Constrained Pseudorandom Functions for Unconstrained Inputs

Apoorvaa Deshpande[1] *, Venkata Koppula[2], and Brent Waters[2]**

[1] Brown University, USA
acdeshpa@cs.brown.edu
[2] University of Texas at Austin, Austin, USA
{kvenkata, bwaters}@cs.utexas.edu

Abstract. A constrained pseudo random function (PRF) behaves like a standard PRF, but with the added feature that the (master) secret key holder, having secret key K, can produce a constrained key, $K\{f\}$, that allows for the evaluation of the PRF on all inputs satisfied by the constraint f. Most existing constrained PRF constructions can handle only bounded length inputs. In a recent work, Abusalah et al. [1] constructed a constrained PRF scheme where constraints can be represented as Turing machines with unbounded inputs. Their proof of security, however, requires risky "knowledge type" assumptions such as differing inputs obfuscation for circuits and SNARKs.

In this work, we construct a constrained PRF scheme for Turing machines with unbounded inputs under weaker assumptions, namely, the existence of indistinguishability obfuscation for circuits (and injective pseudorandom generators).

1 Introduction

Constrained pseudorandom functions (PRFs), as introduced by [7, 9, 23], are a useful extension of standard PRFs [18]. A constrained PRF system is defined with respect to a *family of constraint functions*, and has an additional algorithm Constrain. This algorithm allows a (master) PRF key holder, having PRF key K, to produce a *constrained* PRF key $K\{f\}$ corresponding to a constraint f. This constrained key $K\{f\}$ can be used to evaluate the PRF at all points x accepted by f (that is, $f(x) = 1$). The security notion ensures that even when given multiple constrained keys $K\{f_1\}, \ldots, K\{f_Q\}$, PRF evaluation at a point not accepted by any of the functions f_i 'looks' uniformly random to a computationally bounded adversary. Since their inception, constrained PRFs have found several

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

** Supported by NSF CNS-0952692, CNS-1228599 and CNS-1414082. DARPA through the U.S. Office of Naval Research under Contract N00014-11-1-0382, Google Faculty Research award, the Alfred P. Sloan Fellowship, Microsoft Faculty Fellowship, and Packard Foundation Fellowship.

applications such as broadcast encryption, identity-based key exchange, policy-based key distribution[7] and multi-party key exchange [8]. In particular, even the most basic class of constrained PRFs called puncturable PRFs has found immense application in the area of program obfuscation through the 'punctured programming' technique introduced by [25]. The initial works of [7, 9, 23] showed that the [18] PRF construction can be modified to construct a basic class of constrained PRFs called prefix-constrained PRFs (which also includes puncturable PRFs). Boneh and Waters [7] also showed a construction for the richer class of circuit-constrained PRFs [3] using multilinear maps [14]. Since then, we have seen great progress in this area, leading to constructions from different cryptographic assumptions [8, 10, 4] and constructions with additional properties [12, 1, 10, 4]. However, all the above mentioned works have a common limitation: the corresponding PRF can handle only bounded length inputs.

The problem of constructing constrained PRFs with unbounded length was studied in a recent work by Abusalah, Fuchsbauer and Pietrzak [1], who also showed motivating applications such as broadcast encryption with unbounded recipients and multi-party identity based non-interactive key exchange with no apriori bound on number of parties. Abusalah et al. construct a constrained PRF scheme where the constraint functions are represented as Turing machines with unbounded inputs. The scheme is proven secure under the assumption that differing input obfuscation ($di\mathcal{O}$) for circuits exists. Informally, this assumption states that there exists an 'obfuscation' program $\mathcal{O}$ that takes as input a circuit C, and outputs another circuit $\mathcal{O}(C)$ with the following security guarantee: if an efficient adversary can distinguish between $\mathcal{O}(C_1)$ and $\mathcal{O}(C_2)$, then there exists an efficient extraction algorithm that can find an input x such that $C_1(x) \neq C_2(x)$. However, the $di\mathcal{O}$ assumption is believed to be a risky one due to its 'extractability nature'. Furthermore, the work of [16] conjectures that there exist certain function classes for which $di\mathcal{O}$ is impossible to achieve.

A natural direction then is to try to base the security on the relatively weaker assumption of indistinguishability obfuscation ($i\mathcal{O}$) for circuits. An obfuscator $\mathcal{O}$ is an indistinguishability obfuscator for circuits if for any two circuits C_1 and C_2 that have identical functionality, their obfuscations $\mathcal{O}(C_1)$ and $\mathcal{O}(C_2)$ are computationally indistinguishable. Unlike $di\mathcal{O}$, there are no known impossibility results for $i\mathcal{O}$, and moreover, there has been recent progress [17, 2, 6] towards the goal of constructing $i\mathcal{O}$ from standard assumptions. This brings us to the central question of our work:

Can we construct constrained PRFs for Turing machines under the assumptions that indistinguishability obfuscation and one-way functions exist?

Our starting point is three recent works that build indistinguishability obfuscation for Turing Machines with bounded length inputs using $i\mathcal{O}$ for circuits [5, 11, 24]. The works of [5, 11] show how to do this where the encoding time and size of the obfuscated program grows with the maximum space used by the underlying program, whereas the work of [24] achieves this with no such

[3] Where the constraints can be any boolean circuit

restriction. An immediate question is whether we can use a Turing machine obfuscator for constructing constrained PRFs for Turing machines, similar to the circuit-constrained PRF construction of [8]. However, as mentioned above the Turing machine obfuscator constructions are restricted to Turing Machines with bounded size inputs[4]. Thus, we are unable to use the Turning Machine obfuscation scheme in a black box manner and have to introduce new techniques to construct constrained PRFs for unbounded sized inputs.

Our Results: The main result of our work is as follows.

Theorem 1 (informal). *Assuming the existence of secure indistinguishability obfuscators and injective pseudorandom generators, there exists a constrained PRF scheme that is selectively secure.*

Selective Security vs Adaptive Security: Selective security is a security notion where the adversary must specify the 'challenge input' before receiving constrained keys. A stronger notion, called adaptive security, allows the adversary to query for constrained keys before choosing the challenge input. While adaptive security should be the ideal target, achieving adaptive security with only polynomial factor security loss (i.e. without 'complexity leveraging') has been challenging, even for circuit based constrained PRFs. Currently, the best known results for adaptive security either require superpolynomial security loss [13], or work for very restricted functionalities [20], or achieve non-collusion based security [10] or achieve it in the random oracle mode [19].

Moreover, for many applications, it turns out that selective security is sufficient. For example, the widely used punctured programming technique of [25] only requires selectively secure puncturable PRFs. Similarly, as discussed in [1], selectively secure constrained PRFs with unbounded inputs can be used to construct broadcast encryption schemes with unbounded recipients and identity based non-interactive key exchange (ID-NIKE) protocol with no apriori bound on number of parties. Therefore, as a corollary of Theorem 1, we get both these applications using only indistinguishability obfuscation and injective pseudorandom generators. Interestingly, two recent works have shown direct constructions for both these problems using $i\mathcal{O}$. Zhandry [26] showed a broadcast encryption scheme with unbounded recipients, while Khurana et al. [22] showed an ID-NIKE scheme with unbounded number of parties.

We also show how our construction above can be easily adapted to get selectively secure attribute based encryption for Turing machines with unbounded inputs, which illustrates the versatility of our techniques above.

[4] The restriction to bounded length inputs is due to the fact that their iO analysis requires a hybrid over all possible inputs. They absorb this loss by growing the size of the obfuscated program polynomially in the input size using complexity leveraging and a sub-exponential hardness assumption on the underlying circuit iO. Currently, there is no known way to avoid this.

Theorem 2 (informal). *Assuming the existence of secure indistinguishability obfuscators and injective pseudorandom generators, there exists an ABE scheme for Turing machines that is selectively secure.*

Recently, Ananth and Sahai [3] had an exciting result where they show adaptively secure functional encryption for Turing machines with unbounded inputs. While our adaptation is limited to ABE, we believe that the relative simplicity of our construction is an interesting feature. In addition, we were able to apply our tail-hybrid approach to get an end-to-end polynomial time reduction.

1.1 Overview of our constrained PRF construction

To begin, let us consider the simple case of standard PRFs with unbounded inputs. Any PRF (with sufficient input size) can be extended to handle unbounded inputs by first compressing the input using a collision-resistant hash function (CRHF), and then computing the PRF on this hash value. Abusalah et al. [1] showed that by using $di\mathcal{O}$, this approach can be extended to work for constrained PRFs. However, the proof of security relies on the extractability property of $di\mathcal{O}$ in a fundamental way. In particular, this approach will not work if $i\mathcal{O}$ is used instead of $di\mathcal{O}$ because general CRHFs are not '$i\mathcal{O}$-compatible'[5] (see Section 2 for a more detailed discussion on $i\mathcal{O}$-compatibility).

Challenges of a similar nature were addressed in [24] by introducing new tools and techniques that guarantee program functional equivalence at different stages of the proof. Let us review one such tool called *positional accumulators*, and see why it is $i\mathcal{O}$-compatible. A positional accumulator scheme is a cryptographic primitive used to provide a short commitment to a much larger storage. This commitment (also referred to as an accumulation of the storage) has two main features: succinct verifiability (there exists a short proof to prove that an element is present at a particular position) and succinct updatability (using short auxiliary information, the accumulation can be updated to reflect an update to the underlying storage). The scheme also has a setup algorithm which generates the parameters, and can operate in two computationally indistinguishable modes. It can either generate parameters 'normally', or it can be *enforcing* at a particular position p. When parameters are generated in the enforcing mode, the accumulator is *information-theoretically binding* to position p of the underlying storage. This information theoretic enforcing property is what makes it compatible for proofs involving $i\mathcal{O}$.

Returning to our constrained PRF problem, we need a special hash function that can be used with $i\mathcal{O}$. That brings us to the main insight of our work: the KLW positional accumulator can be repurposed to be an $i\mathcal{O}$-friendly hash

[5] Consider the following toy example. Let C_0, C_1 be circuits such that $C_0(x, y) = 0\ \forall (x, y)$ and $C_1(x, y) = 1$ iff $\mathsf{CRHF}(x) = \mathsf{CRHF}(y)$ for $x \neq y$. Now, under the $di\mathcal{O}$ assumption, the obfuscations of C_0 and C_1 are computationally indistinguishable. However, we cannot get the same guarantee by using $i\mathcal{O}$, since the circuits are not functionally identical

function.[6] Besides giving us an $i\mathcal{O}$-friendly hash function, this also puts the input in a data structure that is already suitable for the KLW framework.[7]

Our Construction : We will now sketch out our construction. Our constrained PRF scheme uses a puncturable PRF F with key k. Let $\mathsf{Hash\text{-}Acc}(x)$ represent the accumulation of storage initialized with input $x = x_1 \dots x_n$. The PRF evaluation (in our scheme) is simply $F(k, \mathsf{Hash\text{-}Acc}(x))$.

The interesting part is the description of our constrained keys, and how they can be used to evaluate at an input x. The constrained key for machine M consists of two programs. The first one is an obfuscated circuit which takes an input, and outputs a signature on that input. The second one is an obfuscated circuit which essentially computes the next-step of the Turing machine, and eventually, if it reaches the 'accepting state', it outputs $F(k, \mathsf{Hash\text{-}Acc}(x))$. This circuit also performs additional authenticity checks to prevent illegal inputs - it takes a signature and accumulator as input, verifies the signature and accumulator before computing the next step, and finally updates the accumulator and outputs a signature on the new state and accumulator.

Evaluating the PRF at input x using the constrained key consists of two steps. The first one is the initialization step, where the evaluator first computes $\mathsf{Hash\text{-}Acc}(x)$ and then computes a signature on $\mathsf{Hash\text{-}Acc}(x)$ using the signing program. Then, it iteratively runs the obfuscated next-step circuit (also including $\mathsf{Hash\text{-}Acc}(x)$ as input at each time step) until the circuit either outputs the PRF evaluation, or outputs $\perp$. While this is similar to the KLW message hiding encoding scheme, there are some major differences. One such difference is with regard to accumulation of the input. In KLW, the input is accumulated by the 'honest' encoding party, while in our case, the (possibly corrupt) evaluator generates the accumulation and feeds it at each step of the iteration. As a result, the KLW proof for message-hiding encoding scheme needs to be tailored to fit our setting.

Proof of Security : Recall we are interested in proving selective security, where the adversary sends the challenge input x^* before requesting for constrained keys. Our goal is to replace the (master) PRF key k in all constrained keys with one that is punctured at $\mathsf{acc\text{-}inp}^* = \mathsf{Hash\text{-}Acc}(x^*)$. Once this is done, the security of puncturable PRFs guarantees that the adversary cannot distinguish between $\mathsf{F}(k, \mathsf{acc\text{-}inp}^*)$ and a truly random string. Let us focus our attention on one constrained key query corresponding to machine M, and suppose M runs for t^* steps on input x^* and finally outputs 'reject'.

To replace k with a punctured key, we need to ensure that the obfuscated program for M does not reach the 'accepting state' on inputs with $\mathsf{acc\text{-}inp} =$

[6] More formally, it gives us an $i\mathcal{O}$ friendly universal one way hash function.

[7] We note that the *somewhat statistically binding* hash of [21] has a similar spirit to positional accumulators in that they have statistical binding at a selected position. However, they are not sufficient for our purposes as positional accumulators provide richer semantics such as interleaved reads, writes, and overwrites that are necessary here.

$\mathsf{acc\text{-}inp}^*$. This is done via two main hybrid steps. First, we alter the program so that it does not reach the accepting state within t^* steps on inputs with $\mathsf{acc\text{-}inp} = \mathsf{acc\text{-}inp}^*$. Then, we have the *tail hybrid*, where we ensure that on inputs with $\mathsf{acc\text{-}inp} = \mathsf{acc\text{-}inp}^*$, the program does not reach accepting state even at time steps $t > t^*$. For the first step, we follow the KLW approach, and define a sequence of t^* sub-hybrids, where in the i^{th} hybrid, the obfuscated circuit does not reach accepting state at time steps $t \leq i$ for inputs with $\mathsf{acc\text{-}inp} = \mathsf{acc\text{-}inp}^*$. We use the KLW selective enforcement techniques to show that consecutive hybrids are computationally indistinguishable.

We have a novel approach for handling the tail hybrids Let $T(= 2^\lambda)$ denote the upper bound on the running time of any machine M on any input. In KLW, the tail hybrid step was handled by defining $T - t^*$ intermediate hybrids. If we adopt a similar approach for our construction, it results in an exponential factor security loss, which is undesirable for our application[8]. Our goal would be to overcome this to get an end to end polynomial reduction to $i\mathcal{O}$. Therefore, we propose a modification to our scheme which will allow us to handle the tail hybrid with only a polynomial factor security loss. First, let us call the time step 2^i as the i^{th} *landmark*, while the interval $[2^i, 2^{i+1} - 1]$ is the i^{th} interval. The obfuscated program now takes a PRG seed as input at each time step, and performs some additional checks on the input PRG seed. At time steps just before a landmark, it outputs a new (pseudorandomly generated) PRG seed, which is then used in the next interval. Using standard $i\mathcal{O}$ techniques, we can show that if the program outputs $\perp$ just before a landmark, then we can alter the program indistinguishably so that it outputs $\perp$ at all time steps in the next interval. Since we know that the program outputs $\perp$ at $(\mathsf{acc\text{-}inp}^*, t^* - 1)$, we can ensure that the program outputs $\perp$ for all $(\mathsf{acc\text{-}inp}^*, t)$ such that $t^* \leq t \leq 2t^*$. Proceeding inductively, we can ensure that the program never reaches accepting state if $\mathsf{acc\text{-}inp} = \mathsf{acc\text{-}inp}^*$.

1.2 Attribute Based Encryption for Turing Machines with Unbounded Inputs

We will now describe our ABE scheme for Turing machines with unbounded inputs. Let $\mathcal{PKE}$ be a public key encryption scheme. Our ABE scheme's master secret key is a puncturable PRF key k and the public key is an obfuscated program $\mathsf{Prog\text{-}PK}$ and accumulator parameters. The program $\mathsf{Prog\text{-}PK}$ takes as input a string $\mathsf{acc\text{-}inp}$, computes $r = F(k, \mathsf{acc\text{-}inp})$ and uses r as randomness for $\mathsf{PKE.setup}$. It finally outputs the $\mathcal{PKE}$ public key. To encrypt a message m for attribute x, one must first accumulate the input x, then feed the accumulated input to $\mathsf{Prog\text{-}PK}$ to get a $\mathcal{PKE}$ public key pk, and finally encrypts m using public key pk. The secret keys corresponding to Turing machine M is simply the constrained PRF key for M. This key can be used to compute $F(k, \mathsf{Hash\text{-}Acc}(x))$ if $M(x) = 1$, and therefore can decrypt messages encrypted for x.

[8] An exponential loss in the security proof of randomized encodings in KLW was acceptable because the end goal was indistinguishability obfuscation, which already requires an exponential number of hybrids.

1.3 Paper Organization

We present the required preliminaries in Section 2 and the notions of constrained PRFs for Turing machines in Section 3. The construction of our constrained PRF scheme can be found in Section 4, while our ABE scheme can be found in 5. Due to space constraints, part of our constrained PRF security proof is deferred to the full version of the paper.

2 Preliminaries

2.1 Notations

In this work, we will use the following notations for Turing machines.

Turing machines A Turing machine is a 7-tuple $M = \langle Q, \Sigma_{\text{tape}}, \Sigma_{\text{inp}}, \delta, q_0, q_{\text{ac}}, q_{\text{rej}} \rangle$ with the following semantics:

- Q is the set of states with start state q_0, accept state q_{ac} and reject state q_{rej}.
- Σ_{inp} is the set of inputs symbols
- Σ_{tape} is the set of tape symbols. We will assume $\Sigma_{\text{inp}} \subset \Sigma_{\text{tape}}$ and there is a special blank symbol '$\sqcup$' $\in \Sigma_{\text{tape}} \setminus \Sigma_{\text{inp}}$.
- $\delta : Q \times \Sigma_{\text{tape}} \to Q \times \Sigma_{\text{tape}} \times \{+1, -1\}$ is the transition function.

2.2 Obfuscation

We recall the definition of indistinguishability obfuscation from [15, 25].

Definition 1. *(Indistinguishability Obfuscation) Let $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda \in \mathbb{N}}$ be a family of polynomial-size circuits. Let $i\mathcal{O}$ be a uniform PPT algorithm that takes as input the security parameter λ, a circuit $C \in \mathcal{C}_\lambda$ and outputs a circuit C'. $i\mathcal{O}$ is called an indistinguishability obfuscator for a circuit class $\{\mathcal{C}_\lambda\}$ if it satisfies the following conditions:*

- *(Preserving Functionality) For all security parameters $\lambda \in \mathbb{N}$, for all $C \in \mathcal{C}_\lambda$, for all inputs x, we have that $C'(x) = C(x)$ where $C' \leftarrow i\mathcal{O}(1^\lambda, C)$.*
- *(Indistinguishability of Obfuscation) For any (not necessarily uniform) PPT distinguisher $\mathcal{B} = (Samp, \mathcal{D})$, there exists a negligible function $negl(\cdot)$ such that the following holds: if for all security parameters $\lambda \in \mathbb{N}, \Pr[\forall x, C_0(x) = C_1(x) : (C_0; C_1; \sigma) \leftarrow Samp(1^\lambda)] > 1 - negl(\lambda)$, then*

$$|\Pr[\mathcal{D}(\sigma, i\mathcal{O}(1^\lambda, C_0)) = 1 : (C_0; C_1; \sigma) \leftarrow Samp(1^\lambda)] -$$
$$\Pr[\mathcal{D}(\sigma, i\mathcal{O}(1^\lambda, C_1)) = 1 : (C_0; C_1; \sigma) \leftarrow Samp(1^\lambda)]| \leq negl(\lambda).$$

In a recent work, [15] showed how indistinguishability obfuscators can be constructed for the circuit class $P/poly$. We remark that $(Samp, \mathcal{D})$ are two algorithms that pass state, which can be viewed equivalently as a single stateful algorithm $\mathcal{B}$. In our proofs we employ the latter approach, although here we state the definition as it appears in prior work.

2.3 iO-Compatible Primitives

In this section, we define extensions of some cryptographic primitives that makes them 'compatible' with indistinguishability obfuscation [9]. All of the primitives described here can be constructed from $i\mathcal{O}$ and one way functions. Their constructions can be found in [24].

Splittable Signatures A splittable signature scheme is a normal deterministic signature scheme, augmented by some additional algorithms and properties that we require for our application. Such a signature scheme has four different kinds of signing/verification key pairs. First, we have the standard signing/verification key pairs, where the signing key can compute signatures on any message, and the verification key can verify signatures corresponding to any message. Next, we have 'all-but-one' signing/verification keys. These keys, which correspond to a special message m^*, work for all messages except m^*; that is, the signing key can sign all messages except m^*, and the verification key can verify signatures for all messages except m^* (it does not accept any signature corresponding to m^*). Third, we have 'one' signing/verification keys. These keys correspond to a special message m', and can only be used to sign/verify signatures for m'. For all other messages, the verification algorithm does not accept any signatures. Finally, we have the rejection verification key which does not accept any signatures. The setup algorithm outputs a standard signing/verification key together with a rejection verification key, while a 'splitting' algorithm uses a standard signing key to generate 'all-but-one' and 'one' signing/verification keys.

At a high level, we require the following security properties. First, the standard verification key and the rejection verification key must be computationally indistinguishable. Intuitively, this is possible because an adversary does not have any secret key or signatures. Next, we require that if an adversary is given an 'all-but-one' secret key for message m^*, then he/she cannot distinguish between a standard verification key and an 'all-but-one' verification key corresponding to m^*. We also have a similar property for the 'one' keys. No PPT adversary, given a 'one' signing key, can distinguish between a standard verification key and a 'one' verification key. Finally, we have the 'splittability' property, which states that the keys generated by splitting one signing key are indistinguishable from the case where the 'all-but-one' key pair and the 'one' key pair are generated from different signing keys.

We will now formally describe the syntax and correctness/security properties of splittable signatures.

Syntax: A splittable signature scheme $\mathcal{S}$ for message space $\mathcal{M}$ consists of the following algorithms:

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

[9] In the full version of our paper, we describe a toy example to illustrate why we need to extend/modify certain primitives in order to use them with $i\mathcal{O}$.

$\mathsf{Sign\text{-}Spl}(\mathrm{SK}, m)$ The signing algorithm is a deterministic algorithm that takes as input a signing key SK and a message $m \in \mathcal{M}$. It outputs a signature σ.

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

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

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

Correctness Let $m^* \in \mathcal{M}$ be any message. Let $(\mathrm{SK}, \mathrm{VK}, \mathrm{VK}_{\mathrm{rej}}) \leftarrow \mathsf{Setup\text{-}Spl}(1^\lambda)$ and $(\sigma_{\mathrm{one}}, \mathrm{VK}_{\mathrm{one}}, \mathrm{SK}_{\mathrm{abo}}, \mathrm{VK}_{\mathrm{abo}}) \leftarrow \mathsf{Split}(\mathrm{SK}, m^*)$. Then, we require the following correctness properties:

1. For all $m \in \mathcal{M}$, $\mathsf{Verify\text{-}Spl}(\mathrm{VK}, m, \mathsf{Sign\text{-}Spl}(\mathrm{SK}, m)) = 1$.
2. For all $m \in \mathcal{M}, m \neq m^*$, $\mathsf{Sign\text{-}Spl}(\mathrm{SK}, m) = \mathsf{Sign\text{-}Spl\text{-}abo}(\mathrm{SK}_{\mathrm{abo}}, m)$.
3. For all σ, $\mathsf{Verify\text{-}Spl}(\mathrm{VK}_{\mathrm{one}}, m^*, \sigma) = \mathsf{Verify\text{-}Spl}(\mathrm{VK}, m^*, \sigma)$.
4. For all $m \neq m^*$ and σ, $\mathsf{Verify\text{-}Spl}(\mathrm{VK}, m, \sigma) = \mathsf{Verify\text{-}Spl}(\mathrm{VK}_{\mathrm{abo}}, m, \sigma)$.
5. For all $m \neq m^*$ and σ, $\mathsf{Verify\text{-}Spl}(\mathrm{VK}_{\mathrm{one}}, m, \sigma) = 0$.
6. For all σ, $\mathsf{Verify\text{-}Spl}(\mathrm{VK}_{\mathrm{abo}}, m^*, \sigma) = 0$.
7. For all σ and all $m \in \mathcal{M}$, $\mathsf{Verify\text{-}Spl}(\mathrm{VK}_{\mathrm{rej}}, m, \sigma) = 0$.

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

Definition 2 ($\mathrm{VK}_{\mathrm{rej}}$ indistinguishability).
A splittable signature scheme $\mathcal{S}$ is said to be $\mathrm{VK}_{\mathrm{rej}}$ indistinguishable if any PPT adversary $\mathcal{A}$ has negligible advantage in the following security game:

$\mathsf{Exp}\text{-}\mathrm{VK}_{\mathrm{rej}}(1^\lambda, \mathcal{S}, \mathcal{A})$*:*

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

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

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

Definition 3 ($\mathrm{VK}_{\mathrm{one}}$ indistinguishability). *A splittable signature scheme $\mathcal{S}$ is said to be $\mathrm{VK}_{\mathrm{one}}$ indistinguishable if any PPT adversary $\mathcal{A}$ has negligible advantage in the following security game:*

$\mathsf{Exp}\text{-}\mathrm{VK}_{\mathrm{one}}(1^\lambda, \mathcal{S}, \mathcal{A})$*:*

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

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

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

Definition 4 ($\mathrm{VK}_{\mathrm{abo}}$ indistinguishability). *A splittable signature scheme $\mathcal{S}$ is said to be $\mathrm{VK}_{\mathrm{abo}}$ indistinguishable if any PPT adversary $\mathcal{A}$ has negligible advantage in the following security game:*

$\mathsf{Exp\text{-}VK}_{\mathrm{abo}}(1^\lambda, \mathcal{S}, \mathcal{A})$*:*

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

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

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

Definition 5 (Splitting indistinguishability). *A splittable signature scheme $\mathcal{S}$ is said to be splitting indistinguishable if any PPT adversary $\mathcal{A}$ has negligible advantage in the following security game:*

$\mathsf{Exp\text{-}Spl}(1^\lambda, \mathcal{S}, \mathcal{A})$*:*

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

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

In the game above, $\mathcal{A}$ is either given a system of $\sigma_{\mathrm{one}}, \mathrm{VK}_{\mathrm{one}}, \mathrm{SK}_{\mathrm{abo}}, \mathrm{VK}_{\mathrm{abo}}$ generated together by one call of $\mathsf{Setup\text{-}Spl}$ or a "split" system of $(\sigma'_{\mathrm{one}}, \mathrm{VK}'_{\mathrm{one}}, \mathrm{SK}_{\mathrm{abo}}, \mathrm{VK}_{\mathrm{abo}})$ where the all but one keys are generated separately

from the signature and key for the one message m^*. Since the correctness conditions do not link the behaviors for the all but one keys and the one message values, this split generation is not detectable by testing verification for the σ_{one} that $\mathcal{A}$ receives or for any signatures that $\mathcal{A}$ creates honestly by signing with SK_{abo}.

Positional Accumulators An accumulator can be seen as a special hash function mapping unbounded[10] length strings to fixed length strings. It has two additional properties: succinct verifiability and succinct updatability. Let $\mathsf{Hash\text{-}Acc}(\cdot)$ be the hash function mapping $x = x_1 \dots x_n$ to y. Then, succinct verifiability means that there exists a 'short' proof π to prove that bit x_i is present at the i^{th} position of x. Note that this verification only requires the hash value y and the short proof π. Succinct updatability means that given y, a bit x'_i, position i and some 'short' auxiliary information, one can update y to obtain $y' = \mathsf{Hash\text{-}Acc}(x_1 \dots x'_i \dots x_n)$. We will refer to y as the tape, and x_i the symbol written at position i.

The notion of accumulators is not sufficient for using with $i\mathcal{O}$, and we need a stronger primitive called *positional accumulators* that is iO-compatible. In a positional accumulator, we have three different setup modes. The first one is the standard setup which outputs public parameters and the initial accumulation corresponding to the empty tape. Next, we have the read-enforced setup mode. In this mode, the algorithm takes as input a sequence of k pairs $(\mathsf{sym}_i, \mathsf{pos}_i)$ which represent the first k symbols written and their positions. It also takes as input the enforcing position pos, and outputs public parameters and an accumulation of the empty tape. As the name might suggest, this mode is read enforcing at position pos - if the first k symbols written are $(\mathsf{sym}_1, \dots, \mathsf{sym}_k)$, and their write positions are $(\mathsf{pos}_1, \dots, \mathsf{pos}_k)$, then there *exists* exactly one opening for position pos : the correct symbol written at pos. Similarly, we have a write-enforcing setup which takes as input k (symbol, position) pairs $\{(\mathsf{sym}_i, \mathsf{pos}_i)\}_{i \le k}$ representing the first k writes, and outputs public parameters and an accumulation of the empty tape. The write-enforcing property states that if $(\mathsf{sym}_i, \mathsf{pos}_i)$ are the first k writes, and acc_{k-1} is the correct accumulation after the first $k-1$ writes, then there is a unique accumulation after the k^{th} write (irrespective of the auxiliary string). Note that both the read and write enforcing properties are information theoretic. This is important when we are using these primitives with indistinguishability obfuscation.

For security, we require that the different setup modes are computationally indistinguishable. We will now give a formal description of the syntax and properties. A positional accumulator for message space $\mathcal{M}_\lambda$ consists of the following algorithms.

- $\mathsf{Setup\text{-}Acc}(1^\lambda, T) \to (\text{PP}, \mathsf{acc}_0, \text{STORE}_0)$ The setup algorithm takes as input a security parameter λ in unary and an integer T in binary representing the

[10] Unbounded, but polynomial in the security parameter

maximum number of values that can stored. It outputs public parameters PP, an initial accumulator value acc_0, and an initial storage value STORE_0.

- $\mathsf{Setup\text{-}Acc\text{-}Enf\text{-}Read}(1^\lambda, T, (m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k), \text{INDEX}^*) \rightarrow (\text{PP}, \mathsf{acc}_0, \text{STORE}_0)$: The setup enforce read algorithm takes as input a security parameter λ in unary, an integer T in binary representing the maximum number of values that can be stored, and a sequence of symbol, index pairs, where each index is between 0 and $T-1$, and an additional INDEX^* also between 0 and $T-1$. It outputs public parameters PP, an initial accumulator value acc_0, and an initial storage value STORE_0.
- $\mathsf{Setup\text{-}Acc\text{-}Enf\text{-}Write}(1^\lambda, T, (m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k)) \rightarrow (\text{PP}, \mathsf{acc}_0, \text{STORE}_0)$: The setup enforce write algorithm takes as input a security parameter λ in unary, an integer T in binary representing the maximum number of values that can be stored, and a sequence of symbol, index pairs, where each index is between 0 and $T-1$. It outputs public parameters PP, an initial accumulator value acc_0, and an initial storage value STORE_0.
- $\mathsf{Prep\text{-}Read}(\text{PP}, \text{STORE}_{in}, \text{INDEX}) \rightarrow (m, \pi)$: The prep-read algorithm takes as input the public parameters PP, a storage value STORE_{in}, and an index between 0 and $T-1$. It outputs a symbol m (that can be ϵ) and a value π.
- $\mathsf{Prep\text{-}Write}(\text{PP}, \text{STORE}_{in}, \text{INDEX}) \rightarrow aux$: The prep-write algorithm takes as input the public parameters PP, a storage value STORE_{in}, and an index between 0 and $T-1$. It outputs an auxiliary value aux.
- $\mathsf{Verify\text{-}Read}(\text{PP}, \mathsf{acc}_{in}, m_{read}, \text{INDEX}, \pi) \rightarrow \{True, False\}$: The verify-read algorithm takes as input the public parameters PP, an accumulator value acc_{in}, a symbol, m_{read}, an index between 0 and $T-1$, and a value π. It outputs $True$ or $False$.
- $\mathsf{Write\text{-}Store}(\text{PP}, \text{STORE}_{in}, \text{INDEX}, m) \rightarrow \text{STORE}_{out}$: The write-store algorithm takes in the public parameters, a storage value STORE_{in}, an index between 0 and $T-1$, and a symbol m. It outputs a storage value STORE_{out}.
- $\mathsf{Update}(\text{PP}, \mathsf{acc}_{in}, m_{write}, \text{INDEX}, aux) \rightarrow \mathsf{acc}_{out}$ or $Reject$: The update algorithm takes in the public parameters PP, an accumulator value acc_{in}, a symbol m_{write}, and index between 0 and $T-1$, and an auxiliary value aux. It outputs an accumulator value acc_{out} or $Reject$.

In general we will think of the $\mathsf{Setup\text{-}Acc}$ algorithm as being randomized and the other algorithms as being deterministic. However, one could consider non-deterministic variants.

Correctness We consider any sequence $(m_1, \text{INDEX}_1), \ldots, (m_k, \text{INDEX}_k)$ of symbols $m_1, \ldots, m_k$ and indices $\text{INDEX}_1, \ldots, \text{INDEX}_k$ each between 0 and $T-1$. We fix any $\text{PP}, \mathsf{acc}_0, \text{STORE}_0 \leftarrow \mathsf{Setup\text{-}Acc}(1^\lambda, T)$. For j from 1 to k, we define STORE_j iteratively as $\text{STORE}_j := \mathsf{Write\text{-}Store}(\text{PP}, \text{STORE}_{j-1}, \text{INDEX}_j, m_j)$. We similarly define aux_j and acc_j iteratively as $aux_j := \mathsf{Prep\text{-}Write}(\text{PP}, \text{STORE}_{j-1}, \text{INDEX}_j)$ and $\mathsf{acc}_j := Update(\text{PP}, \mathsf{acc}_{j-1}, m_j, \text{INDEX}_j, aux_j)$. Note that the algorithms other than $\mathsf{Setup\text{-}Acc}$ are deterministic, so these definitions fix precise values, not random values (conditioned on the fixed starting values $\text{PP}, \mathsf{acc}_0, \text{STORE}_0$).

We require the following correctness properties:

1. For every INDEX between 0 and $T-1$, $\mathsf{Prep\text{-}Read}(\mathrm{PP}, \mathrm{STORE}_k, \mathrm{INDEX})$ returns m_i, π, where i is the largest value in $[k]$ such that $\mathrm{INDEX}_i = \mathrm{INDEX}$. If no such value exists, then $m_i = \epsilon$.
2. For any INDEX, let $(m, \pi) \leftarrow \mathsf{Prep\text{-}Read}(\mathrm{PP}, \mathrm{STORE}_k, \mathrm{INDEX})$. Then $\mathsf{Verify\text{-}Read}(\mathrm{PP}, \mathsf{acc}_k, m, \mathrm{INDEX}, \pi) = True$.

Remarks on Efficiency In our construction, all algorithms will run in time polynomial in their input sizes. More precisely, Setup-Acc will be polynomial in λ and $\log(T)$. Also, accumulator and π values should have size polynomial in λ and $\log(T)$, so Verify-Read and Update will also run in time polynomial in λ and $\log(T)$. Storage values will have size polynomial in the number of values stored so far. Write-Store, Prep-Read, and Prep-Write will run in time polynomial in λ and T.

Security Let Acc = (Setup-Acc, Setup-Acc-Enf-Read, Setup-Acc-Enf-Write, Prep-Read, Prep-Write, Verify-Read, Write-Store, Update) be a positional accumulator for symbol set $\mathcal{M}$. We require Acc to satisfy the following notions of security.

Definition 6 (Indistinguishability of Read Setup). *A positional accumulator* Acc *is said to satisfy indistinguishability of read setup if any PPT adversary $\mathcal{A}$'s advantage in the security game* $\mathsf{Exp\text{-}Setup\text{-}Acc}(1^\lambda, \mathsf{Acc}, \mathcal{A})$ *is at most negligible in λ, where* Exp-Setup-Acc *is defined as follows.*

Exp-Setup-Acc(1^λ, Acc, $\mathcal{A}$)

1. *Adversary chooses a bound $T \in \Theta(2^\lambda)$ and sends it to challenger.*
2. *$\mathcal{A}$ sends k messages $m_1, \ldots, m_k \in \mathcal{M}$ and k indices $\mathrm{INDEX}_1, \ldots, \mathrm{INDEX}_k \in \{0, \ldots, T-1\}$ to the challenger.*
3. *The challenger chooses a bit b. If $b = 0$, the challenger outputs $(\mathrm{PP}, \mathsf{acc}_0, \mathrm{STORE}_0) \leftarrow \mathsf{Setup\text{-}Acc}(1^\lambda, T)$. Else, it outputs $(\mathrm{PP}, \mathsf{acc}_0, \mathrm{STORE}_0) \leftarrow$ Setup-Acc-Enf-Read $(1^\lambda, T, (m_1, \mathrm{INDEX}_1), \ldots, (m_k, \mathrm{INDEX}_k))$.*
4. *$\mathcal{A}$ sends a bit b'.*

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

Definition 7 (Indistinguishability of Write Setup). *A positional accumulator* Acc *is said to satisfy indistinguishability of write setup if any PPT adversary $\mathcal{A}$'s advantage in the security game* $\mathsf{Exp\text{-}Setup\text{-}Acc}(1^\lambda, \mathsf{Acc}, \mathcal{A})$ *is at most negligible in λ, where* Exp-Setup-Acc *is defined as follows.*

Exp-Setup-Acc(1^λ, Acc, $\mathcal{A}$)

1. *Adversary chooses a bound $T \in \Theta(2^\lambda)$ and sends it to challenger.*
2. *$\mathcal{A}$ sends k messages $m_1, \ldots, m_k \in \mathcal{M}$ and k indices $\mathrm{INDEX}_1, \ldots, \mathrm{INDEX}_k \in \{0, \ldots, T-1\}$ to the challenger.*

3. *The challenger chooses a bit b. If $b = 0$, the challenger outputs* $(\mathrm{PP}, \mathsf{acc}_0, \mathrm{STORE}_0) \leftarrow \mathsf{Setup\text{-}Acc}(1^\lambda, T)$. *Else, it outputs* $(\mathrm{PP}, \mathsf{acc}_0, \mathrm{STORE}_0) \leftarrow \mathsf{Setup\text{-}Acc\text{-}Enf\text{-}Write}$ $(1^\lambda, T, (m_1, \mathrm{INDEX}_1), \ldots, (m_k, \mathrm{INDEX}_k))$.
4. $\mathcal{A}$ *sends a bit* b'.

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

Definition 8 (Read Enforcing). *Consider any* $\lambda \in \mathbb{N}$, $T \in \Theta(2^\lambda)$, $m_1, \ldots, m_k \in \mathcal{M}$, $\mathrm{INDEX}_1, \ldots, \mathrm{INDEX}_k \in \{0, \ldots, T-1\}$ *and any* $\mathrm{INDEX}^* \in \{0, \ldots, T-1\}$.

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

Definition 9 (Write Enforcing). *Consider any* $\lambda \in \mathbb{N}$, $T \in \Theta(2^\lambda)$, $m_1, \ldots, m_k \in \mathcal{M}$, $\mathrm{INDEX}_1, \ldots, \mathrm{INDEX}_k \in \{0, \ldots, T-1\}$. *Let* $(\mathrm{PP}, \mathsf{acc}_0, \mathrm{STORE}_0) \leftarrow \mathsf{Setup\text{-}Acc\text{-}Enf\text{-}Write}(1^\lambda, T, (m_1, \mathrm{INDEX}_1), \ldots, (m_k, \mathrm{INDEX}_k))$. *For j from 1 to k, we define* STORE_j *iteratively as* $\mathrm{STORE}_j := \mathsf{Write\text{-}Store}(\mathrm{PP}, \mathrm{STORE}_{j-1}, \mathrm{INDEX}_j, m_j)$. *We similarly define* aux_j *and* acc_j *iteratively as* $aux_j := \mathsf{Prep\text{-}Write}(\mathrm{PP}, \mathrm{STORE}_{j-1}, \mathrm{INDEX}_j)$ *and* $\mathsf{acc}_j := Update(\mathrm{PP}, \mathsf{acc}_{j-1}, m_j, \mathrm{INDEX}_j, aux_j)$. Acc *is said to be* write enforcing *if* $\mathsf{Update}(\mathrm{PP}, \mathsf{acc}_{k-1}, m_k, \mathrm{INDEX}_k, aux) = \mathsf{acc}_{out} \neq Reject$, *for any aux, then* $\mathsf{acc}_{out} = \mathsf{acc}_k$. *Note that this is an information-theoretic property: we are requiring that an aux value producing an accumulated value other than* acc_k *or* $Reject$ *does not exist.*

Iterators In this section, we define the notion of *cryptographic iterators*. A cryptographic iterator essentially consists of a small state that is updated in an iterative fashion as messages are received. An update to apply a new message given current state is performed via some public parameters.

Since states will remain relatively small regardless of the number of messages that have been iteratively applied, there will in general be many sequences of messages that can lead to the same state. However, our security requirement will capture that the normal public parameters are computationally indistinguishable from specially constructed "enforcing" parameters that ensure that a particular *single* state can be only be obtained as an output as an update to precisely one other state, message pair. Note that this enforcement is a very localized property to a particular state, and hence can be achieved information-theoretically when we fix ahead of time where exactly we want this enforcement to be.

Syntax Let ℓ be any polynomial. An iterator $\mathcal{I}$ with message space $\mathcal{M}_\lambda = \{0,1\}^{\ell(\lambda)}$ and state space $\mathcal{S}_\lambda$ consists of three algorithms - $\mathsf{Setup\text{-}Itr}$, $\mathsf{Setup\text{-}Itr\text{-}Enf}$ and $\mathsf{Iterate}$ defined below.

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

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

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

For simplicity of notation, we will drop the dependence of ℓ on λ. Also, for any integer $k \leq T$, we will use the notation $\mathsf{Iterate}^k(\mathrm{PP}, v_0, (m_1, \ldots, m_k))$ to denote $\mathsf{Iterate}(\mathrm{PP}, v_{k-1}, m_k)$, where $v_j = \mathsf{Iterate}(\mathrm{PP}, v_{j-1}, m_j)$ for all $1 \leq j \leq k-1$.

Security Let $\mathcal{I} = (\mathsf{Setup\text{-}Itr}, \mathsf{Setup\text{-}Itr\text{-}Enf}, \mathsf{Iterate})$ be an interator with message space $\{0,1\}^\ell$ and state space $\mathcal{S}_\lambda$. We require the following notions of security.

Definition 10 (Indistinguishability of Setup). *An iterator $\mathcal{I}$ is said to satisfy indistinguishability of Setup phase if any PPT adversary $\mathcal{A}$'s advantage in the security game $\mathsf{Exp\text{-}Setup\text{-}Itr}(1^\lambda, \mathcal{I}, \mathcal{A})$ at most is negligible in λ, where $\mathsf{Exp\text{-}Setup\text{-}Itr}$ is defined as follows.*

$\mathsf{Exp\text{-}Setup\text{-}Itr}(1^\lambda, \mathcal{I}, \mathcal{A})$

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

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

Definition 11 (Enforcing). *Consider any $\lambda \in \mathbb{N}$, $T \in \Theta(2^\lambda)$, $k < T$ and $m_1, \ldots, m_k \in \{0,1\}^\ell$. Let $(\mathrm{PP}, v_0) \leftarrow \mathsf{Setup\text{-}Itr\text{-}Enf}(1^\lambda, T, \mathbf{m} = (m_1, \ldots, m_k))$ and $v_j = \mathsf{Iterate}^j(\mathrm{PP}, v_0, (m_1, \ldots, m_j))$ for all $1 \leq j \leq k$. Then, $\mathcal{I} = (\mathsf{Setup\text{-}Itr}, \mathsf{Setup\text{-}Itr\text{-}Enf}, \mathsf{Iterate})$ is said to be* enforcing *if*

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

Note that this is an information-theoretic property.

2.4 Attribute Based Encryption

An ABE scheme where policies are represented by Turing machines comprises of the following four algorithms ($\mathsf{ABE.setup}, \mathsf{ABE.enc}, \mathsf{ABE.keygen}, \mathsf{ABE.dec}$):

- $\mathsf{ABE.setup}(1^\lambda) \rightarrow (\mathrm{PK}_{\mathsf{ABE}}, \mathrm{MSK}_{\mathsf{ABE}})$: The setup algorithm takes as input the security parameter λ and outputs the public key $\mathrm{PK}_{\mathsf{ABE}}$ and the master secret key $\mathrm{MSK}_{\mathsf{ABE}}$
- $\mathsf{ABE.enc}(m, x, \mathrm{PK}_{\mathsf{ABE}}) \rightarrow \mathsf{ct}$: The encryption algorithm takes as input the message m, the attribute string x of unbounded length and the public key $\mathrm{PK}_{\mathsf{ABE}}$ and it outputs the corresponding ciphertext ct_x specific to the attribute string.
- $\mathsf{ABE.keygen}(\mathrm{MSK}_{\mathsf{ABE}}, M) \rightarrow \mathrm{SK}\{M\}$: The key generation algorithm takes as input $\mathrm{MSK}_{\mathsf{ABE}}$ and a Turing machine M and outputs the secret key $\mathrm{SK}\{M\}$ specific to M
- $\mathsf{ABE.dec}(\mathrm{SK}\{M\}, \mathsf{ct}) \rightarrow m$ or $\perp$: The decryption algorithm takes in $\mathrm{SK}\{M\}$ and ciphertext ct and outputs either a message m or $\perp$.

The *correctness* of the scheme guarantees that if $\mathsf{ABE.enc}(m, x, \mathrm{PK}_{\mathsf{ABE}}) \rightarrow \mathsf{ct}_x$ and $\mathsf{ABE.keygen}(\mathrm{MSK}_{\mathsf{ABE}}, M) \rightarrow \mathrm{SK}\{M\}$ then $\mathsf{ABE.dec}(\mathrm{SK}\{M\}, \mathsf{ct}_x) \rightarrow m$.

2.5 Selective Security

Consider the following experiment between a challenger $\mathcal{C}$ and a stateful adversary $\mathcal{A}$:

- **Setup Phase**: $\mathcal{A}$ sends the challenge attribute string x^* of his choice to $\mathcal{C}$. $\mathcal{C}$ runs the $\mathsf{ABE.setup}(1^\lambda)$ and sends across $\mathrm{PK}_{\mathsf{ABE}}$ to $\mathcal{A}$.
- **Pre-Challenge Query Phase**: $\mathcal{A}$ gets to query for secret keys corresponding to Turing machines. For each query M such that $M(x^*) = 0$, the challenger computes $\mathrm{SK}\{M\} \leftarrow \mathsf{ABE.keygen}(\mathrm{MSK}_{\mathsf{ABE}}, .)$ and sends it to $\mathcal{A}$.
- **Challenge Phase** $\mathcal{A}$ sends two messages m_0, m_1 with $|m_0| = |m_1|$, the challenger chooses bit b uniformly at random and outputs $\mathsf{ct}^* = \mathsf{ABE.enc}(m_b, x^*, \mathrm{PK}_{\mathsf{ABE}})$.
- **Post-Challenge Query Phase**: This is identical to the Pre-Challenge Phase.
- **Guess**: Finally, $\mathcal{A}$ sends its guess b' and wins if $b = b'$.

The advantage of $\mathcal{A}$, $\mathsf{Adv}_{\mathcal{A}}^{ABE}(\lambda)$ in the above experiment is defined to be $|\Pr[b' = b] - \frac{1}{2}|$.

Definition 12. *An ABE scheme is said to be* selectively secure *if for all PPT adversaries $\mathcal{A}$, the advantage $\mathsf{Adv}_{\mathcal{A}}^{ABE}(\lambda)$ is a negligible function in λ.*

3 Constrained Pseudorandom Functions for Turing Machines

The notion of constrained pseudorandom functions was introduced in the concurrent works of [7, 9, 23]. Informally, a constrained PRF extends the notion of standard PRFs, enabling the master PRF key holder to compute 'constrained keys' that allow PRF evaluations on certain inputs, while the PRF evaluation on remaining inputs 'looks' random. In the above mentioned works, these constraints could only handle bounded length inputs. In order to allow unbounded inputs, we need to ensure that the constrained keys correspond to polynomial time Turing Machines. A formal definition is as follows.

Let $\mathcal{M}_\lambda$ be a family of Turing machines with (worst case) running time bounded by 2^λ. Let $\mathcal{K}$ denote the key space, $\mathcal{X}$ the input domain and $\mathcal{Y}$ the range space. A pseudorandom $\mathsf{PRF} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is said to be *constrained* with respect to the Turing machine family $\mathcal{M}_\lambda$ if there is an additional key space $\mathcal{K}_c$, and three algorithms $\mathsf{PRF.setup}$, $\mathsf{PRF.constrain}$ and $\mathsf{PRF.eval}$ as follows:

- $\mathsf{PRF.setup}(1^\lambda)$ is a PPT algorithm that takes the security parameter λ as input and outputs a key $K \in \mathcal{K}$.
- $\mathsf{PRF.constrain}(K, M)$ is a PPT algorithm that takes as input a PRF key $K \in \mathcal{K}$ and a Turing machine $M \in \mathcal{M}_\lambda$ and outputs a constrained key $K\{M\} \in \mathcal{K}_c$.
- $\mathsf{PRF.eval}(K\{M\}, x)$ is a deterministic polynomial time algorithm that takes as input a constrained key $K\{M\} \in \mathcal{K}_c$ and $x \in \mathcal{X}$ and outputs an element $y \in \mathcal{Y}$. Let $K\{M\}$ be the output of $\mathsf{PRF.constrain}(K, M)$. For correctness, we require the following:

$$\mathsf{PRF.eval}(K\{M\}, x) = F(K, x) \text{ if } M(x) = 1.$$

For simplicity of notation, we will use $\mathsf{PRF}(K\{M\}, x)$ to denote $\mathsf{PRF.eval}(K\{M\}, x)$.

3.1 Security of Constrained Pseudorandom Functions

Intuitively, we require that even after obtaining several constrained keys, no polynomial time adversary can distinguish a truly random string from the PRF evaluation at a point not accepted by the queried Turing machines. In this work, we achieve a weaker notion of security called *selective security*, which is formalized by the following security game between a challenger and an adversary Att.

Let $\mathsf{PRF} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ be a constrained PRF with respect to a Turing machine family $\mathcal{M}$. The security game consists of three phases.

Setup Phase The adversary sends the challenge input x^*. The challenger chooses a random key $K \leftarrow \mathcal{K}$ and a random bit $b \leftarrow \{0, 1\}$. If $b = 0$, the challenger outputs $\mathsf{PRF}(K, x^*)$. Else, the challenger outputs a random element $y \leftarrow \mathcal{Y}$.

Query Phase In this phase, Att is allowed to ask for the following queries:

- **Evaluation Query** Att sends $x \in \mathcal{X}$, and receives $\mathsf{PRF}(K, x)$.
- **Key Query** Att sends a Turing machine $M \in \mathcal{M}$ such that $M(x^*) = 0$, and receives $\mathsf{PRF.constrain}(K, M)$.

Guess Finally, A outputs a guess b' of b.

A wins if $b = b'$ and the advantage of Att is defined to be $\mathsf{Adv}_{\mathsf{Att}}(\lambda) = \left| \Pr[\mathsf{Att} \text{ wins}] - 1/2 \right|$.

Definition 13. *The PRF* PRF *is a secure constrained PRF with respect to* $\mathcal{M}$ *if for all PPT adversaries* A $\mathsf{Adv}_{\mathsf{Att}}(\lambda)$ *is negligible in* λ.

3.2 Puncturable Pseudorandom Functions

A special class of constrained PRFs, called *puncturable PRFs*, was introduced in the work of [25]. In a puncturable PRF, the constrained key queries correspond to points in the input domain, and the constrained key is one that allows PRF evaluations at all points except the punctured point.

Formally, a PRF $\mathsf{F} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is a puncturable pseudorandom function if there is an additional key space $\mathcal{K}_p$ and three polynomial time algorithms $\mathsf{F.setup}$, $\mathsf{F.eval}$ and $\mathsf{F.puncture}$ as follows:

- $\mathsf{F.setup}(1^\lambda)$ is a randomized algorithm that takes the security parameter λ as input and outputs a description of the key space $\mathcal{K}$, the punctured key space $\mathcal{K}_p$ and the PRF F.
- $\mathsf{F.puncture}(K, x)$ is a randomized algorithm that takes as input a PRF key $K \in \mathcal{K}$ and $x \in \mathcal{X}$, and outputs a key $K_x \in \mathcal{K}_p$.
- $\mathsf{F.eval}(K_x, x')$ is a deterministic algorithm that takes as input a punctured key $K_x \in \mathcal{K}_p$ and $x' \in \mathcal{X}$. Let $K \in \mathcal{K}$, $x \in \mathcal{X}$ and $K_x \leftarrow \mathsf{F.puncture}(K, x)$. For correctness, we need the following property:

$$\mathsf{F.eval}(K_x, x') = \begin{cases} \mathsf{F}(K, x') & \text{if } x \neq x' \\ \perp & \text{otherwise} \end{cases}$$

The selective security notion is analogous to the security notion of constrained PRFs.

4 Construction

A high level description of our construction: Our constrained PRF construction uses a puncturable PRF F as the base pseudorandom function. The setup algorithm chooses a puncturable PRF key K together with the public parameters of the accumulator and an accumulation of the empty tape (it also outputs additional parameters for the authenticity checks described in the next paragraph).

To evaluate the constrained PRF on input x, one first accumulates the input x. Let y denote this accumulation. The PRF evaluation is $F(K, y)$.

Next, let us consider the constrained key for Turing machine M. The major component of this key is an obfuscated program Prog. At a very high level, this program evaluates the next-step circuit of M. Its main inputs are the time step t, hash y of the input and the symbol, state, position of TM used at step t. Using the state and symbol, it computes the next state and the symbol to be written. If the state is accepting, it outputs $F(K, y)$, else it outputs the next state and symbol. However, this is clearly not enough, since the adversary could pass illegal states and symbols as inputs. So the program first performs some additional authenticity checks then evaluates the next state, symbol , and finally outputs authentication required for the next step evaluation. These authenticity checks are imposed via the accumulator, signature scheme and iterator. For these checks, Prog takes additional inputs: accumulation of the current tape acc, proof π that the input symbol is the correct symbol at the tape-head position, auxiliary string aux to update the accumulation, iterated value and signature σ. The iterated value and the signature together ensure that the correct state and accumulated value is input at each step, while the accumulation ensures that the adversary cannot send a wrong symbol. Finally, to perform the 'tail-cutting', the program requires an additional input seed. The first and last step of the program are for checking the validity of seed, and to output the new seed if required. The constrained key also has another program Init-Sign which is used to sign the accumulation of the input. In the end, if all the checks go through, the final output will be the PRF evaluation using the constrained key.

Formal description: Let $\mathsf{Acc} = (\mathsf{Setup\text{-}Acc}, \mathsf{Setup\text{-}Acc\text{-}Enf\text{-}Read}, \mathsf{Setup\text{-}Acc\text{-}Enf\text{-}Write}, \mathsf{Prep\text{-}Read}, \mathsf{Prep\text{-}Write}, \mathsf{Verify\text{-}Read}, \mathsf{Write\text{-}Store}, \mathsf{Update})$ be a positional accumulator, $\mathsf{Itr} = (\mathsf{Setup\text{-}Itr}, \mathsf{Setup\text{-}Itr\text{-}Enf}, \mathsf{Iterate})$ an iterator, $\mathcal{S} = (\mathsf{Setup\text{-}Spl}, \mathsf{Sign\text{-}Spl}, \mathsf{Verify\text{-}Spl}, \mathsf{Split}, \mathsf{Sign\text{-}Spl\text{-}abo})$ a splittable signature scheme and $\mathrm{PRG} : \{0,1\}^{\lambda} \to \{0,1\}^{2\lambda}$ a length doubling injective pseudorandom generator.

Let F be a puncturable pseudorandom function whose domain and range are chosen appropriately, depending on the accumulator, iterator and splittable signature scheme. For simplicity, we assume that F takes inputs of bounded length, instead of fixed length inputs. This assumption can be easily removed by using different PRFs for different input lengths (in our case, we will require three different fixed-input-length PRFs). Also, to avoid confusion, the puncturable PRF keys (both master and punctured) are represented using lower case letters (e.g. k, $k\{z\}$), while the constrained PRF keys are represented using upper case letters (e.g. K, $K\{M\}$).

– $\mathsf{PRF.setup}(1^{\lambda})$: The setup algorithm takes the security parameter λ as input. It first chooses a puncturable PRF keys $k \leftarrow \mathsf{F.setup}(1^{\lambda})$. Next, it runs the accumulator setup to obtain $(\mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0) \leftarrow \mathsf{Setup\text{-}Acc}(1^{\lambda})$. The master PRF key is $K = (k, \mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0)$.

– PRF Evaluation: To evaluate the PRF with key $K = (k, \mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0)$ on input $x = x_1 \dots x_n$, first 'hash' the input using the accumulator. More formally, let $\mathsf{Hash\text{-}Acc}(x) = \mathsf{acc}_n$, where for all $j \leq n$, acc_j is defined as follows:

  - $\mathrm{STORE}_j = \mathsf{Write\text{-}Store}(\mathrm{PP}_{\mathsf{Acc}}, \mathrm{STORE}_{j-1}, j-1, x_j)$
  - $aux_j = \mathsf{Prep\text{-}Write}(\mathrm{PP}_{\mathsf{Acc}}, \mathrm{STORE}_{j-1}, j-1)$
  - $\mathsf{acc}_j = \mathsf{Update}(\mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_{j-1}, x_j, j-1, aux_j)$

  The PRF evaluation is defined to be $F(k, \mathsf{Hash\text{-}Acc}(x))$.

– $\mathsf{PRF.constrain}(K = (k, \mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0), M)$: The constrain algorithm first chooses puncturable PRF keys $k_1, \dots, k_\lambda$ and $k_{\mathrm{sig},A}$ and runs the iterator setup to obtain $(\mathrm{PP}_{\mathsf{Itr}}, \mathsf{it}_0) \leftarrow \mathsf{Setup\text{-}Itr}(1^\lambda, T)$. Next, it computes an obfuscation of program Prog (defined in Figure 1) and Init-Sign (defined in Figure 2). The constrained key $K\{M\} = (\mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0, \mathrm{PP}_{\mathsf{Itr}}, \mathsf{it}_0, i\mathcal{O}(\mathsf{Prog}), i\mathcal{O}(\mathsf{Init\text{-}Sign}))$.

– PRF Evaluation using Constrained Key: Let $K\{M\} = (\mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0, \mathrm{PP}_{\mathsf{Itr}}, \mathsf{it}_0, P_1, P_2)$ be a constrained key corresponding to machine M, and $x = x_1, \dots, x_n$ the input. As in the evaluation using master PRF key, first compute $\mathsf{acc\text{-}inp} = \mathsf{Hash\text{-}Acc}(x)$.

  To begin the evaluation, compute a signature on the initial values using the program P_2. Let $\sigma_0 = P_2(\mathsf{acc\text{-}inp})$.

  Suppose M runs for t^* steps on input x. Run the program P_1 iteratively for t^* steps. Set $\mathsf{pos}_0 = 0$, $\mathsf{seed}_0 =$ ' ', and for $i = 1$ to t^*, compute

  1. Let $(\mathsf{sym}_{i-1}, \pi_{i-1}) = \mathsf{Prep\text{-}Read}(\mathrm{PP}_{\mathsf{Acc}}, \mathrm{STORE}_{i-1}, \mathsf{pos}_{i-1})$.
  2. Compute $aux_{i-1} \leftarrow \mathsf{Prep\text{-}Write}(\mathrm{PP}_{\mathsf{Acc}}, \mathrm{STORE}_{i-1}, \mathsf{pos}_{i-1})$.
  3. Let $\mathrm{out} = P_1(i, \mathsf{seed}_{i-1}, \mathsf{pos}_{i-1}, \mathsf{sym}_{i-1}, \mathsf{st}_{i-1}, \mathsf{acc}_{i-1}, \pi_{i-1}, aux_{i-1}, \mathsf{acc\text{-}inp}, \mathsf{it}_{i-1}, \sigma_{i-1})$.
     If $j = t^*$, output out. Else, parse out as $(\mathsf{sym}_{w,i}, \mathsf{pos}_i, \mathsf{st}_i, \mathsf{acc}_i, \mathsf{it}_i, \sigma_i, \mathsf{seed}_i)$.
  4. Compute $\mathrm{STORE}_i = \mathsf{Write\text{-}Store}(\mathrm{PP}_{\mathsf{Acc}}, \mathrm{STORE}_{i-1}, \mathsf{pos}_{i-1}, \mathsf{sym}_{w,i})$.

  The output at step t^* is the PRF evaluation using the constrained key.

4.1 Proof of Selective Security

Theorem 1. *Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable pseudorandom function, Acc is a secure positional accumulator, Itr is a secure positional iterator and $\mathcal{S}$ is a secure splittable signature scheme, the constrained PRF construction described in Section 4 is selectively secure as defined in Definition 13.*

Our security proof will consist of a sequence of computationally indistinguishable hybrid experiments. Recall that we are proving selective security, where the adversary sends the challenge input x^* before receiving any constrained keys.

Program Prog

Constants : Turing machine $M = \langle Q, \Sigma_{\text{tape}}, \delta, q_0, q_{\text{ac}}, q_{\text{rej}} \rangle$,
time bound T
Public parameters$\text{PP}_{\mathsf{Acc}}, \text{PP}_{\mathsf{ltr}}$
Puncturable PRF keys $k, k_1, \ldots, k_\lambda, k_{\text{sig},A}$

Inputs : Time t, String seed, position pos_{in}, symbol sym_{in},
TM state st_{in}Accumulator value acc_{in}, proof π,
auxiliary value aux, accumulation of input $\mathsf{acc\text{-}inp}$
Iterator value it_{in}, signature σ_{in}.

1. Let μ be an integer such that $2^\mu \leq t < 2^{\mu+1}$.
   If $\text{PRG}(\mathsf{seed}) \neq \text{PRG}(\mathsf{F}(k_\mu, \mathsf{acc\text{-}inp}))$ and $t > 1$, output $\perp$.

2. If $\mathsf{Verify\text{-}Read}(\text{PP}_{\mathsf{Acc}}, \mathsf{acc}_{\text{in}}, \mathsf{sym}_{\text{in}}, \mathsf{pos}_{\text{in}}, \pi) = 0$ output $\perp$.

3. (a) Let $r_A = \mathsf{F}(k_{\text{sig},A}, (\mathsf{acc\text{-}inp}, t-1))$.
   Compute $(\text{SK}_A, \text{VK}_A, \text{VK}_{A,\text{rej}}) = \mathsf{Setup\text{-}Spl}(1^\lambda; r_A)$.
   (b) Let $m_{\text{in}} = (\mathsf{it}_{\text{in}}, \mathsf{st}_{\text{in}}, \mathsf{acc}_{\text{in}}, \mathsf{pos}_{\text{in}})$.
   If $\mathsf{Verify\text{-}Spl}(\text{VK}_A, m_{\text{in}}, \sigma_{\text{in}}) = 0$ output $\perp$.

4. (a) Let $(\mathsf{st}_{\text{out}}, \mathsf{sym}_{\text{out}}, \beta) = \delta(\mathsf{st}_{\text{in}}, \mathsf{sym}_{\text{in}})$ and $\mathsf{pos}_{\text{out}} = \mathsf{pos}_{\text{in}} + \beta$.
   (b) If $\mathsf{st}_{\text{out}} = q_{\text{rej}}$ output $\perp$.
   Else if $\mathsf{st}_{\text{out}} = q_{\text{ac}}$ output $\mathsf{F}(k, \mathsf{acc\text{-}inp})$.

5. (a) Compute $\mathsf{acc}_{\text{out}} = \mathsf{Update}(\text{PP}_{\mathsf{Acc}}, \mathsf{acc}_{\text{in}}, \mathsf{sym}_{\text{out}}, \mathsf{pos}_{\text{in}}, aux)$.
   If $\mathsf{acc}_{\text{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $\mathsf{it}_{\text{out}} = \mathsf{Iterate}(\text{PP}_{\mathsf{ltr}}, \mathsf{it}_{\text{in}}, (\mathsf{st}_{\text{in}}, \mathsf{acc}_{\text{in}}, \mathsf{pos}_{\text{in}}))$.

6. (a) Let $r'_A = \mathsf{F}(k_{\text{sig},A}, (\mathsf{acc\text{-}inp}, t))$.
   Compute $(\text{SK}'_A, \text{VK}'_A, \text{VK}'_{A,\text{rej}}) = \mathsf{Setup\text{-}Spl}(1^\lambda; r'_A)$.
   (b) Let $m_{\text{out}} = (\mathsf{it}_{\text{out}}, \mathsf{st}_{\text{out}}, \mathsf{acc}_{\text{out}}, \mathsf{pos}_{\text{out}})$ and $\sigma_{\text{out}} = \mathsf{Sign\text{-}Spl}(\text{SK}'_\alpha, m_{\text{out}})$.

7. If $t + 1 = 2^{\mu+1}$, set $\mathsf{seed}' = \mathsf{F}(k_{\mu+1}, \mathsf{acc\text{-}inp})$.
   Else, set $\mathsf{seed}' = \text{␣}$.
8. Output $\mathsf{pos}_{\text{out}}, \mathsf{sym}_{\text{out}}, \mathsf{st}_{\text{out}}, \mathsf{acc}_{\text{out}}, \mathsf{it}_{\text{out}}, \sigma_{\text{out}}, \mathsf{seed}'$.

Fig. 1: Program Prog

Sequence of Hybrid Experiments We will first set up some notation for the hybrid experiments. Let q denote the number of constrained key queries made by the adversary. Let x^* denote the challenge input chosen by the adversary,

Program Init-Sign

Constants: Puncturable PRF key $k_{\mathrm{sig},A}$, Initial TM state q_0, Iterator value it_0

Input: Accumulation of input $\mathsf{acc\text{-}inp}$

1. Let $F(k_{\mathrm{sig},A}, (\mathsf{acc\text{-}inp}, 0)) = r_{\mathrm{sig}}$. Compute $(\mathrm{SK}, \mathrm{VK}, \mathrm{VK}_{\mathrm{rej}}) = \mathsf{Setup\text{-}Spl}(1^\lambda; r_{\mathrm{sig}})$.
2. Output $\sigma = \mathsf{Sign\text{-}Spl}(\mathrm{SK}, (\mathsf{it}_0, q_0, \mathsf{acc\text{-}inp}, 0))$.

Fig. 2: Program Init-Sign

$(k, \mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0)$ the master key chosen by challenger, $\mathsf{acc\text{-}inp}^* = \mathsf{Hash\text{-}Acc}(x^*)$ as defined in the construction. Let M_j denote the j^{th} constrained key query, and t_j^* be the running time of machine M_j on input x^*, and τ_j be the smallest power of two greater than t_j. The program Prog_j denotes the program Prog with machine M_j hardwired.

Hybrid_0 This corresponds to the real experiment.

Next, we define q hybrid experiments $\mathsf{Hybrid}_{0,j}$ for $1 \leq j \leq q$.

$\mathsf{Hybrid}_{0,j}$: Let Prog-1 denote the program defined in Figure 3. In this experiment, the challenger sends an obfuscation of the program $\mathsf{Prog\text{-}1}_i$ (Prog-1 with machine M_i hardwired) for the i^{th} query if $i \leq j$. For the remaining queries, the challenger outputs an obfuscation of Prog_i.

Hybrid_1 : This experiment is identical to hybrid $\mathsf{Hybrid}_{0,q}$. In this experiment, the challenger sends an obfuscation of $\mathsf{Prog\text{-}1}_i$ for all constrained key queries.

Hybrid_2 : In this experiment, the challenger punctures the PRF key k at input $\mathsf{acc\text{-}inp}^*$ and uses the punctured key for all key queries. More formally, after receiving the challenge input x^*, it chooses $(\mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0) \leftarrow \mathsf{Setup\text{-}Acc}(1^\lambda)$ and computes $\mathsf{acc\text{-}inp}^* = \mathsf{Hash\text{-}Acc}(x^*)$. It then chooses a PRF key k and computes $k\{\mathsf{acc\text{-}inp}^*\} \leftarrow \mathsf{F.puncture}(k, \mathsf{acc\text{-}inp}^*)$. Next, it receives constrained key queries for machines $M_1, \ldots, M_q$. For each query, it chooses $(\mathrm{PP}_{\mathsf{Itr}}, \mathsf{it}_0) \leftarrow \mathsf{Setup\text{-}Itr}(1^\lambda)$ and PRF keys $k_1, \ldots, k_\lambda$, $k_{\mathrm{sig},A}$. It computes an obfuscation of $\mathsf{Prog\text{-}1}\{M_i\ , \mathrm{PP}_{\mathsf{Acc}}, \mathrm{PP}_{\mathsf{Itr}}, k\{\mathsf{acc\text{-}inp}^*\}, k_{\mathrm{sig},A}\}$.

$$\mathsf{Hybrid}_0 \equiv \mathsf{Hybrid}_{0,0} \xrightarrow{1} \mathsf{Hybrid}_{0,1} \xrightarrow{1} \cdots \xrightarrow{1} \mathsf{Hybrid}_{0,q} \equiv \mathsf{Hybrid}_1$$

$$\mathsf{Hybrid}_1 \downarrow i\mathcal{O}$$

$$0 \overset{\mathsf{PPRF}}{\approx} \mathsf{Hybrid}_2$$

Program Prog-1

Constants : Turing machine $M = \langle Q, \Sigma_{\text{tape}}, \delta, q_0, q_{\text{ac}}, q_{\text{rej}} \rangle$, time $t^* \in [T]$
Public parameters for accumulator PP_{Acc},
Public parameters for Iterator PP_{Itr}
Puncturable PRF keys $k, k_1, \ldots, k_\lambda, k_{\text{sig},A} \in \mathcal{K}$
Hardwired accumulated value $\mathsf{acc\text{-}inp}^*$

Inputs: Time t, String seed, position pos_{in}, symbol sym_{in}, TM state st_{in}
Accumulator value acc_{in}, proof π, auxiliary value aux,
accumulation of input $\mathsf{acc\text{-}inp}$, Iterator value it_{in}, signature σ_{in}.

1. Let μ be an integer such that $2^\mu \leq t < 2^{\mu+1}$.
   If $\text{PRG}(\mathsf{seed}) \neq \text{PRG}(\mathsf{F}(k_\mu, \mathsf{acc\text{-}inp}))$ and $t > 1$, output $\perp$.

2. If $\mathsf{Verify\text{-}Read}(\text{PP}_{\mathsf{Acc}}, \mathsf{acc}_{\text{in}}, \mathsf{sym}_{\text{in}}, \mathsf{pos}_{\text{in}}, \pi) = 0$ output $\perp$.

3. (a) Let $r_{\text{sig}} = F(k_{\text{sig},A}, t-1)$. Compute $(\text{SK}, \text{VK}, \text{VK}_{\text{rej}}) = \mathsf{Setup\text{-}Spl}(1^\lambda; r_{\text{sig}})$.
   (b) Let $m_{\text{in}} = (\mathsf{it}_{\text{in}}, \mathsf{st}_{\text{in}}, \mathsf{acc}_{\text{in}}, \mathsf{pos}_{\text{in}}, \mathsf{acc\text{-}inp})$. If $\mathsf{Verify\text{-}Spl}(\text{VK}, m_{\text{in}}, \sigma_{\text{in}}) = 0$ output $\perp$.

4. (a) Let $(\mathsf{st}_{\text{out}}, \mathsf{sym}_{\text{out}}, \beta) = \delta(\mathsf{st}_{\text{in}}, \mathsf{sym}_{\text{in}})$ and $\mathsf{pos}_{\text{out}} = \mathsf{pos}_{\text{in}} + \beta$.
   (b) If $\mathsf{st}_{\text{out}} = q_{\text{rej}}$ output $\perp$.
   (c) If $\mathsf{st}_{\text{out}} = q_{\text{ac}}$ and $\mathsf{acc\text{-}inp} \neq \mathsf{acc\text{-}inp}^*$, output $\mathsf{F}(k, \mathsf{acc\text{-}inp})$.
   Else If $\mathsf{st}_{\text{out}} = q_{\text{ac}}$ output $\perp$.

5. (a) Compute $\mathsf{acc}_{\text{out}} = \mathsf{Update}(\text{PP}_{\mathsf{Acc}}, \mathsf{acc}_{\text{in}}, \mathsf{sym}_{\text{out}}, \mathsf{pos}_{\text{in}}, aux)$. If $w_{\text{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $\mathsf{it}_{\text{out}} = \mathsf{Iterate}(\text{PP}_{\mathsf{Itr}}, \mathsf{it}_{\text{in}}, (\mathsf{st}_{\text{in}}, \mathsf{acc}_{\text{in}}, \mathsf{pos}_{\text{in}}))$.

6. (a) Let $r'_{\text{sig}} = F(k_{\text{sig},A}, (\mathsf{acc\text{-}inp}, t))$. Compute $(\text{SK}', \text{VK}', \text{VK}'_{\text{rej}}) \leftarrow \mathsf{Setup\text{-}Spl}(1^\lambda; r'_{\text{sig}})$.
   (b) Let $m_{\text{out}} = (\mathsf{it}_{\text{out}}, \mathsf{st}_{\text{out}}, \mathsf{acc}_{\text{out}}, \mathsf{pos}_{\text{out}}, \mathsf{acc\text{-}inp})$ and $\sigma_{\text{out}} = \mathsf{Sign\text{-}Spl}(\text{SK}', m_{\text{out}})$.

7. If $t + 1 = 2^{\mu+1}$, set $\mathsf{seed}' = \mathsf{F}(k_{\mu+1}, \mathsf{acc\text{-}inp})$.
   Else, set $\mathsf{seed}' = \lrcorner$.
8. Output $\mathsf{pos}_{\text{out}}, \mathsf{sym}_{\text{out}}, \mathsf{st}_{\text{out}}, \mathsf{acc}_{\text{out}}, \mathsf{it}_{\text{out}}, \sigma_{\text{out}}, \mathsf{seed}'$.

Fig. 3: Program Prog-1

Analysis Let $\mathsf{Adv}_i^{\mathcal{A}}$ denote the advantage of any PPT adversary $\mathcal{A}$ in the hybrid experiment Hybrid_i (similarly, let $\mathsf{Adv}_{0,j}^{\mathcal{A}}$ denote the advantage of $\mathcal{A}$ in the intermediate hybrid experiment $\mathsf{Hybrid}_{0,j}$).

Recall $\mathsf{Hybrid}_{0,0}$ corresponds to the experiment Hybrid_0, and $\mathsf{Hybrid}_{0,q}$ corresponds to the experiment Hybrid_1. Using the following lemma, we can show that $|\mathsf{Adv}_0^{\mathcal{A}} - \mathsf{Adv}_1^{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Lemma 1. *Assuming F is a puncturable PRF, Acc is a secure positional accumulator, Itr is a secure positional iterator, $\mathcal{S}$ is a secure splittable signature scheme and $i\mathcal{O}$ is a secure indistinguishability obfuscator, for any PPT adversary $\mathcal{A}$, $|\mathsf{Adv}_{0,j}^{\mathcal{A}} - \mathsf{Adv}_{0,j+1}^{\mathcal{A}}| \leq negl(\lambda)$.*

The proof of this lemma involves multiple hybrids. We include a high level outline of the proof in Appendix A, while the complete proof can be found in the full version of our paper.

Lemma 2. *Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, for any PPT adversary $\mathcal{A}$, $|\mathsf{Adv}_1^{\mathcal{A}} - \mathsf{Adv}_2^{\mathcal{A}}| \leq negl(\lambda)$.*

Proof. Let us assume for now that the adversary makes exactly one constrained key query corresponding to machine M_1. This can be naturally extended to the general case via a hybrid argument.

Note that the only difference between the two hybrids is the PRF key hardwired in Prog-1. In one case, the challenger sends an obfuscation of $P_1 = \mathsf{Prog}\text{-}1\{M_1, \mathrm{PP}_{\mathsf{Acc}}, \mathrm{PP}_{\mathsf{Itr}}, k, k_1, \ldots, k_\lambda, k_{\mathrm{sig},A}\}$, while in the other, it sends an obfuscation of $P_2 = \mathsf{Prog}\text{-}1\{M_1, \mathrm{PP}_{\mathsf{Acc}}, \mathrm{PP}_{\mathsf{Itr}}, k\{\mathsf{acc\text{-}inp}^*\}, k_1, \ldots, k_\lambda, k_{\mathrm{sig},A}\}$. To prove that these two hybrids are computationally indistinguishable, it suffices to show that the P_1 and P_2 are functionally identical. Note that program P_1 computes $\mathsf{F}(k, \mathsf{acc\text{-}inp})$ only if $\mathsf{acc\text{-}inp} \neq \mathsf{acc\text{-}inp}^*$. As a result, using the correctness property of puncturable PRFs, the programs have identical functionality.

Lemma 3. *Assuming F is a selectively secure puncturable PRF, for any PPT adversary $\mathcal{A}$, $|\mathsf{Adv}_2^{\mathcal{A}}| \leq negl(\lambda)$.*

Proof. Suppose there exists a PPT adversary $\mathcal{A}$ such that $|\mathsf{Adv}_2^{\mathcal{A}}| = \epsilon$. We will use $\mathcal{A}$ to construct a PPT algorithm $\mathcal{B}$ that breaks the security of the puncturable PRF F.

To begin with, $\mathcal{B}$ receives the challenge input x^* from $\mathcal{A}$. It chooses $(\mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0) \leftarrow \mathsf{Setup\text{-}Acc}(1^\lambda)$. It then computes $\mathsf{acc\text{-}inp}^* = \mathsf{Hash\text{-}Acc}(x^*)$, and sends $\mathsf{acc\text{-}inp}^*$ to the PRF challenger as the challenge input. It receives a punctured key k' and an element y (which is either the pseudorandom evaluation at $\mathsf{acc\text{-}inp}^*$ or a truly random string in the range space). $\mathcal{B}$ sends y to $\mathcal{A}$ as the challenge response.

Next, it receives multiple constrained key requests. For the i^{th} query corresponding to machine M_i, $\mathcal{B}$ chooses PRF keys $k_1, \ldots, k_\lambda, k_{\mathrm{sig},A} \leftarrow \mathsf{F.setup}(1^\lambda)$, $(\mathrm{PP}_{\mathsf{Itr}}, \mathsf{it}_0) \leftarrow \mathsf{Setup\text{-}Itr}(1^\lambda)$ and computes an obfuscation of $\mathsf{Prog}\text{-}1\{M_i, \mathrm{PP}_{\mathsf{Acc}}, \mathrm{PP}_{\mathsf{Itr}}, k', k_1, \ldots, k_\lambda, k_{\mathrm{sig},A}\}$. It sends this obfuscated program to $\mathcal{A}$ as the constrained key.

Finally, after all constrained key queries, $\mathcal{A}$ sends its guess b', which $\mathcal{B}$ forwards to the challenger. Note that if $\mathcal{A}$ wins the security game against PRF, then $\mathcal{B}$ wins the security game against F. This concludes our proof.

5 Attribute Based Encryption for Turing Machines

In this section, we describe an ABE scheme where policies are associated with Turing machines, and as a result, attributes can be strings of unbounded length. Our ABE scheme is very similar to the constrained PRF construction described in Section 4.

Let $\mathcal{PKE} = (\mathsf{PKE.setup}, \mathsf{PKE.enc}, \mathsf{PKE.dec})$ be a public key encryption scheme and F a puncturable PRF for Turing machines, with algorithms $\mathsf{PRF.setup}$ and $\mathsf{PRF.constrain}$. Consider the following ABE scheme:

- $\mathsf{ABE.setup}(1^\lambda)$ The setup algorithm chooses a puncturable PRF key $k \leftarrow \mathsf{F.setup}(1^\lambda)$ and $(\mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0) \leftarrow \mathsf{Setup\text{-}Acc}(1^\lambda, T)$. Next, it computes an obfuscation of $\mathsf{Prog\text{-}PK}\{k\}$ (defined in Figure 4). The public key $\mathrm{PK}_{\mathsf{ABE}} = (\mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0, i\mathcal{O}(\mathsf{Prog\text{-}PK}\{k\}))$, while the master secret key is $\mathrm{MSK}_{\mathsf{ABE}} = k$.

Program Prog-PK

Constants: Puncturable PRF key k
Input: Accumulation of input $\mathsf{acc\text{-}inp}$

1. Let $F(k, \mathsf{acc\text{-}inp}) = r$. Compute $(\mathrm{pk}, \mathrm{sk}) = \mathsf{PKE.setup}(1^\lambda; r)$.
2. Output pk.

Fig. 4: Program $\mathsf{Prog\text{-}PK}$

- $\mathsf{ABE.enc}(m, x, \mathrm{PK}_{\mathsf{ABE}})$ Let $\mathrm{PK}_{\mathsf{ABE}} = (\mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_0, \mathrm{STORE}_0, \mathsf{Program}_{pk})$ and $x = x_1 \dots x_n$. As in Section 4, the encryption algorithm first 'accumulates' the attribute x using the accumulator public parameters. Let $\mathsf{acc\text{-}inp} = \mathsf{acc}_n$, where for all $j \leq n$, acc_j is defined as follows:
  - $\mathrm{STORE}_j = \mathsf{Write\text{-}Store}(\mathrm{PP}_{\mathsf{Acc}}, \mathrm{STORE}_{j-1}, j-1, x_j)$
  - $aux_j = \mathsf{Prep\text{-}Write}(\mathrm{PP}_{\mathsf{Acc}}, \mathrm{STORE}_{j-1}, j-1)$
  - $\mathsf{acc}_j = \mathsf{Update}(\mathrm{PP}_{\mathsf{Acc}}, \mathsf{acc}_{j-1}, x_j, j-1, aux_j)$

  Next, the accumulated value is used to compute a PKE public key. Let $\mathrm{pk} = \mathsf{Program}_{pk}(\mathsf{acc\text{-}inp})$. Finally, the algorithm outputs $\mathsf{ct} = \mathsf{PKE.enc}(m, \mathrm{pk})$.

- $\mathsf{ABE.keygen}(\mathrm{MSK}_{\mathsf{ABE}}, M)$ Let $\mathrm{MSK}_{\mathsf{ABE}} = k$ and $M =$ a Turing machine. The ABE key corresponding to M is exactly the constrained key corresponding to M, as defined in Section 4. In particular, the key generation algorithm chooses $(\mathrm{PP}_{\mathsf{ltr}}, \mathsf{it}_0) \leftarrow \mathsf{Setup\text{-}ltr}(1^\lambda, T)$ and a puncturable PRF key $k_{\mathrm{sig},A}$, and computes an obfuscation of $\mathsf{Prog}\{M, k, k_{\mathrm{sig}}, \mathrm{PP}_{\mathsf{Acc}}, \mathrm{PP}_{\mathsf{ltr}}\}$ (defined in Figure 1) and $\mathsf{Init\text{-}Sign}\{k_{\mathrm{sig},A}$ (defined in Figure 2). The secret key $\mathrm{SK}\{M\} = (\mathrm{PP}_{\mathsf{ltr}}, \mathsf{it}_0, i\mathcal{O}(\mathsf{Prog}), i\mathcal{O}(\mathsf{Init\text{-}Sign}))$.

– $\mathsf{ABE.dec}(\mathrm{SK}\{M\}, \mathsf{ct}, x)$ Let $\mathrm{SK}\{M\} = (\mathrm{PP}_{\mathsf{ltr}}, \mathsf{it}_0, \mathsf{Program}_1, \mathsf{Program}_2)$, and suppose M accepts x in t^* steps. As in the constrained key PRF evaluation, the decryption algorithm first obtains a signature using $\mathsf{Program}_2$ and then runs $\mathsf{Program}_1$ for t^* steps, until it outputs the pseudorandom string r. Using this PRF output r, the decryption algorithm computes $(\mathrm{pk}, \mathrm{sk}) = \mathsf{PKE.setup}(1^\lambda; r)$ and then decrypts ct using sk. The algorithm outputs $\mathsf{PKE.dec}(\mathrm{sk}, \mathsf{ct})$.

5.1 Proof of Security

We will first define a sequence of hybrid experiments, and then show that any two consecutive hybrid experiments are computationally indistinguishable.

Sequence of Hybrid Experiments

Hybrid H_0 This corresponds to the selective security game. Let x^* denote the challenge input, and $\mathsf{acc\text{-}inp}^* = \mathsf{Hash\text{-}Acc}(x^*)$.

Hybrid H_1 In this hybrid, the challenger sends an obfuscation of Prog-1 instead of Prog. Prog-1, on inputs corresponding to $\mathsf{acc\text{-}inp}^*$, never reaches the accepting state q_{ac}. This is similar to Hybrid_1 of the constrained PRF security proof in Section 4.1.

Hybrid H_2 In this hybrid, the challenger first punctures the PRF key k at $\mathsf{acc\text{-}inp}^*$. It computes $k' \leftarrow \mathsf{F.puncture}(k, \mathsf{acc\text{-}inp}^*)$ and $(\mathrm{pk}^*, \mathrm{sk}^*) = \mathsf{PKE.setup}(1^\lambda; F(k, \mathsf{acc\text{-}inp}^*))$. Next, it uses k' and pk^* to define $\mathsf{Prog\text{-}PK}'\{k', \mathrm{pk}^*\}$ (see Figure 5). It sends an obfuscation of $\mathsf{Prog\text{-}PK}'$ as the public key. Next, for each of the secret key queries, it sends an obfuscation of Prog-1. However unlike the previous hybrid, Prog-1 has k' hardwired instead of k.

Program $\mathsf{Prog\text{-}PK}'$

Constants: Punctured PRF key k', Hardwired accumulation $\mathsf{acc\text{-}inp}^*$ and public key pk^*.
Input: Accumulation of input $\mathsf{acc\text{-}inp}$

1. If $\mathsf{acc\text{-}inp} = \mathsf{acc\text{-}inp}^*$, set $\mathrm{pk} = \mathrm{pk}^*$.
   Else let $F(k', \mathsf{acc\text{-}inp}) = r$. Compute $(\mathrm{pk}, \mathrm{sk}) = \mathsf{PKE.setup}(1^\lambda; r)$.
2. Output pk.

Fig. 5: Program $\mathsf{Prog\text{-}PK}'$

Hybrid H_3 In this hybrid, the challenger chooses $(\mathrm{pk}^*, \mathrm{sk}^*) \leftarrow \mathsf{PKE.setup}(1^\lambda)$; that is, the public key is computed using true randomness. It then hardwires pk^* in $\mathsf{Prog\text{-}PK}$. The secret key queries are same as in previous hybrids.

Analysis Let $\mathsf{Adv}_i^{\mathcal{A}}$ denote the advantage of $\mathcal{A}$ in hybrid H_i.

Lemma 4. *Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator,* Acc *is a secure positional accumulator,* Itr *is a secure iterator, $\mathcal{S}$ is a secure splittable signature scheme and* F *is a secure puncturable PRF, for any adversary $\mathcal{A}$, $|\mathsf{Adv}_0^{\mathcal{A}} - \mathsf{Adv}_1^{\mathcal{A}}| \leq negl(\lambda)$.*

The proof of this lemma is identical to the proof of Lemma 1.

Lemma 5. *Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, for any PPT adversary $\mathcal{A}$, $|\mathsf{Adv}_1^{\mathcal{A}} - \mathsf{Adv}_2^{\mathcal{A}}| \leq negl(\lambda)$.*

Proof. Similar to the proof of Lemma 2, k can be replaced with k' in all the secret key queries, since $F(k, \mathsf{acc\text{-}inp}^*)$ is never executed. As far as $\mathsf{Prog\text{-}PK}$ and $\mathsf{Prog\text{-}PK}'$ are concerned, $(\mathrm{pk}^*, \mathrm{sk}^*)$ is set to be $\mathsf{PKE.setup}(1^\lambda; F(k, \mathsf{acc\text{-}inp}^*))$, and therefore, the programs are functionally identical.

Lemma 6. *Assuming* F *is a selectively secure puncturable PRF, for any PPT adversary $\mathcal{A}$, $|\mathsf{Adv}_2^{\mathcal{A}} - \mathsf{Adv}_3^{\mathcal{A}}| \leq negl(\lambda)$.*

Proof. The proof of this follows immediately from the security definition of puncturable PRFs. Suppose there exists an adversary that can distinguish between H_2 and H_3 with advantage ϵ. Then, there exists a PPT algorithm $\mathcal{B}$ that can break the selective security of F. $\mathcal{B}$ first receives x^* from the adversary. It computes $\mathsf{acc\text{-}inp}^*$, sends $\mathsf{acc\text{-}inp}^*$ to the PRF challenger and receives k', y, where y is either the PRF evaluation at $\mathsf{acc\text{-}inp}^*$, or a truly random string. Using y, it computes $(\mathrm{pk}^*, \mathrm{sk}^*) = \mathsf{PKE.setup}(1^\lambda; y)$, and uses k', pk^* to define the public key $i\mathcal{O}(\mathsf{Prog\text{-}PK}'\{k', \mathrm{pk}^*\})$. The secret key queries are same in both hybrids, and can be answered using k' only. As a result, $\mathcal{B}$ simulates either H_2 or H_3 perfectly. This concludes our proof.

Lemma 7. *Assuming $\mathcal{PKE}$ is a secure public key encryption scheme, for any PPT adversary $\mathcal{A}$, $\mathsf{Adv}_3^{\mathcal{A}} \leq negl(\lambda)$.*

Proof. Suppose there exists a PPT adversary $\mathcal{A}$ such that $\mathsf{Adv}_3^{\mathcal{A}} = \epsilon$. Then there exists a PPT adversary $\mathcal{B}$ that breaks IND-CPA security of $\mathcal{PKE}$. $\mathcal{B}$ receives a public key pk^* from the challenger. It chooses PRF key k, punctures it at $\mathsf{acc\text{-}inp}^*$ and sends the public key $i\mathcal{O}\{\mathsf{Prog\text{-}PK}'\}$. Next, it responds to the secret key queries, and finally, on receiving challenge messages m_0, m_1, it forwards them to the challenger, and receives ct^*, which it forwards to the adversary. The post challenge key query phase is also simulated perfectly, since it has all the required components.

References

[1] Abusalah, H., Fuchsbauer, G., Pietrzak, K.: Constrained prfs for unbounded inputs. IACR Cryptology ePrint Archive 2014, 840 (2014), http://eprint.iacr.org/2014/840

[2] Ananth, P., Jain, A.: Indistinguishability obfuscation from compact functional encryption. IACR Cryptology ePrint Archive 2015, 173 (2015), http://eprint.iacr.org/2015/173
[3] Ananth, P., Sahai, A.: Functional encryption for turing machines. Cryptology ePrint Archive, Report 2015/776 (2015), http://eprint.iacr.org/
[4] Banerjee, A., Fuchsbauer, G., Peikert, C., Pietrzak, K., Stevens, S.: Key-homomorphic constrained pseudorandom functions. In: Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II. pp. 31–60 (2015), http://dx.doi.org/10.1007/978-3-662-46497-7_2
[5] Bitansky, N., Garg, S., Lin, H., Pass, R., Telang, S.: Succinct randomized encodings and their applications. In: Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015. pp. 439–448 (2015), http://doi.acm.org/10.1145/2746539.2746574
[6] Bitansky, N., Vaikuntanathan, V.: Indistinguishability obfuscation from functional encryption. IACR Cryptology ePrint Archive 2015, 163 (2015)
[7] Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. In: ASIACRYPT. pp. 280–300 (2013)
[8] Boneh, D., Zhandry, M.: Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. In: CRYPTO (2014)
[9] Boyle, E., Goldwasser, S., Ivan, I.: Functional signatures and pseudorandom functions. In: PKC. pp. 501–519 (2014)
[10] Brakerski, Z., Vaikuntanathan, V.: Constrained key-homomorphic prfs from standard lattice assumptions - or: How to secretly embed a circuit in your PRF. In: Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II. pp. 1–30 (2015), http://dx.doi.org/10.1007/978-3-662-46497-7_1
[11] Canetti, R., Holmgren, J., Jain, A., Vaikuntanathan, V.: Succinct garbling and indistinguishability obfuscation for RAM programs. In: Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015. pp. 429–437 (2015), http://doi.acm.org/10.1145/2746539.2746621
[12] Chandran, N., Raghuraman, S., Vinayagamurthy, D.: Constrained pseudorandom functions: Verifiable and delegatable. Cryptology ePrint Archive, Report 2014/522 (2014), http://eprint.iacr.org/
[13] Fuchsbauer, G., Konstantinov, M., Pietrzak, K., Rao, V.: Adaptive security of constrained prfs. IACR Cryptology ePrint Archive 2014, 416 (2014)
[14] Garg, S., Gentry, C., Halevi, S.: Candidate multilinear maps from ideal lattices. In: EUROCRYPT (2013)
[15] Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: FOCS (2013)
[16] Garg, S., Gentry, C., Halevi, S., Wichs, D.: On the implausibility of differing-inputs obfuscation and extractable witness encryption with auxiliary input. In: Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I. pp. 518–535 (2014)
[17] Gentry, C., Lewko, A., Sahai, A., Waters, B.: Indistinguishability obfuscation from the multilinear subgroup elimination assumption. Cryptology ePrint Archive, Report 2014/309 (2014), http://eprint.iacr.org/

[18] Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions (extended abstract). In: FOCS. pp. 464–479 (1984)
[19] Hofheinz, D., Kamath, A., Koppula, V., Waters, B.: Adaptively secure constrained pseudorandom functions. IACR Cryptology ePrint Archive 2014, 720 (2014), http://eprint.iacr.org/2014/720
[20] Hohenberger, S., Koppula, V., Waters, B.: Adaptively secure puncturable pseudorandom functions in the standard model. IACR Cryptology ePrint Archive 2014, 521 (2014), http://eprint.iacr.org/2014/521
[21] Hubacek, P., Wichs, D.: On the communication complexity of secure function evaluation with long output. In: Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015. pp. 163–172 (2015)
[22] Khurana, D., Rao, V., Sahai, A.: Multi-party key exchange for unbounded parties from indistinguishability obfuscation. In: ASIACRYPT (2015)
[23] Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: ACM Conference on Computer and Communications Security. pp. 669–684 (2013)
[24] Koppula, V., Lewko, A.B., Waters, B.: Indistinguishability obfuscation for turing machines with unbounded memory. In: Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing. pp. 419–428. STOC '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2746539.2746614
[25] Sahai, A., Waters, B.: How to use indistinguishability obfuscation: deniable encryption, and more. In: STOC. pp. 475–484 (2014)
[26] Zhandry, M.: Adaptively secure broadcast encryption with small system parameters (2014)

A Proof Outline of Lemma 1

In this section, we provide an outline of the proof of Lemma 1. The detailed proof is included in the full version of our paper. Let us assume the key query is for TM M, and M does not accept the challenge input x^*, and let $\mathsf{acc\text{-}inp}^*$ denote the accumulation of x^*. Our goal in this hybrid is to ensure that the program will never output $F(K, \mathsf{acc\text{-}inp}^*)$. This is done via a sequence of hybrids, where we use the security properties of splittable signatures, accumulators and iterators together with $i\mathcal{O}$ security.

Preprocessing hybrid: The first step is to modify the program Prog to allow additional valid signatures without being detected. In particular, we have an additional PRF key in the program, and this generates 'bad' signing/verification keys. The program first checks if the input signature is accepted by the usual 'good' verification key. If not, it checks if it is accepted by the 'bad' verification key. If the incoming signature is bad, then the output signature is also computed using the bad signing key. Let us call this hybrid Hyb-1. This switch is indistinguishable because the Init-Sign program only outputs a good signature, and we use the rejection-verification key indistinguishability property to show that this change is indistinguishable.

Intermediate hybrids $\mathsf{Hyb}\text{-}(1, i)$*:* Next, we gradually ensure that the program does not output the PRF evaluation on $\mathsf{acc\text{-}inp}^*$ in the first i steps. If $i = T$, then we are done. Here, we need to define our intermediate hybrid carefully. In the i^{th} intermediate hybrid, the program does not output PRF evaluation if $t \leq i$. Moreover, if $\mathsf{acc\text{-}inp} = \mathsf{acc\text{-}inp}^*$, it only accepts good signatures for the first $i-1$ steps. For the i^{th} step, if $\mathsf{acc\text{-}inp} = \mathsf{acc\text{-}inp}^*$, it accepts only good signatures, but outputs a bad signature if the input iterated value, accumulated value or state are not the correct ones for time step i (here, the program has the correct values for step i hardwired). We now need to go from step $\mathsf{Hyb}\text{-}(1, i)$ to step $\mathsf{Hyb}\text{-}(1, i + 1)$.

For this, we will first ensure that if $\mathsf{acc\text{-}inp} = \mathsf{acc\text{-}inp}^*$, the only signature accepted at step $i + 1$ is the one corresponding to the correct (iterated value, accumulated value, state) input tuple at step $i + 1$. Intuitively, this is true because the program, at step i, outputs a bad signature for all other tuples. To enforce this, we use the properties of the splittable signature schemes. Next, we make the accumulator read-enforcing. This would mean that both the state and symbol input at step $i+1$ are the correct ones. As a result, the program cannot output the PRF evaluation at step $i + 1$ if $\mathsf{acc\text{-}inp} = \mathsf{acc\text{-}inp}^*$. So now, the state and symbol output at step $i+1$ also have to be the correct ones. To ensure that the accumulated value and iterated value output are also correct, we make the accumulator write-enforcing and iterator enforcing respectively. Together, these will ensure that the transition from $\mathsf{Hyb}\text{-}(1, i)$ and $\mathsf{Hyb}\text{-}(1, i + 1)$ are computationally indistinguishable.

Continuing this way, we can ensure, step by step, that the program does not output the PRF evaluation on $\mathsf{acc\text{-}inp}^*$. However, the approach described above will require exponential hybrids. To make the number of intermediate hybrids polynomial, we use the 'tail-cutting' technique described in Section 1. Note that the program, after t^* steps, only outputs $\perp$. Suppose t^* is a power of two. Using a PRG trick, we can wipe out steps t^* to $2t^*$ in one shot. At every step where t is a power of two, the program outputs a new PRG seed, and this PRG seed's validity is checked till t reaches the next power of two. Now, if no PRG seed is output at step t^*, then using the PRG security, one can ensure that the PRG seed validity check fails. As a result, for all $t \in (t^*, 2t^*)$, the program outputs $\perp$.