynomial number of patch queries.

We consider adaptive security where the adversary submits both the challenge message pair as well as the key query at the beginning of the game itself but the patch queries are made adaptively. We also require the adversary to specify in advance an index $i \in [L]$ where L is the number of patch queries made by the adversary. The index i determines the updated public key that is then used to compute the challenge ciphertext sent to the adversary.

We formalize security in terms of the following security experiment between a challenger Ch and a PPT adversary $\mathcal{A}$.

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

1. $\mathcal{A}$ sends to Ch a tuple consisting of a Turing machine M_0, an attribute x, two messages $(\mathsf{msg}_0, \mathsf{msg}_1)$ and an index $\mathbf{i} \geq 0$.
2. Ch computes the following: (a) $(\mathsf{PABE.PP}_0, \mathsf{PABE.SK}) \leftarrow \mathsf{PABE}, \mathsf{Setup}(1^\lambda)$, (b) $(\mathsf{PABE}.sk_{M_0}, \mathsf{st}_0) \leftarrow \mathsf{PABE.KeyGen}(\mathsf{PABE.SK}, M_0)$. It sends $(\mathsf{PABE.PP}_0, \mathsf{PABE}.sk_{M_0})$ to $\mathcal{A}$.
3. The following is repeated polynomially many times:
   (a) $\mathcal{A}$ sends to Ch a patch $P_j \in \mathcal{P}$.
   (b) Ch computes the following: (a) $(\widetilde{P_j}, \mathsf{PABE.PP}_j, \mathsf{st}_j) \leftarrow \mathsf{PABE.GenPatch}(\mathsf{PABE.SK}, P_j, \mathsf{st}_{j-1})$, (b) $\mathsf{PABE.CT}_{(x,\mathsf{msg}_b)} \leftarrow \mathsf{PABE.Enc}(\mathsf{PABE.PP}_j, x, \mathsf{msg}_b)$ if $j = \mathbf{i}$.

(c) If $j = \mathbf{i}$, Ch sends $(\mathsf{PABE.PP}_j, \widetilde{P_j}, \mathsf{PABE.CT}_{(x,\mathsf{msg}_b)})$ to $\mathcal{A}$. If $j \neq \mathbf{i}$, Ch sends $(\mathsf{PABE.PP}_j, \widetilde{P_j})$ to $\mathcal{A}$.

4. Finally, the adversary outputs the bit b'.

5. The adversary wins the game if (a) $b = b'$, and (b) $M_{\mathbf{i}}(x) = 0$ where $M_j = \mathsf{Update}(M_{j-1}, P_j)$.

Definition 6. *A 1-key PABE scheme is said to be* **adaptively secure** *if for every PPT adversary $\mathcal{A}$, there exists a negligible function* negl *s.t.*

$$\mathsf{Pr}\Big[\mathcal{A} \textit{ wins } \mathsf{Expt}_{\mathcal{A}}^{\mathsf{PABE}}(1^\lambda, b)\Big] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

Remark 3. *Henceforth, we omit the term "adaptive" when referring to the security of ABE schemes.*

5.1 Construction

Overview. Our construction of 1-key PABE for TMs is built upon the message hiding encodings paradigm of KLW. Indeed, a message hiding encoding can be thought of as an ABE scheme where only one ciphertext is issued. But for our application, it is required that multiple ciphertexts are issued. Directly adopting KLW causes serious issues since the fact that only a single branch of computation exists is crucially used in the proof of security of KLW. To do this, we adopt a signature synchronization mechanism that was recently used in a different context in [AJS15]. The result of adopting this mechanism is that our ABE key is just a machine M and a signature σ (computed on $f(M)$, which is "short"). Now to update a machine M with patch P, all we have to do is to compute a new signature σ' on $f(M')$, where M' is a new machine. So our secure patch is just (P, σ')! Using this information, a key for M' can be recovered.

While the structural properties of the ABE scheme developed by AJS already yields us a patchable ABE construction, the security analysis is more involved. Unlike AJS, in the security proof we have to contend with the fact that there exists different ABE keys that are correlated with each other. And in particular, we have to prevent "mix-and-match" attacks. It should not be possible to combine ciphertexts computed using i^{th} updated public key with an attribute key updated at a different time. Nonetheless, we observe some crucial properties satisfied by our construction that enable us to prove security in the presence of patches. We now delve into the technical details.

We import the tools of storage accumulators, splittable signatures and iterators from the work of KLW. The formal definitions of these primitives are provided in Section 2. The primitive of storage accumulators enables computing a short value that represents the entire memory in such a way that it information theoretically binds one particular location (specified in advance) but computationally binds the entire memory. Splittable signatures are signature schemes that allow splitting of signature keys into two "constrained" keys such that each key has capability to sign only one partition of the message space. Lastly, iterators are used to bind the state information.

We denote the positional accumulator scheme we use by $\mathsf{Acc} = (\mathsf{SetupAcc}, \mathsf{EnforceRead}, \mathsf{EnforceWrite}, \mathsf{PrepRead}, \mathsf{PrepWrite}, \mathsf{VerifyRead}, \mathsf{WriteStore}, \mathsf{Update})$. It is associated with the message space Σ_{tape} with accumulated value of size ℓ_{Acc} bits. The iterator scheme we use is denoted by $\mathsf{Itr} = (\mathsf{SetupItr}, \mathsf{ItrEnforce}, \mathsf{Iterate})$. It is associated with the message space $\{0,1\}^{2\lambda+\ell_{\mathsf{Acc}}}$ with iterated value of size ℓ_{Itr} bits. We denote the splittable signatures scheme by $\mathsf{SplScheme} = (\mathsf{SetupSpl}, \mathsf{SignSpl}, \mathsf{VerSpl},$

$\mathsf{SplitSpl}, \mathsf{SignSplAbo}$). It is associated with the message space $\{0,1\}^{\ell_{\mathsf{Itr}}+\ell_{\mathsf{Acc}}+2\lambda}$. In addition to the above tools, we also use a puncturable PRF family denoted by F.

We now describe the scheme $\mathsf{PABE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{GenPatch}, \mathsf{ApplyPatch}, \mathsf{Dec})$ below. Let the scheme PABE be associated with the class of Turing machines $\mathcal{M}$. Without loss of generality, the start state of every Turing machine in $\mathcal{M}$ is q_0. We denote the message space to be MSG.

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

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

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

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

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

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

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

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

[Note: The key generation does not output the storage tree $store_0$ but instead it just outputs the initial store value $\widetilde{store}_0$. The evaluator in possession of M, $\widetilde{store}_0$ and $\mathsf{PP_{Acc}}$ can reconstruct the tree $store_0$.]

$\underline{\mathsf{PABE.Enc}(\mathsf{PABE.PP}, x, \mathsf{msg})}$: On input the public key $\mathsf{PABE.PP} = (\mathsf{VK_{tm}}, \mathsf{PP_{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP_{Itr}}, v_0)$, attribute $x \in \{0,1\}^*$ and message $\mathsf{msg} \in \mathsf{MSG}$, it executes the following steps:

1. It first samples a PRF key K_A at random from the family F.

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

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

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

Program NxtMsg

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

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

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

Figure 1: Program NxtMsg

$\underline{\mathsf{PABE.GenPatch}(\mathsf{PABE.SK}, P, \mathsf{st})}$: On input the secret key $\mathsf{PABE.SK}$, a description of a patch $P \in \mathcal{P}$, and state $\mathsf{st} = M$, it essentially executes the Setup algorithm. It generates $(\mathsf{PABE.PP}' = (\mathsf{VK}'_{\mathsf{tm}}, \mathsf{PP}'_{\mathsf{Acc}}, \widetilde{w}'_0, \widetilde{store}'_0, \mathsf{PP}'_{\mathsf{Itr}}, v'_0), \mathsf{PABE.SK}' = (\mathsf{PABE.PP}', \mathsf{SK}'_{\mathsf{tm}}) \leftarrow \mathsf{PABE.Setup}(1^\lambda)$. It then executes $\mathsf{Update}(M, P)$ to obtain the updated machine M'. We then compute a new ABE key of M' as follows:

1. **Initialization of the storage tree**: Let $\ell_{\mathsf{tm}} = |M'|$ be the length of the machine M'. For $1 \le j \le \ell_{\mathsf{tm}}$, it computes $\widetilde{store}_j = \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, \widetilde{store}_{j-1}, j-1, M'_j)$, $aux_j = \mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}},$

Program $\mathsf{SignProg}$

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

1. If $\mathsf{VerSpl}(\mathsf{VK}_{\mathsf{tm}}, y, \sigma_{\mathsf{tm}}) = 0$ then output $\perp$.
2. Execute the pseudorandom function on input 0 to obtain $r_A \leftarrow F(K, 0)$. Generate the setup of splittable signatures scheme, $(\mathsf{SK}_0, \mathsf{VK}_0) \leftarrow \mathsf{SetupSpl}(1^\lambda; r_A)$.
3. Compute the signature $\sigma_0 \leftarrow \mathsf{SignSpl}(\mathsf{SK}_0, y)$.
4. Output σ_0.

Figure 2: Program $\mathsf{SignProg}$

$\widetilde{store}_{j-1}, j-1)$, $\widetilde{w}_j = \mathsf{Update}(\mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_{j-1}, M'_j, j-1, aux_j)$, where M'_j denotes the j^{th} bit of M'. Finally, it sets the root $w'_0 = \widetilde{w}_{\ell_{\mathsf{tm}}}$.

2. **Signing the accumulator value**: It generates the signature on the message $(v'_0, q_0, w'_0, 0)$, $\sigma_{M'} \leftarrow \mathsf{SignSpl}(\mathsf{SK}'_{\mathsf{tm}}, \mu = (v'_0, q_0, w'_0, 0))$, where q_0 is the start state of M'.

It outputs the secure patch (P, σ'_0), sets the updated public key to be $\mathsf{PABE}.pp'$ and the updated state st' is now assigned to be M'.

$\underline{\mathsf{PABE.ApplyPatch}(\mathsf{PABE}.sk_M, \widetilde{P})}$: On input the current ABE key $\mathsf{PABE}.sk_M = (M, \sigma_M)$ and secure patch $\widetilde{P} = (P, \sigma_{M'})$, it generates $\mathsf{Update}(M, P)$ to obtain M'. It outputs the updated key $\mathsf{PABE}.fesk_{M'} = (M', \sigma_{M'})$.

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

1. **Reconstructing the storage tree**: Suppose $\ell_{\mathsf{tm}} = |M|$ be the length of the TM M. For $1 \leq j \leq \ell_{\mathsf{tm}}$, it then repeatedly updates the storage tree by computing, $\widetilde{store}_j = \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, \widetilde{store}_{j-1}, j-1, M_j)$. Finally, it sets $store_0 = \widetilde{store}_{\ell_{\mathsf{tm}}}$.
2. **Executing N one step at a time**: For $i = 1$ to 2^λ,
   (a) Compute the proof that validates the storage value $store_{i-1}$ (storage value at $(i-1)^{th}$ time step) at position pos_{i-1}. Let $(\mathsf{sym}_{i-1}, \pi_{i-1}) \leftarrow \mathsf{PrepRead}(\mathsf{PP}_{\mathsf{Acc}}, store_{i-1}, \mathsf{pos}_{i-1})$.
   (b) Compute the auxiliary value, $aux_{i-1} \leftarrow \mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}}, store_{-1}, \mathsf{pos}_{i-1})$.
   (c) Run the obfuscated next message function. Compute $\mathsf{out} \leftarrow N(i, \mathsf{sym}_{i-1}, \mathsf{pos}_{i-1}, \mathsf{st}_{i-1}, w_{i-1}, v_{i-1}, \sigma_{i-1}, \pi_{i-1}, aux_{i-1})$. If $\mathsf{out} \in \mathsf{MSG} \cup \{\perp\}$. output out.
   Else parse out as $(\mathsf{sym}_{w,i}, \mathsf{pos}_i, \mathsf{st}_i, w_i, v_i, \sigma_i)$.
   (d) Compute the storage value, $store_i \leftarrow \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, store_{i-1}, \mathsf{pos}_{i-1}, \mathsf{sym}_{w,i})$.

Remark 4. *In the description of Koppula et al., the accumulator and the iterator algorithms also took the time bound T as input. Here, we set $T = 2^\lambda$ since we are only concerned with Turing machines that run in time polynomial in λ.*

This completes the description of the scheme. We note that the size of the secure patch of patch P is $|P| + \mathrm{poly}(\lambda)$ and thus the above scheme satisfies the efficiency property. We describe the proof of correctness below.

Correctness. We first define some notation.

Definition 7. *We say that* $\mathsf{PABE}.sk_M$ *is a* **valid PABE key of** M **w.r.t** $\mathsf{PABE.PP}$ *if there exists pair of randomness* (R_1, R_2) *such that* $(\mathsf{PABE.PP}, \mathsf{PABE.SK}) \leftarrow \mathsf{PABE.Setup}(1^\lambda; R_1)$ *and* $\mathsf{PABE}.sk_M \leftarrow \mathsf{PABE.KeyGen}(\mathsf{PABE.SK}, M; R_2)$.

We state two lemmas that will prove the correctness of the PABE scheme.

Lemma 1. *The decryption of* $\mathsf{PABE.CT} = \mathsf{PABE}.enc(\mathsf{PABE.PP}, x, \mathsf{msg})$ *using a valid PABE key of* M *w.r.t* $\mathsf{PABE.PP}$, $\mathsf{PABE}.sk_M$, *is* msg *if* $M(x) = 1$. *That is,* $\mathsf{msg} \leftarrow \mathsf{PABE.Dec}(\mathsf{PABE}.sk_M, \mathsf{PABE.CT})$ *if* $M(x) = 1$.

Proof Sketch. Suppose $\mathsf{PABE.CT}$ is a ciphertext of message msg w.r.t attribute x and $\mathsf{PABE}.sk_M$ is an ABE key of machine M. We claim that in the i^{th} iteration of the decryption of $\mathsf{PABE.CT}$ using $\mathsf{PABE}.sk_M$, the storage corresponds to the work tape of the execution of $M(x)$ at the i^{th} time step, denoted by $W_{t=i}$ [2]. Once we show this, the lemma follows.

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

The following lemma states that updating a valid PABE key of a machine M using a secure patch $\widetilde{P}$ leads to a valid PABE key of M', where $M' \leftarrow \mathsf{Update}(M, P)$. The proof of this lemma follows by inspection.

Lemma 2. *Consider the following process.*

1. $(\mathsf{PABE.PP}_0, \mathsf{PABE.SK}) \leftarrow \mathsf{PABE.Setup}(1^\lambda)$,
2. $(\mathsf{PABE}.sk_{M_0}, \mathsf{st}_0) \leftarrow \mathsf{PABE.KeyGen}(\mathsf{PABE.SK}, M_0)$,
3. *Repeat the following for* $i = 1, \ldots, L$:
   - $(\widetilde{P}_i, \mathsf{PABE.PP}_i, \mathsf{st}_i) \leftarrow \mathsf{PABE.GenPatch}(\mathsf{PABE.SK}, P_i, \mathsf{st}_{i-1})$,
   - $\mathsf{PABE}.sk_{M_i} \leftarrow \mathsf{PABE.ApplyPatch}(\mathsf{PABE}.sk_{M_{i-1}}, \widetilde{P}_i)$,

For every $i \in [L]$, *we have that* $\mathsf{PABE}.sk_{M_i}$ *is a valid PABE key w.r.t* $\mathsf{PABE.PP}_i$.

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

5.2 Security

To prove the security of our PABE scheme we make use of a theorem proved in Koppula et al. Before we recall their theorem, we first define the following distribution that would be useful to state the theorem. This distribution is identical to the output distribution of input encoding of the message hiding encoding scheme by [KLW15]. We denote the distribution by $\mathcal{D}_{M,U_x(\cdot),\mathsf{msg}}$, where M is a Turing machine, $x \in \{0,1\}^*$ and $\mathsf{msg} \in \mathsf{MSG}$. We define the sampler for the distribution below. We use the same notation to denote both the distribution as well as its sampler.

$\underline{\mathcal{D}_{M,x,\mathsf{msg}}(1^\lambda)}$: It first computes $(\mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0) \leftarrow \mathsf{SetupAcc}(1^\lambda, T)$. Let $\ell_{\mathsf{tm}} = |M|$ be the length of the Turing machine M. It computes $\widetilde{store}_j = \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}},\ \widetilde{store}_{j-1},\ j-1,\ M_j)$, $aux_j = \mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}},\ \widetilde{store}_{j-1},\ j-1)$, $\widetilde{w}_j = \mathsf{Update}(\mathsf{PP}_{\mathsf{Acc}},\ \widetilde{w}_{j-1},\ \mathsf{inp}_j,\ j-1,\ aux_j)$ for $1 \leq j \leq \ell_{\mathsf{tm}}$. Finally, it sets $w_0 = \widetilde{w}_{\ell_{\mathsf{tm}}}$ and $s_0 = \widetilde{store}_{\ell_{\mathsf{tm}}}$. Next, it computes the iterator parameters $(\mathsf{PP}_{\mathsf{Itr}}, v_0) \leftarrow \mathsf{SetupItr}(1^\lambda, T)$. It chooses a puncturable PRF key $K_A \leftarrow F.\mathsf{Setup}(1^\lambda)$. It also computes an obfuscation $N \leftarrow i\mathcal{O}(\mathsf{NxtMsg}\{U_x(\cdot), \mathsf{msg}, \mathsf{PP}_{\mathsf{Acc}}, \mathsf{PP}_{\mathsf{Itr}}, K_A\})$ where NxtMsg is defined in Figure 1. Let $r_A = F(K_A, 0)$, $(\mathsf{SK}_0, \mathsf{VK}_0) = \mathsf{SetupSpl}(1^\lambda; r_A)$ and $\sigma_0 = \mathsf{SignSpl}(\mathsf{SK}_0, (v_0, q_0, w_0, 0))$.

The distribution finally outputs the following:

$$\Big(N, w_0, v_0, \sigma_0, store_0, \mathsf{init} = (\mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP}_{\mathsf{Itr}})\Big)$$

[*Remark: The values* $\widetilde{w_0}$, $\widetilde{store}_0$ *and* $\mathsf{PP}_{\mathsf{Itr}}$ *are not explicitly given out in the message hiding encodings construction of KLW. But in their specific accumulator construction (which even we are utilizing),* $\widetilde{w_0}$ *is set to be* $\perp$ *and* $\widetilde{store}_0$ *is set to be* $\perp$*. Although not made explicit, even the iterator public parameters* $\mathsf{PP}_{\mathsf{Itr}}$ *can be given out in their construction without any modification in the proof of security.*]

The following theorem was shown in [KLW15].

Theorem 7 ([KLW15],Theorem 6.1)**.** *For all TMs* $M \in \mathcal{M}$, $x \in \{0,1\}^*, \mathsf{msg}_0, \mathsf{msg}_1 \in \mathsf{MSG}$ *such that* $M(x) = 0$ *and* $|\mathsf{msg}_0| = |\mathsf{msg}_1|$*, we have that the distributions* $\mathcal{D}_{M,x,\mathsf{msg}_0}$ *and* $\mathcal{D}_{M,x,\mathsf{msg}_1}$ *are computationally indistinguishable assuming the security of indistinguishability obfuscators* $i\mathcal{O}$*, accumulators scheme* Acc*, iterators scheme* Itr*, splittable signatures scheme* $\mathsf{SplScheme}$.

We prove the security of our PABE scheme as follows: We first consider the simpler case, when there are no updates and argue that the security of our scheme holds in this case. We define the security experiment in this case to be $\mathsf{OneTimeExpt}$. We then use $\mathsf{OneTimeExpt}$ to argue about the security of PABE.

We begin by describing $\mathsf{OneTimeExpt}$. Let $\mathcal{A}$ be a PPT adversary.

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

1. $\mathcal{A}$ sends to Ch a tuple consisting of a Turing machine M, an attribute x, two messages $(\mathsf{msg}_0, \mathsf{msg}_1)$ and an index $\mathbf{i} \geq 0$.

2. Ch computes the following: (a) $(\mathsf{PABE.PP}_0, \mathsf{PABE.SK}) \leftarrow \mathsf{PABE}, \mathsf{Setup}(1^\lambda)$, (b) $(\mathsf{PABE}.sk_M, \mathsf{st}_0) \leftarrow \mathsf{PABE.KeyGen}(\mathsf{PABE.SK}, M)$. It sends $(\mathsf{PABE.PP}_0, \mathsf{PABE}.sk_M)$ to $\mathcal{A}$.

3. Finally, the adversary outputs the bit b'.

We prove the following theorem. The proof of this theorem was shown in [AJS15].

Theorem 8. *For every PPT adversary $\mathcal{A}$, we have* $\mathsf{Pr}[b \leftarrow \mathsf{OneTimeExpt}_{\mathcal{A}}^{\mathsf{PABE}}(1^\lambda, b) \,:\, b \xleftarrow{\$} \{0,1\}] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$, *assuming the security of indistinguishability obfuscators $i\mathcal{O}$, accumulators scheme* Acc, *iterators scheme* Itr *and splittable signatures scheme* $\mathsf{SplScheme}$.

Proof. Consider the following sequence of hybrids. The first hybrid corresponds to the real experiment (as described in the security game) when the challenger picks a bit b at random and sets the challenge bit to be b. We then describe a series of intermediate hybrids such that every two consecutive hybrids are computationally indistinguishable. In the final hybrid, the challenger picks a bit b at random but sets the challenge bit to be 0. At this point the probability that the PPT adversary $\mathcal{A}$ can guess the bit b is $\frac{1}{2}$.

We denote $\mathsf{Adv}_{\mathcal{A},i}$ to be the advantage of $\mathcal{A}$ in Hyb_i.

Hybrid Hyb_1: The challenger receives from $\mathcal{A}$, a Turing machine M, an attribute x and two messages $\mathsf{msg}_0, \mathsf{msg}_1 \in \mathsf{MSG}$. The challenger then responds with the public key $\mathsf{PABE.PP}$, an ABE key of M, namely $\mathsf{PABE}.sk_M$ and an encryption of msg_b w.r.t attribute x, namely $\mathsf{PABE.CT}^*$, where b is picked at random. All the parameters are generated honestly by the challenger.

The output of the hybrid is the output of the adversary.

Hybrid Hyb_2: The verification key $\mathsf{VK}_{\mathsf{tm}}$ is replaced by a verification key that only verifies on the root of the accumulator storage, initialized with the TM M, and rejects signatures on all other messages. The rest of the hybrid is the same as the previous hybrid.

The challenger upon receiving a TM M, attribute x and messages $\mathsf{msg}_0, \mathsf{msg}_1 \in \mathsf{MSG}$, does the following. It first picks a bit b at random. It generates the accumulator and the iterator parameters $\mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP}_{\mathsf{Itr}}, v_0$ as in the setup algorithm. It then initializes the accumulator storage with the Turing machine M as follows: as before, let $\ell_{\mathsf{tm}} = |M|$ be the length of the Turing machine. It computes $\widetilde{store}_j = \mathsf{WriteStore}(\mathsf{PP}_{\mathsf{Acc}}, \widetilde{store}_{j-1}, j-1, M_j)$, $aux_j = \mathsf{PrepWrite}(\mathsf{PP}_{\mathsf{Acc}}, \widetilde{store}_{j-1}, j-1)$, $\widetilde{w}_j = \mathsf{Update}(\mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_{j-1}, \mathsf{inp}_j, j-1, aux_j)$ for $1 \leq j \leq \ell_{\mathsf{tm}}$. Finally, it sets $\mathbf{w} = \widetilde{w}_{\ell_{\mathsf{tm}}}$.

It then executes the setup of splittable signatures scheme, $(\mathsf{SK}_{\mathsf{tm}}, \mathsf{VK}_{\mathsf{tm}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$. It then executes the split algorithm of the signatures scheme to obtain, $(\sigma_{\mathsf{tm}}^{\mathbf{y}}, \mathsf{VK}_{\mathsf{tm}}^{\mathbf{y}}, \mathsf{SK}_{\backslash \mathbf{y}}, \mathsf{VK}_{\backslash \mathbf{y}}) \leftarrow \mathsf{SplitSpl}(\mathsf{SK}_{\mathsf{tm}}, \mathbf{y} = (v_0, q_0, \mathbf{w}, 0))$. Of particular interest to us is $\sigma_{\mathsf{tm}}^{\mathbf{y}}$, which is the (deterministic) signature on $\mathbf{y}$ and $\mathsf{VK}_{\mathsf{tm}}^{\mathbf{y}}$, which is the verification key that only validates the message-signature pair $(\mathbf{y}, \sigma_{\mathsf{tm}}^{\mathbf{y}})$ and invalidates all other message-signature pairs. It finally sets the public key as $\big(\mathsf{PABE.PP} = (\mathsf{VK}_{\mathsf{tm}}^{\mathbf{y}}, \mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP}_{\mathsf{Itr}}, v_0)\big)$.

The challenger then sets $\mathsf{PABE}.sk_M = (M, \mathbf{w}, \sigma_{\mathsf{tm}}^{\mathbf{y}}, v_0)$. It generates the challenge ciphertext by computing $\mathsf{PABE.CT}^* \leftarrow \mathsf{PABE.Enc}(\mathsf{PABE.PP}, x, \mathsf{msg}_b)$. It then sends $(\mathsf{PABE.PP}, \mathsf{PABE}.sk_M, \mathsf{PABE.CT}^*)$ to $\mathcal{A}$.

Claim 1. *Assuming that* $\mathsf{SplScheme}$ *satisfies* $\mathsf{VK}_{\mathrm{one}}$ *indistinguishability, for any PPT adversary $\mathcal{A}$ we have* $|\mathsf{Adv}_{\mathcal{A},1} - \mathsf{Adv}_{\mathcal{A},2}| \leq \mathsf{negl}(\lambda)$.

Proof. The only message-signature pair, with respect to the instantiation of the key pair $(\mathsf{SK}_{\mathsf{tm}}, \mathsf{VK}_{\mathsf{tm}})$, provided to the adversary $\mathcal{A}$ is $(\mathbf{y}, \sigma_{\mathsf{tm}}^{\mathbf{y}})$. Even with this additional information, the verification keys $\mathsf{VK}_{\mathsf{tm}}$ from $\mathsf{VK}_{\mathsf{tm}}^{\mathbf{y}}$, defined as in Hyb_1 and Hyb_2, are computationally indistinguishable from the $\mathsf{VK}_{\mathrm{one}}$ property of $\mathsf{SplScheme}$. The proof of the claim follows. □

Hybrid Hyb_3: The program $\mathsf{SignProg}$, which is part of the encryption process, is now modified with the output hardwired into it. The rest of the hybrid is as before.

The challenger upon receiving a TM M, attribute x and messages $\mathsf{msg}_0, \mathsf{msg}_1 \in \mathsf{MSG}$, does the following. It first picks a bit b at random. It sets $\mathsf{msg}^* = \mathsf{msg}_b$. It then computes $\mathsf{PABE.PP} = (\mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}, \mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP}_{\mathsf{Itr}}, v_0)$ as in Hyb_2. Further, it computes the ABE key of M, namely $\mathsf{PABE}.sk_M = (M, \mathbf{w}, \sigma^{\mathbf{y}}_{\mathsf{tm}}, v_0)$, as in Hyb_2.

It then samples a PRF key K_A at random. It computes the obfuscation of the program $U_x(\cdot)$, $N \leftarrow i\mathcal{O}(\mathsf{NxtMsg}\{U_x(\cdot), \mathsf{msg}^*, \mathsf{PP}_{\mathsf{Acc}}, \mathsf{PP}_{\mathsf{Itr}}, K_A\})$ where $U_x(\cdot)$ is defined as in $\mathsf{PABE.Enc}$ and NxtMsg is defined in Figure 1. From here onwards, the challenger deviates from the honest execution of the encryption algorithm. It generates the signing key-verification key pair $(\mathsf{SK}_0, \mathsf{VK}_0) \leftarrow \mathsf{SetupSpl}(1^\lambda; r_A)$, where r_A is the output of $F(K, 0)$. It computes the signature $\sigma_0 \leftarrow \mathsf{SignSpl}(\mathsf{SK}_0, \mathbf{y} = (v_0, q_0, \mathbf{w}, 0))$. As before, it generates $(\sigma^{\mathbf{y}}_{\mathsf{tm}}, \mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}, \mathsf{SK}_{\backslash \mathbf{y}}, \mathsf{VK}_{\backslash \mathbf{y}}) \leftarrow \mathsf{SplitSpl}(\mathsf{SK}_{\mathsf{tm}}, \mathbf{y} = (v_0, q_0, \mathbf{w}, 0))$. It then computes the obfuscation of the program $S^* \leftarrow (\mathsf{HybSgn}\{\mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}, \sigma_0\})$ where HybSgn is defined in Figure 3. It sets the ciphertext $\mathsf{PABE.CT}^* = (N, S^*)$. The challenger then sends $(\mathsf{PABE.PP}, \mathsf{PABE}.sk_M, \mathsf{PABE.CT}^*)$ to $\mathcal{A}$.

Claim 2. *Assuming the security of the scheme $i\mathcal{O}$, for any PPT adversary $\mathcal{A}$ we have that* $|\mathsf{Adv}_{\mathcal{A},2} - \mathsf{Adv}_{\mathcal{A},3}| \leq \mathsf{negl}(\lambda)$.

Proof. Suppose $S \leftarrow iO(\mathsf{SignProg}\{K_A, \mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}\})$ as in Hyb_2 and $S^* \leftarrow iO(\mathsf{HybSgn}\{\mathsf{VK}^{\mathbf{w}}_{\mathsf{tm}}, \sigma_0\})$ as in Hyb_3. To prove the claim, it suffices to show that it is computationally hard to distinguish S and S^*. This further reduces, courtesy security of iO, to showing that $\mathsf{SignProg}\{K_A, \mathsf{VK}_{\mathsf{tm}}\}$ and $\mathsf{HybSgn}\{\mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}, \sigma_0\}$ are functionally equivalent. Consider the input (y, σ) to both the programs. There are two cases to consider:

- **Case** $(y, \sigma) \neq (\mathbf{y}, \sigma^{\mathbf{y}}_{\mathsf{tm}})$: The program $\mathsf{SignProg}\{K_A, \mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}\}(y, \sigma)$ outputs $\perp$ because (y, σ) is invalid with respect to $\mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}$. For the same reason, program $\mathsf{HybSgn}\{\mathsf{VK}^{\mathbf{w}}_{\mathsf{tm}}, \sigma_0\}(y, \sigma)$ also outputs $\perp$.

- **Case** $(y, \sigma) = (\mathbf{y}, \sigma^{\mathbf{y}}_{\mathsf{tm}})$: The program $\mathsf{SignProg}\{K_A, \mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}\}(y, \sigma)$ outputs the signature σ_0 computed by first running $r_A \leftarrow F(K, 0)$, then $(\mathsf{SK}_0, \mathsf{VK}_0) \leftarrow \mathsf{SetupSpl}(1^\lambda)$ and finally $\sigma_0 \leftarrow \mathsf{SignSpl}(\mathsf{SK}_0, \mathbf{y})$. The program HybSgn outputs the hardwired σ_0, where σ_0 is pre-computed exactly as in $\mathsf{SignProg}$.

Thus the programs $\mathsf{SignProg}$ and HybSgn are functionally equivalent. This proves the claim. $\square$

$\mathsf{HybSgn}\{\mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}, \sigma_0\}$

Constants: PRF key K_A, verification key $\mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}$ and signature σ_0.
Input: Message y and a signature σ_{tm}.

1. If $\mathsf{VerSpl}(\mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}, y, \sigma_{\mathsf{tm}}) = 0$ then output $\perp$. Otherwise output σ_0.

Figure 3: Program HybSgn

Hybrid Hyb_4: This is identical to Hyb_3 except that the message msg^* to be encrypted is now set to msg_0, where $(\mathsf{msg}_0, \mathsf{msg}_1)$ is the challenge message pair submitted by the adversary. Recall that in Hyb_3, msg^* was set to msg_b, where b is picked at random.

Claim 3. *From Theorem 7, we have* $|\mathsf{Adv}_{\mathcal{A},3} - \mathsf{Adv}_{\mathcal{A},4}| \leq \mathsf{negl}(\lambda)$

Proof. Assume that the claim is not true. We then construct a reduction $\mathcal{B}$ that uses the adversary $\mathcal{A}$ to contradict Theorem 7.

$\mathcal{A}$ first sends the Turing machine $M \in \mathcal{M}$, input x and message pair $(\mathsf{msg}_0, \mathsf{msg}_1) \in \mathsf{MSG}^2$ to $\mathcal{B}$. The reduction then obtains a sample from the distribution $\mathcal{D}_{M,x,\mathsf{msg}_b}$, where b is either picked at random or set to 0. It then parses the sample as $\big(N, \mathbf{w}, v_0, \sigma_0, store_0, \mathsf{init} = (\mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP}_{\mathsf{Itr}})\big)$. The reduction $\mathcal{B}$ then samples a signature key-verification key pair by running the setup of $\mathsf{SplScheme}$, $(\mathsf{SK}_{\mathsf{tm}}, \mathsf{VK}_{\mathsf{tm}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$. It then executes the split algorithm, $(\sigma^{\mathbf{y}}_{\mathsf{tm}}, \mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}, \mathsf{SK}_{\backslash\mathbf{y}}, \mathsf{VK}_{\backslash\mathbf{y}}) \leftarrow \mathsf{SplitSpl}(\mathsf{SK}_{\mathsf{tm}}, \mathbf{y} = (v_0, q_0, \mathbf{w}, 0)$. Finally, $\mathcal{B}$ generates the obfuscation of the program HybSgn described in Figure 3, $S^* \leftarrow (\mathsf{HybSgn}\{\mathsf{VK}^{\mathbf{w}}_{\mathsf{tm}}, \sigma_0\})$. The reduction then prepares the ABE public key, attribute key and challenge ciphertext as below:

- The public key is set to be $\mathsf{PABE.PP} = (\mathsf{VK}^{\mathbf{y}}_{\mathsf{tm}}, \mathsf{PP}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP}_{\mathsf{Itr}}, v_0)$.
- The attribute key of M to be $\mathsf{PABE}.sk_M = (M, \mathbf{w}, \widetilde{store}_0, \sigma^{\mathbf{y}}_{\mathsf{tm}}, v_0)$.
- The challenge ciphertext is set to be $\mathsf{PABE.CT}^* = (N, S^*)$.

$\mathcal{B}$ then sends $(\mathsf{PABE.PP}, \mathsf{PABE}.sk_M, \mathsf{PABE.CT}^*)$ across to $\mathcal{A}$. The output of $\mathcal{B}$ is set to be the output of $\mathcal{A}$.

If the bit b in $\mathcal{D}_{M,x,\mathsf{msg}_b}$ is picked at random then we are in Hyb_3 and if it is set to be 0 then we are in Hyb_4. From our hypothesis (that the claim is not true), this means that the hybrids Hyb_3 and Hyb_4 are computationally indistinguishable. Thus we arrive at a contradiction of Theorem 7. This completes the proof. □

The probability that $\mathcal{A}$ outputs the bit b in Hyb_4 is $1/2$. From Claims 1, 2 and 3, we have that the probability that $\mathcal{A}$ outputs bit b in Hyb_1 is negligibly close to $1/2$. This completes the proof.

□

Before we show Theorem 8 can be used to prove the security of the PABE scheme, we first define the following property.

Definition 8 (Decomposability)**.** *A PABE scheme satisfies* **decomposability property** *if the following holds:*

1. *A PABE key* $(\mathsf{PABE}.sk_M, \mathsf{st}) \leftarrow \mathsf{PABE.KeyGen}(\mathsf{PABE.SK}, M)$ *is of the form* $\mathsf{PABE}.sk_M = (M, \mu)$.
2. *The patch generated by the patch generation algorithm,* $\widetilde{P} \leftarrow \mathsf{PABE.GenPatch}(\mathsf{PABE.SK}, P, \mathsf{st})$ *is of the form* $(\widetilde{P}, \mu')$.
3. *The state information* st_i *contains the description of the* i^{th} *updated machine. That is, let* $(\mathsf{PABE}.skM_0, \mathsf{st}_0) \leftarrow \mathsf{PABE.KeyGen}(\mathsf{PABE.SK}, M)$. *And let* $\mathsf{PABE}.sk_{M_i} \leftarrow \mathsf{PABE.ApplyPatch}(\mathsf{PABE}.sk_{M_{i-1}}, \widetilde{P}_i)$, *where* $(\widetilde{P}_i, \mathsf{st}_i) \leftarrow \mathsf{PABE.GenPatch}(\mathsf{PABE.SK}, P_i, \mathsf{st}_{i-1})$. *Then we have,* $\mathsf{st}_i = M_i$, *where* $M_i \leftarrow \mathsf{Update}(M_{i-1}, P_i)$.
4. *The algorithm* $\mathsf{ApplyPatch}$, *on input* $\big(\widetilde{P} = (P, \mu'), \mathsf{PABE}.sk_M = (M, \mu)\big)$, *executes in two steps: (i) Execute* $M' \leftarrow \mathsf{Update}(M, P)$, *(ii) Output* (M', μ').

By inspection, our scheme PABE satisfies the decomposability property. Consider the following hybrid experiment:

$\underline{\mathsf{HybExpt}.i_{\mathcal{A}}^{\mathsf{PABE}}(1^\lambda, b)}$:

1. $\mathcal{A}$ sends to Ch a tuple consisting of a Turing machine M_0, an attribute x, two messages $(\mathsf{msg}_0, \mathsf{msg}_1)$ and an index $\mathbf{i} \geq 0$.
2. Ch computes the following: (a) $(\mathsf{PABE.PP}_0, \mathsf{PABE.SK}) \leftarrow \mathsf{PABE}, \mathsf{Setup}(1^\lambda)$, (b) $(\mathsf{PABE}.sk_{M_0}, \mathsf{st}_0) \leftarrow \mathsf{PABE.KeyGen}(\mathsf{PABE.SK}, M_0)$. It sends $(\mathsf{PABE.PP}_0, \mathsf{PABE}.sk_{M_0})$ to $\mathcal{A}$.
3. The following is repeated polynomially many times:
   (a) $\mathcal{A}$ sends to Ch a patch $P_j \in \mathcal{P}$.
   (b) If $j \neq i$, Ch computes the following: (a) $(\widetilde{P_j}, \mathsf{PABE.PP}_j, \mathsf{st}_j) \leftarrow \mathsf{PABE.GenPatch}(\mathsf{PABE.SK}, P_j, \mathsf{st}_{j-1})$ (b) If $j = \mathbf{i}$, $\mathsf{PABE.CT}_{(x,\mathsf{msg}_b)} \leftarrow \mathsf{PABE.Enc}(\mathsf{PABE.PP}_j, x, \mathsf{msg}_b)$.
   (c) If $j = i$, Ch computes the following: (a) $M_j \leftarrow \mathsf{Update}(M_{j-1}, P_j)$, (b) It executes the PABE setup algorithm, $(\mathsf{PABE.PP}_j, \mathsf{PABE.SK}_j) \leftarrow \mathsf{PABE.Setup}(1^\lambda)$ (c) $\big(\mathsf{PABE}.sk_{M_j} = (M_j, \mu_j), \mathsf{st}^*\big) \leftarrow \mathsf{PABE.KeyGen}(\mathsf{PABE.SK}_j, M_j)$. It computes $\mathsf{st}_j = M_j$, (d) It computes the secure patch $\widetilde{P_j} = (P_j, \mu_j)$ (e) If $j = \mathbf{i}$, $\mathsf{PABE.CT}_{(x,\mathsf{msg}_b)} \leftarrow \mathsf{PABE.Enc}(\mathsf{PABE.PP}_j, x, \mathsf{msg}_b)$.
   (d) If $j = \mathbf{i}$, Ch sends $(\mathsf{PABE.PP}_j, \widetilde{P_j}, \mathsf{PABE.CT}_{(x,\mathsf{msg}_b)})$ to $\mathcal{A}$. If $j \neq \mathbf{i}$, Ch sends $(\mathsf{PABE.PP}_j, \widetilde{P_j})$ to $\mathcal{A}$.
4. Finally, the adversary outputs the bit b'.
5. The adversary wins the game if (a) $b = b'$, and (b) $M_{\mathbf{i}}(x) = 0$ where $M_j = \mathsf{Update}(M_{j-1}, P)$ and $M_0 = M$.

By inspection, we have the following lemma.

Lemma 3. *For every PPT adversary $\mathcal{A}$, every $b \in \{0,1\}$, every $i = \mathrm{poly}(\lambda)$, we have* $\Pr[\mathsf{Expt}_{\mathcal{A}}^{\mathsf{PABE}}(1^\lambda, b)] = \Pr[\mathsf{HybExpt}.i_{\mathcal{A}}^{\mathsf{PABE}}(1^\lambda, b)]$

We prove the following theorem that establishes the security of PABE scheme.

Theorem 9. *For every PPT adversary $\mathcal{A}$, every bit $b \in \{0,1\}$, we have* $\Pr[b \leftarrow \mathsf{Expt}_{\mathcal{A}}^{\mathsf{PABE}}(1^\lambda, b)\ b \xleftarrow{\$} \{0,1\}] \leq 1/2 + \mathsf{negl}(\lambda)$.

Proof. Let $\mathcal{B}$ be a reduction that uses $\mathcal{A}$ to break the security of $\mathsf{OneTimeExpt}$. It works as follows: it first receives $\big((\mathsf{msg}_0, \mathsf{msg}_1), M_0, x, \mathbf{i}\big)$ from $\mathcal{A}$. For every $j \neq \mathbf{i}$, $\mathcal{B}$ upon receiving $P_j \in \mathcal{P}$ from $\mathcal{A}$, generates all the parameters and sends it to $\mathcal{A}$. In more detail, it generates: $(\widetilde{P_j}, \mathsf{PABE.PP}_j, \mathsf{st}_j) \leftarrow \mathsf{PABE.GenPatch}(\mathsf{PABE.SK}, P_j, \mathsf{st}_{i-1})$. It then sends $(\widetilde{P_j}, \mathsf{PABE.PP}_j)$ to $\mathcal{A}$.

For $i = \mathbf{i}$, upon receiving $P_{\mathbf{i}}$ from $\mathcal{A}$, reduction $\mathcal{B}$ first computes $M_{\mathbf{i}}$, where $M_i \leftarrow \mathsf{Update}(M_{i-1}, P_i)$ for $1 \leq i \leq \mathbf{i}$. It forwards $\big((\mathsf{msg}_0, \mathsf{msg}_1), x, M_i\big)$ to the challenger of $\mathsf{OneTimeExpt}_{\mathcal{A}}^{\mathsf{PABE}}$. In turn it receives, $(\mathsf{PABE.PP}^*, \mathsf{PABE}.sk^*_{M_i}, \mathsf{PABE.CT}^*)$. From the decomposability property of PABE (Definition 8), we parse $\mathsf{PABE}.sk_{M_i}$ as (M_i, μ_i). It then computes $\widetilde{P}_i = (P_i, \mu_i)$ and sets $\mathsf{st}_i = M_i$. It forwards $(\widetilde{P}_i, \mathsf{PABE.PP}^*, \mathsf{PABE.CT}^*)$ to $\mathcal{A}$.

Suppose, let b be the bit used by the challenger of $\mathsf{OneTimeExpt}$. We define the following notation:

- $\mathsf{View}_{\mathcal{A}}^{\mathcal{B},b}$: This is a random variable (R.V.) that denotes the view of the adversary while interacting with $\mathcal{B}$ and $\mathcal{B}$ in turn is interacting with $\mathsf{OneTimeExpt}_{\mathcal{B}}^{\mathsf{PABE}}(1^\lambda, b)$. We denote $\mathsf{View}_{\mathcal{A}}^{\mathcal{B}}|_{\mathbf{i}}$, to be the R.V. by conditioning on the fact that $\mathcal{A}$ sends $\mathbf{i}$ in the first message of the game.
- $\mathsf{View}_{\mathcal{A}}^{\mathsf{HybExpt}.\mathbf{i},b}$: This is a R.V. that denotes the view of the adversary in the experiment $\mathsf{HybExpt}.\mathbf{i}_{\mathcal{A}}^{\mathsf{PABE}}(1^\lambda, b)$.
- $\mathsf{View}_{\mathcal{A}}^{\mathsf{Expt},b}$: This is a R.V. that denotes the view of the adversary in the experiment $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{PABE}}(1^\lambda, b)$.

By inspection, we have that $\mathsf{View}_{\mathcal{A}}^{\mathcal{B},b}|_{\mathbf{i}} \equiv \mathsf{View}_{\mathcal{A}}^{\mathsf{HybExpt}.\mathbf{i},b}$ (views are identically distributed) and from Lemma 3, we have $\mathsf{View}_{\mathcal{A}}^{\mathsf{Expt},b} \equiv \mathsf{View}_{\mathcal{A}}^{\mathsf{HybExpt}.\mathbf{i},b}$. Thus, we have $\mathsf{View}_{\mathcal{A}}^{\mathcal{B},b}|_{\mathbf{i}} \equiv \mathsf{View}_{\mathcal{A}}^{\mathsf{Expt},b}$

Further, from the security of $\mathsf{OneTimeExpt}$ (Theorem 8), we have that $\mathsf{View}_{\mathcal{A}}^{\mathcal{B},0}|_{\mathbf{i}} \cong \mathsf{View}_{\mathcal{A}}^{\mathcal{B},1}|_{\mathbf{i}}$.

Summarizing the above observations, we thus have: $\mathsf{View}_{\mathcal{A}}^{\mathsf{Expt},0} \cong \mathsf{View}_{\mathcal{A}}^{\mathsf{Expt},1}$. This completes the proof. □

By instantiating the tools of Acc, Itr and $\mathcal{S}$ from indistinguishability obfuscation and one-way functions [KLW15], we thus have the following corollary.

Corollary 1. *There exists a patchable attribute based encryption scheme assuming the existence of indistinguishability obfuscation and one-way functions.*

5.3 Two-Outcome PABE for TMs

5.3.1 Definition

Syntax. A 1-key two-outcome PABE for TMs scheme, defined for a class of Turing machines $\mathcal{M}$, patch family $\mathcal{P}$, update algorithm Update and message space MSG, consists of six PPT algorithms $\mathsf{TwoPABE} = (\mathsf{TwoPABE.Setup}, \mathsf{TwoPABE.KeyGen}, \mathsf{TwoPABE.GenPatch}, \mathsf{TwoPABE.ApplyPatch}, \mathsf{TwoPABE.Enc}, \mathsf{TwoPABE.Dec})$ described below.

1. **Setup,** $\mathsf{TwoPABE.Setup}(1^\lambda)$: On input a security parameter λ in unary, it outputs a secret key $\mathsf{TwoPABE.SK}$ and public key $\mathsf{TwoPABE.PP}$.
2. **Key Generation,** $\mathsf{TwoPABE.KeyGen}(\mathsf{TwoPABE.SK}, M \in \mathcal{M})$: On input a secret key $\mathsf{TwoPABE.SK}$ and a TM $M \in \mathcal{M}$, it outputs a key $\mathsf{TwoPABE}.sk_M$ and state st_0.
3. **Generation of Patch,** $\mathsf{TwoPABE.GenPatch}(\mathsf{TwoPABE}.sk, P \in \mathcal{P}, \mathsf{st})$: On input a secret key $\mathsf{TwoPABE.SK}$, a patch $P \in \mathcal{P}$ and state st, it outputs a patch encoding $\widetilde{P}$, new public parameters $\mathsf{TwoPABE.PP}'$ and updated state st'.
4. **Application of Patch,** $\mathsf{TwoPABE.ApplyPatch}(\mathsf{TwoPABE}.sk_M, \widetilde{P})$: On input a attribute key $\mathsf{TwoPABE}.sk_M$ and patch encoding $\widetilde{P}$, it outputs an updated key $\mathsf{TwoPABE}.sk_{M'}$.
5. **Encryption,** $\mathsf{TwoPABE.Enc}(\mathsf{TwoPABE.PP}, x, \mathsf{msg}_0, \mathsf{msg}_1)$: On input a (possibly updated) public key $\mathsf{TwoPABE.PP}$, attribute $x \in \{0,1\}^*$ and a pair of messages $\mathsf{msg}_0, \mathsf{msg}_1 \in \mathsf{MSG}$, it outputs a ciphertext $\mathsf{TwoPABE.CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)}$.
6. **Decryption,** $\mathsf{TwoPABE.Dec}(\mathsf{TwoPABE}.sk_M, \mathsf{TwoPABE.CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)})$: On input a (possibly updated) two-outcome ABE key $\mathsf{TwoPABE}.sk_M$ and ciphertext $\mathsf{TwoPABE.CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)}$, it outputs a value out.

Correctness. We say that a two-outcome PABE scheme is correct if for any Turing machine $M_0 \in \mathcal{M}$, every $L \geq 0$, patch sequence $(P_1, \ldots, P_L) \in \mathcal{P}^L$, every $x \in \{0,1\}^*$, and $\mathsf{msg}_0, \mathsf{msg}_1 \in \mathsf{MSG}$,

$$\Pr\left[\mathsf{TwoPABE.Dec}\left(\mathsf{TwoPABE}.sk_{M_L}, \mathsf{TwoPABE.CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)}\right) = \mathsf{msg}_b : M_L(x) = b\right] = 1$$

where:

- $(\mathsf{TwoPABE.PP}_0, \mathsf{TwoPABE.SK}) \leftarrow \mathsf{TwoPABE.Setup}(1^\lambda)$,
- $(\mathsf{TwoPABE}.sk_{M_0}, \mathsf{st}_0) \leftarrow \mathsf{TwoPABE.KeyGen}(\mathsf{TwoPABE.SK}, M_0)$,
- $(\widetilde{P_i}, \mathsf{TwoPABE.PP}_i, \mathsf{st}_i) \leftarrow \mathsf{TwoPABE.GenPatch}(\mathsf{TwoPABE.SK}, P_i, \mathsf{st}_{i-1})$,
- $\mathsf{TwoPABE}.sk_{M_i} \leftarrow \mathsf{TwoPABE.ApplyPatch}(\mathsf{TwoPABE}.sk_{M_{i-1}}, \widetilde{P_i})$,
- $\mathsf{TwoPABE.CT}_{(x,\mathsf{msg}_0,\mathsf{msg}_1)} \leftarrow \mathsf{TwoPABE.Enc}(\mathsf{TwoPABE.PP}_L, x, \mathsf{msg}_0, \mathsf{msg}_1)$,
- $M_j = \mathsf{Update}(M_{j-1}, P_j)$.

Efficiency. We say that a two-outcome PABE scheme satisfies efficiency property if $|\widetilde{P}| = \text{poly}(\lambda, |P|)$, where $(\widetilde{P}, \mathsf{TwoPABE.PP}', \mathsf{st}') \leftarrow \mathsf{TwoPABE.GenPatch}(\mathsf{TwoPABE.SK}, P, \mathsf{st})$ and in particular independent of the size of st.

Security. We extend the security definition of PABE to two-outcome PABE. Since we only consider the single-key setting, the adversary is restricted to making one key query. However, we allow the adversary to submit a polynomial number of patch queries. We consider adaptive security where the adversary submits both the challenge message pair as well as the key query at the beginning of the game itself but the patch queries are made adaptively. We also require the adversary to specify in advance an index $i \in [L]$ where L is the number of patch queries made by the adversary. The index i determines the updated public key that is then used to compute the challenge ciphertext sent to the adversary.

We formalize security in terms of the following security experiment between a challenger Ch and a PPT adversary $\mathcal{A}$.

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

1. $\mathcal{A}$ sends to Ch a tuple consisting of a Turing machine M_0, an attribute x, two pairs of messages $(\mathsf{msg}_{0,0}, \mathsf{msg}_{0,1}), (\mathsf{msg}_{1,0}, \mathsf{msg}_{1,1})$ and an index $\mathbf{i} \geq 0$.
2. Ch computes the following values : (a) $(\mathsf{TwoPABE.PP}_0, \mathsf{TwoPABE.SK}) \leftarrow \mathsf{TwoPABE, Setup}(1^\lambda)$, (b) $(\mathsf{TwoPABE}.sk_{M_0}, \mathsf{st}_0) \leftarrow \mathsf{TwoPABE.KeyGen}(\mathsf{TwoPABE.SK}, M_0)$. It sends $(\mathsf{TwoPABE.PP}_0,$ $\mathsf{TwoPABE}.sk_{M_0})$ to $\mathcal{A}$.
3. The following is repeated polynomially many times:
   (a) $\mathcal{A}$ sends to Ch a patch $P_j \in \mathcal{P}$.
   (b) Ch computes the following: (a) $(\widetilde{P_j}, \mathsf{TwoPABE.PP}_j, \mathsf{st}_j) \leftarrow \mathsf{TwoPABE.GenPatch}(\mathsf{TwoPABE.SK},$ $P_j, \mathsf{st}_{j-1})$, (b) $\mathsf{TwoPABE.CT}_{(x,\mathsf{msg}_{b,0},\mathsf{msg}_{b,1})} \leftarrow \mathsf{TwoPABE.Enc}(\mathsf{TwoPABE.PP}_j, x, \mathsf{msg}_{b,0}, \mathsf{msg}_{b,1})$ if $j = \mathbf{i}$.

(c) If $j = \mathbf{i}$, Ch sends $(\mathsf{TwoPABE.PP}_j, \widetilde{P_j}, \mathsf{TwoPABE.CT}_{(x,\mathsf{msg}_b)})$ to $\mathcal{A}$. If $j \neq \mathbf{i}$, Ch sends $(\mathsf{TwoPABE.PP}_j, \widetilde{P_j})$ to $\mathcal{A}$.

4. Finally, the adversary outputs the bit b'.

5. The adversary wins the game if (a) $b = b'$, and (b) $M_{\mathbf{i}}(x) = 0$ and $\mathsf{msg}_{0,0} = \mathsf{msg}_{1,0}$ or if $M_{\mathbf{i}}(x) = 1$ and $\mathsf{msg}_{0,1} = \mathsf{msg}_{1,1}$ where $M_j = \mathsf{Update}(M_{j-1}, P_j)$.

Definition 9. *A 1-key two-outcome PABE scheme is said to be* **adaptively secure** *if for every PPT adversary $\mathcal{A}$, there exists a negligible function* negl *s.t.*

$$\Pr\Big[\mathcal{A} \textit{ wins } \mathsf{Expt}_{\mathcal{A}}^{\mathsf{TwoPABE}}(1^\lambda, b)\Big] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

5.3.2 Construction

We realize this primitive along the same lines as as in [AJS15] (which, in turn, builds on ideas from [GKP+13]). The idea is to have two instantiations of a PABE scheme. To encrypt an attribute x and two messages $(\mathsf{msg}_0, \mathsf{msg}_1)$, we encrypt (x, msg_0) in one instantiation and (x, msg_1) in the other. Given two PABE keys of M, one w.r.t. to each instantiation, and the two ciphertexts, exactly one of $(\mathsf{msg}_0, \mathsf{msg}_1)$ will remain hidden depending on the value of $M(x)$.

We formally give the construction below of $\mathsf{TwoPABE}$ for Turing machine family $\mathcal{M}$, patch family $\mathcal{P}$, update algorithm Update and message space MSG. The only tool we use in our construction is a 1-key PABE for TMs $\mathsf{TwoPABE} = (\mathsf{TwoPABE.Setup}, \mathsf{TwoPABE.KeyGen}, \mathsf{TwoPABE.Enc}, \mathsf{TwoPABE.Dec})$ for the TM family $\mathcal{M}$, patch family $\mathcal{P}$, update algorithm Update and message space MSG.

$\underline{\mathsf{TwoPABE.Setup}(1^\lambda)}$: On input a security parameter λ in unary, execute $\mathsf{PABE.Setup}$ twice to obtain $(\mathsf{PABE.PP}_0^0, \mathsf{PABE.SK}_0^0) \leftarrow \mathsf{PABE.Setup}(1^\lambda)$ and $(\mathsf{PABE.PP}_1^0, \mathsf{PABE.SK}_1^0) \leftarrow \mathsf{PABE.Setup}(1^\lambda)$. Output $\big(\mathsf{TwoPABE.PP}_0 = (\mathsf{PABE.PP}_0^0, \mathsf{PABE.PP}_1^0), \mathsf{TwoPABE.SK}_0 = (\mathsf{PABE.SK}_0^0, \mathsf{PABE.SK}_1^0)\big)$.

$\underline{\mathsf{TwoPABE.KeyGen}(\mathsf{TwoPABE.SK}, M)}$: On input a secret key $\mathsf{TwoPABE.SK}_0 = (\mathsf{PABE.SK}_0^0, \mathsf{PABE.SK}_1^0)$ and a Turing machine $M \in \mathcal{M}$, first compute two PABE keys $\mathsf{PABE}.sk_M^0 \leftarrow \mathsf{PABE.KeyGen}(\mathsf{PABE.SK}_0^0, M \in \mathcal{M})$ and $\mathsf{PABE}.sk_M^1 \leftarrow \mathsf{PABE.KeyGen}(\mathsf{PABE.SK}_1^0, \overline{M})$, where $\overline{M}$ (complement of M) on input x outputs $1 - M(x)$.[3] Output the attribute key, $\mathsf{TwoPABE}.sk_M = (\mathsf{PABE}.sk_M^0, \mathsf{PABE}.sk_M^1)$.

$\underline{\mathsf{TwoPABE.GenPatch}(\mathsf{TwoPABE.SK}, P, \mathsf{st})}$: On input a a secret key $\mathsf{TwoPABE.SK}_0 = (\mathsf{PABE.SK}_0^0, \mathsf{PABE.SK}_1^0)$, patch $P \in \mathcal{P}$, and state $\mathsf{st} = (\mathsf{st}_{\mathsf{PABE},0}, \mathsf{st}_{\mathsf{PABE},1})$, perform the following steps:

- Compute $(\widetilde{P_0}, \mathsf{PABE.PP}'_0, \mathsf{st}'_{\mathsf{PABE},0}) \leftarrow \mathsf{PABE.GenPatch}(\mathsf{PABE.SK}_0^0, P, \mathsf{st}_{\mathsf{PABE},0})$.
- Compute $(\widetilde{P_1}, \mathsf{PABE.PP}'_1, \mathsf{st}'_{\mathsf{PABE},1}) \leftarrow \mathsf{PABE.GenPatch}(\mathsf{PABE.SK}_1^0, P, \mathsf{st}_{\mathsf{PABE},1})$.[4]

Set $\mathsf{st} = (\mathsf{st}'_{\mathsf{PABE},0}, \mathsf{st}'_{\mathsf{PABE},1})$ and $\mathsf{TwoPABE.PP}' = (\mathsf{PABE.PP}'_0, \mathsf{PABE.PP}'_1)$. Output $\widetilde{P} = (\widetilde{P_0}, \widetilde{P_1})$.

$\underline{\mathsf{TwoPABE.ApplyPatch}(\mathsf{TwoPABE}.sk_M, \widetilde{P})}$: On input a key $\mathsf{TwoPABE}.sk_M = (\mathsf{PABE}.sk_M^0, \mathsf{PABE}.sk_M^1)$ and patch encoding $\widetilde{P} = (\widetilde{P_0}, \widetilde{P_1})$, perform the following steps:

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

[4]Note that since $\overline{M}$ simply computes $1 - M(x)$ on any input x, we can apply the same patch P on both M and $\overline{M}$.

- Compute $\mathsf{PABE}.sk^0_{M'} \leftarrow \mathsf{PABE.ApplyPatch}(\mathsf{PABE}.sk^0_M, \widetilde{P_0})$.
- Compute $\mathsf{PABE}.sk^1_{M'} \leftarrow \mathsf{PABE.ApplyPatch}(\mathsf{PABE}.sk^1_M, \widetilde{P_1})$.

Output $\mathsf{TwoPABE}.sk_{M'} = (\mathsf{PABE}.sk^0_{M'}, \mathsf{PABE}.sk^1_{M'})$.

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

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

This completes the description of the scheme. The correctness and efficiency of the above scheme follows directly from the correctness and efficiency of the 1-key PABE scheme.

Security. The security of the above scheme follows in a straightforward manner, along the same lines as in [AJS15]. We omit the proof from this manuscript.

Theorem 10. *Assuming the security of* PABE*, the scheme* TwoPABE *is secure.*

6 Patchable Oblivious Evaluation Encodings

We first define a *basic* patchable oblivious evaluation encodings (BPOEE) scheme. And then we equip the BPOEE scheme with auxiliary algorithms and additional properties to define a patchable oblivious evaluation encodings (POEE) scheme.

Basic POEE. We describe the syntax of a basic patchable oblivious evaluation encoding scheme POEE defined for a class of Turing machines $\mathcal{M}$ and a family of patches $\mathcal{P}$. For simplicity, we denote the input space as $\{0,1\}^*$; however, during the generation of the system parameters, we place an upper bound on the running time of the machines which automatically puts an upper bound on the length of the inputs.

- $\mathsf{POEE.Setup}(1^\lambda)$: It takes as input a security parameter λ and outputs a secret key $\mathsf{POEE.sk}$.
- $\mathsf{POEE.TMEncode}(\mathsf{POEE.sk}, M_0, M_1)$: It takes as input a secret key $\mathsf{POEE.sk}$, a pair of Turing machines $M_0, M_1 \in \mathcal{M}$ and outputs a (joint) machine encoding $\langle M_0, M_1 \rangle$ along with initial state st .
- $\mathsf{POEE.GenPatch}(\mathsf{POEE.sk}, P_0, P_1, \mathsf{st})$: It takes as input a secret key $\mathsf{POEE.sk}$, a pair of patches $P_0, P_1 \in \mathcal{P}$ and current state st. It outputs a patch encoding $\langle P_0, P_1 \rangle$ and the updated state st.
- $\mathsf{POEE.ApplyPatch}\Big(\langle M_0, M_1 \rangle, \langle P_0, P_1 \rangle\Big)$: It takes as input a machine encoding $\langle M_0, M_1 \rangle$ and a patch encoding $\langle P_0, P_1 \rangle$. It outputs an updated machine encoding $\langle M'_0, M'_1 \rangle$.

- $\mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, b)$: It takes as input a secret key $\mathsf{POEE.sk}$, an input $x \in \{0,1\}^*$, an index $i \geq 0$ and a choice bit $b \in \{0,1\}$. It outputs an input encoding $\langle x, i, b\rangle$.
- $\mathsf{POEE.Decode}\Big(\langle M_0, M_1\rangle, \langle x, i, b\rangle\Big)$: It takes as input a (possibly updated) machine encoding $\langle M_0, M_1\rangle$ and an input encoding $\langle x, i, b\rangle$. It outputs a value z.

We define the correctness of encode, patching and decode property below.

Correctness of Encode, Patching and Decode: For all $M_0^0, M_1^0 \in \mathcal{M}$, every $L \geq 0$, patch sequence $P_0^1, P_1^1, \ldots, P_0^L, P_1^L \in \mathcal{P}$, $x \in \{0,1\}^*$ and $b \in \{0,1\}$,

$$\mathsf{POEE.Decode}\Big(\langle M_0^L, M_1^L\rangle, \langle x, L, b\rangle\Big) = M_b^L(x)$$

where:

- $\mathsf{POEE.sk} \leftarrow \mathsf{POEE.Setup}(1^\lambda)$,
- $\Big(\langle M_0^0, M_1^0\rangle, \mathsf{st}_0\Big) \leftarrow \mathsf{POEE.TMEncode}(\mathsf{POEE.sk}, M_0, M_1)$,
- $\Big(\langle P_0^i, P_1^i\rangle, \mathsf{st}_i\Big) \leftarrow \mathsf{POEE.GenPatch}(\mathsf{POEE.sk}, P_0^i, P_1^i, \mathsf{st}_{i-1})$,
- $\langle M_0^i, M_1^i\rangle \leftarrow \mathsf{POEE.ApplyPatch}\Big(\langle M_0^{i-1}, M_1^{i-1}\rangle, \langle P_0^i, P_1^i\rangle\Big)$,
- $\langle x, L, b\rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, L, b)$,
- $M_j^b = \mathsf{Update}(M_{j-1}^b, P_j^b)$.

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

1. The running time of $\mathsf{POEE.TMEncode}(\mathsf{POEE.sk}, M_0, M_1)$ is a polynomial in $(\lambda, |M_0|, |M_1|)$, where $\mathsf{POEE.sk} \leftarrow \mathsf{POEE.Setup}(1^\lambda)$.
2. The running time of $\mathsf{POEE.GenPatch}(\mathsf{POEE.sk}, P_0, P_1, \mathsf{st})$ is a polynomial in $(\lambda, |P_0|, |P_1|)$. In particular it is independent of the size of st.
3. The running time of $\mathsf{POEE.ApplyPatch}(\langle M_0, M_1\rangle, \langle P_0, P_1\rangle)$ is a polynomial in (λ, t), where $t = \max\Big\{\text{runtime of } \mathsf{Update}(M_0, P_0), \text{runtime of } \mathsf{Update}(M_1, P_1)\Big\}$.
4. The running time of $\mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, b)$ is a polynomial in $(\lambda, |x|)$, where $\mathsf{POEE.sk} \leftarrow \mathsf{POEE.Setup}(1^\lambda)$.

5. The running time of $\mathsf{POEE.Decode}\Big(\langle M_0, M_1\rangle, \langle x, i, b\rangle\Big)$ is a polynomial in $(\lambda, |M_0|, |M_1|, |x|, t)$, where $\mathsf{POEE.sk} \leftarrow \mathsf{POEE.Setup}(1^\lambda)$, $\langle M_0, M_1\rangle \leftarrow \mathsf{POEE.TMEncode}(\mathsf{POEE.sk}, M_0, M_1)$, $\langle x, i, b\rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, b)$ and t is the running time of the Turing machine M_b on x.

Definition 10. *A basic POEE scheme (BPOEE) scheme is a tuple of algorithms* ($\mathsf{POEE.Setup}$, $\mathsf{POEE.TMEncode}$, $\mathsf{POEE.GenPatch}$, $\mathsf{POEE.ApplyPatch}$, $\mathsf{POEE.InpEncode}$, $\mathsf{POEE.Decode}$) *that satisfies the above correctness of encode, patch and decode property and the efficiency property.*

POEE. We now augment the basic POEE scheme with additional helper algorithms defined below.

- $\mathsf{POEE.puncInp}(\mathsf{POEE.sk}, x, i)$: It takes as input a secret key $\mathsf{POEE.sk}$, input $x \in \{0,1\}^*$ and index i. It outputs a punctured key $\mathsf{POEE.sk}_{x,i}$.
- $\mathsf{POEE.PIEncode}(\mathsf{POEE.sk}_{x,i}, x', i', b)$: It takes as input a punctured secret key $\mathsf{POEE.sk}_{x,i}$, an input x', index i' and a bit b s.t. $(x, i) \neq (x', i')$. It outputs an input encoding $\langle x', i', b\rangle$.
- $\mathsf{POEE.puncBit}(\mathsf{POEE.sk}, b)$: It takes as input a secret key $\mathsf{POEE.sk}$ and an input bit b. It outputs a key $\mathsf{POEE}.sk_b$.
- $\mathsf{POEE.PBEncode}(\mathsf{POEE}.sk_b, x, i)$: It takes as input a key $\mathsf{POEE}.sk_b$, an input x and an index i. It outputs an input encoding $\langle x, i, b\rangle$.

We associate the above scheme with correctness and security properties as described below.

Correctness. We say that a POEE scheme is correct if it satisfies the following properties:

1. *Correctness of Input Puncturing:* For all $M_0, M_1 \in \mathcal{M}$, every $L \geq 0$, patch sequence $P_0^1, P_1^1, \ldots, P_0^L, P_1^L \in \mathcal{P}$, every $x, x' \in \{0,1\}^*$ and $i, i' \leq L$ s.t. $(x, i) \neq (x', i')$ and $b \in \{0,1\}$,
$$\mathsf{POEE.Decode}\Big(\langle M_0^{i'}, M_1^{i'}\rangle, \langle x', i', b\rangle\Big) = M_b(x'),$$
where:
   - $\mathsf{POEE.sk} \leftarrow \mathsf{POEE.Setup}(1^\lambda)$,
   - $\Big(\langle M_0, M_1\rangle, \mathsf{st}_0\Big) \leftarrow \mathsf{POEE.TMEncode}(\mathsf{POEE.sk}, M_0, M_1)$,
   - $\Big(\langle P_0^j, P_1^j\rangle, \mathsf{st}_j\Big) \leftarrow \mathsf{POEE.GenPatch}(\mathsf{POEE.sk}, P_0^j, P_1^i, \mathsf{st}_{j-1})$,
   - $\langle M_0^j, M_1^j\rangle \leftarrow \mathsf{POEE.ApplyPatch}\Big(\langle M_0^{j-1}, M_1^{j-1}\rangle, \langle P_0^j, P_1^j\rangle\Big)$,
   - $\langle x', i', b\rangle \leftarrow \mathsf{POEE.PIEncode}(\mathsf{POEE.puncInp}(\mathsf{POEE.sk}, x, i), x', i', b)$,
   - $M_\ell^b = \mathsf{Update}(M_{\ell-1}^b, P_\ell^b)$.
2. *Correctness of Bit Puncturing:* For all $M_0, M_1 \in \mathcal{M}$, every $L \geq 0$, patch sequence $P_0^1, P_1^1, \ldots, P_0^L, P_1^L \in \mathcal{P}$, $x \in \{0,1\}^*$ and $b \in \{0,1\}$,
$$\mathsf{POEE.Decode}\Big(\langle M_0^L, M_1^L\rangle, \langle x, L, b\rangle\Big) = M_b^L(x)$$
where:

- $\mathsf{POEE.sk} \leftarrow \mathsf{POEE.Setup}(1^\lambda)$,
- $\Big(\langle P_0^i, P_1^i\rangle, \mathsf{st}_i\Big) \leftarrow \mathsf{POEE.GenPatch}(\mathsf{POEE.sk}, P_0^i, P_1^i, \mathsf{st}_{i-1})$,
- $\langle M_0^i, M_1^i\rangle \leftarrow \mathsf{POEE.ApplyPatch}\Big(\langle M_0^{i-1}, M_1^{i-1}\rangle, \langle P_0^i, P_1^i\rangle\Big)$,
- $\langle x, L, b\rangle \leftarrow \mathsf{POEE.PBEncode}(\mathsf{POEE.puncBit}\,(\mathsf{POEE.sk}, b)\,, x, L)$,
- $M_j^b = \mathsf{Update}(M_{j-1}^b, P_j^b)$.

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

- Setup: This phase consists of the following steps:
  - $\mathcal{A}$ chooses two Turing machines $M_0, M_1 \in \mathcal{M}$ such that $|M_0| = |M_1|$, an input x and an index $\mathbf{i} \geq 0$. $\mathcal{A}$ sends the tuple $(M_0, M_1, x, \mathbf{i})$ to the challenger.
  - The challenger chooses a bit $b \in \{0,1\}$ and computes the following: (a) $\mathsf{POEE.sk} \leftarrow \mathsf{POEE.Setup}(1^\lambda)$, (b) $\Big(\langle M_0, M_1\rangle, \mathsf{st}_0\Big) \leftarrow \mathsf{POEE.TMEncode}(\mathsf{POEE.sk}, M_0, M_1)$, (c) $\mathsf{POEE.sk}_{x,\mathbf{i}} \leftarrow \mathsf{POEE.puncInp}(\mathsf{POEE.sk}, x, \mathbf{i})$. It sends $\Big(\langle M_0, M_1\rangle, \mathsf{POEE.sk}_{x,\mathbf{i}}\Big)$ to $\mathcal{A}$.
  - If $\mathbf{i} = 0$, then the challenger also computes $\langle x, 0, b\rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, 0, b)$ and sends $\langle x, 0, b\rangle$ to $\mathcal{A}$.
- Patch Query phase: The following is repeated polynomially many times:
  - $\mathcal{A}$ chooses two patches $P_0^i, P_1^i \in \mathcal{P}$ and sends them to the challenger.
  - The challenger computes $\Big(\langle P_0^i, P_1^i\rangle, \mathsf{st}_i\Big) \leftarrow \mathsf{POEE.GenPatch}(\mathsf{POEE.sk}, P_0^i, P_1^i, \mathsf{st}_{i-1})$ and sends $\langle P_0^i, P_1^i\rangle$ to $\mathcal{A}$.
  - If $i = \mathbf{i}$, then the challenger also computes $\langle x, i, b\rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, b)$ and sends $\langle x, i, b\rangle$ to $\mathcal{A}$.
- Guess: $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$.

$\mathcal{A}$ is required to choose his queries s.t. $\mathsf{Update}(M_0^{\mathbf{i}-1}, P_0^{\mathbf{i}})(x)$ and $\mathsf{Update}(M_1^{\mathbf{i}-1}, P_1^{\mathbf{i}})(x)$, where $M_0^0 = M_0$ and $M_1^0 = M_1$. The advantage of $\mathcal{A}$ in this game is defined as $\mathsf{adv}_{\mathcal{A}} = \Pr[b' = b] - \frac{1}{2}$.

Definition 11 (Indistinguishability of encoding bit)**.** *A POEE scheme satisfies indistinguishability of encoding bit if there exists a neglible function* $\mathsf{negl}(\cdot)$ *such that for every PPT adversary* $\mathcal{A}$ *in the above security game,* $\mathsf{adv}_{\mathcal{A}} = \mathsf{negl}(\lambda)$.

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

- Setup: $\mathcal{A}$ chooses two Turing machines $M_0, M_1 \in \mathcal{M}$ such that $|M_0| = |M_1|$ and a bit $c \in \{0,1\}$. $\mathcal{A}$ sends the tuple (M_0, M_1, c) to the challenger.

  The challenger chooses a bit $b \in \{0,1\}$ and computes the following: (a) $\mathsf{POEE.sk} \leftarrow \mathsf{POEE.Setup}(1^\lambda)$, (b) $\Big(\langle \mathsf{TM}_1, \mathsf{TM}_2\rangle, \mathsf{st}_0\Big) \leftarrow \mathsf{POEE.TMEncode}(\mathsf{POEE.sk}, \mathsf{TM}_1, \mathsf{TM}_2)$, where $\mathsf{TM}_1 = M_0, \mathsf{TM}_2 =$

$M_{1\oplus b}$ if $c = 0$ and $\mathsf{TM}_1 = M_{0\oplus b}, \mathsf{TM}_2 = M_1$ otherwise, and (c) $\mathsf{POEE.sk}_b \leftarrow \mathsf{POEE.puncBit}(\mathsf{POEE.sk}, b)$. Finally, it sends the following tuple to $\mathcal{A}$:

$$\Big(\langle \mathsf{TM}_1, \mathsf{TM}_2 \rangle, \mathsf{POEE.sk}_b\Big).$$

- Patch Query phase: The following is repeated polynomially many times:
  - $\mathcal{A}$ chooses two patches $P_0^i, P_1^i \in \mathcal{P}$ and sends them to the challenger.
  - The challenger computes $\Big(\langle \mathsf{PT}_1^i, \mathsf{PT}_2^i \rangle, \mathsf{st}_i\Big) \leftarrow \mathsf{POEE.GenPatch}(\mathsf{POEE.sk}, \mathsf{PT}_1^i, \mathsf{PT}_2^i, \mathsf{st}_{i-1})$, where $\mathsf{PT}_1^i = P_0^i, \mathsf{PT}_2^i = P_{1\oplus b}^i$ if $c = 0$ and $\mathsf{PT}_1^i = P_{0\oplus b}^i, \mathsf{PT}_2^i = M_1^i$ otherwise. It sends $\langle \mathsf{PT}_1^i, \mathsf{PT}_2^i \rangle$ to $\mathcal{A}$.
- Guess: $\mathcal{A}$ outputs a bit $b' \in \{0, 1\}$.

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

Definition 12 (Indistinguishability of machine encoding)**.** *A POEE scheme satisfies indistinguishability of machine encoding if there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every PPT adversary* $\mathcal{A}$ *in the above security game,* $\mathsf{adv}_{\mathcal{A}} = \mathsf{negl}(\lambda)$.

We now formally define a POEE scheme.

Definition 13. *A POEE scheme is a basic POEE scheme and in addition is equipped with helper algorithms* ($\mathsf{POEE.puncInp}, \mathsf{POEE.PIEncode}, \mathsf{POEE.puncBit}, \mathsf{POEE.PBEncode}$)*. In addition, it satisfies (i) Correctness of bit puncturing and input puncturing, (ii) Indistinguishability of Encoding Bit and, (iii) Indistinguishability of Machine Encoding.*

6.1 Construction

In this section, we present a transformation from a two-outcome PABE scheme to a POEE scheme. A similar transformation from two-outcome ABE to oblivious evaluation encodings (OEE) was recently explored in [AJS15]. However, since we are considering a more general scenario (of allowing for patching), the transformation of [AJS15] fails in our setting. One important reason is the following: in [AJS15], the two-outcome ABE parameters are part of the OEE secret key. This creates a problem for us because in the case of two-outcome PABE, the parameters are "refreshed" after every patch is generated, whereas in the definition of patchable OEE, there is no provision to update the OEE secret key.[5] To get around this issue, we use a puncturable PRF key to succinctly compress the two-outcome PABE keys across all the update phases. This does not work in general since it could be the case every updated two-outcome PABE key is a complex function of the previous two-outcome PABE keys. However in our patchable ABE construction (Section 5.1), in every phase, the public parameters produced are generated *independently of the previous phases.* This is what crucially allows us to make our proof work.

[5]Even if we were to allow for updating the key, during the puncturing phase we need to provide a succinct key that allows for encoding inputs across all the phases. This further means that the size of the punctured key would blow up proportional to the number of patches issued.

Notation. We denote the class of Turing machines associated with patchable oblivious evaluation encoding to be $\mathcal{M}$. The family of patches associated with $\mathcal{M}$ is denoted as $\mathcal{P}$ and the update algorithm is denoted as Update. We make the following notational simplifications: $\mathcal{M}$ consists of only single-bit output Turing machines. In every machine M in $\mathcal{M}$, there is a special location on the work tape in which the output of the Turing machine (0 or 1) is written. Until the termination of the Turing machine, this location contains the symbol $\perp$. We use the notation $M(x)$ to denote the value contained in this special location.
We use the following ingredients in our construction.

1. A puncturable PRF family PRF.
2. A garbling scheme GC for circuits, denoted by $\mathsf{GC} = (\mathsf{Garble}, \mathsf{EvalGC})$.
3. A fully homomorphic encryption scheme for circuits with additive overhead (Section 2), denoted by $\mathsf{FHE} = (\mathsf{FHE.Setup}, \mathsf{FHE.Enc}, \mathsf{FHE.Eval}, \mathsf{FHE.Dec})$.
4. A 1-key two-outcome PABE for TMs scheme denoted by $\mathsf{TwoPABE} = (\mathsf{TwoPABE.Setup}, \mathsf{TwoPABE.KeyGen}, \mathsf{TwoPABE.GenPatch}, \mathsf{TwoPABE.ApplyPatch}, \mathsf{TwoPABE.Enc}, \mathsf{TwoPABE.Dec})$.
   - The Turing machine family associated with $\mathsf{TwoPABE}$ is denotes as $\mathcal{N}$ where every $N \in \mathcal{N}$ is of the form $N = N_{\left[\{\mathsf{FHE.pk}_b, \mathsf{FHE.CT}_{M_b}\}_{b\in\{0,1\}}\right]}$ where $\mathsf{FHE.pk}_b$ denotes a public-key of the FHE scheme FHE and $\mathsf{FHE.CT}_{M_b}$ denotes an encryption of $M_b \in \mathcal{M}$ w.r.t. $\mathsf{FHE.pk}_b$.
   - The patch family $\mathcal{Q}$ associated with $\mathcal{N}$ consists of patches $Q \in \mathcal{Q}$ of the form $Q = \{\mathsf{FHE.CT}_{P_b}\}_{b\in\{0,1\}}$ where $P_b \in \mathcal{P}$.
   - The update algorithm associated with $\mathcal{N}$ and $\mathcal{Q}$ is denoted as $\mathsf{Update}_{\mathcal{N},\mathcal{Q}}$. On input (N, Q), where $N = N_{\left[\{\mathsf{FHE.pk}_b, \mathsf{FHE.CT}_{M_b}\}_{b\in\{0,1\}}\right]}$ and $Q = \{\mathsf{FHE.CT}_{P_b}\}_{b\in\{0,1\}}$, $\mathsf{Update}_{\mathcal{N},\mathcal{Q}}$ outputs $N' = N_{\left[\{\mathsf{FHE.pk}_b, \mathsf{FHE.CT}_{M'_b}\}_{b\in\{0,1\}}\right]}$ where $\mathsf{FHE.CT}_{M'_b} = \mathsf{FHE.Eval}(\mathsf{FHE.pk}_b, \mathsf{Update}, \mathsf{FHE.CT}_{M_b}, \mathsf{FHE.CT}_{P_b})$.

Construction. We denote the patchable oblivious evaluation encoding scheme as $\mathsf{POEE} = (\mathsf{POEE.Setup}, \mathsf{POEE.InpEncode}, \mathsf{POEE.TMEncode}, \mathsf{POEE.Decode})$. We denote the auxiliary algorithms of POEE as $(\mathsf{POEE.puncInp}, \mathsf{POEE.PIEncode}, \mathsf{POEE.puncBit}, \mathsf{POEE.PBEncode})$. The construction of POEE is presented below.

$\underline{\mathsf{POEE.Setup}(1^\lambda)}$: On input a security parameter λ, execute the following steps.

- Sample a random key K^0 for the puncturable PRF family PRF.
- Execute $\mathsf{FHE.Setup}(1^\lambda)$ twice to obtain two public key,secret key pairs $(\mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0)$ and $(\mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$ for FHE.

Output $\mathsf{POEE.sk} = (K, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$.

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

- Parse $\mathsf{POEE.sk} = (\mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$.
- Compute $\mathsf{FHE.CT}_{M_0} \leftarrow \mathsf{FHE.Enc}(\mathsf{FHE.pk}_0, M_0)$ and $\mathsf{FHE.CT}_{M_1} \leftarrow \mathsf{FHE.Enc}(\mathsf{FHE.pk}_1, M_1)$.
- Compute $r \leftarrow \mathsf{PRF}_K(0)$.
- Compute $(\mathsf{TwoPABE.PP}, \mathsf{TwoPABE.SK}) \leftarrow \mathsf{TwoPABE.Setup}(1^\lambda)$ using randomness r.
- Compute a key $(\mathsf{TwoPABE.}sk_N, \mathsf{st}_{\mathsf{tpabe}}) \leftarrow \mathsf{TwoPABE.KeyGen}(\mathsf{TwoPABE.SK}, N)$ for the machine $N = N_{\left[\{\mathsf{FHE.pk}_b, \mathsf{FHE.CT}_{M_b}\}_{b\in\{0,1\}}\right]}$, where N is described in Figure 4.

Set $\mathsf{st} = (\mathsf{st}_{\mathsf{tpabe}}, \mathsf{ctr})$, where ctr is initialized to 0. Output the TM encoding, $\langle M_0, M_1\rangle = \mathsf{TwoPABE.}sk_N$.

$$N_{\left[\{\mathsf{FHE.pk}_b, \mathsf{FHE.CT}_{M_b}\}_{b\in\{0,1\}}\right]}(x, j, \mathsf{ind}_t)$$

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

  [*Note: If the running time of M is t then the universal Turing machine runs in time $O(t\cdot\log(t))$. This logarithmic overhead is unimportant and will be ignored in our analysis.*]
- Transform the universal Turing machine U into a circuit $C \leftarrow \mathsf{TMtoCKT}(U)$ using Theorem 5 (Section 2).
- Execute $\mathsf{FHE.Eval}(\mathsf{FHE.pk}_0, C, \mathsf{FHE.CT}_{M_0})$ and $\mathsf{FHE.Eval}(\mathsf{FHE.pk}_1, C, \mathsf{FHE.CT}_{M_1})$ to obtain z_1 and z_2, respctively.
- Set $z = (z_1||z_2)$. Output the j^{th} bit of z.

Figure 4: Description of TM N.

$\mathsf{POEE.GenPatch}(\mathsf{POEE.sk}, \mathsf{st}, P_0, P_1)$: On input a secret key $\mathsf{POEE.sk}$, state st and pair of patches $P_0, P_1 \in \mathcal{P}$, execute the following steps:

- Parse $\mathsf{POEE.sk} = (K, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$ and $\mathsf{st} = (\mathsf{st}_{\mathsf{tpabe}}, \mathsf{ctr})$.
- Compute $\mathsf{FHE.CT}_{P_0} \leftarrow \mathsf{FHE.Enc}(\mathsf{FHE.pk}_0, P_0)$ and $\mathsf{FHE.CT}_{P_1} \leftarrow \mathsf{FHE.Enc}(\mathsf{FHE.pk}_1, P_1)$.
- Set $\mathsf{ctr} = \mathsf{ctr} + 1$.
- Compute $r \leftarrow \mathsf{PRF}_K(\mathsf{ctr})$.
- Compute $(\widetilde{Q}, \mathsf{st}'_{\mathsf{tpabe}}) \leftarrow \mathsf{TwoPABE.GenPatch}(\mathsf{st}_{\mathsf{tpabe}}, Q)$ using randomness $s = s_1\|s_2$, where $s_1 = r$, s_2 is chosen uniformly at random and the patch $Q = \{\mathsf{FHE.CT}_{P_b}\}_{b\in\{0,1\}}$.

Set $\mathsf{st} = (\mathsf{st}'_{\mathsf{tpabe}}, \mathsf{ctr})$. Output $\langle P_0, P_1\rangle = \widetilde{Q}$.

$\mathsf{POEE.ApplyPatch}(\langle M_0, M_1\rangle, \langle P_0, P_1\rangle)$: On input a TM encoding $\langle M_0, M_1\rangle$ and patch encoding $\langle P_0, P_1\rangle$), execute the following steps:

- Parse $\langle M_0, M_1\rangle = \mathsf{TwoPABE.}sk_N$ and $\langle P_0, P_1\rangle = \widetilde{Q}$.
- Compute $\mathsf{TwoPABE.}sk_{N'} \leftarrow \mathsf{TwoPABE.ApplyPatch}(\mathsf{TwoPABE.}sk_N, \widetilde{Q})$.

Output $\langle M_0', M_1' \rangle = \mathsf{TwoPABE}.sk_{N'}$.

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

- Parse $\mathsf{POEE.sk}$ as $(K, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$.
- For $\mathsf{ind}_t \in [\lambda]$, compute a garbled circuit and wire keys $\big(\mathcal{GC}_{\mathsf{ind}_t}, \{w_{j,0}^{\mathsf{ind}_t}, w_{j,1}^{\mathsf{ind}_t}\}_{j \in [q]}\big) \leftarrow \mathsf{Garble}(1^\lambda, G)$, where $G = G_{(\mathsf{FHE.sk}_b, b)}(\cdot)$ is a circuit that takes as input FHE ciphertexts $(\mathsf{FHE.CT}_0, \mathsf{FHE.CT}_1)$ and outputs $\mathsf{FHE.Dec}(\mathsf{FHE.sk}_b, \mathsf{FHE.CT}_b)$. Here, q denotes the total length of two FHE ciphertexts $(\mathsf{FHE.CT}_0, \mathsf{FHE.CT}_1)$.
- Compute $r \leftarrow \mathsf{PRF}_K(i)$.
- Compute $(\mathsf{TwoPABE.PP}^i, \mathsf{TwoPABE.SK}^i) \leftarrow \mathsf{TwoPABE.Setup}(1^\lambda)$ using randomness r.
- For every $j \in [q]$ and $\mathsf{ind}_t \in [\lambda]$, compute an ABE ciphertext $\mathsf{TwoPABE.CT}_{j,\mathsf{ind}_t} \leftarrow \mathsf{TwoPABE.Enc}$ $\big(\mathsf{TwoPABE.PP}^i, (x, j, \mathsf{ind}_t), w_{j,0}^{\mathsf{ind}_t}, w_{j,1}^{\mathsf{ind}_t}\big)$ of message pair $(w_{j,0}^{\mathsf{ind}_t}, w_{j,1}^{\mathsf{ind}_t})$ associated to the attribute (x, j, ind_t).

Finally, it outputs the encoding:

$$\langle x, i, b \rangle = \Big(\{\mathcal{GC}\}_{\mathsf{ind}_t \in [\lambda]}, \{\mathsf{TwoPABE.CT}_{j,\mathsf{ind}_t}\}_{j \in [q], \mathsf{ind}_t \in [\lambda]}\Big).$$

$\underline{\mathsf{POEE.Decode}\Big(\langle M_0, M_1 \rangle, \langle x, i, b \rangle\Big)}$: On input a (possibly updated) TM encoding $\langle M_0, M_1 \rangle$ and input encoding $\langle x, i, b \rangle$, execute the following steps:

- Parse the TM encoding $\langle M_0, M_1 \rangle = \mathsf{TwoPABE}.sk_N$ and the input encoding $\langle x, i, b \rangle = \Big(\{\mathcal{GC}\}_{\mathsf{ind}_t \in [\lambda]},$ $\{\mathsf{TwoPABE.CT}_{j,\mathsf{ind}_t}\}_{j \in [q], \mathsf{ind}_t \in [\lambda]}\Big)$.
- For every $\mathsf{ind}_t \in [\lambda]$,
  1. For every $j \in [q]$, execute the decryption procedure of $\mathsf{TwoPABE}$ to obtain the wire keys of the garbled circuit, $\widetilde{w}_j^{\mathsf{ind}_t} \leftarrow \mathsf{TwoPABE.Dec}(\mathsf{TwoPABE}.sk_N, \mathsf{TwoPABE.CT}_{j,\mathsf{ind}_t})$.
  2. Executes $\mathsf{EvalGC}(\mathcal{GC}_{\mathsf{ind}_t}, \widetilde{w}_1^{\mathsf{ind}_t}, \ldots, \widetilde{w}_q^{\mathsf{ind}_t})$ to obtain $\mathsf{out}_{\mathsf{ind}_t}$.
  3. If $\mathsf{out}_{\mathsf{ind}_t} \neq \perp$ then output $\mathsf{out} = \mathsf{out}_{\mathsf{ind}_t}$. Otherwise, continue.

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

$\underline{\mathsf{POEE.puncInp}(\mathsf{POEE.sk}, x, i)}$: On input a secret key $\mathsf{POEE.sk} = (K, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$ and (x, i), it computes:

- $r \leftarrow \mathsf{PRF}_K(i)$.
- $(\mathsf{TwoPABE.PP}^i, \mathsf{TwoPABE.SK}^i) \leftarrow \mathsf{TwoPABE.Setup}(1^\lambda)$ using randomness r.
- $K_i \leftarrow \mathsf{PRFPunc}(K, i)$.

It outputs $\mathsf{POEE.sk}_{x,i} = (\mathsf{TwoPABE.PP}^i, K_i, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$. That is, the punctured key is same as the original secret key except that the PRF key K is replaced with $\mathsf{TwoPABE.PP}^i$ and K_i.

$\underline{\mathsf{POEE.PIEncode}(\mathsf{POEE.sk}_{x,i}, x', i', b)}$: On input a punctured key $\mathsf{POEE.sk}_{x,i}$ and input (x', i', b), it executes the following steps:

- Parse $\mathsf{POEE.sk}_{x,i} = (\mathsf{TwoPABE.PP}^i, K_i, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$.
- For $\mathsf{ind}_t \in [\lambda]$, compute a garbled circuit and wire keys $\left(\mathcal{GC}_{\mathsf{ind}_t}, \{w_{j,0}^{\mathsf{ind}_t}, w_{j,1}^{\mathsf{ind}_t}\}_{j\in[q]}\right) \leftarrow \mathsf{Garble}(1^\lambda, G)$, where $G = G_{(\mathsf{FHE.sk}_b, b)}(\cdot)$ is a circuit that takes as input FHE ciphertexts $(\mathsf{FHE.CT}_0, \mathsf{FHE.CT}_1)$ and outputs $\mathsf{FHE.Dec}(\mathsf{FHE.sk}_b, \mathsf{FHE.CT}_b)$. Here, q denotes the total length of two FHE ciphertexts $(\mathsf{FHE.CT}_0, \mathsf{FHE.CT}_1)$.
- If $i' \neq i$, then compute $r \leftarrow \mathsf{PRF}_{K_i}(i')$ and $(\mathsf{TwoPABE.PP}^{i'}, \mathsf{TwoPABE.SK}^{i'}) \leftarrow \mathsf{TwoPABE.Setup}(1^\lambda)$ using randomness r.
- For every $j \in [q]$ and $\mathsf{ind}_t \in [\lambda]$, compute an ABE ciphertext $\mathsf{TwoPABE.CT}_{j,\mathsf{ind}_t} \leftarrow \mathsf{TwoPABE.Enc}$ $\left(\mathsf{TwoPABE.PP}^{i'}, (x', j, \mathsf{ind}_t), w_{j,0}^{\mathsf{ind}_t}, w_{j,1}^{\mathsf{ind}_t}\right)$ of message pair $(w_{j,0}^{\mathsf{ind}_t}, w_{j,1}^{\mathsf{ind}_t})$ associated to the attribute (x, j, ind_t)

Finally, it outputs the encoding:

$$\langle x', i', b\rangle = \left(\{\mathcal{GC}\}_{\mathsf{ind}_t\in[\lambda]}, \{\mathsf{TwoPABE.CT}_{j,\mathsf{ind}_t}\}_{j\in[q],\mathsf{ind}_t\in[\lambda]}\right).$$

$\underline{\mathsf{POEE.puncBit}(\mathsf{POEE.sk}, b)}$: On input a secret key $\mathsf{POEE.sk}$ and bit $b \in \{0,1\}$, it first parses $\mathsf{POEE.sk} = (K, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$. It then outputs the punctured key $\mathsf{POEE.sk}_b = (K, \mathsf{FHE.pk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_b)$. That is, the punctured key is same as the original key except that it does not contain the FHE key $\mathsf{FHE.sk}_{\overline{b}}$.

$\underline{\mathsf{POEE.PBEncode}(\mathsf{POEE.sk}_b, x, i)}$: On input a punctured key $\mathsf{POEE.sk}_b$, it computes and outputs $\langle x, i, b\rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}_b, x, i, b)$.
[*Note: The algorithm* $\mathsf{POEE.InpEncode}$ *can be executed on the punctured key* $\mathsf{POEE.sk}_b$*, input* x*, index* i *and bit* b *because the FHE secret key* $\mathsf{FHE.sk}_{\overline{b}}$ *is never used by it.*]
This completes the description of the auxiliary algorithms. Below we argue the efficiency property of POEE. The proof of correctness is given in Appendix 6.2 and the proof of security properties is given in Appendix 6.3.

Efficiency. From the description of the scheme and the efficiency of the TwoPABE scheme, the puncturable PRF PRF and the garbling scheme GC, it follows that $\mathsf{POEE.Setup}(1^\lambda)$ runs in time poly(λ), $\mathsf{POEE.TMEncode}(\mathsf{POEE.sk}, M_0, M_1)$ runs in time poly($\lambda, |M_0|, |M_1|$), $\mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, b)$ runs in time poly($\lambda, |x|$), $\mathsf{POEE.GenPatch}(\mathsf{POEE.sk}, \mathsf{st}, P_0, P_1)$ runs in time poly(λ, $|P_0|, |P_1|$), $\mathsf{POEE.ApplyPatch}(\langle M_0, M_1\rangle, \langle P_0, P_1\rangle)$ runs in time poly(λ, t_0, t_1) where t_b is the running time of $\mathsf{Update}(M_b, P_b)$.

The running time of $\mathsf{POEE.Decode}\Big(\langle M_0, M_1\rangle, \langle x, i, b\rangle\Big)$ is poly(λ, t^*), where t^* is the time taken to execute M_b on x. The main bottleneck in the running time of POEE.Decode is the number of the iterations it executes. Note that the j^{th} iteration takes time polynomial in λ and 2^j. If $\mathsf{ind}_t \in [\lambda]$ is the smallest number such that $2^{\mathsf{ind}_t} \geq t^*$ then the number of the iterations in the execution of decode is ind_t. Thus, the total running time of decode is $(\sum_{j=1}^{\mathsf{ind}_t} 2^j)\text{poly}(\lambda) = \text{poly}(\mathrm{t}^*, \lambda)$.

6.2 Proof of Correctness

Correctness of Encode, Patching and Decode. Let $M_0^0, M_1^0 \in \mathcal{M}$, $P_0^1, P_1^1, \ldots, P_0^L, P_1^L \in \mathcal{P}$, $x \in \{0,1\}^*$ and $b \in \{0,1\}$.

- It follows from the description of the scheme that $\langle M_0^0, M_1^0 \rangle = \mathsf{TwoPABE}.sk_{N^0}$, where $N^0 = N_{\left[\{\mathsf{FHE.pk}_b, \mathsf{FHE.CT}_{M_b^0}\}_{b\in\{0,1\}}\right]}$.
- Further, for every index $i \in [L]$, $\langle P_0^i, P_1^i \rangle = \widetilde{Q_i}$ where $(\widetilde{Q_i}, \mathsf{st}'_{\mathsf{tpabe}}) \leftarrow \mathsf{TwoPABE.GenPatch}(\mathsf{st}_{\mathsf{tpabe}}, Q_i)$, $Q_i = \mathsf{FHE.CT}_{P_0^i}, \mathsf{FHE.CT}_{P_1^i}$.
- For every $i \in [L]$, let $\langle M_0^i, M_1^i \rangle = \mathsf{ApplyPatch}(\langle M_0^{i-1}, M_1^{i-1} \rangle, \langle P_0^i, P_1^i \rangle)$. From the correctness of $\mathsf{TwoPABE.ApplyPatch}$, we have that $\langle M_0^i, M_1^i \rangle = \mathsf{TwoPABE}.sk_{N^i}$ where $N^i = \mathsf{Update}_{\mathcal{N},\mathcal{Q}}(N^{i-1}, Q_i) = N^i_{\left[\{\mathsf{FHE.pk}_b, \mathsf{FHE.CT}_{M_b^i}\}_{b\in\{0,1\}}\right]}$ and $M_b^i = \mathsf{Update}(M_b^{i-1}, P_b^i)$.
- Let $\langle x, L, b \rangle = \big(\{\mathcal{GC}\}_{\mathsf{ind}_t \in [\lambda]}, \{\mathsf{TwoPABE.CT}_{j,\mathsf{ind}_t}\}_{j\in[q],\mathsf{ind}_t\in[\lambda]}\big) \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, L, b)$.
- From the correctness of $\mathsf{TwoPABE}$, we have that the output of $\mathsf{TwoPABE.Dec}(\mathsf{TwoPABE}.sk_{N^L})$, $\mathsf{TwoPABE.CT}_{j,\mathsf{ind}_t})$ is the j^{th} wire key of $\mathcal{GC}_{\mathsf{ind}_t}$ which corresponds to the j^{th} bit of ($\mathsf{FHE.CT}_0$, $\mathsf{FHE.CT}_1$). Furthermore, from the correctness of FHE it follows that $\mathsf{FHE.CT}_0$ (resp., $\mathsf{FHE.CT}_1$) is an encryption of $M_0^L(x)$ (resp., $M_1^L(x)$), at 2^{ind_t} number of steps, under $\mathsf{FHE.pk}_0$ (resp., $\mathsf{FHE.pk}_1$).
- Let t^* be the runtime of M_b^L on input x. From the correctness of garbling schemes, it follows that the output of garbled circuit evaluation, $\mathsf{EvalGC}(\mathcal{GC}_{\mathsf{ind}_t}, \widetilde{w}_1^{\mathsf{ind}_t}, \ldots, \widetilde{w}_q^{\mathsf{ind}_t})$ is $M_b^L(x)$ when $2^{\mathsf{ind}_t} \geq t^*$ and is $\perp$ otherwise. Since M_b^L runs in polynomial time on all inputs, there will exist at least one $\mathsf{ind}_t \in [\lambda]$ such that $2^{\mathsf{ind}_t} \geq t^*$.

Therefore, the output of $\mathsf{POEE.Decode}\Big(\langle M_0^L, M_1^L \rangle, \langle x, L, b \rangle\Big)$ in this case would be $M_b^L(x)$, as desired.

Correctness of Input Puncturing. The proof of correctness in this case is similar to the above case. We only highlight the differences. Let $M_0, M_1 \in \mathcal{M}$, $P_0^1, P_1^1, \ldots, P_0^L, P_1^L \in \mathcal{P}$, $x, x' \in \{0,1\}^*$ and $i, i' \leq L$ s.t. $(x,i) \neq (x',i')$, and $b \in \{0,1\}$.

The main difference from the previous case is that here, input encoding $\langle x', i', b \rangle$ is computed for an index $i' \leq L$ using punctured key $\mathsf{POEE.sk}_{x,i}$ where $\mathsf{POEE.sk}_{x,i} \leftarrow \mathsf{POEE.puncInp}(\mathsf{POEE.sk}, x, i)$. Recall that $\mathsf{POEE.sk} = (K, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$ and $\mathsf{POEE.puncInp}(\mathsf{POEE.sk}, x, i) = (\mathsf{TwoPABE.PP}_i, K_i, \mathsf{FHE.pk}_0, \mathsf{FHE.sk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_1)$ where $(\mathsf{TwoPABE.PP}^i, \mathsf{TwoPABE.SK}^i) \leftarrow \mathsf{TwoPABE.Setup}(1^\lambda)$ is computed using randomness $r \leftarrow \mathsf{PRF}_K(i)$ and $K_i \leftarrow \mathsf{PRFPunc}(K, i)$.

Now, note that if $i = i'$, then $\mathsf{TwoPABE.PP}^{i'}$ is used to compute the input encoding $\langle x', i', b \rangle$ in the same manner as in the honest input encoding algorithm and therefore correctness follows from the correctness of the decryption procedure of $\mathsf{TwoPABE}$. On the other hand, if $i \neq i'$, then the punctured input encoding algorithm first computes $(\mathsf{TwoPABE.PP}^{i'}, \mathsf{TwoPABE.SK}^{i'}) \leftarrow \mathsf{TwoPABE.Setup}(1^\lambda)$ using randomness $r \leftarrow \mathsf{PRF}_{K_i}(i')$ and then computes the input encoding using $\mathsf{TwoPABE.PP}^{i'}$. The only difference here from the honest input encoding algorithm is that the randomness r is computed using the punctured PRF key K_i as opposed to the un-punctured PRF key K. However, from the correctness of puncturing of the PRF, it follows that $\mathsf{PRF}_{K_i}(i') = \mathsf{PRF}_K(i')$ when $i \neq i'$. Once again, correctness then follows from the correctness of the decryption procedure of TwoPABE.

Combining the above with the arguments in the proof of the previous case, we can establish that the output of $\mathsf{POEE.Decode}\Big(\langle M_0^{i'}, M_1^{i'}\rangle, \langle x', i', b\rangle\Big)$ in this case would be $M_b^{i'}(x')$, as desired.

Correctness of Bit Puncturing. Correctness follows easily in this case by the definition of POEE.PBEncode and correctness of the regular input encoding algorithm POEE.InpEncode (as established in the first case).

6.3 Proof of Security

We first prove that POEE satisfies the indistinguishability of encoding bit property and later we prove that it satisfies the indistinguishability of machine encoding property.

Theorem 11. *The scheme* POEE *satisfies indistinguishability of encoding bit property assuming the security of* TwoPABE *and security of garbling scheme* GC*.*

Proof. We first design a series of hybrids. The first hybrid corresponds to the real experiment where the challenger picks a bit b at random. In the last hybrid Hyb_3, the bit b is information theoretically hidden from the adversary. The probability that the adversary guesses b is with probability $1/2$. Then by arguing that every two consecutive hybrids are computationally indistinguishable, it follows that the probability that the adversary outputs b is negligibly close to $1/2$.

We denote the advantage of the adversary in Hyb_i to be $\mathsf{adv}_{\mathcal{A},i}$.

Hybrid Hyb_1: On receiving the TM pair (M_0, M_1), input x and index $\mathbf{i}$, the challenger first picks a bit $b \in \{0, 1\}$ at random and computes:

- $\mathsf{POEE.sk} \leftarrow \mathsf{POEE.Setup}(1^\lambda)$
- $\Big(\langle M_0, M_1\rangle, \mathsf{st}_0\Big) \leftarrow \mathsf{POEE.TMEncode}(\mathsf{POEE.sk}, M_0, M_1)$
- $\mathsf{POEE.sk}_{x,\mathbf{i}} \leftarrow \mathsf{POEE.puncInp}(\mathsf{POEE.sk}, x, \mathbf{i})$

It sends $\Big(\langle M_0, M_1\rangle, \mathsf{POEE.sk}_{x,\mathbf{i}}\Big)$ to $\mathcal{A}$. If $\mathbf{i} = 0$, then it also computes $\langle x, 0, b\rangle \leftarrow$ POEE.InpEncode $(\mathsf{POEE.sk}, x, 0, b)$ and sends $\langle x, 0, b\rangle$ to $\mathcal{A}$.

Next, upon receiving a patch query (P_0^i, P_1^i) from $\mathcal{A}$, the challenger computes $\Big(\langle P_0^i, P_1^i\rangle, \mathsf{st}_i\Big) \leftarrow$ $\mathsf{POEE.GenPatch}(\mathsf{POEE.sk}, \mathsf{st}_{i-1}, P_0^i, P_1^i)$ and sends $\langle P_0^i, P_1^i\rangle$ to $\mathcal{A}$. If $\mathbf{i} = i$, then it also computes $\langle x, i, b\rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, b)$ and sends $\langle x, i, b\rangle$ to $\mathcal{A}$.

Hybrid Hyb_2: Same as Hyb_1, except that we change the TwoPABE ciphertexts in the challenge input encoding $\langle x, \mathbf{i}, b\rangle$:

- For every $\mathsf{ind}_t \in [\lambda]$, the challenger computes a garbled circuit and wire keys $\big(\mathcal{GC}_{\mathsf{ind}_t}, \{w_{i,0}^{\mathsf{ind}_t}, w_{j,1}^{\mathsf{ind}_t}\}_{j\in[q]}\big) \leftarrow \mathsf{Garble}(1^\lambda, G)$ in the same manner as in the honest input encoding procedure.

- Let $M_b^0 = M_b$ and $N^0 = N_{[\{\mathsf{FHE.pk}_b, \mathsf{FHE.CT}_{M_b^0}\}_{b\in\{0,1\}}]}$ be as defined in the honest TM encoding procedure. Let $N^i \leftarrow \mathsf{Update}_{\mathcal{N},\mathcal{Q}}(N^{i-1}, Q^i)$ and $Q^i = \{\mathsf{FHE.CT}_{P_b^i}\}_{b\in\{0,1\}}$ for every i.[6]

  For every $j \in [q]$ and $\mathsf{ind}_t \in [\lambda]$, the challenger sets $\mathcal{W}_{j,\mathsf{ind}_t} = (w_{j,0}^{\mathsf{ind}_t}, 0^{\ell_w})$ if $N^{\mathbf{i}}(x, j, \mathsf{ind}_t) = 0$, otherwise it sets $\mathcal{W}_{j,\mathsf{ind}_t} = (0^{\ell_w}, w_{j,1}^{\mathsf{ind}_t})$, where ℓ_w is the length of a wire key of the garbled circuit. It then computes $\widetilde{y}_{j,\mathsf{ind}_t} \leftarrow \mathsf{TwoPABE.Enc}\big(\mathsf{TwoPABE.PP}^i, (x, j, \mathsf{ind}_t), \mathcal{W}_{j,\mathsf{ind}_t}\big)$ where $\mathsf{TwoPABE.PP}^i$ is defined in the same manner as in the honest input encoding algorithm.

The input encoding is set to be $\langle x, \mathbf{i}, b\rangle = \big(\{\mathcal{GC}_{\mathsf{ind}_t\in[\lambda]}\}_{\mathsf{ind}_t\in[\lambda]}, \{\widetilde{y}_{j,\mathsf{ind}_t}\}_{j\in[q],\mathsf{ind}_t\in[\lambda]}\big)$. This completes the description of Hyb_2.

Hybrid Hyb_3: The challenger now simulates the garbled circuits instead of generating them honestly during the computation of challenge input encoding $\langle x, \mathbf{i}, b\rangle$.

To accomplish this task, the challenger uses a simulated garbling procedure denoted by SimGC. It takes as input $(1^\lambda, |G|, \mathsf{out})$ and outputs a garbling of a circuit of size $|G|$ along with input wire keys such that the evaluation of the garbled circuit on the wire keys yields the result out.

The challenge input encoding $\langle x, \mathbf{i}, b\rangle$ is computed by executing the steps below:

- Let $M_b^0 = M_b$ and $M_b^i = \mathsf{Update}(M_b^{i-1}, P_b^i)$ for every i. Let the output of $M_b^{\mathbf{i}}$ on x be out and let t^* be running time of $M_b^{\mathbf{i}}$ on input x. (Note that t^* is also the running time of $M_{\overline{b}}^{\mathbf{i}}$ on input x.)

  For every $\mathsf{ind}_t \in [\lambda]$, set $\mathsf{out}_{\mathsf{ind}_t} = \mathsf{out}$ if $2^{\mathsf{ind}_t} \geq t^*$, and otherwise $\mathsf{out}_{\mathsf{ind}_t} = \perp$. Next, compute the simulated garbled circuit and wire keys $\big(\mathsf{SimGC}_{\mathsf{ind}_t}, \{w_j^{\mathsf{ind}_t}\}_{j\in[q]}\big) \leftarrow \mathsf{SimGarble}(1^\lambda, 1^{|G|}, \mathsf{out}_{\mathsf{ind}_t})$, where circuit G is as defined in the honest input encoding procedure.

- Compute the $\mathsf{TwoPABE}$ ciphertexts $\widetilde{y}_{j,\mathsf{ind}_t}$, for every $j \in [q], \mathsf{ind}_t \in [\lambda]$ in the same manner as in the previous hybrid.

The input encoding is set to be $\langle x, \mathbf{i}, b\rangle = \big(\{\mathsf{SimGC}_{\mathsf{ind}_t}\}_{\mathsf{ind}_t\in[\lambda]}, \{\widetilde{y}_{j,\mathsf{ind}_t}\}_{j\in[q],\mathsf{ind}_t\in[\lambda]}\big)$. This completes the description of Hyb_3.

Lemma 4. *Assuming the security of* $\mathsf{TwoPABE}$*,* $|\mathsf{adv}_{\mathcal{A},1} - \mathsf{adv}_{\mathcal{A},2}| \leq \mathsf{negl}(\lambda)$*, where* negl *is a negligible function.*

Proof. To transition from Hyb_1 to Hyb_2, we change the $\mathsf{TwoPABE}$ ciphertexts in the challenge input encoding one at a time. Consider the following sequence of intermediate hybrids, $\mathsf{Hyb}_{1:\ell}$, for $\ell \in [q\lambda]$. The first hybrid $\mathsf{Hyb}_{1:1}$ is identical to Hyb_1 and the final intermediate hybrid $\mathsf{Hyb}_{1:q\lambda}$ is identical to Hyb_2.

Intermediate hybrid $\mathsf{Hyb}_{1:\ell}$, for $1 < \ell < q\lambda$: This is the same as $\mathsf{Hyb}_{1:\ell-1}$ except that the $\mathsf{TwoPABE}$ ciphertext $\widetilde{y}_{j^*,\mathsf{ind}_t^*}$, where $\ell = (j^* - 1)\cdot\lambda + \mathsf{ind}_t^*$ with $1 \leq j^* \leq q$ and $1 \leq \mathsf{ind}_t^* \leq \lambda$, is composed as follows: the challenger computes $\widetilde{y}_{j^*,\mathsf{ind}_t^*} \leftarrow \mathsf{TwoPABE.Enc}(\mathsf{TwoPABE.PP}, (x, j^*, \mathsf{ind}_t^*), \mathcal{W}_c)$, where $\mathcal{W}_c$ is defined below. As in the description of Hyb_2, here $(w_{j^*,0}^{\mathsf{ind}_t^*}, w_{j^*,1}^{\mathsf{ind}_t^*})$ denotes the j^{*th} wire keys

[6]Note that when computed in the manner as described above, we obtain $N^i = N^i_{[\{\mathsf{FHE.pk}_b, \mathsf{FHE.CT}_{M_b^i}\}_{b\in\{0,1\}}]}$.

corresponding to the ind_t^*-th garbled circuit.

$$\mathcal{W}_c = \begin{cases} (w_{j^*,0}^{\mathsf{ind}_t^*}, \perp) & \text{if } N^{\mathbf{i}}(x, j^*, \mathsf{ind}_t^*) = 0, \\ \\ (\perp, w_{j^*,1}^{\mathsf{ind}_t^*}) & \text{if } N^{\mathbf{i}}(x, j^*, \mathsf{ind}_t^*) = 1 \end{cases}$$

The rest of the hybrid is as in $\mathsf{Hyb}_{1:\ell-1}$.

We thus have the following claim.

Claim 4. *Assuming the security of* TwoPABE, $|\mathsf{adv}_{\mathcal{A},1:\ell-1} - \mathsf{adv}_{\mathcal{A},1:\ell}| \leq \mathsf{negl}(\lambda)$ *for every* $1 < \ell \leq q\lambda$, *where* negl *is a negligible function.*

Hence,

$$|\mathsf{adv}_{\mathcal{A},1} - \mathsf{adv}_{\mathcal{A},2}| = \sum_{\ell=2}^{q\lambda} |\mathsf{adv}_{\mathcal{A},1:\ell-1} - \mathsf{adv}_{\mathcal{A},1:\ell}| \leq \mathsf{negl}(\lambda)$$

□

Lemma 5. *Assuming the security of the garbling scheme* GC, $|\mathsf{adv}_{\mathcal{A},2} - \mathsf{adv}_{\mathcal{A},3}| \leq \mathsf{negl}(\lambda)$, *where* negl *is a negligible function.*

Proof. We consider a sequence of intermediate hybrids where we change one garbled circuit at a time. Consider the following sequence of intermediate hybrids $\mathsf{Hyb}_{2:j}$, for $j \in [\lambda]$. The first hybrid $\mathsf{Hyb}_{2:1}$ is identical to Hyb_2 and the final intermediate hybrid $\mathsf{Hyb}_{2:\lambda}$ is identical to Hyb_3. For $j \in [\lambda]$ and $j > 1$ we define the following sequence of hybrids,

Intermediate hybrid, $\mathsf{Hyb}_{2:\ell}$: This hybrid is identical to $\mathsf{Hyb}_{2:\ell-1}$ except in the generation of ℓ^{th} garbled circuit during the computation of the challenge input encoding. Let t^* be such that $M_b^{\mathbf{i}}(x)$ takes t^* number of steps. If ℓ is such that $2^\ell < t^*$ then generate $\left(\mathsf{SimGC}_\ell, \{w_j^\ell\}_{j\in[q]}\right) \leftarrow \mathsf{SimGarble}(1^\lambda, |G|, \perp)$. Otherwise, generate $\left(\mathsf{SimGC}_\ell, \{w_j^\ell\}_{j\in[q]}\right) \leftarrow \mathsf{SimGarble}(1^\lambda, |G|, M_b^{\mathbf{i}}(x))$. The rest of the garbled circuits and the TwoPABE ciphertexts are generated as in $\mathsf{Hyb}_{2:\ell-1}$.

We thus have the following claim.

Claim 5. *Assuming the security of the garbling scheme* GC, *we have* $|\mathsf{adv}_{\mathcal{A},2:\ell-1} - \mathsf{adv}_{\mathcal{A},2:\ell}| \leq \mathsf{negl}(\lambda)$ *for every* $1 < \ell \leq \lambda$, *where* negl *is a negligible function.*

We thus have,

$$|\mathsf{adv}_{\mathcal{A},2} - \mathsf{adv}_{\mathcal{A},3}| = \sum_{\ell=2}^{\lambda} |\mathsf{adv}_{\mathcal{A},2:\ell-1} - \mathsf{adv}_{\mathcal{A},2:\ell}| \leq \mathsf{negl}(\lambda)$$

□

The probability that $\mathcal{A}$ outputs b in Hyb_3 is $1/2$ since b is information theoretically hidden. Further from Lemmas 4, 5, we have that $|\mathsf{adv}_{\mathcal{A},1} - \mathsf{adv}_{\mathcal{A},3}| \leq \mathsf{negl}(\lambda)$. Combining these two facts we have, $\mathsf{adv}_{\mathcal{A},1} \leq \mathsf{negl}(\lambda)$, as desired. □

Theorem 12. *Assuming the security of* FHE, *the proposed scheme* POEE *satisfies the indistinguishability of machine encoding property.*

Proof. Let $M_0, M_1 \in \mathcal{M}$ be the TMs and c be the bit sent by the adversary to the challenger. Further, let $P_0^i, P_1^i \in \mathcal{P}$ be the patch queries made by the adversary. Below we prove the theorem for the case when $c = 0$. It is easy to extend the proof to the case when $c = 1$.

Let b be the challenge bit chosen by the challenger. Recall that the challenger sends the following values to the adversary:

- Punctured key $\mathsf{POEE.sk}_b = (K, \mathsf{FHE.pk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_b)$.
- TM encoding $\langle \mathsf{TM}_0, \mathsf{TM}_1 \rangle = \mathsf{TwoPABE}.sk_N$ where $N = N_{[\{\mathsf{FHE.pk}_b, \mathsf{FHE.CT}_{\mathsf{TM}_b}\}_{b\in\{0,1\}}]}$ and $\mathsf{TM}_0 = M_0, \mathsf{TM}_1 = M_{1\oplus b}$.
- Patch encodings $\langle \mathsf{PT}_0^i, \mathsf{PT}_1^i \rangle = \widetilde{Q}$ where $(\widetilde{Q}, \mathsf{st}'_{\mathsf{tpabe}}) \leftarrow \mathsf{TwoPABE.GenPatch}(\mathsf{st}_{\mathsf{tpabe}}, Q)$, $Q = \mathsf{FHE.CT}_{\mathsf{PT}_0^i}, \mathsf{FHE.CT}_{\mathsf{PT}_1^i}$, $\mathsf{st}_{\mathsf{tpabe}}$ is as defined in the honest patch generation algorithm, and $\mathsf{PT}_0^i = P_0^i$, $\mathsf{PT}_1^i = P_{1\oplus b}^i$.

From the semantic security of FHE, the adversary cannot distinguish the case when $(\mathsf{TM}_1, \{\mathsf{PT}_1^i\}) = (M_{1\oplus b}, \{P_{1\oplus b}^i\})$ from the case when $(\mathsf{TM}_1, \{\mathsf{PT}_1^i\}) = (M_{1\oplus \bar{b}}, \{P_{1\oplus \bar{b}}^i\})$. This completes the proof. $\square$

From Theorem 11 and Theorem 12, it follows that POEE is a secure scheme. Now, instantiating the underlying tools we obtain the following theorem.

Theorem 13. *Assuming the security of* $\mathsf{TwoPABE}$, FHE *and* GC, *it follows that the scheme* POEE *is a secure POEE scheme.*

The scheme $\mathsf{TwoPABE}$ can be instantiated with indistinguishability obfuscation and one-way functions from Corollary 1 and Theorem 10. We instantiate FHE by sub-exponentially secure indistinguishability obfuscation and sub-exponentially secure one-way functions using the work of Canetti et al. [CLTV15]. And finally, GC can be constructed one-way functions [Yao86]. Thus, we have the following corollary,

Corollary 2. *Assuming the existence of $\frac{\epsilon}{2^\lambda}$-secure iO and $\frac{\epsilon'}{2^\lambda}$-secure one-way functions, there exists a δ-secure POEE scheme, where $\epsilon, \epsilon', \delta \leq \frac{1}{\mathrm{poly}(\lambda)}$.*

7 From POEE to Single Program Patchable Obfuscation

In this section, we give a transformation from patchable oblivious evaluation encodings to achieve patchable obfuscation (PO). We use iO for circuits as an intermediary tool. The starting point is the transformation of (non-patchable) OEE to iO described in [AJS15]. As part of the obfuscation of a Turing machine M, we provide an OEE encoding of M as well as an obfuscated input encoder that produces input encodings of OEE for any input x to M. The puncturing properties of the underlying OEE scheme allows for switching from one branch of computation to another in the proof of security.

While our transformation from POEE to PO is similar in spirit to [AJS15], we have to deal with technical obstacles: the input encoder needs to now work with different (updated) machine encodings as against just one in the case of [AJS15]. We overcome this challenge by giving a fresh input encoder along with every patch encoding. But now we need to have a common secret key (POEE secret key) shared across all the input encoders. So puncturing the key in one input encoder affects the other encoders as well. Also, we need to now go over the input space *once for every patch issued* unlike [AJS15]. Our carefully designed properties of patchable OEE come in handy when dealing with these issues. We provide more details in the formal proof of security.

Notation. We denote the class of Turing machines associated with the patchable obfuscation scheme as $\mathcal{M}$. The family of patches associated with $\mathcal{M}$ is denoted as $\mathcal{P}$ and the update algorithm is denoted as Update.

7.1 Construction

We use the following ingredients in our construction.

1. A puncturable PRF family, denoted by PRF.
2. An indistinguishability obfuscator for general circuits, denoted by $i\mathcal{O}$.
3. A POEE scheme $\mathsf{POEE} = (\mathsf{POEE.Setup}, \mathsf{POEE.InpEncode}, \mathsf{POEE.GenPatch}, \mathsf{POEE.ApplyPatch}, \mathsf{POEE.TMEncode}, \mathsf{POEE.Decode})$ for the Turing machine family $\mathcal{M}$ with associated patch family $\mathcal{P}$ and update algorithm Update.

We now give our construction of an adaptive-IND secure patchable obfuscation scheme pO.

$\underline{\mathsf{Setup}(1^\lambda)}$: Compute $\mathsf{POEE.sk} \leftarrow \mathsf{POEE.Setup}(1^\lambda, T)$, where T is the bound on the running time of M. Output $\mathsf{Obf.SK} = \mathsf{POEE.sk}$.

$\underline{\mathsf{Obfuscate}(\mathsf{Obf.SK}, M)}$: Compute the following:

1. $(\langle M, M\rangle_{\mathsf{poee}}, \mathsf{st}) \leftarrow \mathsf{POEE.TMEncode}(\mathsf{POEE.sk}, M, M)$.
2. A random key K^0 for the puncturable PRF family.
3. $\langle C_0\rangle_{\mathsf{io}} \leftarrow i\mathcal{O}\left(C_{[0,K^0,\mathsf{POEE.sk}]}\right)$, where $C_{[i,K^i,\mathsf{POEE.sk}]}$ is the circuit described in Figure 5.

$C_{[i,K^i,\mathsf{POEE.sk}]}\,(x)$

(a) Compute $r \leftarrow \mathsf{PRF}_{K^i}(x\|0)$.

(b) Compute $\langle x, i, 0\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, 0)$ using randomness r.

(c) Output $\langle x, i, 0\rangle_{\mathsf{poee}}$.

Figure 5: Circuit $C_{[i,K^i,\mathsf{POEE.sk}]}$

Set $\mathsf{ctr} = 0$ and output $\langle M\rangle = \Big(\langle M, M\rangle_{\mathsf{poee}}, \langle C_0\rangle_{\mathsf{io}}\Big)$.

$\underline{\mathsf{GenPatch}(\mathsf{Obf.SK}, P, \mathsf{st})}$: Compute the following:

1. $\mathsf{ctr} = \mathsf{ctr} + 1$.
2. $\Big(\langle P, P\rangle_{\mathsf{poee}}, \mathsf{st}'\Big) \leftarrow \mathsf{POEE.GenPatch}(\mathsf{POEE.sk}, P, P, \mathsf{st})$. Update $\mathsf{st} = \mathsf{st}'$.
3. A random key K^{ctr} for the puncturable PRF family.
4. $\langle C_{\mathsf{ctr}}\rangle_{\mathsf{io}} \leftarrow i\mathcal{O}\left(C_{[\mathsf{ctr},K^{\mathsf{ctr}},\mathsf{POEE.sk}]}\right)$, where $C_{[i,K^i,\mathsf{POEE.sk}]}$ is the circuit described in Figure 5.

Output $\langle P\rangle = \Big(\langle P, P\rangle_{\mathsf{poee}}, \langle C_{\mathsf{ctr}}\rangle_{\mathsf{io}}\Big)$.

$\underline{\mathsf{ApplyPatch}\Big(\langle M\rangle, \langle P\rangle\Big)}$: Execute the following steps:

1. Parse $\langle M \rangle = \Big(\langle M, M\rangle_{\mathsf{poee}}, \langle C_{\mathsf{ctr}-1}\rangle_{\mathsf{io}}\Big)$ and $\langle P \rangle = \Big(\langle P, P\rangle_{\mathsf{poee}}, \langle C_{\mathsf{ctr}}\rangle_{\mathsf{io}}\Big)$.

2. Compute $\langle M_{new}, M_{new}\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.ApplyPatch}\Big(\langle M, M\rangle_{\mathsf{poee}}, \langle P, P\rangle_{\mathsf{poee}}\Big)$.

Output $\langle M_{new}\rangle = \Big(\langle M_{new}, M_{new}\rangle_{\mathsf{poee}}, \langle C_{\mathsf{ctr}}\rangle_{\mathsf{io}}\Big)$.

$\underline{\mathsf{Evaluate}\Big(\langle M\rangle, x\Big)}$: Execute the following steps:

1. Parse $\langle M\rangle = (\langle M, M\rangle_{\mathsf{poee}}, \langle C_i\rangle_{\mathsf{io}})$.

2. Compute $\langle x, i, 0\rangle_{\mathsf{poee}} \leftarrow \langle C_i\rangle_{\mathsf{io}}$.

3. Compute $y \leftarrow \mathsf{POEE.Decode}\Big(\langle M, M\rangle_{\mathsf{poee}}, \langle x, i, 0\rangle_{\mathsf{poee}}\Big)$

Output y.

This completes the description of pO.

Theorem 14. *Let $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$, $\widetilde{\mathcal{C}} = \{\widetilde{\mathcal{C}_\lambda}\}_{\lambda\in\mathbb{N}}$, where $\mathcal{C}_\lambda, \widetilde{\mathcal{C}_\lambda}$ each consist of circuits with input length λ. If* PRF *is a $\frac{\epsilon}{2^{\ell(\lambda)}}$-secure puncturable PRF,* POEE *is a $\frac{\epsilon'}{2^{\ell(\lambda)}}$-secure patchable OEE scheme and $i\mathcal{O}$ is a $\frac{\epsilon''}{2^{\ell(\lambda)}}$-secure indistinguishability obfuscator for $\widetilde{\mathcal{C}_{\ell(\lambda)}}$, then* pO *is a δ-secure patchable indistinguishability obfuscator for $\mathcal{C}_\lambda$, where $\epsilon, \epsilon', \epsilon'', \delta \leq \frac{1}{p(\lambda)}$ with p being a polynomial.*

We start by arguing that pO satisfies the correctness and efficiency properties. We then proceed to prove its security.

Correctness. Let $M_0 \in \mathcal{M}$ be a TM that we want to obfuscate and let $P_1, \ldots, P_L \in \mathcal{P}$ be a sequence of patches for $L > 0$. We argue correctness of pO for $L = 1$. The proof extends in a straightforward manner to the case when $L > 1$.

We first argue that $\langle M_0\rangle = \Big(\langle M_0, M_0\rangle_{\mathsf{poee}}, \langle C_0\rangle_{\mathsf{io}}\Big)$ is functionally equivalent to M_0. To evaluate $\Big(\langle M_0, M_0\rangle_{\mathsf{poee}}, \langle C_0\rangle_{\mathsf{io}}\Big)$ on any input x, the evaluator first computes $\langle C_0\rangle_{\mathsf{io}}(x)$. From the correctness of $i\mathcal{O}$, it follows that the output of $\langle C_0\rangle_{\mathsf{io}}(x) = C_{[0,K^0,\mathsf{POEE.sk}]}(x)$. From the definition of $C_{[0,K^0,\mathsf{POEE.sk}]}(\cdot)$, the correctness of the puncturable PRF scheme and the correctness of the POEE scheme, it follows that $\langle C_0\rangle_{\mathsf{io}}(x) = \langle x, 0, 0\rangle_{\mathsf{poee}}$. In the second step, the evaluator computes $y \leftarrow \mathsf{POEE.Decode}\Big(\langle M_0, M_0\rangle_{\mathsf{poee}}, \langle x, 0, 0\rangle_{\mathsf{poee}}\Big)$. From the correctness of the POEE scheme, it follows that $y = M_0(x)$, as required.

Now consider the encoded patch $\langle P_1\rangle = \Big(\langle P_1, P_1\rangle_{\mathsf{poee}}, \langle C_1\rangle_{\mathsf{io}}\Big)$. Let $M_1 = \mathsf{Update}(M_0, P_1)$. We argue that $\mathsf{ApplyPatch}\Big(\langle M_0\rangle, \langle P_1\rangle\Big)$ is functionally equivalent to M_1. Recall that $\mathsf{ApplyPatch}$ $\Big(\langle M_0\rangle, \langle P_1\rangle\Big)$ outputs $\langle M_1\rangle = \Big(\langle M_1, M_1\rangle_{\mathsf{poee}}, \langle C_1\rangle_{\mathsf{io}}\Big)$ where $\langle M_1, M_1\rangle_{\mathsf{poee}} = \mathsf{POEE.ApplyPatch}\Big($ $\langle M_0, M_0\rangle_{\mathsf{poee}}, \langle P_1, P_1\rangle_{\mathsf{poee}}\Big)$. From the correctness of $i\mathcal{O}$, it follows that the output of $\langle C_1\rangle_{\mathsf{io}}(x) = C_{[1,K^1,\mathsf{POEE.sk}]}(x)$. Further, from the definition of $C_{[1,K^1,\mathsf{POEE.sk}]}(\cdot)$, the correctness of the puncturable PRF scheme and the correctness of the POEE scheme, it follows that for any input x, $\langle C_1\rangle_{\mathsf{io}}(x) = \langle x, 1, 0\rangle_{\mathsf{poee}}$. Now, let $y \leftarrow \mathsf{POEE.Decode}\Big(\langle M_1, M_1\rangle_{\mathsf{poee}}, \langle x, 1, 0\rangle_{\mathsf{poee}}\Big)$. Then, it follows from the correctness of the POEE scheme that $y = M_1(x)$, as required.

Efficiency. The efficiency properties of pO follow from the efficiency of POEE. Recall that for any patch $P \in \mathcal{P}$, $\langle P \rangle = \Big(\langle P, P\rangle_{\mathsf{poee}}, \langle C_i \rangle_{\mathsf{io}}\Big)$ for some index $i > 0$. From the efficiency of POEE, it follows that $|\langle P, P\rangle_{\mathsf{poee}}| = \mathrm{poly}(\lambda, |P|)$. Further, from the efficiency of $i\mathcal{O}$, we have that $|\langle C_i\rangle_{\mathsf{io}}| = \mathrm{poly}(\lambda, |C_{[i,K^i,\mathsf{POEE.sk}]}|)$. Let I denote the input length bound for the TM family $\mathcal{M}$. Then, from the description of $C_{[i,K^i,\mathsf{POEE.sk}]}$, it follows that $|\langle C_i\rangle_{\mathsf{io}}| = \mathrm{poly}(\lambda, I)$. Putting it all together, we have that $\langle P \rangle = \mathrm{poly}(\lambda, |P|, I)$, as required.

7.2 Proof of Security

Let $M_0, M_1 \in \mathcal{M}$ and $P_0^1, P_1^1, \ldots, P_0^L, P_1^L \in \mathcal{P}$ be the adaptive queries made by the adversary in the adaptive security game such that:

- M_0 and M_1 are functionally equivalent and $|M_0| = |M_1|$.
- For every $i \in [L]$, $M_0^i = \mathsf{Update}(M_0^{i-1}, P_0^i)$ and $M_1^i = \mathsf{Update}(M_1^{i-1}, P_1^i)$ are functionally equivalent and $|M_i^0| = |M_i^1|$, where $M_0^0 = M_0$ and $M_1^0 = M_1$.

For simplicity, we assume that $T = 2^I$ is the total number of inputs to every M_0^i and M_1^i.[7] We now prove ϵ-security of pO, where $\epsilon = \mathsf{adv}^{\mathsf{me}}_{\mathsf{poee}} + T \cdot \big(\mathsf{adv}_{\mathsf{PRF}}(\lambda) + \mathsf{adv}_{i\mathcal{O}}(\lambda) + \mathsf{adv}^{\mathsf{eb}}_{\mathsf{poee}}(\lambda)\big)$, ignoring multiplicative factors of $O(L)$. Here $\mathsf{adv}^{\mathsf{me}}_{\mathsf{poee}}$ denotes the advantage of an adversary in the indistinguishability of machine encoding experiment for POEE. Similarly, $\mathsf{adv}^{\mathsf{eb}}_{\mathsf{poee}}$ denotes the advantage of an adversary in the indistinguishability of encoding bit experiment for POEE. Since punctured PRF can be based on one-way functions and our construction of POEE is based on one-way functions and $i\mathcal{O}$ for circuits, we get $\epsilon = T \cdot \mathrm{poly}\left(\mathsf{adv}_{\mathrm{OWF}}(\lambda) + \mathsf{adv}_{i\mathcal{O}}(\lambda)\right)$. Choosing $\mathsf{adv}_{\mathrm{OWF}}(\lambda)$ and $\mathsf{adv}_{i\mathcal{O}}(\lambda)$ to be sub-exponentially small (s.t. $\mathsf{adv}_{\mathrm{OWF}}(\lambda) + \mathsf{adv}_{i\mathcal{O}}(\lambda) \leq \frac{\mathsf{negl}(\lambda)}{T}$), we obtain $\epsilon = \mathsf{negl}(\lambda)$.

We prove the security of the construction by a hybrid argument. We will consider a sequence of five main hybrids $H_0, \ldots, H_5$ such that H_0 (resp., H_5) denotes the real world experiment where the adversary is given the obfuscations of M_0 and P_0^i (resp., M_1 and P_1^i).

Hybrid H_0: Real world experiment where machine M_0 and patches P_0^i are obfuscated. The adversary receives $\langle M_0 \rangle = \Big(\langle M_0, M_0\rangle_{\mathsf{poee}}, \langle C_0\rangle_{\mathsf{io}}\Big)$ and $\langle P_0^i \rangle = \Big(\langle P_0^i, P_0^i\rangle_{\mathsf{poee}}, \langle C_i\rangle_{\mathsf{io}}\Big)$ for every $i \in [L]$.

Hybrid H_1: Same as H_0, except that for every $i \in \{0, \ldots, L\}$, $\langle C_i\rangle_{\mathsf{io}}$ is now computed as $\langle C_i\rangle_{\mathsf{io}} \leftarrow i\mathcal{O}\left(C^1_{[i,K^i,\mathsf{POEE.sk}_0]}\right)$, where $\mathsf{POEE.sk}_0 \leftarrow \mathsf{POEE.puncBit}(\mathsf{POEE.sk}, 0)$ and $C^1_{[i,K^i,\mathsf{POEE.sk}_0]}$ is the circuit described in Figure 6.

$C^1_{[i,K^i,\mathsf{POEE.sk}_0]}(x)$

1. Compute $r \leftarrow \mathsf{PRF}_{K^i}(x\|0)$.
2. Compute $\langle x, i, 0\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.PBEncode}(\mathsf{POEE.sk}_0, x)$ using randomness r.
3. Output $\langle x, i, 0\rangle_{\mathsf{poee}}$.

Figure 6: Circuit $C^1_{[i,K^i,\mathsf{POEE.sk}_0]}$.

[7]Our proof easily extends to the case where the total number of inputs to M_0^i, M_1^i increases with i.

Hybrid H_2: Same as H_1, except that we replace the encodings $\langle M_0, M_0\rangle_{\mathsf{poee}}$ and $\langle P_0^i, P_0^i\rangle_{\mathsf{poee}}$ with $\langle M_0, M_1\rangle_{\mathsf{poee}}$ and $\langle P_0^i, P_1^i\rangle_{\mathsf{poee}}$, respectively.

Hybrid H_3: Same as H_2, except that every $i \in \{0, \ldots, L\}$, $\langle C_i\rangle_{\mathsf{io}}$ is now computed as $\langle C_i\rangle_{\mathsf{io}} \leftarrow i\mathcal{O}\left(C^3_{[i,K^i,\mathsf{POEE.sk}_1]}\right)$, where $\mathsf{POEE.sk}_1 \leftarrow \mathsf{POEE.puncBit}(\mathsf{POEE.sk}, 1)$ and $C^3_{[i,K^i,\mathsf{POEE.sk}_1]}$ is the circuit described in Figure 7.

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

1. Compute $r \leftarrow \mathsf{PRF}_{K^i}(x\|1)$.
2. Compute $\langle x, i, 1\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.PBEncode}(\mathsf{POEE.sk}_1, x)$ using randomness r.
3. Output $\langle x, i, 1\rangle_{\mathsf{poee}}$.

Figure 7: Circuit $C^3_{[i,K^i,\mathsf{POEE.sk}_1]}$.

Hybrid H_4: Same as H_3, except that we replace the encodings $\langle M_0, M_1\rangle_{\mathsf{poee}}$ and $\langle P_0^i, P_1^i\rangle_{\mathsf{poee}}$ with $\langle M_1, M_1\rangle_{\mathsf{poee}}$ and $\langle P_1^i, P_1^i\rangle_{\mathsf{poee}}$, respectively.

Hybrid H_5: Same as H_4, except that $\langle C_i\rangle_{\mathsf{io}}$ is now computed as $\langle C_i\rangle_{\mathsf{io}} \leftarrow i\mathcal{O}\left(C_{[i,K^i,\mathsf{POEE.sk}]}\right)$ where $C_{[i,K^i,\mathsf{POEE.sk}]}$ is the circuit described in Figure 5. This is the real world experiment where machine M_1 and patches P_1^i are obfuscated.

This completes the description of the main hybrids.

Indistinguishability of H_0 and H_1. We show that for every $i \in \{0, \ldots, L\}$, circuits $C_{[i,K^i,\mathsf{POEE.sk}]}$ and $C^1_{[i,K^i,\mathsf{POEE.sk}_0]}$ are functionally equivalent. The indistinguishability of H_0 and H_1 then follows from the security of the indistinguishability obfuscator $i\mathcal{O}$.[8]

Circuit $C_{[i,K^i,\mathsf{POEE.sk}]}$ on input x computes $\langle x, i, 0\rangle_{\mathsf{poee}}$ using $\mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, 0)$ while $C^1_{[i,K^i,\mathsf{POEE.sk}_0]}$ computes $\langle x, i, 0\rangle_{\mathsf{poee}}$ using $\mathsf{POEE.PBEncode}(\mathsf{POEE.sk}_0, x)$. From the correctness of the bit puncturing property of the POEE scheme, it follows that $\mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, 0) = \mathsf{POEE.PBEncode}(\mathsf{POEE.sk}_0, x)$ (note that we generate the randomness r in the same manner in both $C_{[i,K^i,\mathsf{POEE.sk}]}$ and $C^1_{[i,K^i,\mathsf{POEE.sk}_0]}$). Thus, $C_{[i,K^i,\mathsf{POEE.sk}]}$ and $C^1_{[i,K^i,\mathsf{POEE.sk}_0]}$ are functionally equivalent.

Indistinguishability of H_1 and H_2. Note that in both H_1 and H_2, only the punctured key $\mathsf{POEE.sk}_0$ is used. Then, the indistinguishability of H_1 and H_2 follows from the indistinguishability of machine encoding property of the POEE scheme.

ϵ'-Indistinguishability of H_2 and H_3. We will prove that the experiments H_2 and H_3 are ϵ'-indistinguishable, where $\epsilon' = T \cdot (\mathsf{adv}_{\mathsf{PRF}}(\lambda) + \mathsf{adv}_{i\mathcal{O}}(\lambda) + \mathsf{adv}_{\mathrm{OEE}_1}(\lambda))$, ignoring multiplicative factors of $O(L)$.

The proof of this case involves several intermediate hybrids. We describe it in Section 7.2.1.

Indistinguishability of H_3 and H_4. Note that in both H_3 and H_4, only the punctured key $\mathsf{POEE.sk}_1$ is used. Then, the indistinguishability of H_3 and H_4 follows from the indistinguishability of machine encoding property of the POEE scheme.

[8] Here, we use security for $i\mathcal{O}$ for $L+1$ program pairs. Note that this follows in a straightforward manner from the security of $i\mathcal{O}$ for one program pair by a standard hybrid argument.

Indistinguishability of H_4 and H_5. The proof of this case follows in the same manner as the proof of indistinguishability of hybrids H_0 and H_1. We omit the details.

Completing the proof. Combining the above claims, it follows that experiments H_0 and H_5 are ϵ-indistinguishable, where $\epsilon = \mathsf{adv}_{\mathrm{OEE}_2} + T \cdot (\mathsf{adv}_{\mathsf{PRF}}(\lambda) + \mathsf{adv}_{i\mathcal{O}}(\lambda) + \mathsf{adv}_{\mathrm{OEE}_1}(\lambda))$, ignoring multiplicative factors of $O(L)$.

7.2.1 ϵ'-Indistinguishability of H_2 and H_3

To argue ϵ'-indistinguishability of H_2 and H_3, we will consider $(L+1)\cdot(T+1)$ internal hybrids $H_{2:\ell:0}, \ldots, H_{2:\ell:T}$, where $\ell \in \{0, \ldots, L\}$. Let $x_1, \ldots, x_T$ denote the T inputs, sorted in lexicographic order, to the machines M_0^ℓ and M_1^ℓ, where $M_b^0 = M_b$ and $M_b^i = \mathsf{Update}(M_b^{i-1}, P_b^i)$ for $b \in \{0,1\}$, $i \in [L]$.

We start by describing hybrids $H_{2:\ell:0}$ for $0 \le \ell \le L$.

Hybrid $H_{2:\ell:0}$: Same as H_2, except that:

- For $i < \ell$, $\langle C_i \rangle_{\mathsf{io}}$ is computed as $\langle C_i \rangle_{\mathsf{io}} \leftarrow i\mathcal{O}\left(C^{2:i<\ell:0}_{[i,K^i,\mathsf{POEE.sk}]}\right)$, where $C^{2:i<\ell:0}_{[i,K^i,\mathsf{POEE.sk}]}$ is described in Figure 8.

- $C^{2:i<\ell:0}_{[i,K^i,\mathsf{POEE.sk}]}(x)$

1. Compute $r \leftarrow \mathsf{PRF}_{K^i}(x\|1)$.
2. Compute $\langle x, i, 1\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, 1)$ using randomness r.
3. Output $\langle x, i, 1\rangle_{\mathsf{poee}}$.

Figure 8: Circuit $C^{2:i<\ell:0}_{[i,K^i,\mathsf{POEE.sk}]}$

- For $i \ge \ell$, $\langle C_i \rangle_{\mathsf{io}}$ is computed as $\langle C_i \rangle_{\mathsf{io}} \leftarrow i\mathcal{O}\left(C^{2:i\ge\ell:0}_{[i,K^i,\mathsf{POEE.sk}]}\right)$, where $C^{2:i\ge\ell:0}_{[i,K^i,\mathsf{POEE.sk}]}$ as described in Figure 9.

$C^{2:i\ge\ell:0}_{[i,K^i,\mathsf{POEE.sk}]}(x)$

1. Compute $r \leftarrow \mathsf{PRF}_{K^i}(x\|0)$.
2. Compute $\langle x, i, 0\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, 0)$ using randomness r.
3. Output $\langle x, i, 0\rangle_{\mathsf{poee}}$.

Figure 9: Circuit $C^{2:i\ge\ell:0}_{[i,K^i,\mathsf{POEE.sk}]}$

This completes the description of $H_{2:\ell:0}$. Next, we describe hybrids $H_{2:\ell:j}$, where $0 \le \ell \le L$ and $0 < j \le T$.

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

$C^{2:\ell:j}_{[\ell,K^\ell,\mathsf{POEE.sk}]}(x)$

1. If $x \le x_j$, then $b = 1$, else $b = 0$.
2. Compute $r \leftarrow \mathsf{PRF}_{K^\ell}(x\|b)$.
3. Compute $\langle x, \ell, b\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, \ell, b)$ using randomness r.
4. Output $\langle x, \ell, b\rangle_{\mathsf{poee}}$.

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

This completes the description of Hybrid $H_{2:\ell:j}$.

For every $0 \le j \le T$, we want to argue the indistinguishability of $H_{2:\ell:j}$ and $H_{2:\ell:j+1}$. To facilitate this, we consider another sequence of intermediate hybrids $H_{2:\ell:j:1}, \ldots, H_{2:\ell:j:4}$, where $0 \le j < T$. We describe them below.

Hybrid $H_{2:\ell:j:1}$: Same as $H_{2:\ell:j}$, except that:

1. $\langle C_\ell\rangle_{\mathsf{io}}$ is now computed as $\langle C_\ell\rangle_{\mathsf{io}} \leftarrow i\mathcal{O}\left(C^{2:\ell:j:1}_{\left[\ell,K^\ell_{x_{j+1}},\mathsf{POEE.sk}_{x_{j+1},\ell},\langle x_{j+1},\ell,0\rangle_{\mathsf{poee}}\right]}\right)$, where:
   - $K^\ell_{x_{j+1}} \leftarrow \mathsf{PRFPunc}(K^\ell, x_{j+1})$.
   - $\mathsf{POEE.sk}_{x_{j+1},\ell} \leftarrow \mathsf{POEE.puncInp}(\mathsf{POEE.sk}, x_{j+1}, \ell)$.
   - $\langle x_{j+1}, \ell, 0\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x_{j+1}, \ell, 0)$ using randomness $r \leftarrow \mathsf{PRF}(K^\ell, x_{j+1}\|0)$.
   - Circuit $C^{2:\ell:j:1}_{\left[\ell,K^\ell_{x_{j+1}},\mathsf{POEE.sk}_{x_{j+1},\ell},\langle x_{j+1},\ell,0\rangle_{\mathsf{poee}}\right]}$ contains the values $K^\ell_{x_{j+1}}$, $\mathsf{POEE.sk}_{x_{j+1},\ell}$ and $\langle x_{j+1}, \ell, 0\rangle_{\mathsf{poee}}$ hardwired, and is described in Figure 11.

$C^{2:\ell:j:1}_{\left[\ell,K^\ell_{x_{j+1}},\mathsf{POEE.sk}_{x_{j+1},\ell},\langle x_{j+1},\ell,0\rangle_{\mathsf{poee}}\right]}(x)$

(a) If $x = x_{j+1}$, output $\langle x_{j+1}, \ell, 0\rangle_{\mathsf{poee}}$.
(b) If $x \le x_j$, then $b = 1$, else $b = 0$.
(c) Compute $r \leftarrow \mathsf{PRF}_{K^\ell_{x_{j+1}}}(x\|b)$.
(d) Compute $\langle x, \ell, b\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.PIEncode}(\mathsf{POEE.sk}_{x_{j+1},\ell}, x, \ell, b)$ using randomness r.
(e) Output $\langle x, \ell, b\rangle_{\mathsf{poee}}$.

Figure 11: Circuit $C^{2:\ell:j:1}_{\left[\ell,K^\ell_{x_{j+1}},\mathsf{POEE.sk}_{x_{j+1},\ell},\langle x_{j+1},\ell,0\rangle_{\mathsf{poee}}\right]}(x)$.

2. For $i \ne \ell$, $\langle C_i\rangle_{\mathsf{io}}$ is now computed as $\langle C_i\rangle_{\mathsf{io}} \leftarrow i\mathcal{O}\left(C^{2:\ell:j:1}_{\left[i,K^i,\mathsf{POEE.sk}_{x_{j+1},\ell}\right]}\right)$, where $C^{2:\ell:j:1}_{\left[i,K^i,\mathsf{POEE.sk}_{x_{j+1},\ell}\right]}$ is the same as $C^{2:\ell:j}_{[i,K^i,\mathsf{POEE.sk}]}$, except that it uses $\mathsf{POEE.sk}_{x_{j+1},\ell}$ to compute POEE input encodings instead of using $\mathsf{POEE.sk}$.

Hybrid $H_{2:\ell:j:2}$**:** Same as $H_{2:\ell:j:1}$, except that in the circuit $C^{2:\ell:j:1}_{\left[\ell,K^{\ell}_{x_{j+1}},\mathsf{POEE.sk}_{x_{j+1},\ell},\langle x_{j+1},\ell,0\rangle_{\mathsf{poee}}\right]}(x)$, the hardwired value $\langle x_{j+1},\ell,0\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk},x_{j+1},\ell,0)$ is now computed using true randomness (as opposed to PRF generated randomness).

Hybrid $H_{2:\ell:j:3}$**:** Same as $H_{2:\ell:j:2}$, except that we now replace the hardwired value $\langle x_{j+1},\ell,0\rangle_{\mathsf{poee}}$ with $\langle x_{j+1},\ell,1\rangle_{\mathsf{poee}}$, where $\langle x_{j+1},\ell,1\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk},x_{j+1},\ell,1)$ is computed using true randomness.

Hybrid $H_{2:j:4}$**:** Same as $H_{2:\ell:j:3}$, except that the hardwired value $\langle x_{j+1},\ell,1\rangle_{\mathsf{poee}} \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk},x_{j+1},\ell,1)$ is now computed using randomness $r \leftarrow \mathsf{PRF}_{K^{\ell}}(x_{j+1}\|1)$.

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

- For $0 \le j < T$,
  - $H_{2:\ell:j} \approx H_{2:\ell:j:1}$
  - $H_{2:\ell:j:1} \approx H_{2:\ell:j:2}$
  - $H_{2:\ell:j:2} \approx H_{2:\ell:j:3}$
  - $H_{2:\ell:j:3} \approx H_{2:\ell:j:4}$
  - $H_{2:\ell:j:4} \approx H_{2:\ell:j+1}$
- For $0 \le \ell < L$, $H_{2:\ell:T} \approx H_{2:\ell+1:0}$
- $H_2 \approx H_{2:0:0}$
- $H_{2:L:T} \approx H_3$

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

Indistinguishability of $H_{2:\ell:j}$ **and** $H_{2:\ell:j:1}$**.** We show that the two circuits $C^{2:\ell:j}_{[\ell,K^{\ell},\mathsf{POEE.sk}]}$ and $C^{2:\ell:j:1}_{\left[\ell,K^{\ell}_{x_{j+1}},\mathsf{POEE.sk}_{x_{j+1},\ell},\langle x_{j+1},\ell,0\rangle_{\mathsf{poee}}\right]}$ are functionally equivalent. Further, we will also argue that for all $i \ne \ell$, $C^{2:\ell:j:1}_{\left[i,K^{i},\mathsf{POEE.sk}_{x_{j+1},\ell}\right]}$ and $C^{2:\ell:j}_{[i,K^{i},\mathsf{POEE.sk}]}$ are functionally equivalent. The indistinguishability of $H_{2:j}$ and $H_{2:j:1}$ then follows from the security of the indistinguishability obfuscator $i\mathcal{O}$.

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

Next, it follows from the correctness of input puncturing property of POEE scheme that for all $i \neq \ell$, the behavior of circuits $C^{2:\ell:j:1}_{\left[i,K^i,\mathsf{POEE.sk}_{x_{j+1},\ell}\right]}$ and $C^{2:\ell:j}_{[i,K^i,\mathsf{POEE.sk}]}$ is identical on all inputs x.

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

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

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

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

Indistinguishability of $H_{2:\ell:T}$ and $H_{2:\ell+1:0}$. The proof of this is similar to the proof of indistinguishability of hybrids $H_{2:\ell:j}$ and $H_{2:\ell:j:1}$. We omit the details.

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

Circuit $C^{2:0:0}_{[i,K^i,\mathsf{POEE.sk}]}$ on input x computes $\mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, 0)$ using randomness $r \leftarrow \mathsf{PRF}_{K^i}(x\|0)$ while $C^{2}_{[i,K^i,\mathsf{POEE.sk}_1]}$ computes $\mathsf{POEE.PBEncode}(\mathsf{POEE.sk}_0, x, i)$ using randomness r. From the correctness of bit puncturing property of the POEE scheme, we have that $\mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, 0) = \mathsf{POEE.PBEncode}(\mathsf{POEE.sk}_0, x, i)$. Thus, $C^{2:0:0}_{[i,K^i,\mathsf{POEE.sk}]}$ and $C^{2}_{[i,K^i,\mathsf{POEE.sk}_0]}$ are functionally equivalent.

Indistinguishability of $H_{2:L:T}$ and H_3. Let $C^{2:L:T}_{[i,K^i,\mathsf{POEE.sk}]}$ denote the circuits used in hybrid $H_{2:L:T}$. We will show that for every $i \in \{0,\ldots,L\}$, the circuits $C^{2:L:T}_{[i,K^i,\mathsf{POEE.sk}]}$ and $C^{3}_{[i,K^i,\mathsf{POEE.sk}_1]}$ are functionally equivalent. The indistinguishability of $H_{2:L:T}$ and H_3 then follows from the security of the indistinguishability obfuscator $i\mathcal{O}$.

Circuit $C^{2:L:T}_{[i,K^i,\mathsf{POEE.sk}]}$ on input x computes $\mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, 1)$ using randomness $r \leftarrow \mathsf{PRF}_{K^i}(x\|1)$ while $C^{3}_{[i,K^i,\mathsf{POEE.sk}_1]}$ computes $\mathsf{POEE.PBEncode}(\mathsf{POEE.sk}_1, x, i)$ using randomness r. From the correctness of bit puncturing property of the POEE scheme, we have that $\mathsf{POEE.InpEncode}(\mathsf{POEE.sk}, x, i, 1) = \mathsf{POEE.PBEncode}(\mathsf{POEE.sk}_1, x, i)$. Thus, $C^{2:L:T}_{[i,K^i,\mathsf{POEE.sk}]}$ and $C^{3}_{[i,K^i,\mathsf{POEE.sk}_1]}$ are functionally equivalent.

Completing the proof of ϵ'-Indistinguishability of H_2 and H_3. Combining the above claims, we can first establish that $H_{2:\ell:j}$ and $H_{2:\ell:j+1}$ are ϵ''-indistinguishable, where $\epsilon'' = \mathsf{adv}_{\mathsf{PRF}}(\lambda) + \mathsf{adv}_{i\mathcal{O}}(\lambda) + \mathsf{adv}^{\mathsf{eb}}_{\mathsf{poee}}(\lambda)$, ignoring constant multiplicative factors. This is true for every j such that $0 \leq j < T$. Iterating over all values of j, we obtain that $H_{2:\ell:0}$ and $H_{2:\ell:T}$ are $T\cdot\epsilon''$-indistinguishable. Further, iterating over all values of ℓ s.t. $0 \leq \ell \leq L$, and using that $H_{2:\ell:T} \approx H_{2:\ell+1:0}$, we have that $H_{2:0:0}$ and $H_{2:L:T}$ are $T \cdot \epsilon''$-indistinguishable, ignoring multiplicative factor of $O(L)$. Finally, using that $H_2 \approx H_{2:0:0}$ and $H_{2:L:T} \approx H_3$, we have that H_2 and H_3 are ϵ'-indistinguishable, where $\epsilon' = T \cdot \big(\mathsf{adv}_{\mathsf{PRF}}(\lambda) + \mathsf{adv}_{i\mathcal{O}}(\lambda) + \mathsf{adv}^{\mathsf{eb}}_{\mathsf{poee}}(\lambda)\big)$, ignoring multiplicative factors of $O(L)$. This completes the proof.

8 Multi-Program Patchable Obfuscation

The concept of multi-program patchable obfuscation allows for obfuscating a priori unbounded number of programs in such a way that it is possible to issue a secure patch, whose size is independent of the number of programs obfuscated, that updates all the obfuscated programs at once. This is unlike the setting of single-program patchable obfuscation defined in Section 4, where the secure patch is issued for a specific program.

Syntax. A multi-program patchable obfuscation 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 $\mathsf{mp.pO} = (\mathsf{Setup}, \mathsf{Obfuscate}, \mathsf{GenPatch}, \mathsf{ApplyPatch}, \mathsf{Evaluate})$ 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 $\mathsf{Obf.SK}$.
- **Obfuscate**, $\mathsf{Obfuscate}(\mathsf{Obf.SK}, M)$: It takes as input the secret key $\mathsf{Obf.SK}$ and a TM $M \in \mathcal{M}$. It outputs an obfuscated TM $\langle M \rangle$.
- **Secure Patch Generation**, $\mathsf{GenPatch}(\mathsf{Obf.SK}, P)$: It takes as input the secret key $\mathsf{Obf.SK}$ and a description of a patch $P \in \mathcal{P}$. It outputs a secure patch $\langle P \rangle$.
- **Applying Patch**, $\mathsf{ApplyPatch}\Big(\langle M \rangle, \langle P \rangle\Big)$: It takes as input an obfuscated TM $\langle M \rangle$ and a secure patch $\langle P \rangle$. It outputs an updated obfuscation $\langle M_{new} \rangle$.
- **Evaluation**, $\mathsf{Evaluate}\Big(\langle M \rangle, x\Big)$: It takes as input an obfuscated TM $\langle M \rangle$ and an input x. It outputs a value y.

Correctness. At a high level, the correctness property states that executing Update on a TM M and a patch P is equivalent to executing $\mathsf{ApplyPatch}$ on the obfuscation of M and a secure patch of P. In fact we require that this hold for *multiple TMs* and multiple 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) $\mathsf{Obf.SK} \leftarrow \mathsf{Setup}(1^\lambda)$, (b) $\langle M_0^j \rangle \leftarrow \mathsf{Obfuscate}(\mathsf{Obf.SK}, M_0^j)$, (c) $\langle P_i \rangle \leftarrow \mathsf{GenPatch}(\mathsf{Obf.SK}, P_i)$, (d) $\langle M_i^j \rangle \leftarrow \mathsf{ApplyPatch}\Big(\langle M_{i-1}^j \rangle, \langle P_i \rangle\Big)$.
- **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 $\mathsf{Evaluate}\Big(\langle M_i^j \rangle, x\Big) = M_i^j(x)$.

8.1 Indistinguishability-Based Security

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. There are two main ways of formalizing an IND-style definition for multi-program patchable obfuscation:

- **Adaptive security**: In this notion, the adversary has to declare all the TM queries in the beginning of the game itself. The patch queries, however, can done in an adaptive manner.
- **Selective security**: In this notion, the adversary has to declare all the TM queries as well as the patch queries in the beginning of the game itself.

We formally define both the types of security notions below.

Adaptive security. We describe the experiment below.

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

1. $\mathcal{A}$ submits a sequence of TM pairs $\Big((M_0^1, M_1^1), \ldots, (M_0^Q, M_1^Q)\Big)$.
2. Challenger executes the setup algorithm to obtain $\mathsf{Obf.SK} \leftarrow \mathsf{Setup}(1^\lambda)$.
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 $\langle P_b^i \rangle \leftarrow \mathsf{GenPatch}(\mathsf{Obf.SK}, P_b^i)$. 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 $M_{i,0}^i \equiv M_{i,1}^i$, where $M_{i,0}^j \leftarrow \mathsf{Update}(M_{i-1,0}^j, P_0^i)$ and $M_{i,1}^i \leftarrow \mathsf{Update}(M_{i-1,1}^j, P_1^i)$. If check fails then the challenger aborts the experiment.
5. $\mathcal{A}$ outputs the bit b'.

Definition 14 (Adaptive security)**.** *A multi-program patchable obfuscation scheme* $\mathsf{mp.pO}$ *is said to be adaptively secure if for any PPT adversary $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *s.t.*

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

Selective security. We first describe the experiment.

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

1. $\mathcal{A}$ submits a sequence of TM pairs $\Big((M_0^1, M_1^1), \ldots, (M_0^Q, M_1^Q)\Big)$. It also submits a sequence of patch pairs $\Big((P_0^1, P_1^1), \ldots, (P_0^L, P_1^L)\Big)$.

2. Challenger executes the setup algorithm to obtain $\mathsf{Obf.SK} \leftarrow \mathsf{Setup}(1^\lambda)$. For every $j \in \{1,\ldots,Q\}$, it generates $\langle M_b^j\rangle \leftarrow \mathsf{Obfuscate}(\mathsf{Obf.SK}, M_b^j)$. For every $i \in \{1,\ldots,L\}$, it generates $\langle P_b^i\rangle \leftarrow \mathsf{GenPatch}(\mathsf{Obf.SK}, P_b^i)$.

3. Challenger sends $\left(\{\langle M_b^j\rangle\}_{j\in[Q]}, \{\langle P_b^i\rangle\}_{i\in[L]}\right)$ to $\mathcal{A}$.

4. For every $i \in \{1,\ldots,L\}$, every $j \in \{1,\ldots,Q\}$ the challenger checks if $M_{i,0}^i \equiv M_{i,1}^i$, 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 15 (Selective security)**.** *A multi-program patchable obfuscation scheme* $\mathsf{mp.pO}$ *is said to be selectively secure if for any PPT adversary* $\mathcal{A}$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *s.t.*

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

Overview of Construction of Multi-Program PO. The construction of multi-program patchable obfuscation is divided into three main steps:

1. **Step I: Stateless PABE.** We first consider a notion, termed as *stateless patchable ABE.* This is a special class of PABE schemes (Section 5). The important aspect about this notion is that no state is maintained during the patching process and in particular, the patch generation algorithm only takes as input the secret key and the patch. This is the unlike the PABE scheme, where the state is also part of the input.

   To construct this primitive, we build upon the construction of PABE in Section 5.1 and along the way using adaptive garbled TMs with persistent memory (Definition 3). Also, our construction of stateless PABE achieves adaptive security.

2. **Step II: Stateless PABE to Multi-Program POEE.** We generalize the notion of patchable OEE to multi-program OEE. In a multi-program OEE, the secret key can be used to produce TM encodings with respect to multiple machines. Furthermore, the patches issued should be applicable on all the machines. As an added feature, we also achieve a patch generation mechanism that does not maintain state.

   We build upon the construction of POEE (Section 6.1), using additional layers, to achieve our goal of multi-program POEE.

3. **Step III: Multi-Program OEE to iO.** The transformation from multi-program OEE to iO is identical to the transformation from single-program OEE to iO (Section 7). The main challenge lies in the security analysis.

9 Stateless PABE

9.1 Syntax

A stateless PABE is a patchable ABE scheme, where no state is maintained in between the executions of the patch generation algorithms. This implies that the issued secure patch is independent of the history of updates made so far. We formally define this notion below.

Definition 16. *A stateless patchable attribute based encryption (SPABE) scheme is a PABE scheme, denoted by* $\mathsf{SPABE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{GenPatch}, \mathsf{ApplyPatch}, \mathsf{Dec})$*, associated with a TM class* $\mathcal{M}$ *equipped with* $(\mathsf{Update}, \mathcal{P})$ *and satisfying the following* stateless *property.*

For every machine $M_0 \in \mathcal{M}$*, sequence of patches* $P_1, \ldots, P_L \in \mathcal{P}$*, consider the following process: (i)* $(\mathsf{SPABE.PP}, \mathsf{SPABE.SK}) \leftarrow \mathsf{SPABE.Setup}(1^{\lambda})$*, (ii)* $(\mathsf{SPABE}.sk_{M_0}, \mathsf{st}_0) \leftarrow \mathsf{SPABE.KeyGen}(\mathsf{SPABE.SK}, M_0)$ *and, (iii) for every* $i \in [L]$*,* $(\widetilde{P}_i, \mathsf{st}_i) \leftarrow \mathsf{SPABE.GenPatch}(\mathsf{SPABE.SK}, P_i, \mathsf{st}_{i-1})$*.*

We say that SPABE *satisfies the stateless property if for every* $i \in \{0, \ldots, L\}$*,* $\mathsf{st}_i = \perp$*.*

In the construction of SPABE scheme, described next, we omit the argument st in the description of the algorithms.

9.2 Construction

Overview of Construction. Our starting point is the (stateful) patchable obfuscation scheme developed earlier. The main challenge is the following contradictory requirement: (i) for the authority to "get rid of its state" and, (ii) authority to produce a patch as a function of the current state. A naive approach to solve this problem is for the authority to delegate the storage of state to the user who holds the attribute key. So whenever the authority wants to issue a secure patch, it delegates the computation of the secure patch to the user. But note that this computation needs to be hidden. A natural idea here is to use randomized encodings. While randomized encodings (RE) is a starting step to what we want, it has scalability issues: RE is a one-time primitive and so this process cannot be repeated for multiple patches. One approach (that does not work) is to issue fresh randomized encodings during every execution but this would mean that the authority computes the encoding of the state afresh every time; thus defeating the purpose of delegating this computation.

To deal with this issue, we use a tool called garbling with *persistent memory.* The concept of persistent memory is that once the evaluation of the encodings is completed, the resulting state will still be in an encoded form. Thus enabling us to re-evaluate on the existing state. Previous literature on garbling with persistent memory mainly dealt with RAM model of computation. However, we note that it would suffice for us to just consider garbling for Turing machines in the persistent memory setting.

We construct a stateless PABE scheme denoted by $\mathsf{SPABE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{GenPatch}, \mathsf{ApplyPatch}, \mathsf{Dec})$. We use the following primitives in our construction.

1. Patchable ABE scheme described in Section 5. We denote this scheme by $\mathsf{pabe} = (\mathsf{pabe.Setup}, \mathsf{pabe.KeyGen}, \mathsf{pabe.Enc}, \mathsf{pabe.GenPatch}, \mathsf{pabe.ApplyPatch}, \mathsf{pabe.Dec})$. As a consequence, we import the primitives used in pabe. We recall the tools below.
    - Positional accumulator scheme, Acc = ($\mathsf{SetupAcc}$, $\mathsf{EnforceRead}$, $\mathsf{EnforceWrite}$, $\mathsf{PrepRead}$, $\mathsf{PrepWrite}$, $\mathsf{VerifyRead}$, $\mathsf{WriteStore}$, Update). It is associated with the message space Σ_{tape} with accumulated value of size ℓ_{Acc} bits.
    - Iterator scheme, Itr =($\mathsf{SetupItr}$, $\mathsf{ItrEnforce}$, $\mathsf{Iterate}$). It is associated with the message space $\{0,1\}^{2\lambda+\ell_{\mathsf{Acc}}}$ with iterated value of size ℓ_{Itr} bits.
    - Splittable signatures scheme, $\mathsf{SplScheme} = (\mathsf{SetupSpl}, \mathsf{SignSpl}, \mathsf{VerSpl}, \mathsf{SplitSpl}, \mathsf{SignSplAbo})$. It is associated with the message space $\{0,1\}^{\ell_{\mathsf{Itr}}+\ell_{\mathsf{Acc}}+2\lambda}$. In addition to the above tools, we also use a puncturable PRF family denoted by F.

2. Garbled TMs with persistent memory, denoted by $\mathsf{GTM} = (\mathsf{Gen}, \mathsf{GarbDB}, \mathsf{GarbTM}, \mathsf{GarbEval})$.

We present the algorithms of SPABE below. The class of Turing machines asssociated with SPABE is $\mathcal{M}$ and we denote the update algorithm by Update.

$\underline{\mathsf{SPABE.Setup}(1^\lambda)}$: On input a security parameter λ in unary, it executes the setup of pabe to obtain the key pair $(\mathsf{pabe.PP}, \mathsf{pabe.SK}) \leftarrow \mathsf{pabe.Setup}(1^\lambda)$. It also executes the garbled TM setup algorithm to obtain $\mathbf{k} \leftarrow \mathsf{Gen}(1^\lambda)$. Output the public key-secret key pair $\big(\mathsf{SPABE.PP} = \mathsf{pabe.PP}, \mathsf{SPABE.SK} = (\mathsf{pabe.SK}, \mathbf{k})\big)$.

$\underline{\mathsf{SPABE.KeyGen}(\mathsf{SPABE.SK}, M)}$: On input a secret key $\mathsf{SPABE.SK} = (\mathsf{pabe.SK}, \mathbf{k})$ and a TM $M \in \mathcal{M}$, it executes $\mathsf{pabe}.sk_M \leftarrow \mathsf{pabe.KeyGen}(\mathsf{pabe.SK}, M)$. It then initializes the input tape with $DB_M = M$ and then garbles the input tape, $\widehat{DB_M} \leftarrow \mathsf{gtm.GarbTM}(\mathbf{k}, DB_M)$. It outputs the SPABE key $\mathsf{SPABE}.sk_M = (\mathsf{pabe}.sk_M, \widehat{DB_M})$.

$\underline{\mathsf{SPABE.GenPatch}(\mathsf{SPABE.SK}, P)}$: On input the secret key $\mathsf{SPABE.SK} = (\mathsf{pabe.SK}, \mathbf{k})$, a description of a patch $P \in \mathcal{P}$, it computes fresh parameters of splittable signatures, accumulators and iterators. In more detail, it executes the setup of splittable signatures scheme, $(\mathsf{SK_{tm}}, \mathsf{VK_{tm}}) \leftarrow \mathsf{SetupSpl}(1^\lambda)$. It executes the setup of the accumulator scheme to obtain the values, $(\mathsf{PP_{Acc}}, \widetilde{w}_0, \widetilde{store}_0) \leftarrow \mathsf{SetupAcc}(1^\lambda)$. It then executes the setup of the iterator scheme to obtain the public parameters, $(\mathsf{PP_{Itr}}, v_0) \leftarrow \mathsf{SetupItr}(1^\lambda)$. It then sets $\mathsf{pabe.PP}' = (\mathsf{VK_{tm}}, \mathsf{PP_{Acc}}, \widetilde{w}_0, \widetilde{store}_0, \mathsf{PP_{Itr}}, v_0)$.

Pick a random string $r \leftarrow \{0,1\}^{\mathrm{poly}(\lambda)}$. It then computes a garbling of the program Prg by executing $\widehat{\mathsf{Prg}} \leftarrow \mathsf{gtm.GarbTM}(\mathbf{k}, \mathsf{Prg}[P, \mathsf{pabe.PP}', \mathsf{SK_{tm}}, r])$ where $\mathsf{Prg}[P, \mathsf{pabe.PP}', \mathsf{SK_{tm}}, r]$ is described in Figure 12.

Finally it outputs the patch $(P, \widehat{\mathsf{Prg}}, \mathsf{pabe.PP}')$ and the new public key $\mathsf{SPABE.PP}' = \mathsf{pabe.PP}'$.

Prg

Input: Database DB_M
Hardwired Values: $(P, \mathsf{pabe.PP}', \mathsf{SK_{tm}}, r)$

1. Parse DB_M as machine M.
2. Generate the TM M' by updating M using P,i.e., $M' \leftarrow \mathsf{Update}(M, P)$. Update the input tape to be M'.
3. Set $\mathsf{pabe.SK}' = (\mathsf{pabe.PP}', \mathsf{SK_{tm}})$. Execute $\mathsf{pabe.KeyGen}(\mathsf{pabe.SK}', M'; r)$ to obtain $\mathsf{pabe}.sk_{M'}$. Recall that $\mathsf{pabe}.sk_{M'}$ is of the form $(M', \sigma_{M'})$.
4. Output $\sigma_{M'}$.

Figure 12: Description of Prg.

$\underline{\mathsf{SPABE.ApplyPatch}(\mathsf{SPABE}.sk_M, \widetilde{P})}$: On input an SPABE key $\mathsf{SPABE}.sk_M = \big(\mathsf{pabe}.sk_M = (M, \sigma_M), \widehat{DB_M}\big)$ and a secure patch $\widetilde{P} = (P, \widehat{\mathsf{Prg}}, \mathsf{pabe.PP}')$, it first runs the update algorithm $M' \leftarrow \mathsf{Update}(M, P)$. Then it executes the garbled TM evaluation algorithm. That is, it generates $(\sigma_{M'}, \widehat{DB_{M'}}) \leftarrow \mathsf{gtm.GarbEval}(\widehat{\mathsf{Prg}}, \widehat{DB_M})$. Note that $\sigma_{M'}$ is the output of the garbled TM evaluation. And, $\widehat{DB_{M'}}$ is the encoding of the input tape initialized with the value M'.

Set $\mathsf{pabe}.sk_{M'} = (M', \sigma_{M'})$. Output the updated attribute key $\mathsf{SPABE}.sk_{M'} = (\mathsf{pabe}.sk_{M'}, \widehat{DB_{M'}})$.

$\underline{\mathsf{SPABE.Enc}(\mathsf{SPABE.PP}, x, \mathsf{msg})}$: On input the (possibly updated) public parameters $\mathsf{SPABE.PP} = \mathsf{pabe.PP}$, attribute $x \in \{0,1\}^*$ and message $\mathsf{msg} \in \mathsf{MSG}$, it generates $\mathsf{pabe.CT}_{(x,\mathsf{msg})} \leftarrow \mathsf{pabe.Enc}(\mathsf{pabe.PP}, x, \mathsf{msg})$. It outputs a ciphertext $\mathsf{SPABE.CT}_{(x,\mathsf{msg})} = \mathsf{pabe.CT}_{(x,\mathsf{msg})}$.

$\underline{\mathsf{SPABE.Dec}(\mathsf{SPABE}.sk_M, \mathsf{SPABE.CT}_{(x,\mathsf{msg})})}$: On input an ABE key $\mathsf{SPABE}.sk_M = (\mathsf{pabe}.sk_M, \widehat{M})$ and ciphertext $\mathsf{SPABE.CT}_{(x,\mathsf{msg})} = \mathsf{pabe.CT}_{(x,\mathsf{msg})}$, it executes the decryption algorithm $\mathsf{out} \leftarrow \mathsf{pabe.Dec}(\mathsf{pabe}.sk_M, \mathsf{pabe.CT}_{(x,\mathsf{msg})})$. Output out.

We now argue the correctness and the security properties.

Correctness. Consider a TM $M \in \mathcal{M}$. Consider a sequence of patches $P_1, \ldots, P_L \in \mathcal{P}$. Consider the following process:

- $(\mathsf{pabe.PP}, \mathsf{pabe.SK}) \leftarrow \mathsf{pabe.Setup}(1^\lambda)$
- $(\mathsf{pabe}.sk_M, \mathsf{st}_0) \leftarrow \mathsf{pabe.KeyGen}(\mathsf{pabe.SK}, M)$
- for $i \in [L]$, $\widetilde{P_i} \leftarrow \mathsf{pabe.GenPatch}(\mathsf{pabe.SK}, P_i)$

Now, let $(\mathsf{pabe}.sk'_M, \widehat{DB_M})$ be the initial attribute key of M in the stateless PABE scheme SPABE. Let the evaluation of the garbled encodings in the i^{th} phase corresponding to patch P_i result in the output $\widetilde{P'_i}$. By inspection it follows that the following two distributions are identical:

1. $\left\{(\mathsf{pabe}.sk_M, \widetilde{P_1}, \ldots, \widetilde{P_L})\right\}$
2. $\left\{(\mathsf{pabe}.sk'_M, \widetilde{P'_1}, \ldots, \widetilde{P'_L})\right\}$

Combining the above observation with the correctness of pabe it follows that the scheme SPABE is correct.

Theorem 15. *Assuming the security of garbled TMs* GTM *and patchable ABE scheme (Section 5), the scheme* SPABE *is secure.*

Proof. We reduce the security of the stateless PABE scheme to the security of our PABE scheme. We take the help of GTM to achieve this. First, we simulate the TM encoding that is part of all the patches and this can be done using the simulator of GTM. And once we do this, we now can reduce the security of our construction to the security of pabe.

Let $\mathcal{A}$ be a PPT adversary in the security experiment of stateless PABE.

Hyb_1: This corresponds to the real experiment. To recall: the adversary first specifies the machine M, attribute x, message pair $(\mathsf{msg}_0, \mathsf{msg}_1)$ and index $\mathbf{i}$ as part of the first message. The challenger is supposed to provide him the ABE key of M. The challenge ciphertext is not generated yet. Then there is the patch query phase when the adversary submits a patch query and in returns a secure patch along with new public parameters. Only in the $\mathbf{i}^{th}$ phase, the challenger also sends a challenge ciphertext, that is generated using the $\mathbf{i}^{th}$ parameters and msg_b, where b is picked at random. In the end, $\mathcal{A}$ guesses b.

Hyb_2: Recall that the secure patches in the previous hybrid are essentially garbled TM encodings. Also, the ABE key is a garbled input encoding. In this hybrid, we simulate the garbled encodings. To be precise, we design our simulator Sim as follows. Our simulator internally executes the simulator $\widetilde{\mathsf{Sim}} = (\widetilde{\mathsf{Sim}_1}, \widetilde{\mathsf{Sim}_2})$ of GTM.

1. Sim picks a bit b at random.
2. $\Big(M, x, (\mathsf{msg}_0, \mathsf{msg}_1), \mathbf{i}\Big) \leftarrow \mathcal{A}(1^\lambda)$
3. Sim generates SPABE parameters $(\mathsf{SPABE.PP} = \mathsf{pabe.PP}, \mathsf{SPABE.SK} = (\mathsf{pabe.SK}, \mathbf{k})) \leftarrow \mathsf{SPABE.Setup}(1^\lambda)$. Note that the GTM key $\mathbf{k}$ will never be used. It first generates a PABE key of M using $\mathsf{pabe.SK}$; $\mathsf{pabe}.sk_M$. It then generates $(\mathsf{st}_{\mathsf{GTM}}, \widehat{DB}_{\mathsf{ideal}}) \leftarrow \mathsf{Sim}_1(1^\lambda, 1^{|M|})$. It then sends across $(\widehat{DB}_{\mathsf{ideal}}, \mathsf{pabe}.sk_M)$ to $\mathcal{A}$.
4. The following is repeated polynomially many times:
   - $\mathcal{A}$ sends patch P_i to Sim.
   - Sim samples fresh parameters $(\mathsf{pabe.PP}_i, \mathsf{SK}^i_{\mathsf{tm}})$. It also samples fresh randomness r_i. Set $\mathsf{Prg}_i = \mathsf{Prg}[P_i, \mathsf{pabe.PP}_i, \mathsf{SK}^i_{\mathsf{tm}}, r_i]$. Compute $y_i \leftarrow \mathsf{Prg}_i(M_{i-1})$, where M_{i-1} is the updated machine. Execute $(\widehat{P_i^{\mathsf{ideal}}}, \mathsf{st}_{\mathsf{GTM}}) \leftarrow \mathsf{Sim}_2(\mathsf{st}_{\mathsf{GTM}}, y_i, 1^{|\mathsf{Prg}_i|})$.
   - Sim sends across $\widehat{P_i^{\mathsf{ideal}}}$ to $\mathcal{A}$.
5. $b' \leftarrow \mathcal{A}(\widehat{P_\ell^{\mathsf{ideal}}})$.

The output of this hybrid is b'.

The indistinguishability of Hyb_1 and Hyb_2 follows from the adaptive security of GTM.

$\mathsf{Hyb}_{3,c}$ for $c \in \{0,1\}$: Instead of the Sim generating $\mathsf{Prg}[P_i, \mathsf{pabe.PP}_i, \mathsf{SK}^i_{\mathsf{tm}}, r_i]$, it gets the output signature σ_i from the external challenger of the pabe scheme. The external challenger is also responsible for generating the ABE public parameters and the challenge ciphertext.

In more detail, upon receiving the first message of $\mathcal{A}$, Sim forwards this message to pabe. In return it receives public parameters and the attribute key which is forwarded to $\mathcal{A}$. During the patch query phase, the patch P_i from $\mathcal{A}$ is forwarded to pabe's challenger. In response, Sim receives the signature along with fresh parameters. That is, in the i^{th} phase, it receives (P_i, σ_i). Sim, as in the previous hybrid, using this signature σ_i simulates the program encoding and sends this encoding along with the patch and the public parameters to $\mathcal{A}$. Finally, the challenge ciphertext is received by Sim from the external challenger. If the challenger uses the bit b to choose which message to encrypt. If $c = 0$, it picks a random bit b and chooses the b^{th} bit to encrypt. Otherwise, it uses the 0^{th} message to encrypt.

The output of this hybrid is the output of $\mathcal{A}$.

The hybrids Hyb_2 and $\mathsf{Hyb}_{3,0}$ are identically distributed. The indistinguishability of $\mathsf{Hyb}_{3,0}$ and $\mathsf{Hyb}_{3,1}$ follows from the security of the underlying pabe scheme.

Note that at $\mathsf{Hyb}_{3,1}$, the secret challenge bit is information-theoretically hidden from $\mathcal{A}$. By computational indistinguishability of consecutive hybrids, this means that $\mathcal{A}$ can guess the challenge bit with probability $1/2 + \mathsf{negl}(\lambda)$ in Hyb_1. This completes the proof.

□

Instantiating the garbled TMs scheme (Section 2.5) and the patchable ABE scheme (Section 5), we get the following corollary.

Corollary 3. *Let $\lambda \in \mathbb{N}$ be a sufficiently large security parameter. Assuming the existence of $\frac{\epsilon}{2^\lambda}$-indistinguishability obfuscation and $\frac{\epsilon'}{2^\lambda}$-secure decisional Diffie-Hellman assumption, there exists a δ-secure stateless PABE scheme, where $\epsilon, \epsilon', \delta \leq \frac{1}{p(\lambda)}$, for some polynomial p.*

10 Multi-program POEE

Syntax. We describe the syntax of a multi-program patchable oblivious evaluation encoding (mpPOEE) scheme mpOEE. It is defined for a class of Turing machines $\mathcal{M}$ and a family of patches $\mathcal{P}$. To define this primitive, we make use of the abstraction of basic POEE (BPOEE) scheme described in Section 6 (Definition 10). A multi-program patchable obfuscation scheme is also a BPOEE scheme but it has additional algorithms, similar to the POEE scheme (for the single program case) as described next.

We define four helper algorithms.

- $\mathsf{mpOEE.puncInp}(\mathsf{sk}, x, i, \mathsf{mid})$: It takes as input a secret key sk, input $x \in \{0,1\}^*$, index i and machine id mid. It outputs a punctured key $\mathsf{sk}_{x,i,\mathsf{mid}}$.
- $\mathsf{mpOEE.PIEncode}(\mathsf{sk}_{x,i,\mathsf{mid}}, x', i', \mathsf{mid}', b)$: It takes as input a punctured secret key $\mathsf{sk}_{x,i,\mathsf{mid}}$, an input x', index i', machine id mid' and a bit b s.t. $(x, i, \mathsf{mid}) \neq (x', i', \mathsf{mid}')$. It outputs an input encoding $\langle x', i', \mathsf{mid}', b \rangle$.
- $\mathsf{mpOEE.puncBit}(\mathsf{sk}, \mathsf{mid}, b)$: It takes as input a secret key sk, machine id mid and an input bit b. It outputs a key $\mathsf{sk}_{\mathsf{mid},b}$.
- $\mathsf{mpOEE.PBEncode}(\mathsf{sk}_b, x, i, \mathsf{mid})$: It takes as input a key $\mathsf{sk}_{\mathsf{mid},b}$, an input x, an index i and a machine id mid. It outputs an input encoding $\langle x, i, \mathsf{mid}, b \rangle$.

We associate the above scheme with correctness and security properties as described below.

Correctness. We associate a mpPOEE scheme with the following properties:

1. *Correctness of Input Puncturing:* For all $Q \geq 1, L \geq 0$, every $M_0^1, M_1^1, \ldots, M_0^Q, M_1^Q \in \mathcal{M}$, patch sequence $P_0^1, P_1^1, \ldots, P_0^L, P_1^L \in \mathcal{P}$, every $x^*, x' \in \{0,1\}^*$, $i^*, i' \leq L$ and $j^*, j' \leq Q$ s.t. $(x^*, i^*, j^*) \neq (x', i', j')$ and $b \in \{0,1\}$,

$$\mathsf{mpOEE.Decode}\Big(\langle M_{i',0}^{j'}, M_{i',1}^{j'} \rangle, \langle x', i', j', b \rangle\Big) = M_{b,i'}^{j'}(x'),$$

   where for every $0 \leq i \leq i'$:

   - $\mathsf{sk} \leftarrow \mathsf{mpOEE.Setup}(1^\lambda)$,

- $\left(\langle M_0^{j'}, M_1^{j'}\rangle\right) \leftarrow \mathsf{mpOEE.TMEncode}(\mathsf{sk}, M_0^{j'}, M_1^{j'})$,
- $\left(\langle P_0^i, P_1^i\rangle\right) \leftarrow \mathsf{mpOEE.GenPatch}(\mathsf{sk}, P_0^i, P_1^i)$,
- $\langle M_{0,i}^{j'}, M_{1,i}^{j'}\rangle \leftarrow \mathsf{mpOEE.ApplyPatch}\Big(\langle M_{0,i-1}^{j'}, M_{1,i-1}^{j'}\rangle, \langle P_0^i, P_1^i\rangle\Big)$,
- $\langle x', i', j', b\rangle \leftarrow \mathsf{mpOEE.PIEncode}\Big(\mathsf{sk}_{x^*,i^*,j^*} \leftarrow \mathsf{puncInp}(\mathsf{sk}, x^*, i^*, j^*), x', i', j', b\Big)$,
- $M_{b,i}^{j'} \leftarrow \mathsf{Update}(M_{b,i-1}^{j'}, P_0^i)$.

2. *Correctness of Bit Puncturing:* For all $Q \geq 1, L \geq 0$, every $M_0^1, M_1^1, \ldots, M_0^Q, M_1^Q \in \mathcal{M}$, patch sequence $P_0^1, P_1^1, \ldots, P_0^L, P_1^L \in \mathcal{P}$, $x \in \{0,1\}^*$, machine id $j \in [Q]$ and $b \in \{0,1\}$,

$$\mathsf{mpOEE.Decode}\Big(\langle M_{0,L}^j, M_{1,L}^j\rangle, \langle x, L, j, b\rangle\Big) = M_{b,L}^j(x)$$

where for $i \in \{1, \ldots, L\}$:

- $\mathsf{sk} \leftarrow \mathsf{mpOEE.Setup}(1^\lambda)$,
- $\left(\langle P_0^i, P_1^i\rangle\right) \leftarrow \mathsf{mpOEE.GenPatch}(\mathsf{sk}, P_0^i, P_1^i)$,
- $\langle M_{0,i}^j, M_{1,i}^j\rangle \leftarrow \mathsf{POEE.ApplyPatch}\Big(\langle M_{0,i-1}^j, M_{1,i-1}^j\rangle, \langle P_0^i, P_1^i\rangle\Big)$,
- $\langle x, L, j, b\rangle \leftarrow \mathsf{mpOEE.PBEncode}(\mathsf{mpOEE.puncBit}\,(\mathsf{sk}, b)\,, x, L, j)$,
- $M_{b,i}^j \leftarrow \mathsf{Update}(M_{b,i-1}^j, P_b^i)$.

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

- Setup: $\mathcal{A}$ chooses a sequence of Turing machine pairs $(M_0^1, M_1^1, \ldots, M_0^Q, M_1^Q) \in \mathcal{M}$ such that $|M_0^j| = |M_1^j|$, an input x, an index $\mathbf{i} \geq 0$ and a machine id mid. $\mathcal{A}$ sends the tuple $\big(\{M_0^j, M_1^j\}_{j\in[Q]}, x, \mathbf{i}, \mathsf{mid}\big)$ to the challenger.
  The challenger chooses a bit $b \in \{0,1\}$ and computes the following: (a) $\mathsf{sk} = K \leftarrow \mathsf{mpOEE.Setup}(1^\lambda)$, (b) for $j \in [Q]$, $\left(\langle M_0^j, M_1^j\rangle\right) \leftarrow \mathsf{mpOEE.TMEncode}(\mathsf{sk}, M_0^j, M_1^j)$, (c) $\mathsf{sk}_{x,i,\mathsf{mid}} \leftarrow \mathsf{mpOEE.puncInp}(\mathsf{sk}, x, i, \mathsf{mid})$(d) $\langle x, \mathbf{i}, \mathsf{mid}, b\rangle \leftarrow \mathsf{mpOEE.InpEncode}(\mathsf{sk}, x, \mathbf{i}, \mathsf{mid}, b)$. It sends $\big(\{\langle M_0^j, M_1^j\rangle\}_{j\in[Q]}, \mathsf{sk}_{x,i,\mathsf{mid}}, \langle x, i, \mathsf{mid}, b\rangle\big)$ to $\mathcal{A}$:
- Patch Query phase: The following is repeated polynomially many times:
  - $\mathcal{A}$ chooses two patches $P_0^i, P_1^i \in \mathcal{P}$ and sends them to the challenger.
  - The challenger computes $\left(\langle P_0^i, P_1^i\rangle\right) \leftarrow \mathsf{mpOEE.GenPatch}(\mathsf{sk}, P_0^i, P_1^i)$. It sends $\langle P_0^i, P_1^i\rangle$ to $\mathcal{A}$.
- Guess: $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$.

$\mathcal{A}$ is required to choose his queries s.t. $M_{0,\mathbf{i}}^{\mathsf{mid}}(x) = M_{1,\mathbf{i}}^{\mathsf{mid}}(x)$, where $M_{c,\mathbf{i}}^{\mathsf{mid}} \leftarrow \mathsf{Update}(M_{c,\mathbf{i}-1}^{\mathsf{mid}}, P_c^{\mathbf{i}})$ for $c \in \{0,1\}$. The advantage of $\mathcal{A}$ in this game is defined as $\mathsf{adv}_\mathcal{A} = \Pr[b' = b] - \frac{1}{2}$.

Definition 17 (Indistinguishability of encoding bit)**.** *A mpPOEE scheme satisfies indistinguishability of encoding bit if there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every PPT adversary* $\mathcal{A}$ *in the above security game,* $\mathsf{adv}_\mathcal{A} = \mathsf{negl}(\lambda)$.

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

- Setup: $\mathcal{A}$ chooses a sequence of Turing machine pairs $(M_0^1, M_1^1, \ldots, M_0^Q, M_1^Q) \in \mathcal{M}$ such that $|M_0^j| = |M_1^j|$, $j \in [Q]$ and a bit $c \in \{0,1\}$. $\mathcal{A}$ sends the tuple $\left(\{M_0^j, M_1^j\}_{j\in[Q]}, \mathsf{mid}, c\right)$ to the challenger.

  The challenger chooses a bit $b \in \{0,1\}$ and computes the following: (a) $\mathsf{sk} \leftarrow \mathsf{mpOEE.Setup}(1^\lambda)$, (b) $\left(\langle \mathsf{TM}_1^j, \mathsf{TM}_2^j \rangle\right) \leftarrow \mathsf{mpOEE.TMEncode}(\mathsf{sk}, \mathsf{TM}_1^j, \mathsf{TM}_2^j)$, where $\mathsf{TM}_1^j = M_0^{\mathsf{mid}}, \mathsf{TM}_2^j = M_{1\oplus b}^{\mathsf{mid}}$ if $c = 0$ and $\mathsf{TM}_1^j = M_{0\oplus b}^j, \mathsf{TM}_2^j = M_1^j$ otherwise, and (c) $\mathsf{sk}_b \leftarrow \mathsf{mpOEE.puncBit}(\mathsf{sk}, \mathsf{mid}, b)$. Finally, it sends the following tuple to $\mathcal{A}$:

$$\left(\{\langle \mathsf{TM}_1^j, \mathsf{TM}_2^j \rangle\}_{j\in[Q]}, \mathsf{sk}_b\right).$$

- Patch Query phase: The following is repeated polynomially many times:
  - $\mathcal{A}$ chooses two patches $P_0^i, P_1^i \in \mathcal{P}$ and sends them to the challenger.
  - The challenger computes $\left(\langle \mathsf{PT}_1^i, \mathsf{PT}_2^i \rangle\right) \leftarrow \mathsf{mpOEE.GenPatch}(\mathsf{sk}, \mathsf{PT}_1^i, \mathsf{PT}_2^i)$, where $\mathsf{PT}_1^i = P_0^i, \mathsf{PT}_2^i = P_{1\oplus b}^i$ if $c = 0$ and $\mathsf{PT}_1^i = P_{0\oplus b}^i, \mathsf{PT}_2^i = M_1^i$ otherwise. It sends $\langle \mathsf{PT}_1^i, \mathsf{PT}_2^i \rangle$ to $\mathcal{A}$.

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

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

Definition 18 (Indistinguishability of machine encoding)**.** *An mpPOEE scheme satisfies indistinguishability of machine encoding if there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every PPT adversary* $\mathcal{A}$ *in the above security game,* $\mathsf{adv}_\mathcal{A} = \mathsf{negl}(\lambda)$*.*

We now formally define a mpPOEE scheme below.

Definition 19. *An mpPOEE scheme is a basic POEE scheme and in addition is equipped with the algorithms* $(\mathsf{puncInp}, \mathsf{PIEncode}, \mathsf{puncBit}, \mathsf{PBEncode})$*. It satisfies the correctness of input puncturing and bit puncturing properties. It also satisfies the indistinguishability of encoding bit (Definition 17) and indistinguishability of machine encoding (Definition 18).*

10.1 Construction of Multi-program POEE

We present a construction of multi-program POEE below. We build upon the construction of (single-program) POEE constructed in Section 6.1. This is done in two main steps:

1. We first instantiate the two-outcome PABE scheme in the construction of POEE to be *stateless* PABE that we define next. We observe that the resulting POEE scheme would also be stateless. Meaning that the the TM encoding and the generation patch algorithms in the POEE scheme output $\mathsf{st} = \perp$.

2. In the next step, we show how to go generically from a stateless POEE to a multi-program POEE. In a stateless POEE, a unique key is used for encoding every machine and also its subsequent patches. Similarly, the issued input encodings work only against the associated machine encodings. In order to be able to make the patch encodings and the input encodings universal, we introduce, respectively, patch encoders and input encoders. A patch encoder is a succinct way of encoding patches for multiple machines. Similarly, an input encoder is also a succinct representation of multiple input encodings. We explain the implementation details of the encoders in the construction below.

Stateless Two-outcome PABE. One of the ingredients we use in our construction of multi-program POEE is a version of two-outcome PABE defined in Section 5.3 where the patching algorithms do not maintain any state in between executions. The formal definition is given below.

Definition 20. *A stateless patchable attribute based encryption (2SPABE) scheme is a 2PABE (two-outcome PABE) scheme, denoted by* $\mathsf{2SPABE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{GenPatch}, \mathsf{ApplyPatch}, \mathsf{Dec})$*, associated with a TM class* $\mathcal{M}$ *equipped with* $(\mathsf{Update}, \mathcal{P})$ *and satisfying the following* stateless *property.*

For every machine $M_0 \in \mathcal{M}$*, sequence of patches* $P_1, \ldots, P_L \in \mathcal{P}$*, consider the following process: (i)* $(\mathsf{2SPABE.PP}, \mathsf{2SPABE.SK}) \leftarrow \mathsf{2SPABE.Setup}(1^\lambda)$*, (ii)* $(\mathsf{2SPABE}.sk_{M_0}, \mathsf{st}_0) \leftarrow \mathsf{2SPABE.KeyGen}(\mathsf{2SPABE.SK}, M_0)$ *and, (iii) for every* $i \in [L]$*,* $(\widetilde{P_i}, \mathsf{st}_i) \leftarrow \mathsf{2SPABE.GenPatch}(\mathsf{2SPABE.SK}, P_i, \mathsf{st}_{i-1})$*.*

We say that $\mathsf{2SPABE}$ *satisfies the stateless property if for every* $i \in \{0, \ldots, L\}$*,* $\mathsf{st}_i = \perp$*.*

Similar to the stateless PABE, we omit the argument st in the description of the algorithms.

The construction of stateless two-outcome PABE from stateless PABE is a replica of the construction from (stateful) PABE to (stateful) two-outcome PABE described in Section 5.3. From Corollary 3, we thus have the following theorem,

Theorem 16. *Let* $\lambda \in \mathbb{N}$ *be a sufficiently large security parameter. Assuming the existence of* $\frac{\epsilon}{2^\lambda}$*-secure indistinguishability obfuscation and* $\frac{\epsilon'}{2^\lambda}$*-secure decisional Diffie-Hellman assumption, there exists a* δ*-secure stateless two-outcome PABE scheme, where* $\epsilon, \epsilon', \delta \leq \frac{1}{p(\lambda)}$ *with* p *being a polynomial.*

Stateless Single-Program POEE. We now instantiate the construction of (single-program) patchable oblivious evaluation encodings described in Section 6.1 with the above stateless two-outcome PABE scheme. We denote the resulting scheme to be POEE. We note that the resulting patchable OEE scheme is stateless, meaning that the algorithms $\mathsf{POEE.TMEncode}$ and $\mathsf{POEE.GenPatch}$ output $\mathsf{st} = \perp$. From Corollary 2, we have

Theorem 17. *Assuming the existence of* $\frac{\epsilon}{2^\lambda}$*-secure iO and* $\frac{\epsilon'}{2^\lambda}$*-secure decisional DDH assumption, there exists a* δ*-secure stateless POEE scheme, where* $\epsilon, \epsilon', \delta \leq \frac{1}{\mathrm{poly}(\lambda)}$*.*

Construction. We now move on to constructing a multi-program patchable oblivious evaluation encodings scheme mpOEE. The class of Turing machines associated with mpOEE is $\mathcal{M}$ and we denote the update algorithm by Update. The building blocks in our scheme are:

1. A fully homomorphic encryption scheme, $\mathsf{FHE} = (\mathsf{FHE.Setup}, \mathsf{FHE.Eval}, \mathsf{FHE.Enc}, \mathsf{FHE.Dec})$.

2. A (single-program) patchable OEE scheme described in Section 6.1, denoted by $\mathsf{POEE} = (\mathsf{POEE.Setup}, \mathsf{POEE.InpEncode}, \mathsf{POEE.TMEncode}, \mathsf{POEE.Decode})$. We denote the auxiliary algorithms associated with POEE to be $(\mathsf{POEE.puncInp}, \mathsf{POEE.PIEncode}, \mathsf{POEE.puncBit}, \mathsf{POEE.PBEncode})$. Recall that in the construction of POEE we use a (stateful) two-outcome PABE scheme. But instead we instantiate with a stateless two-outcome PABE scheme. Observe that the resulting scheme POEE is also stateless.

   The class of Turing machines associated with POEE is $\mathcal{M}'$. The class $\mathcal{M}'$ is a function of $\mathcal{M}$. We denote the update algorithm by $\widetilde{\mathsf{Update}}(\cdot,\cdot)$: it takes as input $(\mathsf{FHE.pk}, \mathsf{FHE.CT}, \mathsf{FHE.CT}')$ and executes $\mathsf{FHE.Eval}(\mathsf{FHE.pk}, \mathsf{Update}(\cdot,\cdot), \mathsf{FHE.CT}, \mathsf{FHE.CT}')$.

3. Indistinguishability obfuscation, denoted by $i\mathcal{O}$.

4. Puncturable pseudorandom function family, denoted by $\mathcal{F}$. We denote the puncture algorithm accompanying $\mathcal{F}$ to be $\mathsf{Puncture}$.

We describe the algorithms of mpOEE below.

$\underline{\mathsf{mpOEE.Setup}(1^\lambda)}$: It takes as input a security parameter λ. It samples a puncturable PRF key $K \xleftarrow{\$} \{0,1\}^\lambda$. It runs the FHE setup algorithm twice, $\{(\mathsf{FHE.pk}_b, \mathsf{FHE.sk}_b) \leftarrow \mathsf{FHE.Setup}(1^\lambda)\}_{b\in\{0,1\}}$. It outputs $\mathsf{sk} = \Big(K, \{\mathsf{FHE.pk}_b, \mathsf{FHE.sk}_b\}_{b\in\{0,1\}}\Big)$.

$\underline{\mathsf{mpOEE.TMEncode}(\mathsf{sk}, M_0, M_1)}$: It takes as input a secret key $\mathsf{sk} = \Big(K, \{\mathsf{FHE.pk}_b, \mathsf{FHE.sk}_b\}_{b\in\{0,1\}}\Big)$, a pair of Turing machines $M_0, M_1 \in \mathcal{M}$ and does the following. It picks a machine ID, $\mathsf{mid} \leftarrow \{0,1\}^\lambda$. Evaluate the pseudorandom function, $(R, K_P, K_{inp}) \leftarrow \mathsf{PRF}(K, \mathsf{mid})$.

Generate a POEE secret key $\mathsf{POEE.sk}_{\mathsf{mid}} \leftarrow \mathsf{POEE.Setup}(1^\lambda; R)$. Generate ciphertexts $\Big\{\mathsf{FHE.CT}_b^M \leftarrow \mathsf{FHE.Enc}(\mathsf{FHE.pk}_b, M_b)\Big\}_{b\in\{0,1\}}$. Generate a POEE TM encoding of $(\mathsf{FHE.CT}_0^M, \mathsf{FHE.CT}_1^M)$ by computing the following:

$$\mathsf{POEE}.\langle \mathsf{FHE.CT}_0^M, \mathsf{FHE.CT}_1^M\rangle \leftarrow \mathsf{POEE.TMEncode}(\mathsf{POEE.sk}, \mathsf{FHE.CT}_0^M, \mathsf{FHE.CT}_1^M; R)$$

Output $\langle M_0, M_1\rangle = (\mathsf{mid}, \mathsf{POEE}.\langle \mathsf{FHE.CT}_0^M, \mathsf{FHE.CT}_1^M\rangle)$.

$\underline{\mathsf{mpOEE.GenPatch}(\mathsf{sk}, P_0, P_1)}$: It takes as input a secret key $\mathsf{sk} = (K, \{\mathsf{FHE.pk}_b, \mathsf{FHE.sk}_b\}_{b\in\{0,1\}})$ and a pair of patches $P_0, P_1 \in \mathcal{P}$. We then encrypt P_b using $\mathsf{FHE.pk}_b$ to obtain the FHE ciphertext $\mathsf{FHE.CT}_b^P$, for $b \in \{0,1\}$. Sample a tag $\tau \leftarrow \{0,1\}^\lambda$. It then generates an obfuscated program, $PG \leftarrow i\mathcal{O}(\mathsf{pgen}[\mathsf{sk}, \mathsf{FHE.CT}_0^P, \mathsf{FHE.CT}_1^P, \tau])$, where pgen is described in Figure 13. Output $\langle P_0, P_1\rangle = PG$.

$\underline{\mathsf{mpOEE.ApplyPatch}\Big(\langle M_0, M_1\rangle, \langle P_0, P_1\rangle\Big)}$: It takes as input a machine encoding $\langle M_0, M_1\rangle = (\mathsf{mid}, \mathsf{POEE}.\langle \mathsf{FHE.CT}_0^M, \mathsf{FHE.CT}_1^M\rangle)$ and a patch encoding $\langle P_0, P_1\rangle = PG$. It then computes $\mathsf{POEE}.\langle \mathsf{FHE.CT}_0^P, \mathsf{FHE.CT}_1^P\rangle \leftarrow PG(\mathsf{mid})$. Finally, the apply-patching algorithm of POEE is executed: $\mathsf{POEE}.\langle \mathsf{FHE.CT}_0^{'M}, \mathsf{FHE.CT}_1^{'M}\rangle \leftarrow \mathsf{POEE.ApplyPatch}(\mathsf{POEE}.\langle \mathsf{FHE.CT}_0^M, \mathsf{FHE.CT}_1^M\rangle, \mathsf{POEE}.\langle \mathsf{FHE.CT}_0^P, \mathsf{FHE.CT}_1^P\rangle)$. Output the updated machine encoding, $\langle M_0', M_1'\rangle = (\mathsf{mid}, \mathsf{POEE}.\langle \mathsf{FHE.CT}_0^{'M}, \mathsf{FHE.CT}_1^{'M}\rangle)$.

pgen

Input: mid
Hardwired values: $(\mathsf{sk} = K), \mathsf{FHE.CT}_0^P, \mathsf{FHE.CT}_1^P, \tau$

1. Generate the POEE secret key corresponding to this machine. First, compute $(R, K_P, K_{inp}) \leftarrow \mathsf{PRF}(K, \mathsf{mid})$. Then, compute $\mathsf{POEE.sk}_{\mathsf{mid}} \leftarrow \mathsf{POEE.Setup}(1^\lambda; R)$.
2. Generate the randomness $r_1 \leftarrow \mathsf{PRF}(K_P, \tau)$. Execute $\mathsf{POEE}.\langle \mathsf{FHE.CT}_0^P, \mathsf{FHE.CT}_1^P \rangle \leftarrow \mathsf{POEE.GenPatch}(\mathsf{POEE.sk}_{\mathsf{mid}}, \mathsf{st} = \perp, \mathsf{FHE.CT}_0^P, \mathsf{FHE.CT}_1^P; r_1)$.
3. Output patch encoding $\mathsf{POEE}.\langle P_0, P_1 \rangle$.

Figure 13: Description of Patch Generator.

$\underline{\mathsf{mpOEE.InpEncode}(\mathsf{sk}, x, i, b)}$: It takes as input a secret key $\mathsf{sk} = (K, \{\mathsf{FHE.pk}_b, \mathsf{FHE.sk}_b\}_{b \in \{0,1\}})$, an input $x \in \{0,1\}^*$, an index $i \geq 0$ and a choice bit $b \in \{0,1\}$. It then computes an obfuscated program, $IG \leftarrow i\mathcal{O}(\mathsf{igen}[\mathsf{sk}, x, i, b])$, where igen is described in Figure 17. Output IG.

igen

Input: mid
Hardwired values: $\big(\mathsf{sk} = (K, \{\mathsf{FHE.pk}_b, \mathsf{FHE.sk}_b\}_{b \in \{0,1\}})\big), x, i, b$

1. Generate the POEE secret key corresponding to this machine. First, compute $(R, K_P, K_{inp}) \leftarrow \mathsf{PRF}(K, \mathsf{mid})$. Then, compute $\mathsf{POEE.sk}_{\mathsf{mid}} \leftarrow \mathsf{POEE.Setup}(1^\lambda; R)$.
2. Generate the randomness $r_2 \leftarrow \mathsf{PRF}\big(K_{inp}, (x, i, b)\big)$.
3. Execute $\mathsf{POEE}.\langle x, i, b \rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}_{\mathsf{mid}}, U_{x, \mathsf{FHE.SK}_b}, i, b; r_2)$, where $U_{x, \mathsf{FHE.sk}_b}$ is a TM that takes as input $\mathsf{FHE.CT}$; $M \leftarrow \mathsf{FHE.Dec}(\mathsf{FHE.sk}_b, \mathsf{FHE.CT})$ and outputs $M(x)$.
4. Output input encoding $\langle x, i, b \rangle = \mathsf{POEE}.\langle x, i, b \rangle$.

Figure 14: Description of Input Encoding Generator.

$\underline{\mathsf{mpOEE.Decode}\Big(\langle M_0, M_1 \rangle, \langle x, i, b \rangle\Big)}$: It takes as input a (possibly updated) machine encoding $\langle M_0, M_1 \rangle = \mathsf{POEE}.\langle \mathsf{FHE.CT}_0^M, \mathsf{FHE.CT}_1^M \rangle$ and an input encoding $\langle x, i, b \rangle = \mathsf{POEE}.\langle x, i, b \rangle$. It then outputs $y \leftarrow \mathsf{POEE.Decode}(\mathsf{POEE}.\langle \mathsf{FHE.CT}_0^M, \mathsf{FHE.CT}_1^M \rangle, \mathsf{POEE}.\langle x, i, b \rangle)$.

The helper algorithms are presented next.

$\underline{\mathsf{mpOEE.puncInp}(\mathsf{sk}, x, i, \mathsf{mid})}$: It takes as input a secret key $\mathsf{sk} = (K, \{\mathsf{FHE.pk}_b, \mathsf{FHE.sk}_b\}_{b \in \{0,1\}})$, input $x \in \{0,1\}^*$, index i and machine id mid. It punctures K at mid by executing $K^{\mathsf{mid}} \leftarrow \mathsf{Puncture}(K, \mathsf{mid})$. Let $(R, K_P, K_{inp}) \leftarrow \mathsf{PRF}(K, \mathsf{mid})$.

1. It computes $\mathsf{POEE.sk} \leftarrow \mathsf{POEE.Setup}(1^\lambda; R)$. It then executes $\mathsf{POEE.sk}_{(x,i)} \leftarrow \mathsf{POEE.puncInp}(\mathsf{POEE.sk}, (x, i))$.
2. It punctures K_{inp} at points $(x, i, 0)$ and $(x, i, 1)$. That is, it computes $K_{inp}^{(x,i)} \leftarrow \mathsf{PRF}\Big(K_{inp},$

$\{(x,i,0),(x,i,1)\}\big)$. Note that here we are puncturing K_{inp} at both the points at the same time.

Output the punctured key $\mathsf{sk}_{x,i,\mathsf{mid}} = \Big(K^{\mathsf{mid}}, \mathsf{POEE.sk}_{(x,i)}, K_{inp}^{(x,i)}, \{\mathsf{FHE.pk}_b, \mathsf{FHE.sk}_b\}_{b\in\{0,1\}}\Big)$.

mpOEE.PIEncode($\mathsf{sk}_{x,i,\mathsf{mid}}, x', i', \mathsf{mid}', b$): It takes as input a punctured secret key $\mathsf{sk}_{x,i,\mathsf{mid}} = (K^{\mathsf{mid}}, \mathsf{POEE.sk}_{(x,i)}, K_{inp}^{(x,i)}, \{\mathsf{FHE.pk}_b, \mathsf{FHE.sk}_b\}_{b\in\{0,1\}})$, an input x', index i', machine id mid' and a bit b s.t. $(x,i,\mathsf{mid}) \neq (x',i',\mathsf{mid}')$. It then computes $IG^* \leftarrow i\mathcal{O}(\mathsf{igenINP}[\mathsf{sk}_{(x,i,\mathsf{mid})}, x', i', b, x, i, \mathsf{mid}])$, where igenINP is described in Figure 15. Output IG^*.

igenINP

Input: mid'
Hardwired values: $\mathsf{sk}_{(x,i,\mathsf{mid})}, x', i', b, x, i, \mathsf{mid}$

1. Parse $\mathsf{sk}_{(x,i,\mathsf{mid})} = (K^{\mathsf{mid}}, \mathsf{POEE.sk}_{(x,i)}, K_{inp}^{(x,i)}, \{\mathsf{FHE.pk}_b, \mathsf{FHE.sk}_b\}_{b\in\{0,1\}})$.
2. If $\mathsf{mid}' \neq \mathsf{mid}$ then do the following:
   - Generate the POEE secret key corresponding to this machine. First, compute $(R', K'_P, K'_{inp}) \leftarrow \mathsf{PRF}(K^{\mathsf{mid}}, \mathsf{mid}')$. Then, compute $\mathsf{POEE.sk}_{\mathsf{mid}'} \leftarrow \mathsf{POEE.Setup}(1^\lambda; R')$.
   - Generate the randomness $r'_2 \leftarrow \mathsf{PRF}\big(K'_{inp}, (x',i',b)\big)$.
   - Execute $\mathsf{POEE}.\langle x', i', b\rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}_{\mathsf{mid}'}, U_{x',\mathsf{FHE.sk}_b}, i', b; r'_2)$, where $U_{x',\mathsf{FHE.sk}_b}$ is a TM that takes as input FHE.CT; $M \leftarrow \mathsf{FHE.Dec}(\mathsf{FHE.sk}_b, \mathsf{FHE.CT})$ and outputs $M(x')$.
3. If $\mathsf{mid}' = \mathsf{mid}$ and $(x,i) \neq (x',i')$ then do the following:
   - Generate the randomness $r'_2 \leftarrow \mathsf{PRF}\big(K_{inp}^{(x,i)}, (x',i',b)\big)$.
   - Execute $\mathsf{POEE}.\langle x', i', b\rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}_{(x,i)}, U_{x',\mathsf{FHE.sk}_b}, i', b; r'_2)$, where $U_{x',\mathsf{FHE.sk}_b}$ is a TM that takes as input FHE.CT; $M \leftarrow \mathsf{FHE.Dec}(\mathsf{FHE.sk}_b, \mathsf{FHE.CT})$ and outputs $M(x')$.
4. Output input encoding $\langle x', i', b\rangle = \mathsf{POEE}.\langle x', i', b\rangle$.

Figure 15:

mpOEE.puncBit(sk, b): It takes as input a secret key $\mathsf{sk} = (K, \{\mathsf{FHE.pk}_b, \mathsf{FHE.sk}_b\}_{b\in\{0,1\}})$ and an input bit b. It outputs $\mathsf{sk}_b = (K, \mathsf{FHE.pk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_b)$.

mpOEE.PBEncode($\mathsf{sk}_b, x, i, \mathsf{mid}$): It takes as input a key $\mathsf{sk}_b = (K, \mathsf{FHE.pk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_b)$, an input x, an index i and a machine id mid. It then computes $IG^* \leftarrow i\mathcal{O}(\mathsf{igenBIT}[\mathsf{sk}_b, x, i, b, \mathsf{mid}])$, where igenBIT is described in Figure 16. Output IG^*.

We argue the correctness and the security properties below.

Correctness. We show that mpOEE satisfies all the three correctness properties below. But first we establish some notation.

```
igenBIT

Input: mid
Hardwired values: $\mathsf{sk}_b = (K, \mathsf{FHE.pk}_0, \mathsf{FHE.pk}_1, \mathsf{FHE.sk}_b), x, i, b$

  1. Generate the POEE secret key corresponding to this machine. First, compute $(R, K_P, K_{inp}) \leftarrow \mathsf{PRF}(K, \mathsf{mid})$. Then, compute $\mathsf{POEE.sk}_{\mathsf{mid}} \leftarrow \mathsf{POEE.Setup}(1^\lambda; R)$.
  2. Generate the randomness $r_2 \leftarrow \mathsf{PRF}\big(K_{inp}, (x, i, b)\big)$.
  3. Execute $\mathsf{POEE}.\langle x, i, b\rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}_{\mathsf{mid}}, U_{x,\mathsf{FHE.SK}_b}, i, b; r_2)$, where $U_{x,\mathsf{FHE.sk}_b}$ is a TM that takes as input FHE.CT; $M \leftarrow \mathsf{FHE.Dec}(\mathsf{FHE.sk}_b, \mathsf{FHE.CT})$ and outputs $M(x)$.
  4. Output input encoding $\langle x, i, b\rangle = \mathsf{POEE}.\langle x, i, b\rangle$.
```

Figure 16:

Consider a pair of machines $(M_0, M_1) \in \mathcal{M}$ and an input $x \in \{0,1\}^*$. Let $(P_0^1, P_1^1)\ldots,(P_0^L, P_1^L) \in \mathcal{P}^2$. Correspondingly, we denote $\langle M_0, M_1\rangle = (\mathsf{mid}, \mathsf{POEE}.\langle M_0, M_1\rangle)$ to be the TM encoding of mpOEE and the secure patches are denoted by $\{\langle P_0, P_1\rangle\}_{i\in[L]}$. Further, we denote the input encoding to be $\langle x, i, b\rangle$, for some $i \in [L]$ and bit b. The secret key used in the computation of these encodings is sk. We consider below different ways the input encoding could be generated.

1. *Correctness of Encode, Patching and Decode:* From the correctness of POEE, it follows that:

$$M_{b,L}(x) \leftarrow \mathsf{POEE.Decode}\Big(\mathsf{POEE}.\langle \mathsf{FHE.CT}^M_{0,L}, \mathsf{FHE.CT}^M_{1,L}\rangle, \mathsf{POEE}.\langle U_{x,\mathsf{FHE.sk}_b}, i, b\rangle\Big),$$

   where,

   - $\mathsf{sk} = (K, \{\mathsf{FHE.pk}_c, \mathsf{FHE.sk}_c\}_{c\in\{0,1\}}) \leftarrow \mathsf{mpOEE.Setup}(1^\lambda)$,
   - $\mathsf{POEE.sk}_{\mathsf{mid}} \leftarrow \mathsf{POEE.Setup}(1^\lambda; R)$, where $(R, K_P, K_{inp}) \leftarrow \mathsf{PRF}(K, \mathsf{mid})$
   - $\mathsf{FHE.CT}^M_{c,L} \leftarrow \mathsf{FHE.Enc}(\mathsf{FHE.pk}, M_{c,L})$, for $c \in \{0,1\}$
   - $\langle \mathsf{FHE.CT}^M_{0,L}, \mathsf{FHE.CT}^M_{1,L}\rangle \leftarrow \mathsf{POEE.TMEncode}(\mathsf{POEE.sk}_{\mathsf{mid}}, \mathsf{FHE.CT}^M_{0,L}, \mathsf{FHE.CT}^M_{1,L})$
   - $\mathsf{POEE}.\langle U_{x,\mathsf{FHE.sk}_b}, i, b\rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}_{\mathsf{mid}}, x, i, b)$
   - $M_{b,i} \leftarrow \mathsf{Update}(M_{b,i-1}, P^i_b)$.

2. *Correctness of Input Puncturing:* Suppose $\mathsf{sk}_{(x^*,i^*,\mathsf{mid}^*)} \leftarrow \mathsf{mpOEE.puncInp}(\mathsf{sk}, x^*, i^*, \mathsf{mid}^*)$, where $(x^*, i^*, \mathsf{mid}^*) \neq (x, i, \mathsf{mid})$. And let $IG = \langle x, i, b\rangle \leftarrow \mathsf{mpOEE.PIEncode}(\mathsf{sk}_{x^*,i^*,\mathsf{mid}^*}, x, i, \mathsf{mid}, b)$. By inspection, it follows that $\mathsf{POEE}.\langle x, i, b\rangle \leftarrow IG(\mathsf{mid})$ is a valid POEE input encoding of (x, i, b). Hence, similar to the previous case, we have the output of the decode algorithm on input encodings $\langle M_{0,L}, M_{1,L}\rangle$ and $\langle x, i, b\rangle$ to be $M_{b,L}(x)$.

3. *Correctness of Bit Puncturing:* Suppose the secret key sk is punctured at bit b. And for some $x \in \{0,1\}^*, i \in [L], b \in \{0,1\}$, let $IG^* \leftarrow i\mathcal{O}(\mathsf{igenBIT}[\mathsf{sk}_b, x, i, b, \mathsf{mid}])$, where igenBIT is described in Figure 16 and sk_b is a secret key punctured at b. Then $IG(\mathsf{mid}) = IG^*(\mathsf{mid})$ which means that encoding w.r.t the bit punctured key leads to a valid input encoding and so, the rest follows from the correctness of encode, patching and decode properties.

We argue the security properties below.

Theorem 18. *From Theorem 11 and assuming the security of $i\mathcal{O}$ and $\mathcal{F}$, we have that* mpOEE *satisfies indistinguishability of encoding bit property.*

Proof. Let $\mathcal{A}$ be the PPT adversary in the indistinguishability of bit encoding experiment. Consider the hybrids below. By presenting a sequence of hybrids, we argue that the probability that the adversary outputs the challenge bit is negligibly close to 1/2.

Hyb_1: This corresponds to the real experiment.

In this hybrid, $\mathcal{A}$ chooses a sequence of Turing machine pairs $(M_0^1, M_1^1, \ldots, M_0^Q, M_1^Q) \in \mathcal{M}$ such that $|M_0^j| = |M_1^j|$, an input x, an index $\mathbf{i} \geq 0$ and a machine id mid. $\mathcal{A}$ then sends the tuple $\big(\{M_0^j, M_1^j\}_{j\in[Q]}, x, \mathbf{i}, \mathsf{mid}\big)$ to the challenger. The challenger chooses a bit $b \in \{0,1\}$ and computes the following: (a) $\mathsf{sk} \leftarrow \mathsf{mpOEE.Setup}(1^\lambda)$, (b) for $j \in [Q]$, $\Big(\langle M_0^j, M_1^j\rangle\Big) \leftarrow \mathsf{mpOEE.TMEncode}(\mathsf{sk}, M_0^j, M_1^j)$, (c) $\mathsf{sk}_{x,i,\mathsf{mid}} = \big(K^{\mathsf{mid}}, \mathsf{POEE.sk}_{(x,i)}, K_{inp}^{(x,i)}, \{(\mathsf{FHE.pk}_c, \mathsf{FHE.sk}_c)_{c\in\{0,1\}}\}\big) \leftarrow \mathsf{mpOEE.puncInp}(\mathsf{sk}, x, i, \mathsf{mid})$ (d) $IG^* = \langle x, \mathbf{i}, \mathsf{mid}, b\rangle \leftarrow \mathsf{mpOEE.InpEncode}(\mathsf{sk}, x, \mathbf{i}, \mathsf{mid}, b)$. It sends $\big(\{\langle M_0^j, M_1^j\rangle\}_{j\in[Q]}, \mathsf{sk}_{x,i,\mathsf{mid}}, \langle x, i, \mathsf{mid}, b\rangle\big)$ to $\mathcal{A}$.

In the patch query phase, the following is repeated polynomially many times:

- $\mathcal{A}$ chooses two patches $P_0^i, P_1^i \in \mathcal{P}$ and sends them to the challenger.
- The challenger computes $\Big(\langle P_0^i, P_1^i\rangle\Big) \leftarrow \mathsf{mpOEE.GenPatch}(\mathsf{sk}, P_0^i, P_1^i)$. It sends $\langle P_0^i, P_1^i\rangle$ to $\mathcal{A}$.

In the end, $\mathcal{A}$ outputs b'.

Hyb_2: In this hybrid, we change $IG^* = \langle x, i, b\rangle$, which is the input encoding sent by the challenger to $\mathcal{A}$. The program IG^* is now generated by executing $i\mathcal{O}\Big(\mathsf{igenHYB}\Big[K^{\mathsf{mid}}, \{(\mathsf{FHE.pk}_c, \mathsf{FHE.sk}_c)_{c\in\{0,1\}}\}, x, i, b, y\Big]\Big)$, where $\mathsf{igenHYB}$ is described in Figure 16 and K^{mid} is obtained by puncturing K at adversarially chosen mid. Here, $y = \mathsf{POEE}.\langle x, i, b\rangle$ is the pre-computed output of $\big(i\mathcal{O}(\mathsf{igen}[\mathsf{sk}, x, i, b])\big)(\mathsf{mid})$. The rest of the hybrid is same as before.

The programs igen and $\mathsf{igenHYB}$ are equivalent. Hence, from the security of iO, it follows that hybrids Hyb_1 and Hyb_2 are computationally indistinguishable.

```
                              igenHYB

Input: $\mathsf{mid}'$
Hardwired values: $(K^{\mathsf{mid}}, \{(\mathsf{FHE.pk}_c, \mathsf{FHE.sk}_c)_{c\in\{0,1\}}\}, x, i, b, y)$

  1. If $\mathsf{mid}' = \mathsf{mid}$ then output $y$.
  2. Otherwise do the following:
       • Generate the POEE secret key corresponding to this machine. First, compute $(R', K'_P, K'_{inp}) \leftarrow \mathsf{PRF}(K^{\mathsf{mid}}, \mathsf{mid}')$. Then, compute $\mathsf{POEE.sk}_{\mathsf{mid}'} \leftarrow \mathsf{POEE.Setup}(1^\lambda; R')$.
       • Generate the randomness $r'_2 \leftarrow \mathsf{PRF}\big(K'_{inp}, (x', i', b)\big)$.
       • Execute $\mathsf{POEE}.\langle x, i, b\rangle \leftarrow \mathsf{POEE.InpEncode}(\mathsf{POEE.sk}_{\mathsf{mid}'}, U_{x,\mathsf{FHE.sk}_b}, i, b; r'_2)$.
```

Figure 17: Hybrid Input Encoding Generator.

Hyb_4: This time hardwire the output corresponding to the input mid in all the patches. That is, for every patch (P_0, P_1) requested, generate the secure patch $\langle P_0, P_1\rangle \leftarrow i\mathcal{O}(\mathsf{pgenHYB1}[K^{\mathsf{mid}}, x, i, b, y])$, where $\mathsf{pgenHYB1}$ is described in Figure 18. Here, y is a pre-computed value obtained by executing $y \leftarrow \left(i\mathcal{O}(\mathsf{pgen}[K, \mathsf{FHE.CT}_0^P, \mathsf{FHE.CT}_0^P, \tau])\right)$, where $\{\mathsf{FHE.CT}_b^P\}_{b \in \{0,1\}}$ are FHE ciphertexts of P_0 and P_1. The rest of the hybrid is same as before.

The programs pgen and $\mathsf{pgenHYB1}$ are identical. From the security of $i\mathcal{O}$, it follows that the hybrids Hyb_3 and Hyb_4 are computationally indistinguishable.

```
                              pgenHYB1

Input: $\mathsf{mid}'$
Hardwired values: $(K^{\mathsf{mid}}, \mathsf{FHE.CT}_0^P, \mathsf{FHE.CT}_1^P, \tau, y)$

  1. If $\mathsf{mid}' = \mathsf{mid}$ then output $y$.
  2. Otherwise, do the following:
      • Generate the POEE secret key corresponding to this machine. First, compute $(R', K'_P, K'_{inp}) \leftarrow \mathsf{PRF}(K^{\mathsf{mid}}, \mathsf{mid}')$. Then, compute $\mathsf{POEE.sk}_{\mathsf{mid}} \leftarrow \mathsf{POEE.Setup}(1^\lambda; R')$.
      • Generate the randomness $r_1 \leftarrow \mathsf{PRF}(K_P, \tau)$. Execute $\mathsf{POEE}.\langle \mathsf{FHE.CT}_0^P, \mathsf{FHE.CT}_0^P \rangle \leftarrow \mathsf{POEE.GenPatch}(\mathsf{POEE.sk}_{\mathsf{mid}}, \mathsf{st} = \perp, \mathsf{FHE.CT}_0^P, \mathsf{FHE.CT}_0^P; r_1)$.
      • Output patch encoding $\langle P_0, P_1 \rangle = \mathsf{POEE}.\langle \mathsf{FHE.CT}_0^P, \mathsf{FHE.CT}_0^P \rangle$.
```

Figure 18: Hybrid Patch Generator.

Hyb_5: Unlike the previous hybrid, we use uniform randomness in the key generation of $\mathsf{POEE.sk}_{\mathsf{mid}}$ and this in turn is used to produce the pre-computed output $y = \mathsf{POEE}.\langle x, i, b \rangle$. In the previous hybrid, $\mathsf{PRF}(K, \mathsf{mid})$ was instead used. The rest of the hybrid is as before. The adversary is only given the key K punctured at mid.

From the security of puncturable PRFs, it follows that Hyb_2 and Hyb_3 are computationally indistinguishable.

Hyb_6: In the previous hybrid, uniform randomness was used in the generation procedure of the secret key $\mathsf{POEE.sk}_{\mathsf{mid}}$. However, the randomness used in the encoding of $y = \mathsf{POEE}.\langle x, i, b \rangle$ was still using $\mathsf{PRF}(K_{inp}, (x, i, b))$, where K_{inp} is derived from $\mathsf{PRF}(K, \mathsf{mid})$. In this hybrid, we use uniform randomness in the encoding of y. Similarly, even for the secure patches, we compute every patch encoding using uniform randomness.

Even in the presence of the punctured key $K_{inp}^{(x,i)}$, the security of punctured PRFs imply that the indistinguishability of Hyb_2 and Hyb_3.

Hyb_7: In the final hybrid, we now switch from the encoding $y = \langle x, i, b \rangle$ to the encoding $y = \langle x, i, 0 \rangle$. The rest of the hybrid is the same as before.

The indistinguishability of Hyb_6 and Hyb_7 can be reduced to the indistinguishability of bit encoding of POEE: the reduction forwards the TM queries to the external challenger (of indistinguishability of bit encoding of POEE) and then the answers obtained will now be forwarded to the receiver. For the patch queries, the reduction appropriately computes the obfuscated program with the patches (received from the challenger) hardwired inside these programs. For the challenge input encoding as well, it computes an obfuscated program with the received input encoding hardwired into it.

□

Theorem 19. *Assuming the security of* FHE*, we have that* mpOEE *satisfies indistinguishability of machine encoding property.*

Proof Sketch. Suppose let adversary $\mathcal{A}$ requests for secret key punctured at bit 0. The case when he requests for bit 1 is symmetrical. In this case, he is provided the FHE secret key $\mathsf{FHE.sk}_0$ and in particular, the key $\mathsf{FHE.sk}_1$ is not given to $\mathcal{A}$. This means that in every machine encoding of the form $\langle \mathsf{FHE.CT}^M_{0,j}, \mathsf{FHE.CT}^M_{1,j}\rangle$ (resp., $\langle \mathsf{FHE.CT}^P_{0,i}, \mathsf{FHE.CT}^P_{1,i}\rangle$), the plaintext messages in $\mathsf{FHE}^{\cdot}_{0,\cdot}$ can be modified without the PPT adversary $\mathcal{A}$ noticing the change: follows from the semantic security of FHE. This proves the theorem.

From Theorem 18 and Theorem 19, we have that the scheme mpOEE is secure. By instantiating the underlying tools in mpOEE, we get the following theorem.

Theorem 20. *Assuming the existence of* $\frac{\epsilon}{2^\lambda}$*-secure indistinguishability obfuscation and* $\frac{\epsilon'}{2^\lambda}$*-secure decisional Diffie-Hellman assumption, there exists a* δ*-secure multi-program POEE scheme, where* $\epsilon, \epsilon', \delta \leq \frac{1}{p(\lambda)}$ *for some polynomial* p*.*

11 From Multi-Program OEE to Multi-Program PO

We instantiate the POEE scheme in the transformation from (single-program) POEE to single-program patchable iO in Section 7 with multi-program OEE and the resulting primitive is a multi-program iO. However, the security analysis for the single-program setting does not immediately work in the multi-program case. We need to adopt the security guarantees offered by the multi-program POEE to make the security proof work. Since, the overall structure of the hybrids more or less follows along the lines of the security proof of Section 7, we sketch the main steps.

Let $\Big((M_0^1, M_1^1), \ldots, (M_0^Q, M_1^Q)\Big)$ be the sequence of the machine pairs sent by $\mathcal{A}$. Also let $\Big((P_0^1, P_1^1), \ldots, (P_0^L, P_1^L)\Big)$ be the sequence of patch pairs adaptively queried by $\mathcal{A}$.

- **Step I: Challenge bit = 0.** The challenger obfuscates machine M_0^i, for all $i \in [Q]$. Further, it uses P_0^i, for all $i \in [L]$ to generate the secure patches. Note that an obfuscation of M_0^i is a multi-program POEE encoding of (M_0^i, M_0^i) and a secure patch of P_0^j is a mpPOEE patch encoding of (P_0^j, P_0^j). The mpPOEE secret key is used to encode the inputs that will be later decoded by the machine encodings.

- **Step II: From** (M_0, M_0) **(resp.,** (P_0, P_0)**) to** (M_0, M_1) **(resp.,** (P_0, P_1)**).** In this step, we switch the TM encodings of the form (M_0, M_0) to encodings of the form (M_0, M_1). Similarly, we switch the patch encodings of the form (P_0, P_0) to (P_0, P_0).

  This step is performed by first substituting the mpPOEE secret key with a punctured key that enables only encoding inputs w.r.t bit 0. Once this is done, by the indistinguishability of machine encoding property, we can switch *every* machine encoding from (M_0^i, M_0^i) to (M_0^i, M_1^i). Similarly this switch is also performed for the case of patch encodings.

  We note that the decoding of an input encoding of x with the machine encoding (M_0^i, M_1^i) results in the output $M_0^i(x)$. In other words, M_1^i is not used in any computation in this step.

- **Step III: Switching Computation one at a time.** Now, we have machine encodings (resp., patch encodings) of the form (M_0^i, M_1^i) (resp., (P_0^j, P_1^j)). In this step, we shift the space of inputs that evaluate on M_0^i to evaluating on M_1^i. Initially, the entire space of inputs evaluate on M_0^i (this is from the previous step). One input at a time, we then shift the space of inputs evaluating on M_1^i. In the end, the entire space of inputs evaluate on M_1^i.

  This switching process is enabled by input puncturing. Let us illustrate with a simple example. Consider an input x. Initially, the decoding of x and (M_0^i, M_1^i) results in the evaluation of $M_0^i(x)$. We puncture the secret key at x – this itself will not suffice and we also need to puncture at the patch index[9]. Once the puncturing is done, we then can encode x to be evaluated on M_1^i. This step is possible from the security of indistinguishability of bit encoding property.

  Unlike the single-program case, we also need to puncture the secret key at the machine id. Once we puncture the secret key at a particular machine id, we change the input encodings for that step. We then un-puncture this secret key and then puncture it at a different machine id and so on.

- **Step IV: From (M_0, M_1) (resp., (P_0, P_1)) to (M_1, M_1) (resp., (P_1, P_1)).** This is identical to Step II, except we change the first component of (M_0, M_1) in the machine encoding. We go through the same process sketched in Step II to achieve this end goal.

- **Step V: Challenge bit = 1.** Once the previous step is completed, we notice that the obfuscations issued correspond to only machines M_1^i and the secure patches issued correspond to P_1^j.

12 Implications of Patchable Obfuscation

In this section, we first show how to use multi-program patchable obfuscation to construct a secret-key functional encryption (FE) scheme for unbounded-input Turing machines. We then extend our construction in a simple manner to obtain a secret-key multi-input functional encryption (MIFE) scheme for functions of unbounded arity. Both of our resulting constructions rely on adaptively secure multi-program patchable obfuscation. Furthermore, in both of our constructions, we achieve indistinguishability security against adaptive post-ciphertext key queries, namely, where the adversary first submits all the ciphertext queries and then issues function key queries in an adaptive fashion. We refer the reader to [AS16] and [BGJS15] for formal definitions of FE for unbounded-input Turing machines and MIFE for functions of unbounded arity, respectively.

12.1 FE for Unbounded-Input Turing Machines

Let $\mathcal{M}$ be any family of Turing machines that supports arbitrary length inputs. We describe a secret-key FE scheme $\mathsf{FE} = (\mathsf{FE.Setup}, \mathsf{FE.KeyGen}, \mathsf{FE.Enc}, \mathsf{FE.Dec})$ for $\mathcal{M}$ that achieves indistinguishability security against adaptive post-ciphertext key queries. The only ingredient in our construction is an adaptively secure multi-program patchable obfuscation scheme $\mathsf{mp.pO} = (\mathsf{Setup}, \mathsf{Obfuscate}, \mathsf{GenPatch}, \mathsf{ApplyPatch}, \mathsf{Evaluate})$ for a Turing machine family $\mathcal{M}_{\mathsf{mpo}}$ with associated patch family $\mathcal{P}_{\mathsf{mpo}}$ and update algorithm $\mathsf{Update}_{\mathsf{mpo}}$:

[9]This indicates the number of patches issued so far.

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

We now proceed to describe FE.

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

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

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

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

12.2 MIFE for Unbounded-Arity Functions

Let $\mathcal{M}$ be any family of Turing machines that supports arbitrary number of arbitrary length inputs. We describe a secret-key MIFE scheme $\mathsf{MIFE} = (\mathsf{MIFE.Setup}, \mathsf{MIFE.KeyGen}, \mathsf{MIFE.Enc}, \mathsf{MIFE.Dec})$ for $\mathcal{M}$ that achieves indistinguishability security against adaptive post-ciphertext key queries. The only ingredient in our construction is an adaptively secure multi-program patchable obfuscation scheme $\mathsf{mp.pO} = (\mathsf{Setup}, \mathsf{Obfuscate}, \mathsf{GenPatch}, \mathsf{ApplyPatch}, \mathsf{Evaluate})$ for a Turing machine family $\mathcal{M}_{\mathsf{mpo}}$ with associated patch family $\mathcal{P}_{\mathsf{mpo}}$ and update algorithm $\mathsf{Update}_{\mathsf{mpo}}$:

- A Turing machine $\mathsf{TM} \in \mathcal{M}_{\mathsf{mpo}}$ is of the form $\mathsf{TM} = \mathsf{TM}_{[M,\ell,x_1,\ldots,x_\ell]}$ where $M \in \mathcal{M}$, $\ell \geq 0$ and $x_i \in \{0,1\}^*$. On *any* input y, TM outputs $\perp$ if $\ell = 0$. Otherwise, it computes and outputs $M(x_1, \ldots, x_\ell)$.

- A patch $P \in \mathcal{P}_{\mathsf{mpo}}$ is of the form $P = P_{[x]}$ where $x \in \{0,1\}^*$.

- The update algorithm $\mathsf{Update}_{\mathsf{mpo}}$ on input $(\mathsf{TM}_{[M,\ell,x_1,\ldots,x_\ell]}, P_{[x]})$ outputs $\mathsf{TM}' = \mathsf{TM}_{[M,\ell',x_1,\ldots,x_{\ell'}]}$ where $\ell' = \ell + 1$ and $x_{\ell'} = x$.

We now proceed to describe MIFE.

- $\mathsf{MIFE.Setup}(1^\lambda)$: On input the security parameter λ in unary, compute $\mathsf{Obf.SK} \leftarrow \mathsf{Setup}(1^\lambda)$. Output $\mathsf{MIFE.msk} = \mathsf{Obf.SK}$.

- $\mathsf{MIFE.KeyGen}(\mathsf{MIFE.msk}, M)$: On input $\mathsf{MIFE.msk} = \mathsf{Obf.SK}$ and a Turing machine $M \in \mathcal{M}$, compute $\langle \mathsf{TM}_{M_0} \rangle \leftarrow \mathsf{Obfuscate}(\mathsf{Obf.SK}, \mathsf{TM}_{[M,0]})$ where $\mathsf{TM}_{[M,0]} \in \mathcal{M}_{\mathsf{mpo}}$. Output $\mathsf{MIFE}.sk_M = \langle \mathsf{TM}_{M_0} \rangle$.

- $\mathsf{MIFE.Enc}(\mathsf{MIFE.msk}, x)$: On input $\mathsf{MIFE.msk} = \mathsf{Obf.SK}$ and a message $x \in \{0,1\}^*$, compute $\langle P_x \rangle \leftarrow \mathsf{GenPatch}(\mathsf{Obf.SK}, P_{[x]})$ where $P_{[x]} \in \mathcal{P}_{\mathsf{mpo}}$. Output $\mathsf{ct} = \langle P_x \rangle$.

- $\mathsf{MIFE.Dec}(\mathsf{MIFE}.sk_M, \mathsf{ct}_1, \ldots, \mathsf{ct}_\ell)$: On input a functional key $\mathsf{MIFE}.sk_M = \langle \mathsf{TM}_M \rangle$ and an arbitrary number of ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_\ell$ where $\mathsf{ct}_i = \langle P_{x_i} \rangle$, compute for every $i \in [\ell]$, $\langle \mathsf{TM}_{M_i} \rangle \leftarrow \mathsf{ApplyPatch}\Big(\langle \mathsf{TM}_{M_{i-1}} \rangle, \langle P_{x_i} \rangle\Big)$. Output $\mathsf{Evaluate}\Big(\langle \mathsf{TM}_{M_\ell} \rangle, 0\Big)$.

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

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

Acknowledgements

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

References

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

[ABSV15] Prabhanjan Ananth, Zvika Brakerski, Gil Segev, and Vinod Vaikuntanathan. From selective to adaptive security in functional encryption. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 657–677, 2015.

[ACC$^+$15] Prabhanjan Ananth, Kai-Min Chung, Yu-Chi Chen, Huijia Lin, and Wei-Kai Lin. Delegating ram computations with adaptive soundness and privacy. Manuscript, 2015.

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

[AJS15] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Indistinguishability obfuscation with constant size overhead. Cryptology ePrint Archive, Report 2015/1023, 2015.

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

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

[BGG$^+$14] Dan Boneh, Craig Gentry, Sergey Gorbunov, Shai Halevi, Valeria Nikolaenko, Gil Segev, Vinod Vaikuntanathan, and Dhinakaran Vinayagamurthy. Fully key-homomorphic encryption, arithmetic circuit ABE and compact garbled circuits. In *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, pages 533–556, 2014.

[BGI$^+$12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

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

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

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

[BKS15] Zvika Brakerski, Ilan Komargodski, and Gil Segev. From single-input to multi-input functional encryption in the private-key setting. *IACR Cryptology ePrint Archive*, 2015:158, 2015.

[BV15] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. *IACR Cryptology ePrint Archive*, 2015:163, 2015.

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

[CCC+15] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Computation-trace indistinguishability obfuscation and its applications. *IACR Cryptology ePrint Archive*, 2015:406, 2015.

[CCHR15] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Succinct adaptive garbled ram. Cryptology ePrint Archive, Report 2015/1074, 2015. http://eprint.iacr.org/.

[CH15] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. *IACR Cryptology ePrint Archive*, 2015:388, 2015.

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

[CHV15] Aloni Cohen, Justin Holmgren, and Vinod Vaikuntanathan. Publicly verifiable software watermarking. *IACR Cryptology ePrint Archive*, 2015:373, 2015.

[CLTV15] Ran Canetti, Huijia Lin, Stefano Tessaro, and Vinod Vaikuntanathan. Obfuscation of probabilistic circuits and applications. In *Theory of Cryptography*, pages 468–497. Springer, 2015.

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

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

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

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

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

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In

Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 555–564. ACM, 2013.

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

[GLSW15] Craig Gentry, Allison B. Lewko, Amit Sahai, and Brent Waters. Indistinguishability obfuscation from the multilinear subgroup elimination assumption. In *FOCS*, 2015.

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

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

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

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

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

[LV13] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer Science & Business Media, 2013.

[NW15] Ryo Nishimaki and Daniel Wichs. Watermarking cryptographic programs against arbitrary removal strategies. *IACR Cryptology ePrint Archive*, 2015:344, 2015.

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

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

[Wat15] Brent Waters. A punctured programming approach to adaptively secure functional encryption. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 678–697, 2015.

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