Constrained Pseudorandom Functions for Unconstrained Inputs Revisited: Achieving Verifiability and Key Delegation

Pratish Datta, Ratna Dutta, and Sourav Mukhopadhyay

Department of Mathematics
Indian Institute of Technology Kharagpur
Kharagpur-721302, India
{pratishdatta,ratna,sourav}@maths.iitkgp.ernet.in

Abstract. In EUROCRYPT 2016, Deshpande et al. presented a construction of *constrained pseudorandom function* (CPRF) supporting inputs of *unconstrained* polynomial length based on indistinguishability obfuscation and injective pseudorandom generators. Their construction was claimed to be selectively secure. We demonstrate in this paper that their CPRF construction can actually be proven secure not in the selective model, rather in a *significantly weaker* security model where the adversary is forbidden to query constrained keys adaptively. We also show how to allow *adaptive* constrained key queries in their construction by innovating new technical ideas. We suitably redesign the security proof. We emphasize that our modification does not involve any additional heavy duty cryptographic tool. Our improved CPRF is further enhanced to present the *first* constructions of *constrained verifiable pseudorandom function* (CVPRF) and *delegatable constrained pseudorandom function* (DCPRF) supporting inputs of *unconstrained* polynomial length, employing only standard public key encryption (PKE).

Keywords: constrained pseudorandom functions, verifiable constrained pseudorandom function, key delegation, indistinguishability obfuscation

1 Introduction

Constrained Pseudorandom Functions: *Constrained pseudorandom functions* (CPRF), concurrently introduced by Boneh and Waters [6], Boyle et al. [7], as well as Kiayias et al. [19], are promising extension of the notion of standard *pseudorandom functions* (PRF) [15]. PRF is a fundamental primitive in modern cryptography. A PRF is a deterministic keyed function with the following property: Given a key, the function can be computed in polynomial time at all points of its input domain. But, without the key it is computationally hard to distinguish the PRF output at any arbitrary input from a uniformly random value, even after seeing the PRF evaluations on a polynomial number of inputs. A CPRF is an augmentation of a PRF with an additional *constrain algorithm* which enables a party holding a master PRF key to derive constrained keys

that allow the evaluation of the PRF over certain subsets of the input domain. However, PRF evaluations on the rest of the inputs still remain computationally indistinguishable from random.

Since their inception, CPRF's have found countless applications in various branches of cryptography ranging from broadcast encryption, attribute-based encryption to policy-based key distribution, multi-party on-interactive key exchange. Even the simplest class of CPRF's, known as *puncturable pseudorandom functions* (PPRF) [23], have turned out to be a powerful tool in conjunction with indistinguishability obfuscation [14]. In fact, the combination of these two primitives have led to solutions of longstanding open problems including deniable encryption, full domain hash, adaptively secure functional encryption for general functionalities, and functional encryption for randomized functionalities through the classic punctured programming technique introduced in [23].

Over the last few years there has been a significant progress in the field of CPRF's. In terms of expressiveness of the constraint predicates, starting with the most basic type of constraints such as prefix constraints [6,7,19] (which also encompass puncturing constraints) and bit fixing constraints [6,13], CPRF's have been constructed for highly rich constraint families such as circuit constraints [6,8,4,16] employing diverse cryptographic tools and based on various complexity assumptions. In terms of security, most of the existing CPRF constructions are only *selectively* secure. The stronger and more realistic notion of *adaptive* security seems to be rather challenging to achieve without complexity leveraging. In fact, the best known results so far on adaptive security of CPRF's require super-polynomial security loss [13], or work for very restricted form of constraints [17], or attain the security in non-collusion mode [8], or accomplish security in the random oracle model [16].

Constrained Verifiable Pseudorandom Functions: An interesting enhancement of the usual CPRF's is *verifiability*. A *verifiable constrained pseudorandom function* (CVPRF), independently introduced by Fuchsbauer [12] and Chandran et al. [9], is the unification of the notions of a *verifiable random function* (VRF) [21] and a standard CPRF. In a CVPRF system, a public verification key is set similar to a traditional VRF, along with the master PRF key. Besides enabling the evaluation of the PRF, the master PRF key can be utilized to generate a non-interactive proof of correctness of the evaluation. This proof can be verified by any party using only the public verification key. On the other hand, as in the case of a CPRF, here also the master PRF key holder can give out constrained keys for specific constraint predicates. A constrained key corresponding to some constraint predicate p allows the evaluation of the PRF together with the generation of a non-interactive proof of correct evaluation for only those inputs x for which $p(x) = 1$. In essence, CVPRF's resolve the issue of trust on a CPRF evaluator for the correctness of the received PRF output. In [12, 9], the authors have shown that the CPRF constructions of [6] for the bit fixing and circuit constraints can be augmented with the verifiability feature without incurring any significant additional cost.

Delegatable Constrained Pseudorandom Functions: *Key delegation* is another interesting enrichment of standard CPRF's. This feature empowers the holder of a constrained key, corresponding to some constraint predicate $p \in \mathbb{P}$ with the ability to distribute further restricted keys corresponding to the joint predicates $p \wedge \widetilde{p}$, for constraints $\widetilde{p} \in \mathbb{P}$, where $\mathbb{P}$ is certain constraint family over the input domain of the PRF. Such a delegated key can be utilized to evaluate the PRF on only those inputs x for which $[p(x) = 1] \wedge [\widetilde{p}(x) = 1]$, whereas, the PRF outputs on the rest of the inputs are computationally indistinguishable from random values. The concept of key delegation in the context of CPRF's has been recently introduced by Chandran et al. [9], who have shown how to extend the bit fixing and circuit-based CPRF constructions of [6] to support key delegation.

CPRF's for Unconstrained Inputs: Until recently, the research on CPRF's has been confined to inputs of apriori bounded length. In fact, all the CPRF constructions mentioned above could handle only bounded length inputs. Abusalah et al. [2] have taken a first step forward towards overcoming the barrier of bounded input length. They have also demonstrated highly motivating applications of CPRF's supporting apriori unconstrained length inputs such as broadcast encryption with an unbounded number of recipients and multi-party identity-based non-interactive key exchange with no pre-determined bound on the number of parties. They presented a selectively secure CPRF for unconstrained length inputs by viewing the constraint predicates as *Turing machines* (TM) that can handle inputs of arbitrary polynomial length. In a more recent work, Abusalah and Fuchsbauer [1] have made progress towards efficiency improvements by constructing TM-based CPRF's with much shorter constrained keys compared to the CPRF construction of [2].

However, both the aforementioned CPRF constructions rely on the existence of public-coin differing-input obfuscators and succinct non-interactive arguments of knowledge, which are believed to be risky assumptions due to their inherent extractability nature. In EUROCRYPT 2016, Deshpande et al. [10] presented a CPRF for TM constraints, supporting inputs of unconstrained polynomial length, which they claimed to be selectively secure. Their CPRF construction utilizes indistinguishability obfuscators (IO) for circuits and injective pseudorandom generators. Currently, there is no known impossibility or implausibility result on IO and, moreover, in the last few years, there has been a significant progress towards constructing IO based on standard complexity assumptions.

Our Contributions: Unfortunately, the CPRF construction of [10] can not be proven secure in the selective model, as will be shown in this paper, rather the construction actually derives its security in a *significantly weaker* model. Further, as per as we know, there is no existing construction of CVPRF's or delegatable CPRF's (DCPRF) supporting inputs of unconstrained length. Our work in this paper is two-fold:

- Firstly, we identify a flaw in the security argument of the CPRF construction of [10], by a thorough analysis of the construction and its security proof. Selective security is a security notion for CPRF's where the adversary is bound to declare upfront the challenge input, on which it wishes to distinguish the

PRF output from random, but is allowed to query the legitimate constrained keys and PRF values *adaptively*. We observe that the CPRF construction of [10] can be proven secure only if the adversary is not just forced to declare the challenge input, but also is bound to make all the constrained key queries *prior to setting up the system*. To address the security limitation of the CPRF construction of [10], we carefully modify their construction by innovating new technical ideas, which might be useful elsewhere, and suitably redesign the security proof. For building our improved CPRF system, we additionally use a somewhere statistically binding (SSB) hash function [18, 22] beyond the cryptographic tools used in [10]. Currently, efficient constructions of SSB hash based on standard number theoretic assumptions exist [22]. In effect, our modified CPRF stands out to be the *first* IO-based provably selectively secure CPRF for TM constraints that can handle inputs of arbitrary polynomial length.

– Secondly, we enhance our construction of CPRF with verifiability and key delegation features, thereby, developing the *first* IO-based selectively secure constructions of CVPRF and DCPRF supporting inputs of *unconstrained* polynomial length. Towards achieving these two augmentations of our CPRF, we only assume the existence of a perfectly correct and chosen plaintext attack (CPA) secure public key encryption scheme, which is evidently a minimal assumption. Finally, we note that following [12, 9], our CVPRF construction would imply the *first* selectively unforgeable *policy-based signature* (PBS) scheme [5] where policies are represented as Turing machines.

2 Preliminaries

Here we give the necessary background on various cryptographic primitives we will be using throughout this paper. Let $\lambda \in \mathbb{N}$ denotes the security parameter. For $n \in \mathbb{N}$ and $a, b \in \mathbb{N} \cup \{0\}$ (with $a < b$), we let $[n] = \{1, \ldots, n\}$ and $[a, b] = \{a, \ldots, b\}$. For any set S, $v \xleftarrow{\$} S$ represents the uniform random variable on S. For a randomized algorithm $\mathcal{R}$, we denote by $\psi = \mathcal{R}(v; \rho)$ the random variable defined by the output of $\mathcal{R}$ on input v and randomness ρ, while $\psi \xleftarrow{\$} \mathcal{R}(v)$ has the same meaning with the randomness suppressed. Also, if $\mathcal{R}$ is a deterministic algorithm $\psi = \mathcal{R}(v)$ denotes the output of $\mathcal{R}$ on input v. We will use the alternative notation $\mathcal{R}(v) \to \psi$ as well to represent the output of the algorithm $\mathcal{R}$, whether randomized or deterministic, on input v. For any string $s \in \{0,1\}^*$, $|s|$ represents the length of the string s. For any two strings $s, s' \in \{0,1\}^*$, $s \| s'$ represents the concatenation of s and s'.

2.1 Turing Machines

A Turing machine (TM) M is a 7-tuple $M = \langle Q, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \delta, q_0, q_{\text{AC}}, q_{\text{REJ}} \rangle$ with the following semantics:

– Q: The finite set of possible states of M.
– Σ_{INP}: The finite set of input symbols.

- Σ_{TAPE}: The finite set of tape symbols such that $\Sigma_{\text{INP}} \subset \Sigma_{\text{TAPE}}$ and there exists a special blank symbol '$\_$' $\in \Sigma_{\text{TAPE}} \backslash \Sigma_{\text{INP}}$.
- $\delta : Q \times \Sigma_{\text{TAPE}} \to Q \times \Sigma_{\text{TAPE}} \times \{+1, -1\}$: The transition function of M.
- $q_0 \in Q$: The designated start state.
- $q_{\text{AC}} \in Q$: The designated accept state.
- $q_{\text{REJ}} (\neq q_{\text{AC}}) \in Q$: The distinguished reject state.

For any $t \in [T = 2^\lambda]$, we define the following variables for M, while running on some input (without the explicit mention of the input in the notations):

- $\text{POS}_{M,t}$: An integer which denotes the position of the header of M after the t^{th} step. Initially, $\text{POS}_{M,0} = 0$.
- $\text{SYM}_{M,t} \in \Sigma_{\text{TAPE}}$: The symbol stored on the tape at the $\text{POS}_{M,t}{}^{\text{th}}$ location.
- $\text{SYM}_{M,t}^{(\text{WRITE})} \in \Sigma_{\text{TAPE}}$: The symbol to be written at the $\text{POS}_{M,t-1}{}^{\text{th}}$ location during the t^{th} step.
- $\text{ST}_{M,t} \in Q$: The state of M after the t^{th} step. Initially, $\text{ST}_{M,0} = q_0$.

At each time step, theTM M reads the tape at the header position and based on the current state, computes what needs to be written on the tape at the current header location, the next state, and whether the header must move left or right. More formally, let $(q, \zeta, \beta \in \{+1, -1\}) = \delta(\text{ST}_{M,t-1}, \text{SYM}_{M,t-1})$. Then, $\text{ST}_{M,t} = q, \text{SYM}_{M,t}^{(\text{WRITE})} = \zeta$, and $\text{POS}_{M,t} = \text{POS}_{M,t-1} + \beta$. M accepts at time t if $\text{ST}_{M,t} = q_{\text{AC}}$. In this paper we consider $\Sigma_{\text{INP}} = \{0, 1\}$ and $\Sigma_{\text{TAPE}} = \{0, 1, \_\}$. Given any TM M and string $x \in \{0, 1\}^*$, we define $M(x) = 1$, if M accepts x within T steps, and 0, otherwise.

2.2 Indistinguishability Obfuscation

Definition 2.1 (Indistinguishability Obfuscation: IO [14]). An indistinguishability obfuscator (IO) $\mathcal{IO}$ for a certain circuit class $\{\mathbb{C}_\lambda\}_\lambda$ is a probabilistic polynomial-time (PPT) uniform algorithm satisfying the following conditions:

- **Correctness**: $\mathcal{IO}(1^\lambda, C)$ preserves the functionality of the input circuit C, i.e., for any $C \in \mathbb{C}_\lambda$, if we compute $C' = \mathcal{IO}(1^\lambda, C)$, then $C'(\upsilon) = C(\upsilon)$ for all inputs υ.
- **Indistinguishability**: For any security parameter λ and any two circuits $C_0, C_1 \in \mathbb{C}_\lambda$ with same functionality, the circuits $\mathcal{IO}(1^\lambda, C_0)$ and $\mathcal{IO}(1^\lambda, C_1)$ are computationally indistinguishable. More precisely, for all (not necessarily uniform) PPT adversaries $\mathcal{D} = (\mathcal{D}_1, \mathcal{D}_2)$, there exists a negligible function negl such that, if

$$\mathsf{Pr}\Big[(C_0, C_1, \xi) \xleftarrow{\$} \mathcal{D}_1(1^\lambda) \ : \ \forall\, \upsilon, C_0(\upsilon) = C_1(\upsilon)\Big] \geq 1 - \mathsf{negl}(\lambda),$$

$$\text{then } \Big|\mathsf{Pr}\Big[\mathcal{D}_2(\xi, \mathcal{IO}(1^\lambda, C_0)) = 1\Big] - \mathsf{Pr}\Big[\mathcal{D}_2(\xi, \mathcal{IO}(1^\lambda, C_1)) = 1\Big]\Big| \leq \mathsf{negl}(\lambda).$$

When clear from the context, we will drop 1^λ as an input to $\mathcal{IO}$ and λ as a subscript of $\mathbb{C}$.

2.3 IO-Compatible Cryptographic Primitives

In this section, we present the syntax and correctness requirement of certain IO-friendly cryptographic tools which we will be using in the sequel. The security properties of these primitives can be found in the full version of this paper or in the references provided in the respective subsections below.

2.3.1 Puncturable Pseudorandom Function

Definition 2.2 (Puncturable Pseudorandom Function: PPRF [23]). A puncturable pseudorandom function (PPRF) $\mathcal{F} : \mathcal{K}_{\text{PPRF}} \times \mathcal{X}_{\text{PPRF}} \rightarrow \mathcal{Y}_{\text{PPRF}}$ consists of an additional punctured key space $\mathcal{K}_{\text{PPRF-PUNC}}$ other than the usual key space $\mathcal{K}_{\text{PPRF}}$ and PPT algorithms ($\mathcal{F}$.Setup, $\mathcal{F}$.Eval, $\mathcal{F}$.Puncture, $\mathcal{F}$.Eval-Punctured) described below. Here, $\mathcal{X}_{\text{PPRF}} = \{0,1\}^{\ell_{\text{PPRF-INP}}}$ and $\mathcal{Y}_{\text{PPRF}} = \{0,1\}^{\ell_{\text{PPRF-OUT}}}$, where $\ell_{\text{PPRF-INP}}$ and $\ell_{\text{PPRF-OUT}}$ are polynomials in the security parameter λ,

- $\mathcal{F}.\mathsf{Setup}(1^\lambda) \rightarrow K$: The setup authority takes as input the security parameter 1^λ and uniformly samples a PPRF key $K \in \mathcal{K}_{\text{PPRF}}$.
- $\mathcal{F}.\mathsf{Eval}(K, x) \rightarrow r$: The setup authority takes as input a PPRF key $K \in \mathcal{K}_{\text{PPRF}}$ along with an input $x \in \mathcal{X}_{\text{PPRF}}$. It outputs the PPRF value $r \in \mathcal{Y}_{\text{PPRF}}$ on x. For simplicity, we will represent by $\mathcal{F}(K, x)$ the output of this algorithm.
- $\mathcal{F}.\mathsf{Puncture}(K, x) \rightarrow K\{x\}$: Taking as input a PPRF key $K \in \mathcal{K}_{\text{PPRF}}$ along with an element $x \in \mathcal{X}_{\text{PPRF}}$, the setup authority outputs a punctured key $K\{x\} \in \mathcal{K}_{\text{PPRF-PUNC}}$.
- $\mathcal{F}.\mathsf{Eval\text{-}Puncured}(K\{x\}, x') \rightarrow r$ or $\perp$: An evaluator takes as input a punctured key $K\{x\} \in \mathcal{K}_{\text{PPRF-PUNC}}$ along with an input $x' \in \mathcal{X}_{\text{PPRF}}$. It outputs either a value $r \in \mathcal{Y}_{\text{PPRF}}$ or a distinguished symbol $\perp$ indicating failure. For simplicity, we will represent by $\mathcal{F}(K\{x\}, x')$ the output of this algorithm.

The algorithms $\mathcal{F}$.Setup and $\mathcal{F}$.Puncture are randomized, whereas, the algorithms $\mathcal{F}$.Eval and $\mathcal{F}$.Eval-Punctured are deterministic.

► **Correctness under Puncturing**: Consider any security parameter λ, $K \in \mathcal{K}_{\text{PPRF}}$, $x \in \mathcal{X}_{\text{PPRF}}$, and $K\{x\} \xleftarrow{\$} \mathcal{F}.\mathsf{Puncture}(K, x)$. Then it must hold that

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

2.3.2 Somewhere Statistically Binding Hash Function

Definition 2.3 (Somewhere Statistically Binding Hash Function: SSB [18, 22]). A somewhere statistically binding (SSB) hash consists of PPT algorithms (SSB.Gen, $\mathcal{H}$, SSB.Open,SSB.Verify) along with a block alphabet $\Sigma_{\text{SSB-BLK}} = \{0,1\}^{\ell_{\text{SSB-BLK}}}$, output size $\ell_{\text{SSB-HASH}}$, and opening space $\Pi_{\text{SSB}} = \{0,1\}^{\ell_{\text{SSB-OPEN}}}$, where $\ell_{\text{SSB-BLK}}, \ell_{\text{SSB-HASH}}, \ell_{\text{SSB-OPEN}}$ are some polynomials in the security parameter λ. The algorithms have the following syntax:

- $\mathsf{SSB.Gen}(1^\lambda, n_{\text{SSB-BLK}}, i^*) \rightarrow \text{HK}$: The setup authority takes as input the security parameter 1^λ, an integer $n_{\text{SSB-BLK}} \leq 2^\lambda$ representing the maximum number of blocks that can be hashed, and an index $i^* \in [0, n_{\text{SSB-BLK}} - 1]$ and publishes a public hashing key HK.

$\mathcal{H}_{\text{HK}} : x \in \Sigma_{\text{SSB-BLK}}^{n_{\text{SSB-BLK}}} \to h \in \{0,1\}^{\ell_{\text{SSB-HASH}}}$: This is a deterministic function that has the hash key HK hardwired. A user runs this function on input $x = x_0 \| \ldots \| x_{n_{\text{SSB-BLK}}-1} \in \Sigma_{\text{SSB-BLK}}^{n_{\text{SSB-BLK}}}$ to obtain as output $h = \mathcal{H}_{\text{HK}}(x) \in \{0,1\}^{\ell_{\text{SSB-HASH}}}$.

$\mathsf{SSB.Open}(\text{HK}, x, i) \to \pi_{\text{SSB}}$: Taking as input the hash key HK, input $x \in \Sigma_{\text{SSB-BLK}}^{n_{\text{SSB-BLK}}}$, and an index $i \in [0, n_{\text{SSB-BLK}} - 1]$, a user creates an opening $\pi_{\text{SSB}} \in \Pi_{\text{SSB}}$.

$\mathsf{SSB.Verify}(\text{HK}, h, i, u, \pi_{\text{SSB}}) \to \hat{\beta} \in \{0,1\}$: On input a hash key HK, a hash value $h \in \{0,1\}^{\ell_{\text{SSB-HASH}}}$, an index $i \in [0, n_{\text{SSB-BLK}} - 1]$, a value $u \in \Sigma_{\text{SSB-BLK}}$, and an opening $\pi_{\text{SSB}} \in \Pi_{\text{SSB}}$, a verifier outputs a bit $\hat{\beta} \in \{0,1\}$.

The algorithms $\mathsf{SSB.Gen}$ and $\mathsf{SSB.Open}$ are randomized, while the algorithm $\mathsf{SSB.Verify}$ is deterministic.

► **Correctness**: For any security parameter λ, integer $n_{\text{SSB-BLK}} \leq 2^\lambda$, $i, i^* \in [0, n_{\text{SSB-BLK}} - 1]$, $\text{HK} \xleftarrow{\$} \mathsf{SSB.Gen}(1^\lambda, n_{\text{SSB-BLK}}, i^*)$, $x \in \Sigma_{\text{SSB-BLK}}^{n_{\text{SSB-BLK}}}$, and $\pi_{\text{SSB}} \xleftarrow{\$} \mathsf{SSB.Open}(\text{HK}, x, i)$, we have $\mathsf{SSB.Verify}(\text{HK}, \mathcal{H}_{\text{HK}}(x), i, x_i, \pi_{\text{SSB}}) = 1$.

2.3.3 Positional Accumulator

Definition 2.4 (Positional Accumulator [20, 22]). A positional accumulator consists of PPT algorithms ($\mathsf{ACC.Setup}$, $\mathsf{ACC.Setup\text{-}Enforce\text{-}Read}$, $\mathsf{ACC.Setup\text{-}Enforce\text{-}Write}$, $\mathsf{ACC.Prep\text{-}Read}$, $\mathsf{ACC.Prep\text{-}Write}$, $\mathsf{ACC.Verify\text{-}Read}$, $\mathsf{ACC.Write\text{-}Store}$, $\mathsf{ACC.Update}$) along with a block alphabet $\Sigma_{\text{ACC-BLK}} = \{0,1\}^{\ell_{\text{ACC-BLK}}}$, accumulator size $\ell_{\text{ACC-ACCUMULATE}}$, proof space $\Pi_{\text{ACC}} = \{0,1\}^{\ell_{\text{ACC-PROOF}}}$ where $\ell_{\text{ACC-BLK}}, \ell_{\text{ACC-ACCUMULATE}}$, $\ell_{\text{ACC-PROOF}}$ are some polynomials in the security parameter λ. The algorithms have the following syntax:

$\mathsf{ACC.Setup}(1^\lambda, n_{\text{ACC-BLK}}) \to (\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0)$: The setup authority takes as input the security parameter 1^λ and an integer $n_{\text{ACC-BLK}} \leq 2^\lambda$ representing the maximum number of blocks that can be accumulated. It outputs the public parameters PP_{ACC}, an initial accumulator value w_0, and an initial storage value STORE_0.

$\mathsf{ACC.Setup\text{-}Enforce\text{-}Read}(1^\lambda, n_{\text{ACC-BLK}}, ((x_1, i_1), \ldots, (x_\kappa, i_\kappa)), i^*) \to (\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0)$: Taking as input the security parameter 1^λ, an integer $n_{\text{ACC-BLK}} \leq 2^\lambda$ representing the maximum number of blocks that can be accumulated, a sequence of symbol-index pairs $((x_1, i_1), \ldots, (x_\kappa, i_\kappa)) \in (\Sigma_{\text{ACC-BLK}} \times [0, n_{\text{ACC-BLK}} - 1])^\kappa$, and an additional index $i^* \in [0, n_{\text{ACC-BLK}} - 1]$, the setup authority publishes the public parameters PP_{ACC}, an initial accumulator value w_0, together with an initial storage value STORE_0.

$\mathsf{ACC.Setup\text{-}Enforce\text{-}Write}(1^\lambda, n_{\text{ACC-BLK}}, ((x_1, i_1), \ldots, x_\kappa, i_\kappa))) \to (\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0)$: On input the security parameter 1^λ, an integer $n_{\text{ACC-BLK}} \leq 2^\lambda$ denoting the maximum number of blocks that can be accumulated, and a sequence of symbol-index pairs $((x_1, i_1), \ldots, (x_\kappa, i_\kappa)) \in (\Sigma_{\text{ACC-BLK}} \times [0, n_{\text{ACC-BLK}} - 1])^\kappa$, the setup authority publishes the public parameters PP_{ACC}, an initial accumulator value w_0, as well as, an initial storage value STORE_0.

$\mathsf{ACC.Prep\text{-}Read}(\text{PP}_{\text{ACC}}, \text{STORE}_{\text{IN}}, i_{\text{IN}}) \to (x_{\text{OUT}}, \pi_{\text{ACC}})$: A storage-maintaining party takes as input the public parameter PP_{ACC}, a storage value STORE_{IN}, and an

index $i_{\text{IN}} \in [0, n_{\text{ACC-BLK}} - 1]$. It outputs a symbol $x_{\text{OUT}} \in \Sigma_{\text{ACC-BLK}} \cup \{\epsilon\}$ (ϵ being the empty string) and a proof $\pi_{\text{ACC}} \in \Pi_{\text{ACC}}$.

$\mathsf{ACC.Prep\text{-}Write}(\text{PP}_{\text{ACC}}, \text{STORE}_{\text{IN}}, i_{\text{IN}}) \rightarrow \text{AUX}$: Taking as input the public parameter PP_{ACC}, a storage value STORE_{IN}, together with an index $i_{\text{IN}} \in [0, n_{\text{ACC-BLK}} - 1]$, a storage-maintaining party outputs an auxiliary value AUX.

$\mathsf{ACC.Verify\text{-}Read}(\text{PP}_{\text{ACC}}, w_{\text{IN}}, x_{\text{IN}}, i_{\text{IN}}, \pi_{\text{ACC}}) \rightarrow \hat{\beta} \in \{0,1\}$: A verifier takes as input the public parameter PP_{ACC}, an accumulator value $w_{\text{IN}} \in \{0,1\}^{\ell_{\text{ACC-ACCUMULATE}}}$, a symbol $x_{\text{IN}} \in \Sigma_{\text{ACC-BLK}} \cup \{\epsilon\}$, an index $i_{\text{IN}} \in [0, n_{\text{ACC-BLK}} - 1]$, and a proof $\pi_{\text{ACC}} \in \Pi_{\text{ACC}}$. It outputs a bit $\hat{\beta} \in \{0,1\}$.

$\mathsf{ACC.Write\text{-}Store}(\text{PP}_{\text{ACC}}, \text{STORE}_{\text{IN}}, i_{\text{IN}}, x_{\text{IN}}) \rightarrow \text{STORE}_{\text{OUT}}$: On input the public parameters PP_{ACC}, a storage value STORE_{IN}, an index $i_{\text{IN}} \in [0, n_{\text{ACC-BLK}} - 1]$, and a symbol $x_{\text{IN}} \in \Sigma_{\text{ACC-BLK}}$, a storage-maintaining party computes a new storage value $\text{STORE}_{\text{OUT}}$.

$\mathsf{ACC.Update}(\text{PP}_{\text{ACC}}, w_{\text{IN}}, x_{\text{IN}}, i_{\text{IN}}, \text{AUX}) \rightarrow w_{\text{OUT}}$ or $\perp$: An accumulator-updating party takes as input the public parameters PP_{ACC}, an accumulator value $w_{\text{IN}} \in \{0,1\}^{\ell_{\text{ACC-ACCUMULATE}}}$, a symbol $x_{\text{IN}} \in \Sigma_{\text{ACC-BLK}}$, an index $i_{\text{IN}} \in [0, n_{\text{ACC-BLK}} - 1]$, and an auxiliary value AUX. It outputs the updated accumulator value $w_{\text{OUT}} \in \{0,1\}^{\ell_{\text{ACC-ACCUMULATE}}}$ or the designated reject string $\perp$.

Following [20,10], in this paper we will consider the algorithms ACC.Setup, ACC.Setup-Enforce-Read, and ACC.Setup-Enforce-Write as randomized while all other algorithms as deterministic.

► **Correctness**: Consider any symbol-index pair sequence $((x_1, i_1), \ldots, (x_\kappa, i_\kappa)) \in (\Sigma_{\text{ACC-BLK}} \times [0, n_{\text{ACC-BLK}} - 1])^\kappa$. Fix any $(\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0) \xleftarrow{\$} \mathsf{ACC.Setup}(1^\lambda, n_{\text{ACC-BLK}})$. For $j = 1, \ldots, \kappa$, iteratively define the following:

- $\text{STORE}_j = \mathsf{ACC.Write\text{-}Store}(\text{PP}_{\text{ACC}}, \text{STORE}_{j-1}, i_j, x_j)$
- $\text{AUX}_j = \mathsf{ACC.Prep\text{-}Write}(\text{PP}_{\text{ACC}}, \text{STORE}_{j-1}, i_j)$
- $w_j = \mathsf{ACC.Update}(\text{PP}_{\text{ACC}}, w_{j-1}, x_j, i_j, \text{AUX}_j)$

The following correctness properties are required to be satisfied:

i) For any security parameter λ, $n_{\text{ACC-BLK}} \leq 2^\lambda$, index $i^* \in [0, n_{\text{ACC-BLK}} - 1]$, sequence of symbol-index pairs $((x_1, i_1), \ldots, (x_\kappa, i_\kappa)) \in (\Sigma_{\text{ACC-BLK}} \times [0, n_{\text{ACC-BLK}} - 1])^\kappa$, and $(\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0) \xleftarrow{\$} \mathsf{ACC.Setup}(1^\lambda, n_{\text{ACC-BLK}})$, if STORE_κ is computed as above, then $\mathsf{ACC.Prep\text{-}Read}(\text{PP}_{\text{ACC}}, \text{STORE}_\kappa, i^*)$ returns (x_j, π_{ACC}) where j is the largest value in $[\kappa]$ such that $i_j = i^*$.

ii) For any security parameter λ, $n_{\text{ACC-BLK}} \leq 2^\lambda$, sequence of symbol-index pairs $((x_1, i_1), \ldots, (x_\kappa, i_\kappa)) \in (\Sigma_{\text{ACC-BLK}} \times [0, n_{\text{ACC-BLK}} - 1])^\kappa$, $i^* \in [0, n_{\text{ACC-BLK}} - 1]$, and $(\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0) \xleftarrow{\$} \mathsf{ACC.Setup}(1^\lambda, n_{\text{ACC-BLK}})$, if STORE_κ and w_κ are computed as above and $(x_{\text{OUT}}, \pi_{\text{ACC}}) = \mathsf{ACC.Prep\text{-}Read}(\text{PP}_{\text{ACC}}, \text{STORE}_\kappa, i^*)$, then $\mathsf{ACC.Verify\text{-}Read}(\text{PP}_{\text{ACC}}, w_\kappa, x_{\text{OUT}}, i^*, \pi_{\text{ACC}}) = 1$

2.3.4 Iterator

Definition 2.5 (Iterator [20]). A cryptographic iterator consists of PPT algorithms (ITR.Setup, ITR.Set-Enforce, ITR.Iterate) along with a message space

$\mathcal{M}_{\text{ITR}} = \{0,1\}^{\ell_{\text{ITR-MSG}}}$ and iterator state size $\ell_{\text{ITR-ST}}$, where $\ell_{\text{ITR-MSG}}, \ell_{\text{ITR-ST}}$ are some polynomials in the security parameter λ. Algorithms have the following syntax:

- $\mathsf{ITR.Setup}(1^\lambda, n_{\text{ITR}}) \to (\text{PP}_{\text{ITR}}, v_0)$: The setup authority takes as input the security parameter 1^λ along with an integer bound $n_{\text{ITR}} \leq 2^\lambda$ on the number of iterations. It outputs the public parameters PP_{ITR} and an initial state $v_0 \in \{0,1\}^{\ell_{\text{ITR-ST}}}$.
- $\mathsf{ITR.Setup\text{-}Enforce}(1^\lambda, n_{\text{ITR}}, (\mu_1, \ldots, \mu_\kappa)) \to (\text{PP}_{\text{ITR}}, v_0)$: Taking as input the security parameter 1^λ, an integer bound $n_{\text{ITR}} \leq 2^\lambda$, together with a sequence of κ messages $(\mu_1, \ldots, \mu_\kappa) \in \mathcal{M}_{\text{ITR}}^\kappa$, where $\kappa \leq n_{\text{ITR}}$, the setup authority publishes the public parameters PP_{ITR} and an initial state $v_0 \in \{0,1\}^{\ell_{\text{ITR-ST}}}$.
- $\mathsf{ITR.Iterate}(\text{PP}_{\text{ITR}}, v_{\text{IN}} \in \{0,1\}^{\ell_{\text{ITR-ST}}}, \mu) \to v_{\text{OUT}}$: On input the public parameters PP_{ITR}, a state v_{IN}, and a message $\mu \in \mathcal{M}_{\text{ITR}}$, an iterator outputs an updated state $v_{\text{OUT}} \in \{0,1\}^{\ell_{\text{ITR-ST}}}$. For any integer $\kappa \leq n_{\text{ITR}}$, we will write $\mathsf{ITR.Iterate}^\kappa(\text{PP}_{\text{ITR}}, v_0, (\mu_1, \ldots, \mu_\kappa))$ to denote $\mathsf{ITR.Iterate}(\text{PP}_{\text{ITR}}, v_{\kappa-1}, \mu_\kappa)$, where v_j is defined iteratively as $v_j = \mathsf{ITR.Iterate}(\text{PP}_{\text{ITR}}, v_{j-1}, \mu_j)$ for all $j = 1, \ldots, \kappa - 1$.

The algorithm $\mathsf{ITR.Iterate}$ is deterministic, while the other two are randomized.

2.3.5 Splittable Signature

Definition 2.6 (Splittable Signature: SPS [20]). A splittable signature scheme (SPS) for message space $\mathcal{M}_{\text{SPS}} = \{0,1\}^{\ell_{\text{SPS-MSG}}}$ and signature space $\mathcal{S}_{\text{SPS}} = \{0,1\}^{\ell_{\text{SPS-SIG}}}$, where $\ell_{\text{SPS-MSG}}, \ell_{\text{SPS-SIG}}$ are some polynomials in the security parameter λ, consists of PPT algorithms ($\mathsf{SPS.Setup}$, $\mathsf{SPS.Sign}$, $\mathsf{SPS.Verify}$, $\mathsf{SPS.Split}$, $\mathsf{SPS.Sign\text{-}ABO}$) which are described below:

- $\mathsf{SPS.Setup}(1^\lambda) \to (\text{SK}_{\text{SPS}}, \text{VK}_{\text{SPS}}, \text{VK}_{\text{SPS-REJ}})$: The setup authority takes as input the security parameter 1^λ and generates a signing key SK_{SPS}, a verification key VK_{SPS}, together with a reject verification key $\text{VK}_{\text{SPS-REJ}}$.
- $\mathsf{SPS.Sign}(\text{SK}_{\text{SPS}}, m) \to \sigma_{\text{SPS}}$: A signer given a signing key SK_{SPS} along with a message $m \in \mathcal{M}_{\text{SPS}}$, produces a signature $\sigma_{\text{SPS}} \in \mathcal{S}_{\text{SPS}}$.
- $\mathsf{SPS.Verify}(\text{VK}_{\text{SPS}}, m, \sigma_{\text{SPS}}) \to \hat{\beta} \in \{0,1\}$: A verifier takes as input a verification key VK_{SPS}, a message $m \in \mathcal{M}_{\text{SPS}}$, and a signature $\sigma_{\text{SPS}} \in \mathcal{S}_{\text{SPS}}$. It outputs a bit $\hat{\beta} \in \{0,1\}$.
- $\mathsf{SPS.Split}(\text{SK}_{\text{SPS}}, m^*) \to (\sigma_{\text{SPS-ONE},m^*}, \text{VK}_{\text{SPS-ONE}}, \text{SK}_{\text{SPS-ABO}}, \text{VK}_{\text{SPS-ABO}})$: On input a signing key SK_{SPS} along with a message $m^* \in \mathcal{M}_{\text{SPS}}$, the setup authority generates a signature $\sigma_{\text{SPS-ONE},m^*} = \mathsf{SPS.Sign}(\text{SK}_{\text{SPS}}, m^*)$, a one-message verification key $\text{VK}_{\text{SPS-ONE}}$, and all-but-one signing-verification key pair $(\text{SK}_{\text{SPS-ABO}}, \text{VK}_{\text{SPS-ABO}})$.
- $\mathsf{SPS.Sign\text{-}ABO}(\text{SK}_{\text{SPS-ABO}}, m) \to \sigma_{\text{SPS}}$ or $\perp$: An all-but-one signer given an all-but-one signing key $\text{SK}_{\text{SPS-ABO}}$ and a message $m \in \mathcal{M}_{\text{SPS}}$, outputs a signature $\sigma_{\text{SPS}} \in \mathcal{S}_{\text{SPS}}$ or a distinguished string $\perp$ to indicate failure. For simplicity of notation, we will often use $\mathsf{SPS.Sign}(\text{SK}_{\text{SPS-ABO}}, m)$ to represent the output of this algorithm.

We note that among the algorithms described above, SPS.Setup and SPS.Split are randomized while all the others are deterministic.

► **Correctness**: For any security parameter λ, message $m^* \in \mathcal{M}_{\text{SPS}}$, $(\text{SK}_{\text{SPS}}, \text{VK}_{\text{SPS}}, \text{VK}_{\text{SPS-REJ}}) \xleftarrow{\$} \mathsf{SPS.Setup}(1^\lambda)$, and $(\sigma_{\text{SPS-ONE},m^*}, \text{VK}_{\text{SPS-ONE}}, \text{SK}_{\text{SPS-ABO}}, \text{VK}_{\text{SPS-ABO}}) \xleftarrow{\$} \mathsf{SPS.Split}(\text{SK}_{\text{SPS}}, m^*)$ the following correctness conditions hold:

i) $\forall\, m \in \mathcal{M}_{\text{SPS}},\ \mathsf{SPS.Verify}(\text{VK}_{\text{SPS}}, m, \mathsf{SPS.Sign}(\text{SK}_{\text{SPS}}, m)) = 1.$
ii) $\forall\, m \neq m^* \in \mathcal{M}_{\text{SPS}},\ \mathsf{SPS.Sign}(\text{SK}_{\text{SPS}}, m) = \mathsf{SPS.Sign\text{-}ABO}(\text{SK}_{\text{SPS-ABO}}, m).$
iii) $\forall\, \sigma_{\text{SPS}} \in \mathcal{S}_{\text{SPS}},\ \mathsf{SPS.Verify}(\text{VK}_{\text{SPS-ONE}}, m^*, \sigma_{\text{SPS}}) = \mathsf{SPS.Verify}(\text{VK}_{\text{SPS}}, m^*, \sigma_{\text{SPS}}).$
iv) $\forall\, m \neq m^* \in \mathcal{M}_{\text{SPS}},\ \sigma_{\text{SPS}} \in \mathcal{S}_{\text{SPS}},$
$\mathsf{SPS.Verify}(\text{VK}_{\text{SPS-ABO}}, m, \sigma_{\text{SPS}}) = \mathsf{SPS.Verify}(\text{VK}_{\text{SPS}}, m, \sigma_{\text{SPS}}).$
v) $\forall\, m \neq m^* \in \mathcal{M}_{\text{SPS}},\ \sigma_{\text{SPS}} \in \mathcal{S}_{\text{SPS}},\ \mathsf{SPS.Verify}(\text{VK}_{\text{SPS-ONE}}, m, \sigma_{\text{SPS}}) = 0.$
vi) $\forall\, \sigma_{\text{SPS}} \in \mathcal{S}_{\text{SPS}},\ \mathsf{SPS.Verify}(\text{VK}_{\text{SPS-ABO}}, m^*, \sigma_{\text{SPS}}) = 0.$
vii) $\forall\, m \in \mathcal{M}_{\text{SPS}},\ \sigma_{\text{SPS}} \in \mathcal{S}_{\text{SPS}},\ \mathsf{SPS.Verify}(\text{VK}_{\text{SPS-REJ}}, m, \sigma_{\text{SPS}}) = 0.$

3 Our CPRF for Turing Machines

3.1 Notion

Definition 3.1 (Constrained Pseudorandom Function for Turing Machines: CPRF [10]). Let $\mathbb{M}_\lambda$ be a family of TM's with (worst case) running time bounded by $T = 2^\lambda$. A constrained pseudorandom function (CPRF) with key space $\mathcal{K}_{\text{CPRF}}$, input domain $\mathcal{X}_{\text{CPRF}} \subset \{0,1\}^*$, and output space $\mathcal{Y}_{\text{CPRF}} \subset \{0,1\}^*$ for the TM family $\mathbb{M}_\lambda$ consists of an additional key space $\mathcal{K}_{\text{CPRF-CONST}}$ and PPT algorithms (CPRF.Setup, CPRF.Eval, CPRF.Constrain, CPRF.Eval-Constrained) described as follows:

- $\mathsf{CPRF.Setup}(1^\lambda) \rightarrow \text{SK}_{\text{CPRF}}$: The setup authority takes as input the security parameter 1^λ and generates the master CPRF key $\text{SK}_{\text{CPRF}} \in \mathcal{K}_{\text{CPRF}}$.
- $\mathsf{CPRF.Eval}(\text{SK}_{\text{CPRF}}, x) \rightarrow y$: On input the master CPRF key SK_{CPRF} along with an input $x \in \mathcal{X}_{\text{CPRF}}$, the setup authority computes the value of the CPRF $y \in \mathcal{Y}_{\text{CPRF}}$. For simplicity of notation, we will use $\mathsf{CPRF}(\text{SK}_{\text{CPRF}}, x)$ to indicate the output of this algorithm.
- $\mathsf{CPRF.Constrain}(\text{SK}_{\text{CPRF}}, M) \rightarrow \text{SK}_{\text{CPRF}}\{M\}$: Taking as input the master CPRF key SK_{CPRF} and a TM $M \in \mathbb{M}_\lambda$, the setup authority provides a constrained key $\text{SK}_{\text{CPRF}}\{M\} \in \mathcal{K}_{\text{CPRF-CONST}}$ to a legitimate user.
- $\mathsf{CPRF.Eval\text{-}Constrained}(\text{SK}_{\text{CPRF}}\{M\}, x) \rightarrow y$ or $\perp$: A user takes as input a constrained key $\text{SK}_{\text{CPRF}}\{M\} \in \mathcal{K}_{\text{CPRF-CONST}}$, corresponding to a legitimate TM $M \in \mathbb{M}_\lambda$, along with an input $x \in \mathcal{X}_{\text{CPRF}}$. It outputs either a value $y \in \mathcal{Y}_{\text{CPRF}}$ or $\perp$ indicating failure.

The algorithms CPRF.Setup and CPRF.Constrain are randomized, whereas, the other two are deterministic.

► **Correctness under Constraining**: Consider any security parameter λ, $\text{SK}_{\text{CPRF}} \in \mathcal{K}_{\text{CPRF}}$, $M \in \mathbb{M}_\lambda$, and $\text{SK}_{\text{CPRF}}\{M\} \xleftarrow{\$} \mathsf{CPRF.Constrain}(\text{SK}_{\text{CPRF}}, M)$. The

following must hold:

$$\mathsf{CPRF.Eval\text{-}Constrained}(\mathrm{SK}_{\mathrm{CPRF}}\{M\}, x) = \begin{cases} \mathsf{CPRF}(\mathrm{SK}_{\mathrm{CPRF}}, x), & \text{if } M(x) = 1 \\ \perp, & \text{otherwise} \end{cases}$$

► **Selective Pseudorandomness**: This property of a CPRF is defined through the following experiment between an adversary $\mathcal{A}$ and a challenger $\mathcal{B}$:

- $\mathcal{A}$ submits a challenge input $x^* \in \mathcal{X}_{\mathrm{CPRF}}$ to $\mathcal{B}$.
- $\mathcal{B}$ generates a master CPRF key $\mathrm{SK}_{\mathrm{CPRF}} \xleftarrow{\$} \mathsf{CPRF.Setup}(1^\lambda)$. Next it selects a random bit $b \xleftarrow{\$} \{0,1\}$. If $b = 0$, it computes $y^* = \mathsf{CPRF}(\mathrm{SK}_{\mathrm{CPRF}}, x^*)$. Otherwise, it chooses a random $y^* \xleftarrow{\$} \mathcal{Y}_{\mathrm{CPRF}}$. It returns y^* to $\mathcal{A}$.
- $\mathcal{A}$ may adaptively make a polynomial number of queries of the following kinds to $\mathcal{B}$:
  - **Evaluation query**: $\mathcal{A}$ queries the CPRF value at some input $x \in \mathcal{X}_{\mathrm{CPRF}}$ such that $x \neq x^*$. $\mathcal{B}$ provides the CPRF value $\mathsf{CPRF}(\mathrm{SK}_{\mathrm{CPRF}}, x)$ to $\mathcal{A}$.
  - **Key query**: $\mathcal{A}$ queries a constrained key corresponding to TM $M \in \mathbb{M}_\lambda$ subject to the constraint that $M(x^*) = 0$. $\mathcal{B}$ gives the constrained key $\mathrm{SK}_{\mathrm{CPRF}}\{M\} \xleftarrow{\$} \mathsf{CPRF.Constrain}(\mathrm{SK}_{\mathrm{CPRF}}, M)$ to $\mathcal{A}$
- $\mathcal{A}$ eventually outputs a guess bit $b' \in \{0,1\}$.

The CPRF is said to be selectively pseudorandom if for any PPT adversary $\mathcal{A}$, for any security parameter λ,

$$\mathsf{Adv}_{\mathcal{A}}^{\mathrm{CPRF,SEL\text{-}PR}}(\lambda) = |\mathsf{Pr}[b = b'] - 1/2| \leq \mathsf{negl}(\lambda)$$

for some negligible function negl.

Remark 3.1. As pointed out in [16, 9], note that in the above selective pseudorandomness experiment, without loss of generality we may assume that the adversary $\mathcal{A}$ only makes constrained key queries and no evaluation query. This is because any evaluation query at input $x \in \mathcal{X}_{\mathrm{CPRF}}$ can be replaced by constrained key query for a TM $M_x \in \mathbb{M}_\lambda$ that accepts only x. Since, the restriction on the evaluation queries is that $x \neq x^*$, $M_x(x^*) = 0$, and thus M_x is a valid constrained key query. We will use this simplification in our proof.

3.2 The CPRF Construction of Deshpande et al.

In EUROCRYPT 2016, Deshpande et al. [10] presented a CPRF construction supporting inputs of unconstrained polynomial length based on indistinguishability obfuscation and injective pseudorandom generators, which they claimed to be selectively secure. Unfortunately, their security argument has a flaw. In this section, we give an informal description of their CPRF construction and point out the flaw in their security argument.

Overview of the CPRF Construction of [10]: The principle ideas behind the CPRF construction of [10] are as follows: To produce the CPRF output their construction uses a PPRF $\mathcal{F}$ and a positional accumulator. A master CPRF key consists of a key K for the PPRF $\mathcal{F}$ and a set of public parameters $\mathrm{PP}_{\mathrm{ACC}}$ of the positional accumulator. The CPRF evaluation on some input $x = x_0 \ldots x_{\ell_x - 1} \in$

$\mathcal{X}_{\text{CPRF}} \subset \{0,1\}^*$ is simply $\mathcal{F}(K, w_{\text{INP}})$, where w_{INP} is the accumulation of the bits of x using PP_{ACC}.

A constrained key of the CPRF, corresponding to some TM M, comprises of PP_{ACC} along with two programs $\mathcal{P}_1$ and $\mathcal{P}_{\text{CPRF}}$, which are obfuscated using IO. The first program $\mathcal{P}_1$, also known as the *initial signing program*, takes as input an accumulator value and outputs a signature on it together with the initial state and header position of the TM M. The second program $\mathcal{P}_{\text{CPRF}}$, also called the *next step program*, takes as input a state and header position of M along with an input symbol and an accumulator value. It essentially computes the next step function of M on the input state-symbol pair, and eventually outputs the proper PRF value, if M reaches the accepting state. The program $\mathcal{P}_{\text{CPRF}}$ also performs certain authenticity checks before computing the next step function of M in order to prevent illegal inputs. For this purpose, $\mathcal{P}_{\text{CPRF}}$ additionally takes as input a signature on the input state, header position, and accumulator value, together with a proof for the positional accumulator. The program $\mathcal{P}_{\text{CPRF}}$ verifies the signature as well as checks the accumulator proof to get convinced that the input symbol is indeed the one placed at the input header position of the underlying storage of the input accumulator value. If all these verifications pass, then $\mathcal{P}_{\text{CPRF}}$ determines the next state and header position of M, as well as, the new symbol that needs to be written to the input header position. The program $\mathcal{P}_{\text{CPRF}}$ then updates the accumulator value by placing the new symbol at the input header position as well as signs the updated accumulator value along with the computed next state and header position of M. The signature scheme used by the two programs is a splittable signature. In order to deal with the positional accumulator related verifications and updations, the program $\mathcal{P}_{\text{CPRF}}$ has PP_{ACC} hardwired.

Evaluating the CPRF on some input x using a constrained key, corresponding to some TM M, consists of two steps. In the first step, the evaluator computes the accumulation w_{INP} of the bits of x using PP_{ACC}, which are also included in the constrained key, and then obtains a signature on w_{INP} together with the initial state and header position of M by running the program $\mathcal{P}_1$. The second step is to repeatedly run the program $\mathcal{P}_{\text{CPRF}}$, each time on input the current accumulator value, current state and header position of M, along with the signature on them. Additionally, in each iteration the evaluator also feeds w_{INP} to $\mathcal{P}_{\text{CPRF}}$. The iteration is continued until the program $\mathcal{P}_{\text{CPRF}}$ either outputs the PRF evaluation or the designated null string $\perp$ indicating failure.

The Flaw: In order to prove selective pseudorandomness of the above CPRF construction, the authors of [10] extends the techniques introduced in [20] in the context of proving security of message-hiding encoding scheme for TM's. More precisely, the authors of [10] proceed as follows: During the course of the proof, the authors aim to modify the constrained keys given to the adversary $\mathcal{A}$ in the selective pseudorandomness experiment, discussed in Section 3.1, to embed the punctured PPRF key $K\{w^*_{\text{INP}}\}$ punctured at w^*_{INP} instead of the full PPRF key K, which is part of the master CPRF key sampled by the challenger $\mathcal{B}$. Here, w^*_{INP} is the accumulation of the bits of the challenge input x^*, submitted by

the adversary $\mathcal{A}$, using $\mathrm{PP}_{\mathrm{ACC}}$, included within the master CPRF key generated by the challenger $\mathcal{B}$. In order to make this substitution, it is to be ensured that the obfuscated next step programs included in the constrained keys never outputs the PRF evaluation for inputs corresponding to w^*_{INP} even if reaching the accepting state. The proof transforms the constrained keys one at a time through multiple hybrid steps. Suppose that the total number of constrained keys queried by $\mathcal{A}$ be $\hat{q}$. Consider the transformation of the ν^{th} constrained key ($1 \leq \nu \leq \hat{q}$) corresponding to the TM $M^{(\nu)}$ that runs on the challenge input x^* for $t^{*(\nu)}$ steps and reaches the rejecting state. In the course of transformation, the obfuscated next step program $\mathcal{P}^{(\nu)}_{\mathrm{CPRF}}$ of the ν^{th} constrained key is first altered to one that never outputs the PRF evaluation for inputs corresponding to w^*_{INP} within the first $t^{*(\nu)}$ steps. Towards accomplishing this transition, the challenger $\mathcal{B}$ at various stages needs to generate $\mathrm{PP}_{\mathrm{ACC}}$ in read/write enforcing mode where the enforcing property should be tailored to the steps of execution of the specific TM $M^{(\nu)}$ on x^*. For instance, at some point of transformation of the ν^{th} constrained key, $\mathrm{PP}_{\mathrm{ACC}}$ needs to be set in the read enforcing mode by $\mathcal{B}$ on input (i) the entire sequence of symbol-position pairs arising from iteratively running $M^{(\nu)}$ on x^* upto the t^{th} step and (ii) the enforcing index corresponding to the header position of $M^{(\nu)}$ at the t^{th} step while running on x^*, where $1 < t \leq t^{*(\nu)}$. Evidently, if $\mathcal{A}$ makes the constrained key queries adaptively, which it is allowed to do in the selective pseudorandomness experiment, then $\mathcal{B}$ can determine those symbol-position pairs *only after receiving* the ν^{th} queried TM $M^{(\nu)}$ from $\mathcal{A}$. However, $\mathcal{B}$ would also require $\mathrm{PP}_{\mathrm{ACC}}$ while creating the constrained keys queried by $\mathcal{A}$ before making the ν^{th} constrained key query and even possibly for preparing the challenge value for $\mathcal{A}$. Thus, it is immediate that $\mathcal{B}$ must generate $\mathrm{PP}_{\mathrm{ACC}}$ *prior to receiving* the ν^{th} query from $\mathcal{A}$. This is *impossible* as setting $\mathrm{PP}_{\mathrm{ACC}}$ in read enforcing mode requires the knowledge of the TM $M^{(\nu)}$, which is *not available* before the ν^{th} constrained key query of $\mathcal{A}$. A similar conflict also arises when $\mathcal{B}$ attempts to setup $\mathrm{PP}_{\mathrm{ACC}}$ in the write enforcing mode tailored to $M^{(\nu)}$. This serious flaw renders the proof of selective pseudorandomness of the CPRF construction of [10] invalid. Ofcourse, this problem would clearly not arise if the pseudorandomness of the CPRF construction of [10] is analysed in a weaker model in which the adversary $\mathcal{A}$ is forced to submit all the constrained key queries along with the challenge input at the beginning of the experiment, i.e., before the challenger $\mathcal{B}$ performs the setup. However, this weaker model is rather unrealistic as it renders the adversary $\mathcal{A}$ completely static.

3.3 Our Techniques to Fix the Flaw of [10]

Observe that a set of public parameters of the positional accumulator must be included within each constrained key. This is mandatory due to the required updatability feature of positional accumulator, which is indispensable to keep track of the current situation while running the obfuscated next step program $\mathcal{P}_{\mathrm{CPRF}}$ iteratively in the course of evaluating the CPRF on some input. The root cause of the problem in the selective security argument of [10] is the use of a single set of public parameters $\mathrm{PP}_{\mathrm{ACC}}$ of the positional accumulator throughout the system.

Therefore, as a first step, we attempt to assign a fresh set of public parameters of the positional accumulator to each constrained key. However, for compressing the PRF input to a fixed length, on which $\mathcal{F}$ can be applied producing the PRF output, we need a system-wide compressing tool. We employ SSB hash for this purpose. The idea is that while evaluating the CPRF on some input x using a constrained key, corresponding to some TM M, the evaluator first computes the hash value h by hashing x using the system wide SSB hash key, which is part of the master key. The evaluator also computes the accumulator value w_{INP} by accumulating the bits of x using the public parameters of positional accumulator included in the constrained key. Then, using the obfuscated initial signing program $\mathcal{P}_1$, included in the constrained key, the evaluator will obtain a signature on w_{INP} along with the initial state and header position of M. Finally, the evaluator will repeatedly run the obfuscated next step program $\mathcal{P}_{\text{CPRF}}$, included in the constrained key, each time giving as input all the quantities as in the evaluation algorithm of [10], except that it now feeds the SSB hash value h in place of w_{INP} in each iteration. This is because, in case $\mathcal{P}_{\text{CPRF}}$ reaches the accepting state, it would require h to apply $\mathcal{F}$ for producing the PRF output.

However, this approach is not completely sound yet. Observe that, a possibly malicious evaluator can compute the SSB hash value h on the input x, on which it wishes to evaluate the CPRF although M does not accepts it, and initiates the evaluation by accumulating the bits of only a substring of x or some entirely different input, which is accepted by M. To prevent such malicious behavior, we include another IO-obfuscated program $\mathcal{P}_2$ within the constrained key, known as the *accumulating program*, whose purpose is to *restrict* the evaluator from accumulating the bits of a different input rather than the hashed one. The program $\mathcal{P}_2$ takes as input an SSB hash value h, an index i, a symbol, an accumulator value, a signature on the input accumulator value (along with the initial state and header position of M), and an opening value for SSB. The program $\mathcal{P}_2$ verifies the signature and also checks whether the input symbol is indeed present at the index i of the string that has been hashed to form h, using the input opening value. If all of these verifications pass, then $\mathcal{P}_2$ updates the input accumulator value by writing the input symbol at the i^{th} position of the accumulator storage. We also modify the obfuscated initial signing program $\mathcal{P}_1$, included in the constrained key, to take as input a hash value and output a signature on the accumulator value corresponding to the empty accumulator storage, along with the initial state and header position of M.

Moreover, for forbidding the evaluator from performing the evaluation by accumulating an M-accepted substring of the hashed input, we define our PRF output as the evaluation of $\mathcal{F}$ on the pair (hash value, length) of the input in stead of just the hash value of the input. Note that, without loss of generality, we can set the upper bound of the length of PRF inputs to be 2^λ, where λ is the underlying security parameter in view of the fact that by suitably choosing λ we can accommodate inputs of any polynomial length. This setting of upper bound on the input length is implicitly considered in [10]. Now, as the input length is bounded by 2^λ, the input length can be expressed as a bit strings of length λ.

Thus, the PRF input length can be safely fed along with the SSB hash value of PRF input to $\mathcal{F}$, which can handle only inputs of apriori bounded length. Hence, the obfuscated next step programs $\mathcal{P}_{\text{CPRF}}$ included in our constrained keys must also take as input the length of the PRF input for producing the PRF value if reaching to the accepting state.

Therefore, to evaluate the CPRF on some input using a constrained key, corresponding to some TM M, an evaluator first hash the PRF input. The evaluator also obtains a signature on the empty accumulator value included in the constrained key, by running the obfuscated initial signing program $\mathcal{P}_1$ on input the computed hash value. Next, it repeatedly runs the obfuscated accumulating program $\mathcal{P}_2$ to accumulate the bits of the PRF input. Finally, it runs the obfuscated next step program $\mathcal{P}_{\text{CPRF}}$ iteratively on the current accumulator value along with other legitimate inputs until it obtains either the PRF output or $\perp$.

Regarding the proof of security, notice that the problem with enforcing the public parameters of the positional accumulator while transforming the queried constrained keys will not appear in our case as we have assigned a separate set of public parameters of positional accumulator to each constrained key. However, our actual security proof involves many subtleties that are difficult to describe with this high level description and is provided in full details in the sequel. We would only like to mention here that to cope up with certain issues in the proof we further include another IO-obfuscated program $\mathcal{P}_3$ in the constrained keys, known as the *signature changing program*, that changes the signature on the accumulation of the bits of the PRF input before starting the iterative computation with the obfuscated next step program $\mathcal{P}_{\text{CPRF}}$.

We follow the same novel technique introduced in [10] for handling the tail hybrids in the final stage of transformation of the constrained keys. Note that as in [10], we are also considering TM's which run for at most $T = 2^\lambda$ steps on any input. Unlike [20], the authors of [10] have devised a beautiful approach to obtain an end to end polynomial reduction to the security of IO for the tail hybrids by means of an injective pseudorandom generator (PRG). We directly adopt that technique to deal with the tail hybrids in our security proof. A high level overview of the approach is sketched below. Let us call the time step 2^τ as the τ^{th} landmark and the interval $[2^\tau, 2^{\tau+1} - 1]$ as the τ^{th} interval. Like [10], our obfuscated next step programs $\mathcal{P}_{\text{CPRF}}$ included within the constrained keys take an additional PRG seed as input at each time step, and perform some additional checks on the input PRG seed. At time steps just before a landmark, the programs output a new pseudorandomly generated PRG seed, which is then used in the next interval. Using standard IO techniques, it can be shown that for inputs corresponding to (h^*, ℓ^*), if the program $\mathcal{P}_{\text{CPRF}}$ outputs $\perp$, for all time steps upto the one just before a landmark, then we can alter the program indistinguishably so that it outputs $\perp$ at all time steps in the next interval. Here h^* and ℓ^* are respectively the SSB hash value and length of the challenge input x^* submitted by the adversary $\mathcal{A}$ in the selective pseudorandomness experiment. Employing this technique, we can move across an exponential number of time steps at a single switch of the next step program $\mathcal{P}_{\text{CPRF}}$.

3.4 Formal Description of Our CPRF

Now we will formally present our CPRF construction where the constrained keys are associated with TM's. Let λ be the underlying security parameter. Consider the family $\mathbb{M}_\lambda$ of TM's, the members of which have (worst-case) running time bounded by $T = 2^\lambda$, input alphabet $\Sigma_{\text{INP}} = \{0, 1\}$, and tape alphabet $\Sigma_{\text{TAPE}} = \{0, 1, \_\}$. Our CPRF construction utilizes the following cryptographic building blocks:

i) $\mathcal{IO}$: An indistinguishability obfuscator for general polynomial-size circuits.
ii) SSB = (SSB.Gen, $\mathcal{H}$, SSB.Open, SSB.Verify): A somewhere statistically binding hash function with $\Sigma_{\text{SSB-BLK}} = \{0, 1\}$.
iii) ACC = (ACC.Setup, ACC.Setup-Enforce-Read, ACC.Setup-Enforce-Write, ACC.Prep-Read, ACC.Prep-Write, ACC.Verify-Read, ACC.Write-Store, ACC.Update): A positional accumulator with $\Sigma_{\text{ACC-BLK}} = \{0, 1, \_\}$.
iv) ITR = (ITR.Setup, ITR.Setup-Enforce, ITR.Iterate): A cryptographic iterator with an appropriate message space $\mathcal{M}_{\text{ITR}}$
v) SPS = (SPS.Setup, SPS.Sign, SPS.Verify, SPS.Split, SPS.Sign-ABO): A splittable signature scheme with an appropriate message space $\mathcal{M}_{\text{SPS}}$.
vi) PRG : $\{0, 1\}^\lambda \to \{0, 1\}^{2\lambda}$: A length-doubling pseudorandom generator.
vii) $\mathcal{F} = (\mathcal{F}.\text{Setup}, \mathcal{F}.\text{Puncture}, \mathcal{F}.\text{Eval})$: A puncturable pseudorandom function whose domain and range are chosen appropriately. For simplicity, we assume that $\mathcal{F}$ has inputs and outputs of bounded length instead of fixed length inputs and outputs. This assumption can be easily removed by using different PPRF's for different input and output lengths.

Our CPRF construction is described below:

CPRF.Setup$(1^\lambda) \to \text{SK}_{\text{CPRF}} = (K, \text{HK})$: The setup authority takes as input the security parameter 1^λ and proceeds as follows:
1. It first chooses a PPRF key $K \xleftarrow{\$} \mathcal{F}.\text{Setup}(1^\lambda)$.
2. Next it generates $\text{HK} \xleftarrow{\$} \text{SSB.Gen}(1^\lambda, n_{\text{SSB-BLK}} = 2^\lambda, i^* = 0)$.
3. It sets the master CPRF key as $\text{SK}_{\text{CPRF}} = (K, \text{HK})$.

CPRF.Eval$(\text{SK}_{\text{CPRF}}, x) \to y = \mathcal{F}(K, (h, \ell_x))$: Taking as input the master CPRF key $\text{SK}_{\text{CPRF}} = (K, \text{HK})$ along with an input $x = x_0 \dots x_{\ell_x - 1} \in \mathcal{X}_{\text{CPRF}}$, where $|x| = \ell_x$, the setup authority executes the following steps:
1. It computes $h = \mathcal{H}_{\text{HK}}(x)$.
2. It outputs the CPRF value on input x to be $y = \mathcal{F}(K, (h, \ell_x))$.

CPRF.Constrain$(\text{SK}_{\text{CPRF}}, M) \to \text{SK}_{\text{CPRF}}\{M\} = (\text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0, \text{PP}_{\text{ITR}}, v_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{CPRF}})$: On input the master CPRF key $\text{SK}_{\text{CPRF}} = (K, \text{HK})$ and a TM $M = \langle Q, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \delta, q_0, q_{\text{AC}}, q_{\text{REJ}} \rangle \in \mathbb{M}_\lambda$, the setup authority performs the following steps:
1. At first, it selects PPRF keys $K_1, \dots, K_\lambda, K_{\text{SPS},A}, K_{\text{SPS},E} \xleftarrow{\$} \mathcal{F}.\text{Setup}(1^\lambda)$.
2. Next, it generates $(\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0) \xleftarrow{\$} \text{ACC.Setup}(1^\lambda, n_{\text{ACC-BLK}} = 2^\lambda)$ and $(\text{PP}_{\text{ITR}}, v_0) \xleftarrow{\$} \text{ITR.Setup}(1^\lambda, n_{\text{ITR}} = 2^\lambda)$.

3. Then, it constructs the following obfuscated programs:
   - $\mathcal{P}_1 = \mathcal{IO}(\mathsf{Init\text{-}SPS.Prog}[q_0, w_0, v_0, K_{\text{SPS},E}])$,
   - $\mathcal{P}_2 = \mathcal{IO}(\mathsf{Accumulate.Prog}[n_{\text{SSB-BLK}} = 2^\lambda, \text{HK}, \text{PP}_{\text{ACC}}, \text{PP}_{\text{ITR}}, K_{\text{SPS},E}])$,
   - $\mathcal{P}_3 = \mathcal{IO}(\mathsf{Change\text{-}SPS.Prog}[K_{\text{SPS},A}, K_{\text{SPS},E}])$,
   - $\mathcal{P}_{\text{CPRF}} = \mathcal{IO}(\mathsf{Constrained\text{-}Key.Prog}_{\text{CPRF}}[M,\ T = 2^\lambda,\ \text{PP}_{\text{ACC}},\ \text{PP}_{\text{ITR}},\ K, K_1, \ldots, K_\lambda, K_{\text{SPS},A}])$,

   where the programs Init-SPS.Prog, Accumulate.Prog, Change-SPS.Prog, and $\mathsf{Constrained\text{-}Key.Prog}_{\text{CPRF}}$ are depicted respectively in Figs. 3.1, 3.2, 3.3 and 3.4.
4. It Provides the constrained key $\text{SK}_{\text{CPRF}}\{M\} = (\text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0, \text{PP}_{\text{ITR}}, v_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{CPRF}}) \in \mathcal{K}_{\text{CPRF-CONST}}$ to a legitimate user.

Constants: Initial TM state q_0, Accumulator value w_0, Iterator value v_0, PPRF key $K_{\text{SPS},E}$

Input: SSB hash value h

Output: Signature $\sigma_{\text{SPS,OUT}}$

1. Compute $r_{\text{SPS},E} = \mathcal{F}(K_{\text{SPS},E}, (h, 0))$ and $(\text{SK}_{\text{SPS},E}, \text{VK}_{\text{SPS},E}, \text{VK}_{\text{SPS-REJ},E}) = \mathsf{SPS.Setup}(1^\lambda; r_{\text{SPS},E})$.
2. Output $\sigma_{\text{SPS,OUT}} = \mathsf{SPS.Sign}(\text{SK}_{\text{SPS},E}, (v_0, q_0, w_0, 0))$.

Fig. 3.1. Init-SPS.Prog

Constants: Maximum number of blocks for SSB hash $n_{\text{SSB-BLK}} = 2^\lambda$, SSB hash key HK, Public parameters for positional accumulator PP_{ACC}, Public parameters for iterator PP_{ITR}, PPRF key $K_{\text{SPS},E}$

Inputs: Index i, Symbol SYM_{IN}, TM state ST, Accumulator value w_{IN}, Auxiliary value AUX, Iterator value v_{IN}, Signature $\sigma_{\text{SPS,IN}}$, SSB hash value h, SSB opening value π_{SSB}

Output: (Accumulator value w_{OUT}, Iterator value v_{OUT}, Signature $\sigma_{\text{SPS-OUT}}$), or $\perp$

1.(a) Compute $r_{\text{SPS},E} = \mathcal{F}(K_{\text{SPS},E}, (h, i))$ and $(\text{SK}_{\text{SPS},E}, \text{VK}_{\text{SPS},E}, \text{VK}_{\text{SPS-REJ},E}) = \mathsf{SPS.Setup}(1^\lambda; r_{\text{SPS},E})$.
   (b) Set $m_{\text{IN}} = (v_{\text{IN}}, \text{ST}, w_{\text{IN}}, 0)$. If $\mathsf{SPS.Verify}(\text{VK}_{\text{SPS},E}, m_{\text{IN}}, \sigma_{\text{SPS,IN}}) = 0$, output $\perp$.
2. If $\mathsf{SSB.Verify}(\text{HK}, h, i, \text{SYM}_{\text{IN}}, \pi_{\text{SSB}}) = 0$, output $\perp$.
3.(a) Compute $w_{\text{OUT}} = \mathsf{ACC.Update}(\text{PP}_{\text{ACC}}, w_{\text{IN}}, \text{SYM}_{\text{IN}}, i, \text{AUX})$. If $w_{\text{OUT}} = \perp$, output $\perp$.
   (b) Compute $v_{\text{OUT}} = \mathsf{ITR.Iterate}(\text{PP}_{\text{ITR}}, v_{\text{IN}}, (\text{ST}, w_{\text{IN}}, 0))$.
4.(a) Compute $r'_{\text{SPS},E} = \mathcal{F}(K_{\text{SPS},E}, (h, i+1))$ and $(\text{SK}'_{\text{SPS},E}, \text{VK}'_{\text{SPS},E}, \text{VK}'_{\text{SPS-REJ},E}) = \mathsf{SPS.Setup}(1^\lambda; r'_{\text{SPS},E})$.
   (b) Set $m_{\text{OUT}} = (v_{\text{OUT}}, \text{ST}, w_{\text{OUT}}, 0)$. Compute $\sigma_{\text{SPS,OUT}} = \mathsf{SPS.Sign}(\text{SK}'_{\text{SPS},E}, m_{\text{OUT}})$.
5. Output $(w_{\text{OUT}}, v_{\text{OUT}}, \sigma_{\text{SPS,OUT}})$.

Fig. 3.2. Accumulate.Prog

Constants: PPRF keys $K_{\text{SPS},A}, K_{\text{SPS},E}$

Inputs: TM state ST, Accumulator value w, Iterator value v, SSB hash value h, Length ℓ_{INP}, Signature $\sigma_{\text{SPS,IN}}$

Output: Signature $\sigma_{\text{SPS,OUT}}$, or $\perp$

1.(a) Compute $r_{\text{SPS},E} = \mathcal{F}(K_{\text{SPS},E}, (h, \ell_{\text{INP}}))$ and $(\text{SK}_{\text{SPS},E}, \text{VK}_{\text{SPS},E}, \text{VK}_{\text{SPS-REJ},E}) = \mathsf{SPS.Setup}(1^\lambda; r_{\text{SPS},E})$.
(b) Set $m = (v, \text{ST}, w, 0)$. If $\mathsf{SPS.Verify}(\text{VK}_{\text{SPS},E}, m, \sigma_{\text{SPS,IN}}) = 0$, output $\perp$.
2.(a) Compute $r_{\text{SPS},A} = \mathcal{F}(K_{\text{SPS},A}, (h, \ell_{\text{INP}}, 0))$ and $(\text{SK}_{\text{SPS},A}, \text{VK}_{\text{SPS},A}, \text{VK}_{\text{SPS-REJ},A}) = \mathsf{SPS.Setup}(1^\lambda; r_{\text{SPS},A})$.
(b) Output $\sigma_{\text{SPS,OUT}} = \mathsf{SPS.Sign}(\text{SK}_{\text{SPS},A}, m)$.

Fig. 3.3. Change-SPS.Prog

Constants: TM $M = \langle Q, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \delta, q_0, q_{\text{AC}}, q_{\text{REJ}} \rangle$, Time bound $T = 2^\lambda$, Public parameters for positional accumulator PP_{ACC}, Public parameters for iterator PP_{ITR}, PPRF keys $K, K_1, \ldots, K_\lambda, K_{\text{SPS},A}$

Inputs: Time t, String SEED_{IN}, Header position POS_{IN}, Symbol SYM_{IN}, TM state ST_{IN}, Accumulator value w_{IN}, Accumulator proof π_{ACC}, Auxiliary value AUX, Iterator value v_{IN}, SSB hash value h, length ℓ_{INP}, Signature $\sigma_{\text{SPS,IN}}$

Output: CPRF evaluation $\mathcal{F}(K, (h, \ell_{\text{INP}}))$, or Header Position ($\text{POS}_{\text{OUT}}$, Symbol SYM_{OUT}, TM state ST_{OUT}, Accumulator value w_{OUT}, Iterator value v_{OUT}, Signature $\sigma_{\text{SPS,OUT}}$, String SEED_{OUT}), or $\perp$

1. Identify an integer τ such that $2^\tau \leq t < 2^{\tau+1}$. If $[\mathsf{PRG}(\text{SEED}_{\text{IN}}) \neq \mathsf{PRG}(\mathcal{F}(K_\tau, (h, \ell_{\text{INP}}))] \wedge [t > 1]$, output $\perp$.
2. If $\mathsf{ACC.Verify\text{-}Read}(\text{PP}_{\text{ACC}}, w_{\text{IN}}, \text{SYM}_{\text{IN}}, \text{POS}_{\text{IN}}, \pi_{\text{ACC}}) = 0$, output $\perp$.
3.(a) Compute $r_{\text{SPS},A} = \mathcal{F}(K_{\text{SPS},A}, (h, \ell_{\text{INP}}, t-1))$ and $(\text{SK}_{\text{SPS},A}, \text{VK}_{\text{SPS},A}, \text{VK}_{\text{SPS-REJ},A}) = \mathsf{SPS.Setup}(1^\lambda; r_{\text{SPS},A})$.
(b) Set $m_{\text{IN}} = (v_{\text{IN}}, \text{ST}_{\text{IN}}, w_{\text{IN}}, \text{POS}_{\text{IN}})$. If $\mathsf{SPS.Verify}(\text{VK}_{\text{SPS},A}, m_{\text{IN}}, \sigma_{\text{SPS,IN}}) = 0$, output $\perp$.
4.(a) Compute $(\text{ST}_{\text{OUT}}, \text{SYM}_{\text{OUT}}, \beta) = \delta(\text{ST}_{\text{IN}}, \text{SYM}_{\text{IN}})$ and $\text{POS}_{\text{OUT}} = \text{POS}_{\text{IN}} + \beta$.
(b) If $\text{ST}_{\text{OUT}} = q_{\text{REJ}}$, output $\perp$.
Else if $\text{ST}_{\text{OUT}} = q_{\text{AC}}$, output $\mathcal{F}(K, (h, \ell_{\text{INP}}))$.
5.(a) Compute $w_{\text{OUT}} = \mathsf{ACC.Update}(\text{PP}_{\text{ACC}}, w_{\text{IN}}, \text{SYM}_{\text{OUT}}, \text{POS}_{\text{IN}}, \text{AUX})$. If $w_{\text{OUT}} = \perp$, output $\perp$.
(b) Compute $v_{\text{OUT}} = \mathsf{ITR.Iterate}(\text{PP}_{\text{ITR}}, v_{\text{IN}}, (\text{ST}_{\text{IN}}, w_{\text{IN}}, \text{POS}_{\text{IN}}))$.
6.(a) Compute $r'_{\text{SPS},A} = \mathcal{F}(K_{\text{SPS},A}, (h, \ell_{\text{INP}}, t))$ and $(\text{SK}'_{\text{SPS},A}, \text{VK}'_{\text{SPS},A}, \text{VK}'_{\text{SPS-REJ},A}) = \mathsf{SPS.Setup}(1^\lambda; r'_{\text{SPS},A})$.
(b) Set $m_{\text{OUT}} = (v_{\text{OUT}}, \text{ST}_{\text{OUT}}, w_{\text{OUT}}, \text{POS}_{\text{OUT}})$.
Compute $\sigma_{\text{SPS,OUT}} = \mathsf{SPS.Sign}(\text{SK}'_{\text{SPS},A}, m_{\text{OUT}})$.
7. If $t + 1 = 2^{\tau'}$, set $\text{SEED}_{\text{OUT}} = \mathcal{F}(K_{\tau'}, (h, \ell_{\text{INP}}))$.
Else, set $\text{SEED}_{\text{OUT}} = \epsilon$.
8. Output $(\text{POS}_{\text{OUT}}, \text{SYM}_{\text{OUT}}, \text{ST}_{\text{OUT}}, w_{\text{OUT}}, v_{\text{OUT}}, \sigma_{\text{SPS,OUT}}, \text{SEED}_{\text{OUT}})$.

Fig. 3.4. Constrained-Key.$\text{Prog}_{\text{CPRF}}$

$\mathsf{CPRF.Eval\text{-}Constrained}(\text{SK}_{\text{CPRF}}\{M\}, x) \rightarrow y = \mathcal{F}(K, (h, \ell_x))$ or $\perp$: A user takes as input its constrained key $\text{SK}_{\text{CPRF}}\{M\} = (\text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0, \text{PP}_{\text{ITR}}, v_0,$

$\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{CPRF}}) \in \mathcal{K}_{\text{CPRF-CONST}}$ corresponding to some legitimate TM $M = \langle Q, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \delta, q_0, q_{\text{AC}}, q_{\text{REJ}} \rangle$ and an input $x = x_0 \dots x_{\ell_x - 1} \in \mathcal{X}_{\text{CPRF}}$ with $|x| = \ell_x$. It proceeds as follows:

1. It first computes $h = \mathcal{H}_{\text{HK}}(x)$.
2. Next, it computes $\breve{\sigma}_{\text{SPS},0} = \mathcal{P}_1(h)$.
3. Then for $j = 1, \dots, \ell_x$, it iteratively performs the following:
   (a) It computes $\pi_{\text{SSB},j-1} \xleftarrow{\$} \mathsf{SSB.Open}(\text{HK}, x, j-1)$.
   (b) It computes $\text{AUX}_j = \mathsf{ACC.Prep\text{-}Write}(\text{PP}_{\text{ACC}}, \text{STORE}_{j-1}, j-1)$.
   (c) It computes $\text{OUT} = \mathcal{P}_2(j-1, x_{j-1}, q_0, w_{j-1}, \text{AUX}_j, v_{j-1}, \breve{\sigma}_{\text{SPS},j-1}, h, \pi_{\text{SSB},j-1})$.
   (d) If $\text{OUT} = \perp$, it outputs OUT. Else, it parses OUT as $\text{OUT} = (w_j, v_j, \breve{\sigma}_{\text{SPS},j})$.
   (e) It computes $\text{STORE}_j = \mathsf{ACC.Write\text{-}Store}(\text{PP}_{\text{ACC}}, \text{STORE}_{j-1}, j-1, x_{j-1})$.
4. It computes $\sigma_{\text{SPS},0} = \mathcal{P}_3(q_0, w_{\ell_x}, v_{\ell_x}, h, \ell_x, \breve{\sigma}_{\text{SPS},\ell_x})$.
5. It sets $\text{POS}_{M,0} = 0$ and $\text{SEED}_0 = \epsilon$.
6. Suppose, M runs for t_x steps on input x. For $t = 1, \dots, t_x$, it iteratively performs the following steps:
   (a) It computes $(\text{SYM}_{M,t-1}, \pi_{\text{ACC},t-1}) = \mathsf{ACC.Prep\text{-}Read}(\text{PP}_{\text{ACC}}, \text{STORE}_{\ell_x+t-1}, \text{POS}_{M,t-1})$.
   (b) It computes $\text{AUX}_{\ell_x+t} = \mathsf{ACC.Prep\text{-}Write}(\text{PP}_{\text{ACC}}, \text{STORE}_{\ell_x+t-1}, \text{POS}_{M,t-1})$.
   (c) It computes $\text{OUT} = \mathcal{P}_{\text{CPRF}}(t, \text{SEED}_{t-1}, \text{POS}_{M,t-1}, \text{SYM}_{M,t-1}, \text{ST}_{M,t-1}, w_{\ell_x+t-1}, \pi_{\text{ACC},t-1}, \text{AUX}_{\ell_x+t}, v_{\ell_x+t-1}, h, \ell_x, \sigma_{\text{SPS},t-1})$.
   (d) If $t = t_x$, it outputs OUT. Otherwise, it parses OUT as $\text{OUT} = (\text{POS}_{M,t}, \text{SYM}_{M,t}^{(\text{WRITE})}, \text{ST}_{M,t}, w_{\ell_x+t}, v_{\ell_x+t}, \sigma_{\text{SPS},t}, \text{SEED}_t)$.
   (e) It computes $\text{STORE}_{\ell_x+t} = \mathsf{ACC.Write\text{-}Store}(\text{PP}_{\text{ACC}}, \text{STORE}_{\ell_x+t-1}, \text{POS}_{M,t-1}, \text{SYM}_{M,t}^{(\text{WRITE})})$.

Theorem 3.1. *Assuming $\mathcal{IO}$ is a secure indistinguishability obfuscator for* P/poly, *$\mathcal{F}$ is a secure puncturable pseudorandom function,* SSB *is a somewhere statistically binding hash function,* ACC *is a secure positional accumulator,* ITR *is a secure cryptographic iterator,* SPS *is a secure splittable signature scheme, and* PRG *is a secure injective pseudorandom generator, our* CPRF *construction satisfies correctness under constraining and selective pseudorandomness properties.*

The proof of Theorem 3.1 is provided in the full version of this paper.

Remark 3.2. We note that concurrently and independently of our work, Deshpande et al. [11] have recently provided an alternative fix to the flaw in [10] discussed in Section 3.2, by replacing the standard positional accumulators used in the CPRF construction of [10] with an advanced variant of positional accumulators, namely, history-less positional accumulators [3]. Unlike standard positional accumulators, in case of history-less positional accumulators, setting up the public parameters in read/write enforcing mode does not require any history of symbol-index pairs as input. Consequently, the problem in the simulation of [10] discussed in Section 3.2, resulting from the use of standard positional accumulators, would clearly not arise if history-less positional accumulators are utilized in the CPRF construction of [10] instead. However, we emphasize that our approach towards resolving the flaw of [10] brings about some new subtle technical ideas which might be useful elsewhere as well.

4 Our CVPRF for Turing Machines

4.1 Notion

Definition 4.1 (Constrained Verifiable Pseudorandom Function for Turing Machines: CVPRF). Let $\mathbb{M}_\lambda$ be a family of TM's with (worst-case) running time bounded by $T = 2^\lambda$. A constrained verifiable pseudorandom function (CVPRF) for $\mathbb{M}_\lambda$ with key space $\mathcal{K}_{\text{CVPRF}}$, input domain $\mathcal{X}_{\text{CVPRF}} \subset \{0,1\}^*$, and output space $\mathcal{Y}_{\text{CVPRF}} \subset \{0,1\}^*$ consists of a constrained key space $\mathcal{K}_{\text{CVPRF-CONST}}$, a proof space Π_{CVPRF}, along with PPT algorithms (CVPRF.Setup, CVPRF.Eval, CVPRF.Prove, CVPRF.Constrain, CVPRF.Prove-Constrained, CVPRF.Verify) which are described below:

- $\mathsf{CVPRF.Setup}(1^\lambda) \rightarrow (\text{SK}_{\text{CVPRF}}, \text{VK}_{\text{CVPRF}})$: The setup authority takes as input the security parameter 1^λ and generates a master CVPRF key SK_{CVPRF} along with a public verification key VK_{CVPRF}.
- $\mathsf{CVPRF.Eval}(\text{SK}_{\text{CVPRF}}, x) \rightarrow y$: Taking as input the master CVPRF key SK_{CVPRF} and an input $x \in \mathcal{X}_{\text{CVPRF}}$, the trusted authority outputs the value of the function $y \in \mathcal{Y}_{\text{CVPRF}}$. For simplicity of notation, we will denote by $\mathsf{CVPRF}(\text{SK}_{\text{CVPRF}}, x)$ the output of this algorithm.
- $\mathsf{CVPRF.Prove}(\text{SK}_{\text{CVPRF}}, x) \rightarrow \pi_{\text{CVPRF}}$: Taking as input the master CVPRF key SK_{CVPRF} and an input $x \in \mathcal{X}_{\text{CVPRF}}$, the trusted authority outputs a proof $\pi_{\text{CVPRF}} \in \Pi_{\text{CVPRF}}$.
- $\mathsf{CVPRF.Constrain}(\text{SK}_{\text{CVPRF}}, M) \rightarrow \text{SK}_{\text{CVPRF}}\{M\}$: On input the master CVPRF key SK_{CVPRF} and a TM $M \in \mathbb{M}_\lambda$, the setup authority provides a constrained key $\text{SK}_{\text{CVPRF}}\{M\}$ to a legitimate user.
- $\mathsf{CVPRF.Prove\text{-}Constrained}(\text{SK}_{\text{CVPRF}}\{M\}, x) \rightarrow (y, \pi_{\text{CVPRF}})$ or $\perp$: A user takes as input its constrained key $\text{SK}_{\text{CVPRF}}\{M\}$ corresponding to a legitimate TM $M \in \mathbb{M}_\lambda$ and an input $x \in \mathcal{X}_{\text{CVPRF}}$. It outputs either a value-proof pair $(y, \pi_{\text{CVPRF}}) \in \mathcal{Y}_{\text{CVPRF}} \times \Pi_{\text{CVPRF}}$ or $(\perp, \perp)$ indicating failure.
- $\mathsf{CVPRF.Verify}(\text{VK}_{\text{CVPRF}}, x, y, \pi_{\text{CVPRF}}) \rightarrow \hat{\beta} \in \{0,1\}$: A verifier takes as input the public verification key VK_{CVPRF}, an input $x \in \mathcal{X}_{\text{CVPRF}}$, a value $y \in \mathcal{Y}_{\text{CVPRF}}$, together with a proof $\pi_{\text{CVPRF}} \in \Pi_{\text{CVPRF}}$. It outputs a bit $\hat{beta} \in \{0,1\}$.

The algorithms CVPRF.Setup, CVPRF.Prove, CVPRF.Constrain and CVPRF.Prove-Constrained are randomized, while the other two algorithms are deterministic.

► **Provability**: For any security parameter λ, $(\text{SK}_{\text{CVPRF}}, \text{VK}_{\text{CVPRF}}) \xleftarrow{\$} \mathsf{CVPRF.Setup}(1^\lambda)$, $M \in \mathbb{M}_\lambda$, $\text{SK}_{\text{CVPRF}}\{M\} \xleftarrow{\$} \mathsf{CVPRF.Constrain}(\text{SK}_{\text{CVPRF}}, M)$, $x \in \mathcal{X}_{\text{CVPRF}}$, and $(y, \pi_{\text{CVPRF}}) \xleftarrow{\$} \mathsf{CVPRF.Prove\text{-}Constrained}(\text{SK}_{\text{CVPRF}}\{M\}, x)$, the following holds:

- If $M(x) = 1$, then $y = \mathsf{CVPRF}(\text{SK}_{\text{CVPRF}}, x)$ and $\mathsf{CVPRF.Verify}(\text{VK}_{\text{CVPRF}}, x, y, \pi_{\text{CVPRF}}) = 1$.
- If $M(x) = 0$, then $(y, \pi_{\text{CVPRF}}) = (\perp, \perp)$.

The security requirements of a CVPRF are formally defined in the full version of this paper.

4.2 Techniques Adapted in Our CVPRF Construction

Let us now sketch our technical ideas to extend our CPRF construction to incorporate the verifiability feature. The additional tool that we use for this enhancement is a public key encryption (PKE) scheme which is perfectly correct and chosen plaintext attack (CPA) secure. Besides the PPRF key K, used to generate the PRF output, and the SSB hash key, we include within the master key another PPRF key K_{PKE} to generate randomness for the setup and encryption algorithms of PKE. As earlier, the PRF output on some input x is $\mathcal{F}(K, (h, \ell_x))$, where h and ℓ_x are respectively the SSB hash value and length of x. The non-interactive proof of correctness consists of a PKE public key PK_{PKE} together with a pseudorandom string $r_{\text{PKE},2}$. The randomness $r_{\text{PKE},1}$ for setting up the PKE public key PK_{PKE} along with the pseudorandom string $r_{\text{PKE},2}$ are formed as $r_{\text{PKE},1} \| r_{\text{PKE},2} = \mathcal{F}(K_{\text{PKE}}, (h, \ell_x))$.

The public verification key comprises of the same SSB hash key as included in the master PRF key, together with an IO-obfuscated program $\mathcal{V}_{\text{CVPRF}}$, known as the *verifying program*. The verifying program $\mathcal{V}_{\text{CVPRF}}$ has the PPRF keys K and K_{PKE} hardwired in it. It takes as input an SSB hash value h and PRF input length ℓ_{INP}. It first computes the concatenated pseudorandom strings $\hat{r}_{\text{PKE},1} \| \hat{r}_{\text{PKE},2} = \mathcal{F}(K_{\text{PKE}}, (h, \ell_{\text{INP}}))$. Next, it runs the PKE setup algorithm using the generated randomness $\hat{r}_{\text{PKE},1}$ and creates a PKE public key $\widehat{\text{PK}}_{\text{PKE}}$. The program outputs $\widehat{\text{PK}}_{\text{PKE}}$ together with the ciphertext $\widehat{\text{CT}}_{\text{PKE}}$ encrypting the PRF value $\mathcal{F}(K, (h, \ell_{\text{INP}}))$ under $\widehat{\text{PK}}_{\text{PKE}}$ utilizing the randomness $\hat{r}_{\text{PKE},2}$.

To verify a purported PRF value-proof pair $(y, \pi_{\text{CVPRF}} = (\text{PK}_{\text{PKE}}, r))$ for some input x using the public verification key, a verifier first hashes x using the SSB hash key and then obtains a PKE public key-ciphertext pair $(\widehat{\text{PK}}_{\text{PKE}}, \widehat{\text{CT}}_{\text{PKE}})$ by running the obfuscated verifying program $\mathcal{V}_{\text{CVPRF}}$ on input the computed hash value and length of the input x. The verifier accepts the proof if $\widehat{\text{PK}}_{\text{PKE}}$ matches with PK_{PKE}, as well as $\widehat{\text{CT}}_{\text{PKE}}$ matches with the ciphertext formed by encrypting the purported PRF value y under PK_{PKE} using the string r included within the proof. Observe that the soundness of verification follows directly from the perfect correctness property of the underlying PKE scheme. Specifically, due to the perfect correctness of PKE, it is guaranteed that two different values cannot map to the same ciphertext under the same public key.

Finally, to enable the generation of the proof along with the PRF value using a constrained key, we modify the obfuscated next step program, which we denote as $\mathcal{P}_{\text{CVPRF}}$, included in the constrained key to output the proof together with the PRF value when it reaches the accepting state.

4.3 Formal Description of our CVPRF

Here we will provide our CVPRF for TM's. This construction is obtained by extending our CPRF construction described in Section 3.4. Let λ be the underlying security parameter. Let $\mathbb{M}_\lambda$ be a class of TM's, the members of which have (worst-case) running time bounded by $T = 2^\lambda$, input alphabet $\Sigma_{\text{INP}} = \{0, 1\}$, and

tape alphabet $\Sigma_{\text{TAPE}} = \{0, 1, \_\}$. Our CVPRF construction for TM family $\mathbb{M}_\lambda$ will employ all the building blocks utilized in our CPRF construction. Additionally, we will use a perfectly correct and chosen plaintext attack (CPA) secure public key encryption scheme $\mathsf{PKE} = (\mathsf{PKE.Setup}, \mathsf{PKE.Encrypt}, \mathsf{PKE.Decrypt})$ with an appropriate message space. The formal description of our CVPRF construction follows:

$\mathsf{CVPRF.Setup}(1^\lambda) \rightarrow (\text{SK}_{\text{CVPRF}} = (K, K_{\text{PKE}}, \text{HK}), \text{VK}_{\text{CVPRF}} = (\text{HK}, \mathcal{V}_{\text{CVPRF}}))$: The setup authority takes as input the security parameter 1^λ and proceeds as follows:

1. It first chooses PPRF keys $K, K_{\text{PKE}} \xleftarrow{\$} \mathcal{F}.\mathsf{Setup}(1^\lambda)$.
2. Next it generates $\text{HK} \xleftarrow{\$} \mathsf{SSB.Gen}(1^\lambda, n_{\text{SSB-BLK}} = 2^\lambda, i^* = 0)$.
3. Then, it creates the obfuscated program $\mathcal{V}_{\text{CVPRF}} = \mathcal{IO}(\mathsf{Verify.Prog}_{\text{CVPRF}}[K, K_{\text{PKE}}])$, where the program $\mathsf{Verify.Prog}_{\text{CVPRF}}$ is described in Fig. 4.1.
4. It sets the master CVPRF key as $\text{SK}_{\text{CVPRF}} = (K, K_{\text{PKE}}, \text{HK})$ and publishes the public verification key $\text{VK}_{\text{CVPRF}} = (\text{HK}, \mathcal{V}_{\text{CVPRF}})$.

Constants: PPRF keys K, K_{PKE}
Inputs: SSB hash value h, Length ℓ_{INP}
Output: (PKE public key $\widehat{\text{PK}}_{\text{PKE}}$, Encryption of CVPRF value $\widehat{\text{CT}}_{\text{PKE}}$)

1. Compute $\hat{r}_{\text{PKE},1} \| \hat{r}_{\text{PKE},2} = \mathcal{F}(K_{\text{PKE}}, (h, \ell_{\text{INP}}))$, $(\widehat{\text{PK}}_{\text{PKE}}, \widehat{\text{SK}}_{\text{PKE}}) = \mathsf{PKE.Setup}(1^\lambda; \hat{r}_{\text{PKE},1})$.
2. Compute $\widehat{\text{CT}}_{\text{PKE}} = \mathsf{PKE.Encrypt}(\widehat{\text{PK}}_{\text{PKE}}, \mathcal{F}(K, (h, \ell_{\text{INP}})); \hat{r}_{\text{PKE},2})$.
3. Output $(\widehat{\text{PK}}_{\text{PKE}}, \widehat{\text{CT}}_{\text{PKE}})$.

Fig. 4.1. $\mathsf{Verify.Prog}_{\text{CVPRF}}$

$\mathsf{CVPRF.Eval}(\text{SK}_{\text{CVPRF}}, x) \rightarrow y = \mathcal{F}(K, (h, \ell_x))$: Taking as input the master CVPRF key $\text{SK}_{\text{CVPRF}} = (K, K_{\text{PKE}}, \text{HK})$ along with an input $x = x_0 \dots x_{\ell_x - 1} \in \mathcal{X}_{\text{CVPRF}}$, where $|x| = \ell_x$, the setup authority proceeds in an identical fashion to $\mathsf{CPRF.Eval}(\text{SK}_{\text{CPRF}}, x)$ described in Section 3.4.

$\mathsf{CVPRF.Prove}(\text{SK}_{\text{CVPRF}}, x) \rightarrow \pi_{\text{CVPRF}} = (\text{PK}_{\text{PKE}}, r_{\text{PKE},2})$: The setup authority takes as input the master CVPRF key $\text{SK}_{\text{CVPRF}} = (K, K_{\text{PKE}}, \text{HK})$ along with an input $x = x_0 \dots x_{\ell_x - 1} \in \mathcal{X}_{\text{CVPRF}}$, where $|x| = \ell_x$. It proceeds as follows:

1. At first, it computes $h = \mathcal{H}_{\text{HK}}(x)$.
2. Then, it computes $r_{\text{PKE},1} \| r_{\text{PKE},2} = \mathcal{F}(K_{\text{PKE}}, (h, \ell_x))$, $(\text{PK}_{\text{PKE}}, \text{SK}_{\text{PKE}}) = \mathsf{PKE.Setup}(1^\lambda;\ r_{\text{PKE},1})$.
3. It outputs $\pi_{\text{CVPRF}} = (\text{PK}_{\text{PKE}}, r_{\text{PKE},2})$.

$\mathsf{CVPRF.Constrain}(\text{SK}_{\text{CVPRF}}, M) \rightarrow \text{SK}_{\text{CVPRF}}\{M\} = (\text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0,\ \text{PP}_{\text{ITR}}, v_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{CVPRF}})$: On input the master CVPRF key $\text{SK}_{\text{CVPRF}} = (K,\ K_{\text{PKE}},\ \text{HK})$ and a TM $M = \langle Q, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \delta, q_0, q_{\text{AC}}, q_{\text{REJ}} \rangle \in \mathbb{M}_\lambda$, the setup authority proceeds identically to $\mathsf{CPRF.Constrain}(\text{SK}_{\text{CPRF}}, M)$ with the only difference that in place of $\mathcal{P}_{\text{CPRF}}$ it includes $\mathcal{P}_{\text{CVPRF}} = \mathcal{IO}(\mathsf{Constrained\text{-}Key.Prog}_{\text{CVPRF}}[M, T = 2^\lambda, \text{PP}_{\text{ACC}}, \text{PP}_{\text{ITR}}, K, K_{\text{PKE}}, K_1, \dots, K_\lambda, K_{\text{SPS},A}])$ within the constrained key $\text{SK}_{\text{CVPRF}}\{M\}$, where the program $\mathsf{Constrained\text{-}Key.Prog}_{\text{CVPRF}}$ is depicted in Fig. 4.2.

Constants: PPRF key K_{PKE} along with everything hardwired within the program $\mathsf{Constrained\text{-}Key.Prog}_{\text{CPRF}}$ (Fig. 3.4)

Inputs: Same as those to the program $\mathsf{Constrained\text{-}Key.Prog}_{\text{CPRF}}$ (Fig. 3.4)

Output: (CVPRF evaluation $\mathcal{F}(K,(h,\ell_{\text{INP}}))$, CVPRF proof $\pi_{\text{CVPRF}} = (\text{PK}_{\text{PKE}}, r_{\text{PKE},2})$) or Header Position ($\text{POS}_{\text{OUT}}$, Symbol SYM_{OUT}, TM state ST_{OUT}, Accumulator value w_{OUT}, Iterator value v_{OUT}, Signature $\sigma_{\text{SPS,OUT}}$, String SEED_{OUT}), or $\perp$

The functionality of this program is exactly the same as that of the program $\mathsf{Constrained\text{-}Key.Prog}_{\text{CPRF}}$ (Fig. 3.4) except that Step 4.(b) is replaced with the following:

4.(b) If $\text{ST}_{\text{OUT}} = q_{\text{REJ}}$, output $\perp$.
Else if $\text{ST}_{\text{OUT}} = q_{\text{AC}}$, perform the following:
(I) Compute $r_{\text{PKE},1} \| r_{\text{PKE},2} = \mathcal{F}(K_{\text{PKE}}(h,\ell_{\text{INP}}))$ and $(\text{PK}_{\text{PKE}}, \text{SK}_{\text{PKE}}) = \mathsf{PKE.Setup}(1^{\lambda}; r_{\text{PKE},1})$.
(II) Output $(\mathcal{F}(k,(h,\ell_{\text{INP}})), \pi_{\text{CVPRF}} = (\text{PK}_{\text{PKE}}, r_{\text{PKE},2}))$.

Fig. 4.2. $\mathsf{Constrained\text{-}Key.Prog}_{\text{CVPRF}}$

$\mathsf{CVPRF.Prove\text{-}Constrained}(\text{SK}_{\text{CVPRF}}\{M\}, x) \rightarrow (y = \mathcal{F}(K,(h,\ell_x)),\ \pi_{\text{CVPRF}} = (\text{PK}_{\text{PKE}}, r_{\text{PKE},2}))$ or $\perp$: A user takes as input its constrained key $\text{SK}_{\text{CVPRF}}\{M\} = (\text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0, \text{PP}_{\text{ITR}}, v_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{CVPRF}})$ corresponding to some legitimate TM $M = \langle Q, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \delta, q_0, q_{\text{AC}}, q_{\text{REJ}} \rangle$ and an input $x = x_0 \ldots x_{\ell_x - 1} \in \mathcal{X}_{\text{CVPRF}}$ with $|x| = \ell_x$. It proceeds in the exact same manner as the algorithm $\mathsf{CPRF.Eval\text{-}Constrained}(\text{SK}_{\text{CPRF}}\{M\}, x)$ described in Section 3.4. However, note that now the constrained key $\text{SK}_{\text{CVPRF}}\{M\}$ of the user contains the obfuscated program $\mathcal{P}_{\text{CVPRF}}$ instead of $\mathcal{P}_{\text{CPRF}}$. Thus, it utilizes the program $\mathcal{P}_{\text{CVPRF}}$ in place of $\mathcal{P}_{\text{CPRF}}$ in the course of execution.

$\mathsf{CVPRF.Verify}(\text{VK}_{\text{CVPRF}}, x, y, \pi_{\text{CVPRF}}) \rightarrow \hat{\beta} \in \{0,1\}$: A verifier takes as input the public verification key $\text{VK}_{\text{CVPRF}} = (\text{HK}, \mathcal{V}_{\text{CVPRF}})$, an input $x = x_0 \ldots x_{\ell_x - 1} \in \mathcal{X}_{\text{CVPRF}}$, where $|x| = \ell_x$, a value $y \in \mathcal{Y}_{\text{CVPRF}}$, and a proof $\pi_{\text{CVPRF}} = (\text{PK}_{\text{PKE}}, r) \in \Pi_{\text{CVPRF}}$. It executes the following:

1. It first computes $h = \mathcal{H}_{\text{HK}}(x)$.
2. Next, it computes $(\widehat{\text{PK}}_{\text{PKE}}, \widehat{\text{CT}}_{\text{PKE}}) = \mathcal{V}_{\text{CVPRF}}(h, \ell_x)$.
3. If $[\text{PK}_{\text{PKE}} = \widehat{\text{PK}}_{\text{PKE}}] \wedge [\mathsf{PKE.Encrypt}(\text{PK}_{\text{PKE}}, y; r) = \widehat{\text{CT}}_{\text{PKE}}]$, it outputs 1. Otherwise, it outputs 0.

Theorem 4.1. *Assuming $\mathcal{IO}$ is a secure indistinguishability obfuscator for* P/poly*, $\mathcal{F}$ is a secure puncturable pseudorandom function,* SSB *is a somewhere statistically binding hash function,* ACC *is a secure positional accumulator,* ITR *is a secure cryptographic iterator,* SPS *is a secure splittable signature scheme,* PRG *is a secure injective pseudorandom generator, and* PKE *is a perfectly correct* CPA *secure public key encryption scheme, our* CVPRF *construction satisfies all the properties of a secure* CVPRF*.*

The proof of Theorem 4.1 is given in the full version of this paper.

5 Our DCPRF for Turing Machines

5.1 Notion

Definition 5.1 (Delegatable Constrained Pseudorandom Function for Turing Machines: DCPRF). Let $\mathbb{M}_\lambda$ be a family of TM's with (worst-case) running time bounded by $T = 2^\lambda$. A delegatable constrained pseudorandom function (DCPRF) with key space $\mathcal{K}_{\text{DCPRF}}$, input domain $\mathcal{X}_{\text{DCPRF}} \subset \{0,1\}^*$, and output space $\mathcal{Y}_{\text{DCPRF}} \subset \{0,1\}^*$ for the TM family $\mathbb{M}_\lambda$ consists of an additional key space $\mathcal{K}_{\text{DCPRF-CONST}}$ and PPT algorithms (DCPRF.Setup, DCPRF.Eval, DCPRF.Constrain, DCPRF.Delegate, DCPRF.Eval-Constrained) described as follows:

- $\mathsf{DCPRF.Setup}(1^\lambda) \rightarrow \text{SK}_{\text{DCPRF}}$: The setup authority takes as input the security parameter 1^λ and generates the master DCPRF key $\text{SK}_{\text{DCPRF}} \in \mathcal{K}_{\text{DCPRF}}$.
- $\mathsf{DCPRF.Eval}(\text{SK}_{\text{DCPRF}}, x) \rightarrow y$: On input the master DCPRF key SK_{DCPRF} along with an input $x \in \mathcal{X}_{\text{DCPRF}}$, the setup authority computes the value of the DCPRF $y \in \mathcal{Y}_{\text{DCPRF}}$. For simplicity of notation, we will use $\mathsf{DCPRF}(\text{SK}_{\text{DCPRF}}, x)$ to indicate the output of this algorithm.
- $\mathsf{DCPRF.Constrain}(\text{SK}_{\text{DCPRF}}, M) \rightarrow \text{SK}_{\text{DCPRF}}\{M\}$: Taking as input the master DCPRF key $\text{SK}_{\text{DCPRF}} \in \mathcal{K}_{\text{DCPRF}}$ and a TM $M \in \mathbb{M}_\lambda$, the setup authority provides a constrained key $\text{SK}_{\text{DCPRF}}\{M\} \in \mathcal{K}_{\text{DCPRF-CONST}}$ to a legitimate user.
- $\mathsf{DCPRF.Delegate}(\text{SK}_{\text{DCPRF}}\{M\}, \widetilde{M}) \rightarrow \text{SK}_{\text{DCPRF}}\{M \ \wedge \ \widetilde{M}\}$: Taking as input a constrained key $\text{SK}_{\text{DCPRF}}\{M\} \in \mathcal{K}_{\text{DCPRF-CONST}}$ corresponding to a legitimate TM $M \in \mathbb{M}_\lambda$ along with another TM $\widetilde{M} \in \mathbb{M}_\lambda$, a user gives a delegated constrained key $\text{SK}_{\text{DCPRF}}\{M \ \wedge \ \widetilde{M}\} \in \mathcal{K}_{\text{DCPRF-CONST}}$ to a legitimate delegate.
- $\mathsf{DCPRF.Eval\text{-}Constrained}(\text{SK}_{\text{DCPRF}}\{M\}/\text{SK}_{\text{DCPRF}}\{M \ \wedge \ \widetilde{M}\}, x) \rightarrow y$ or $\perp$: A user takes as input a constrained key $\text{SK}_{\text{DCPRF}}\{M\} \in \mathcal{K}_{\text{DCPRF-CONST}}$ obtained from the setup authority, corresponding to TM $M \in \mathbb{M}_\lambda$, or a delegated constrained key $\text{SK}_{\text{DCPRF}}\{M \ \wedge \ \widetilde{M}\} \in \mathcal{K}_{\text{DCPRF-CONST}}$ delegated by a constrained key holder holding the constrained key $\text{SK}_{\text{DCPRF}}\{M\} \in \mathcal{K}_{\text{DCPRF-CONST}}$, corresponding to TM $\widetilde{M} \in \mathbb{M}_\lambda$, along with an input $x \in \mathcal{X}_{\text{DCPRF}}$. It outputs either a value $y \in \mathcal{Y}_{\text{DCPRF}}$ or $\perp$ indicating failure.

The algorithms DCPRF.Eval and DCPRF.Eval-Constrained are deterministic, while, all the others are randomized.

► **Correctness under Constraining/Delegation**: Let us consider any security parameter λ, $x \in \mathcal{X}_{\text{DCPRF}}$, $\text{SK}_{\text{DCPRF}} \xleftarrow{\$} \mathsf{DCPRF.Setup}(1^\lambda)$, $M, \widetilde{M} \in \mathbb{M}_\lambda$, $\text{SK}_{\text{DCPRF}}\{M\} \xleftarrow{\$} \mathsf{DCPRF.Constrain}(\text{SK}_{\text{DCPRF}}, M)$ and $\text{SK}_{\text{DCPRF}}\{M \ \wedge \ \widetilde{M}\} \xleftarrow{\$} \mathsf{DCPRF.Delegate}(\text{SK}_{\text{DCPRF}}\{M\}, \widetilde{M})$. The following must hold:

$$\mathsf{DCPRF.Eval\text{-}Constrained}(\text{SK}_{\text{DCPRF}}\{M\}/\text{SK}_{\text{DCPRF}}\{M \ \wedge \ \widetilde{M}\}, x) = \begin{cases} \mathsf{DCPRF}(\text{SK}_{\text{DCPRF}}, x), & \text{if } M(x) = 1/[M(x) = 1] \ \wedge \ [\widetilde{M}(x) = 1] \\ \perp, & \text{otherwise} \end{cases}$$

The security notion of a DCPRF, namely, the pseudorandomness property is formally defined in the full version of this paper.

5.2 Techniques Adapted in our DCPRF Construction

Here again our starting point is our CPRF construction. We again use a perfectly correct and CPA secure PKE scheme for accomplishing key delegation. Precisely, while generating a constrained key corresponding to some TM M, we create a PPRF key K' specific to that constrained key. We then modify the output of the next step program, which we refer to as $\mathcal{P}_{\text{DCPRF}}$, when it reaches the accepting state. In stead of outputting the PRF value, the program $\mathcal{P}_{\text{DCPRF}}$ outputs an encryption of the PRF value. For performing this encryption it generates a PKE public key PK_{PKE}. The program computes the randomness $r_{\text{PKE},1}$ for generating the PKE public key PK_{PKE} as well as the randomness $r_{\text{PKE},2}$ for the encryption as $r_{\text{PKE},1} \| r_{\text{PKE},2} = \mathcal{F}(K', (h, \ell_{\text{INP}}))$, where h and ℓ_{INP} denote respectively the SSB hash value and length of the PRF input. We also include the PPRF key K' in the clear within the constrained key. Thus, while evaluating the PRF on some input using the constrained key, the evaluator will be able to recompute the pseudorandom string $r_{\text{PKE},1}$ using K' and then can generate the necessary PKE secret key SK_{PKE} by running the setup algorithm using the randomness $r_{\text{PKE},1}$ on its own. Once the secret key SK_{PKE} is obtained, the evaluator can simply decrypt the ciphertext obtained from the next step program $\mathcal{P}_{\text{DCPRF}}$ to uncover the PRF value. However, if a party does not have the key K' or the randomness that would have to be used for creating the required PKE secret key, then it cannot derive the PRF value from the ciphertext obtained from the next step program $\mathcal{P}_{\text{DCPRF}}$. We encash this idea to design the key delegation functionality.

The structure of our delegated key is as follows: Suppose a party holding a constrained key, corresponding to some TM M, wishes to construct a delegated key for $M \wedge \widetilde{M}$, where $\widetilde{M}$ is some other TM. The party generates all the components and obfuscated programs as those formed while constructing a constrained key for $\widetilde{M}$ with the only exception that it embeds the PPRF key K', included in its constrained key, inside the obfuscated next step program for $\widetilde{M}$ in place of the PPRF key K, which is part of the master PRF key and provides the PRF output. In fact, since the party only has a constrained key and not the master key, it does not possess the key K in the clear and hence cannot embed it within the obfuscated programs that it generates. The delegated key, corresponding to $\widetilde{M \wedge \widetilde{M}}$ consists of all the generated components and obfuscated programs for $\widetilde{M}$ together with all the components and obfuscated programs included in the constrained key for M possessed by the delegator except the PPRF key K'.

The idea is that, while evaluating the PRF on some input x using the delegated key for $M \wedge \widetilde{M}$, the evaluator proceeds in three steps. In the first step, provided $\widetilde{M}(x) = 1$, the evaluator computes the output of $\mathcal{F}$ with key K' on the SSB hash value and length of x by making use of the delegated key components pertaining to $\widetilde{M}$. Next, using the obtained PPRF output, the evaluator runs the PKE setup algorithm to obtain the necessary PKE secret key. In the second step, utilizing the delegated key components associated to M, the evaluator obtains a ciphertext encrypting the PRF output on x, provided $M(x) = 1$. Finally, the evaluator decrypts the ciphertext using the computed PKE secret key to reveal the PRF output.

5.3 Formal Description of Our DCPRF

In this section, we will present our DCPRF for TM's. The construction presented here considers only one level of delegation, however, it can readily be generalized to support multiple delegation levels. Let λ be the underlying security parameter. Consider the class $\mathbb{M}_\lambda$ of TM's, the members of which have (worst-case) running time bounded by $T = 2^\lambda$, input alphabet $\Sigma_{\text{INP}} = \{0,1\}$, and tape alphabet $\Sigma_{\text{TAPE}} = \{0,1,\_\}$. Our DCPRF construction is an augmentation of our CPRF construction with a delegation functionality and employs all the cryptographic building blocks utilized by our CPRF construction. In addition, we use a perfectly correct and CPA secure public key encryption scheme $\mathsf{PKE} = (\mathsf{PKE.Setup}, \mathsf{PKE.Encrypt}, \mathsf{PKE.Decrypt})$ with an appropriate message space. The formal description of our DCPRF follows:

- $\mathsf{DCPRF.Setup}(1^\lambda) \rightarrow \text{SK}_{\text{DCPRF}} = (K, \text{HK})$: The setup authority takes as input the security parameter 1^λ and proceeds the same way as $\mathsf{CPRF.Setup}(1^\lambda)$ described in Section 3.4.
- $\mathsf{DCPRF.Eval}(\text{SK}_{\text{DCPRF}}, x) \rightarrow y = \mathcal{F}(K, (h, \ell_x))$: Taking as input the master DCPRF key $\text{SK}_{\text{DCPRF}} = (K, \text{HK})$ and an input $x = x_0 \ldots x_{\ell_x - 1} \in \mathcal{X}_{\text{DCPRF}}$, where $|x| = \ell_x$, the setup authority executes identical steps as $\mathsf{CPRF.Eval}(\text{SK}_{\text{CPRF}}, x)$ described in Section 3.4.
- $\mathsf{DCPRF.Constrain}(\text{SK}_{\text{DCPRF}}, M) \rightarrow \text{SK}_{\text{DCPRF}}\{M\} = (K', \text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0, \text{PP}_{\text{ITR}}, v_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{DCPRF}})$: On input the master DCPRF key $\text{SK}_{\text{DCPRF}} = (K, \text{HK})$ and a TM $M = \langle Q, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \delta, q_0, q_{\text{AC}}, q_{\text{REJ}} \rangle \in \mathbb{M}_\lambda$, the setup authority performs the following steps:
  1. At first, it selects PPRF keys $K', K_1, \ldots, K_\lambda, K_{\text{SPS},A}, K_{\text{SPS},E} \xleftarrow{\$} \mathcal{F}.\mathsf{Setup}(1^\lambda)$.
  2. Next, it generates $(\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0) \xleftarrow{\$} \mathsf{ACC.Setup}(1^\lambda, n_{\text{ACC-BLK}} = 2^\lambda)$ and $(\text{PP}_{\text{ITR}}, v_0) \xleftarrow{\$} \mathsf{ITR.Setup}(1^\lambda, n_{\text{ITR}} = 2^\lambda)$.
  3. Then, it constructs the obfuscated programs
     - $\mathcal{P}_1 = \mathcal{IO}(\mathsf{Init\text{-}SPS.Prog}[q_0, w_0, v_0, K_{\text{SPS},E}])$,
     - $\mathcal{P}_2 = \mathcal{IO}(\mathsf{Accumulate.Prog}[n_{\text{SSB-BLK}} = 2^\lambda, \text{HK}, \text{PP}_{\text{ACC}}, \text{PP}_{\text{ITR}}, K_{\text{SPS},E}])$,
     - $\mathcal{P}_3 = \mathcal{IO}(\mathsf{Change\text{-}SPS.Prog}[K_{\text{SPS},A}, K_{\text{SPS},E}])$,
     - $\mathcal{P}_{\text{DCPRF}} = \mathcal{IO}(\mathsf{Constrained\text{-}Key.Prog}_{\text{DCPRF}}[M, T = 2^\lambda, \text{PP}_{\text{ACC}}, \text{PP}_{\text{ITR}}, K, K', K_1, \ldots, K_\lambda, K_{\text{SPS},A}])$,

     where the programs Init-SPS.Prog, Accumulate.Prog, and Change-SPS.Prog are depicted respectively in Figs. 3.1, 3.2 and 3.3 in Section 3.4, while the program $\mathsf{Constrained\text{-}Key.Prog}_{\text{DCPRF}}$ is described in Fig. 5.1.
  4. It provides the constrained key $\text{SK}_{\text{DCPRF}}\{M\} = (K', \text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0, \text{PP}_{\text{ITR}}, v_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{DCPRF}})$ to a legitimate user.
- $\mathsf{DCPRF.Delegate}(\text{SK}_{\text{DCPRF}}\{M\}, \widetilde{M}) \rightarrow \text{SK}_{\text{DCPRF}}\{M \wedge \widetilde{M}\} = (\widetilde{K}', \text{HK}, \text{PP}_{\text{ACC}}, \widetilde{\text{PP}}_{\text{ACC}}, w_0, \widetilde{w}_0, \text{STORE}_0, \widetilde{\text{STORE}}_0, \text{PP}_{\text{ITR}}, \widetilde{\text{PP}}_{\text{ITR}}, v_0, \widetilde{v}_0, \mathcal{P}_1, \widetilde{\mathcal{P}}_1, \mathcal{P}_2, \widetilde{\mathcal{P}}_2, \mathcal{P}_3, \widetilde{\mathcal{P}}_3, \mathcal{P}_{\text{DCPRF}}, \widetilde{\mathcal{P}}_{\text{DCPRF}})$: A user takes as input a constrained key $\text{SK}_{\text{DCPRF}}\{M\} = (K', \text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0, \text{PP}_{\text{ITR}}, v_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{DCPRF}})$, corresponding to a legitimate TM $M \in \mathbb{M}_\lambda$ and another TM $\widetilde{M} = \langle \widetilde{Q}, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \widetilde{\delta}, \widetilde{q}_0, \widetilde{q}_{\text{AC}}, \widetilde{q}_{\text{REJ}} \rangle \in \mathbb{M}_\lambda$. It proceeds as follows:

Constants: PPRF key K' along with everything hardwired within the program $\mathsf{Constrained\text{-}Key.Prog}_{\text{CPRF}}$ (Fig. 3.4)

Inputs: Same as those to the program $\mathsf{Constrained\text{-}Key.Prog}_{\text{CPRF}}$ (Fig. 3.4)

Output: Encryption of DCPRF value CT_{PKE}, or Header Position (POS_{OUT}, Symbol SYM_{OUT}, TM state ST_{OUT}, Accumulator value w_{OUT}, Iterator value v_{OUT}, Signature $\sigma_{\text{SPS,OUT}}$, String SEED_{OUT}), or $\perp$

This program functions in the same fashion as the program $\mathsf{Constrained\text{-}Key.Prog}_{\text{CPRF}}$ (Fig. 3.4) except that Step 4.(b) is replaced with the following:

4.(b) If $\text{ST}_{\text{OUT}} = q_{\text{REJ}}$, output $\perp$.
Else if $\text{ST}_{\text{OUT}} = q_{\text{AC}}$, perform the following steps:
(I) Compute $r_{\text{PKE},1} \| r_{\text{PKE},2} = \mathcal{F}(K', (h, \ell_{\text{INP}}))$ and $(\text{PK}_{\text{PKE}}, \text{SK}_{\text{PKE}}) = \mathsf{PKE.Setup}(1^\lambda; r_{\text{PKE},1})$.
(II) Output $\text{CT}_{\text{PKE}} = \mathsf{PKE.Encrypt}(\text{PK}_{\text{PKE}}, \mathcal{F}(K, (h, \ell_{\text{INP}})); r_{\text{PKE},2})$.

Fig. 5.1. $\mathsf{Constrained\text{-}Key.Prog}_{\text{DCPRF}}$

1. It first picks fresh PPRF keys $\widetilde{K}', \widetilde{K}_1, \ldots, \widetilde{K}_\lambda, \widetilde{K}_{\text{SPS},A}, \widetilde{K}_{\text{SPS},E} \xleftarrow{\$} \mathcal{F}.\mathsf{Setup}(1^\lambda)$.
2. Next it generates $(\widetilde{\text{PP}}_{\text{ACC}}, \widetilde{w}_0, \widetilde{\text{STORE}}_0) \xleftarrow{\$} \mathsf{ACC.Setup}(1^\lambda, n_{\text{ACC-BLK}} = 2^\lambda)$ and $(\widetilde{\text{PP}}_{\text{ITR}}, \widetilde{v}_0) \xleftarrow{\$} \mathsf{ITR.Setup}(1^\lambda, n_{\text{ITR}} = 2^\lambda)$ afresh.
3. Then, it constructs the obfuscated programs
   - $\widetilde{\mathcal{P}}_1 = \mathcal{IO}(\mathsf{Init\text{-}SPS.Prog}[\widetilde{q}_0, \widetilde{w}_0, \widetilde{v}_0, \widetilde{K}_{\text{SPS},E}])$,
   - $\widetilde{\mathcal{P}}_2 = \mathcal{IO}(\mathsf{Accumulate.Prog}[n_{\text{SSB-BLK}} = 2^\lambda, \text{HK}, \widetilde{\text{PP}}_{\text{ACC}}, \widetilde{\text{PP}}_{\text{ITR}}, \widetilde{K}_{\text{SPS},E}])$,
   - $\widetilde{\mathcal{P}}_3 = \mathcal{IO}(\mathsf{Change\text{-}SPS.Prog}[\widetilde{K}_{\text{SPS},A}, \widetilde{K}_{\text{SPS},E}])$,
   - $\widetilde{\mathcal{P}}_{\text{DCPRF}} = \mathcal{IO}(\mathsf{Constrained\text{-}Key.Prog}_{\text{DCPRF}}[\widetilde{M}, T = 2^\lambda, \widetilde{\text{PP}}_{\text{ACC}}, \widetilde{\text{PP}}_{\text{ITR}}, K', \widetilde{K}', \widetilde{K}_1, \ldots, \widetilde{K}_\lambda, \widetilde{K}_{\text{SPS},A}])$,

   where the programs $\mathsf{Init\text{-}SPS.Prog}$, $\mathsf{Accumulate.Prog}$, and $\mathsf{Change\text{-}SPS.Prog}$ are depicted respectively in Figs. 3.1, 3.2 and 3.3 in Section 3.4, while the program $\mathsf{Constrained\text{-}Key.Prog}_{\text{DCPRF}}$ is described in Fig. 5.1.
4. It gives the delegated key $\text{SK}_{\text{DCPRF}}\{M \wedge \widetilde{M}\} = (\widetilde{K}', \text{HK}, \text{PP}_{\text{ACC}}, \widetilde{\text{PP}}_{\text{ACC}}, w_0, \widetilde{w}_0, \text{STORE}_0, \widetilde{\text{STORE}}_0, \text{PP}_{\text{ITR}}, \widetilde{\text{PP}}_{\text{ITR}}, v_0, \widetilde{v}_0, \mathcal{P}_1, \widetilde{\mathcal{P}}_1, \mathcal{P}_2, \widetilde{\mathcal{P}}_2, \mathcal{P}_3, \widetilde{\mathcal{P}}_3, \mathcal{P}_{\text{DCPRF}}, \widetilde{\mathcal{P}}_{\text{DCPRF}})$ to a legitimate delegate.

$\mathsf{DCPRF.Eval\text{-}Constrained}(\text{SK}_{\text{DCPRF}}\{M\}/\text{SK}_{\text{DCPRF}}\{M \wedge \widetilde{M}\}, x) \to y = \mathcal{F}(K, (h, \ell_x))$ or $\perp$: A user takes as input a constrained key $\text{SK}_{\text{DCPRF}}\{M\} = (K', \text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0, \text{PP}_{\text{ITR}}, v_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{DCPRF}})$ obtained from the setup authority, corresponding to some legitimate TM $M = \langle Q, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \delta, q_0, q_{\text{AC}}, q_{\text{REJ}} \rangle \in \mathbb{M}_\lambda$, or a delegated key $\text{SK}_{\text{DCPRF}}\{M \wedge \widetilde{M}\} = (\widetilde{K}', \text{HK}, \text{PP}_{\text{ACC}}, \widetilde{\text{PP}}_{\text{ACC}}, w_0, \widetilde{w}_0, \text{STORE}_0, \widetilde{\text{STORE}}_0, \text{PP}_{\text{ITR}}, \widetilde{\text{PP}}_{\text{ITR}}, v_0, \widetilde{v}_0, \mathcal{P}_1, \widetilde{\mathcal{P}}_1, \mathcal{P}_2, \widetilde{\mathcal{P}}_2, \mathcal{P}_3, \widetilde{\mathcal{P}}_3, \mathcal{P}_{\text{DCPRF}}, \widetilde{\mathcal{P}}_{\text{DCPRF}})$ obtained from the holder of the constrained key $\text{SK}_{\text{DCPRF}}\{M\}$, corresponding to TM $\widetilde{M} = \langle \widetilde{Q}, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \widetilde{\delta}, \widetilde{q}_0, \widetilde{q}_{\text{AC}}, \widetilde{q}_{\text{REJ}} \rangle \in \mathbb{M}_\lambda$, along with an input $x = x_0 \ldots x_{\ell_x - 1} \in \mathcal{X}_{\text{DCPRF}}$ with $|x| = \ell_x$. It proceeds as follows:

(A) If $M(x) = 0$, it outputs $\perp$. Otherwise, it performs the following steps:

1. It first computes $h = \mathcal{H}_{\mathrm{HK}}(x)$.
2. Next, it computes $\breve{\sigma}_{\mathrm{SPS},0} = \mathcal{P}_1(h)$.
3. Then for $j = 1, \ldots, \ell_x$, it iteratively performs the following:
   (a) It computes $\pi_{\mathrm{SSB},j-1} \xleftarrow{\$} \mathsf{SSB.Open}(\mathrm{HK}, x, j-1)$.
   (b) It computes $\mathrm{AUX}_j = \mathsf{ACC.Prep\text{-}Write}(\mathrm{PP}_{\mathrm{ACC}}, \mathrm{STORE}_{j-1}, j-1)$.
   (c) It computes $\mathrm{OUT} = \mathcal{P}_2(j-1, x_{j-1}, q_0, w_{j-1}, \mathrm{AUX}_j, v_{j-1}, \breve{\sigma}_{\mathrm{SPS},j-1}, h, \pi_{\mathrm{SSB},j-1})$.
   (d) If $\mathrm{OUT} = \perp$, it outputs OUT. Else, it parses OUT as $\mathrm{OUT} = (w_j, v_j, \breve{\sigma}_{\mathrm{SPS},j})$.
   (e) It computes $\mathrm{STORE}_j = \mathsf{ACC.Write\text{-}Store}(\mathrm{PP}_{\mathrm{ACC}}, \mathrm{STORE}_{j-1}, j-1, x_{j-1})$.
4. It computes $\sigma_{\mathrm{SPS},0} = \mathcal{P}_3(q_0, w_{\ell_x}, v_{\ell_x}, h, \ell_x, \breve{\sigma}_{\mathrm{SPS},\ell_x})$.
5. It sets $\mathrm{POS}_{M,0} = 0$ and $\mathrm{SEED}_0 = \epsilon$.
6. Suppose, M accepts x in t_x steps. For $t = 1, \ldots, t_x$, it iteratively performs the following steps:
   (a) It computes $(\mathrm{SYM}_{M,t-1}, \pi_{\mathrm{ACC},t-1}) = \mathsf{ACC.Prep\text{-}Read}(\mathrm{PP}_{\mathrm{ACC}}, \mathrm{STORE}_{\ell_x+t-1}, \mathrm{POS}_{M,t-1})$.
   (b) It computes $\mathrm{AUX}_{\ell_x+t} = \mathsf{ACC.Prep\text{-}Write}(\mathrm{PP}_{\mathrm{ACC}}, \mathrm{STORE}_{\ell_x+t-1}, \mathrm{POS}_{M,t-1})$.
   (c) It computes $\mathrm{OUT} = \mathcal{P}_{\mathrm{DCPRF}}(t, \mathrm{SEED}_{t-1}, \mathrm{POS}_{M,t-1}, \mathrm{SYM}_{M,t-1}, \mathrm{ST}_{M,t-1}, w_{\ell_x+t-1}, \pi_{\mathrm{ACC},t-1}, \mathrm{AUX}_{\ell_x+t}, v_{\ell_x+t-1}, h, \ell_x, \sigma_{\mathrm{SPS},t-1})$.
   (d) If $t = t_x$, it sets $\mathrm{CT}_{\mathrm{PKE}} = \mathrm{OUT}$. Otherwise, it parses OUT as $\mathrm{OUT} = (\mathrm{POS}_{M,t}, \mathrm{SYM}_{M,t}^{(\mathrm{WRITE})}, \mathrm{ST}_{M,t}, w_{\ell_x+t}, v_{\ell_x+t}, \sigma_{\mathrm{SPS},t}, \mathrm{SEED}_t)$.
   (e) It computes $\mathrm{STORE}_{\ell_x+t} = \mathsf{ACC.Write\text{-}Store}(\mathrm{PP}_{\mathrm{ACC}}, \mathrm{STORE}_{\ell_x+t-1}, \mathrm{POS}_{M,t-1}, \mathrm{SYM}_{M,t}^{(\mathrm{WRITE})})$.

(B) If the user is using the constrained key $\mathrm{SK}_{\mathrm{DCPRF}}\{M\}$, then it computes $r_{\mathrm{PKE},1} \| r_{\mathrm{PKE},2} = \mathcal{F}(K', (h, \ell_x)), (\mathrm{PK}_{\mathrm{PKE}}, \mathrm{SK}_{\mathrm{PKE}}) = \mathsf{PKE.Setup}(1^\lambda; r_{\mathrm{PKE},1})$, and outputs $\mathsf{PKE.Decrypt}(\mathrm{SK}_{\mathrm{PKE}}, \mathrm{CT}_{\mathrm{PKE}})$. On the other hand, if the user is using the delegated key $\mathrm{SK}_{\mathrm{DCPRF}}\{M \wedge \widetilde{M}\}$ and $\widetilde{M}(x) = 0$, then it outputs $\perp$, while if $\widetilde{M}(x) = 1$, it further executes the following steps:

1. It computes $\widetilde{\breve{\sigma}}_{\mathrm{SPS},0} = \widetilde{\mathcal{P}}_1(h)$.
2. Then for $j = 1, \ldots, \ell_x$, it iteratively performs the following:
   (a) It computes $\widetilde{\pi}_{\mathrm{SSB},j-1} \xleftarrow{\$} \mathsf{SSB.Open}(\mathrm{HK}, x, j-1)$.
   (b) It computes $\widetilde{\mathrm{AUX}}_j = \mathsf{ACC.Prep\text{-}Write}(\widetilde{\mathrm{PP}}_{\mathrm{ACC}}, \widetilde{\mathrm{STORE}}_{j-1}, j-1)$.
   (c) It computes $\widetilde{\mathrm{OUT}} = \widetilde{\mathcal{P}}_2(j-1, x_{j-1}, \widetilde{q}_0, \widetilde{w}_{j-1}, \widetilde{\mathrm{AUX}}_j, \widetilde{v}_{j-1}, \widetilde{\breve{\sigma}}_{\mathrm{SPS},j-1}, h, \widetilde{\pi}_{\mathrm{SSB},j-1})$.
   (d) If $\widetilde{\mathrm{OUT}} = \perp$, it outputs $\widetilde{\mathrm{OUT}}$. Else, it parses $\widetilde{\mathrm{OUT}}$ as $\widetilde{\mathrm{OUT}} = (\widetilde{w}_j, \widetilde{v}_j, \widetilde{\breve{\sigma}}_{\mathrm{SPS},j})$.
   (e) It computes $\widetilde{\mathrm{STORE}}_j = \mathsf{ACC.Write\text{-}Store}(\widetilde{\mathrm{PP}}_{\mathrm{ACC}}, \widetilde{\mathrm{STORE}}_{j-1}, j-1, x_{j-1})$.
3. It computes $\widetilde{\sigma}_{\mathrm{SPS},0} = \widetilde{\mathcal{P}}_3(\widetilde{q}_0, \widetilde{w}_{\ell_x}, \widetilde{v}_{\ell_x}, h, \ell_x, \widetilde{\breve{\sigma}}_{\mathrm{SPS},\ell_x})$.
4. It sets $\mathrm{POS}_{\widetilde{M},0} = 0$ and $\widetilde{\mathrm{SEED}}_0 = \epsilon$.
5. Suppose, $\widetilde{M}$ accepts x in $\widetilde{t}_x$ steps. For $t = 1, \ldots, \widetilde{t}_x$, it iteratively performs the following steps:

(a) It computes $(\mathrm{SYM}_{\widetilde{M},t-1}, \widetilde{\pi}_{\mathrm{ACC},t-1}) = \mathsf{ACC.Prep\text{-}Read}(\widetilde{\mathrm{PP}}_{\mathrm{ACC}}, \widetilde{\mathrm{STORE}}_{\ell_x+t-1}, \mathrm{POS}_{\widetilde{M},t-1})$.

(b) It computes $\widetilde{\mathrm{AUX}}_{\ell_x+t} = \mathsf{ACC.Prep\text{-}Write}(\widetilde{\mathrm{PP}}_{\mathrm{ACC}}, \widetilde{\mathrm{STORE}}_{\ell_x+t-1}, \mathrm{POS}_{\widetilde{M},t-1})$.

(c) It computes $\widetilde{\mathrm{OUT}} = \widetilde{\mathcal{P}}_{\mathrm{DCPRF}}(t, \widetilde{\mathrm{SEED}}_{t-1}, \mathrm{POS}_{\widetilde{M},t-1}, \mathrm{SYM}_{\widetilde{M},t-1}, \mathrm{ST}_{\widetilde{M},t-1}, \widetilde{w}_{\ell_x+t-1}, \widetilde{\pi}_{\mathrm{ACC},t-1}, \widetilde{\mathrm{AUX}}_{\ell_x+t}, \widetilde{v}_{\ell_x+t-1}, h, \ell_x, \widetilde{\sigma}_{\mathrm{SPS},t-1})$.

(d) If $t = \widetilde{t}_x$, it sets $\widetilde{\mathrm{CT}}_{\mathrm{PKE}} = \widetilde{\mathrm{OUT}}$. Otherwise, it parses $\widetilde{\mathrm{OUT}}$ as $\widetilde{\mathrm{OUT}} = (\mathrm{POS}_{\widetilde{M},t}, \mathrm{SYM}^{(\mathrm{WRITE})}_{\widetilde{M},t}, \mathrm{ST}_{\widetilde{M},t}, \widetilde{w}_{\ell_x+t}, \widetilde{v}_{\ell_x+t}, \widetilde{\sigma}_{\mathrm{SPS},t}, \widetilde{\mathrm{SEED}}_t)$.

(e) It computes $\widetilde{\mathrm{STORE}}_{\ell_x+t} = \mathsf{ACC.Write\text{-}Store}(\widetilde{\mathrm{PP}}_{\mathrm{ACC}}, \widetilde{\mathrm{STORE}}_{\ell_x+t-1}, \mathrm{POS}_{\widetilde{M},t-1}, \mathrm{SYM}^{(\mathrm{WRITE})}_{\widetilde{M},t})$.

(C) Finally, it computes

- $\widetilde{r}_{\mathrm{PKE},1} \| \widetilde{r}_{\mathrm{PKE},2} = \mathcal{F}(\widetilde{K}', (h, \ell_x))$,
- $(\widetilde{\mathrm{PK}}_{\mathrm{PKE}}, \widetilde{\mathrm{SK}}_{\mathrm{PKE}}) = \mathsf{PKE.Setup}(1^\lambda;\ \widetilde{r}_{\mathrm{PKE},1})$,
- $r_{\mathrm{PKE},1} \| r_{\mathrm{PKE},2} = \mathsf{PKE.Decrypt}(\widetilde{\mathrm{SK}}_{\mathrm{PKE}}, \widetilde{\mathrm{CT}}_{\mathrm{PKE}})$,
- $(\mathrm{PK}_{\mathrm{PKE}}, \mathrm{SK}_{\mathrm{PKE}}) = \mathsf{PKE.Setup}(1^\lambda;\ r_{\mathrm{PKE},1})$,

and outputs $\mathsf{PKE.Decrypt}(\mathrm{SK}_{\mathrm{PKE}}, \mathrm{CT}_{\mathrm{PKE}})$.

Theorem 5.1. *Assuming $\mathcal{IO}$ is a secure indistinguishability obfuscator for* P/poly*, $\mathcal{F}$ is a secure puncturable pseudorandom function,* SSB *is a somewhere statistically binding hash function,* ACC *is a secure positional accumulator,* ITR *is a secure cryptographic iterator,* SPS *is a secure splittable signature scheme,* PRG *is a secure injective pseudorandom generator, and* PKE *is* CPA *secure, our* DCPRF *construction satisfies the correctness and selective pseudorandomness properties.*

The proof of Theorem 5.1 is given in the full version of this paper.

References

1. Abusalah, H., Fuchsbauer, G.: Constrained prfs for unbounded inputs with short keys. In: Applied Cryptography and Network Security–ACNS 2016. pp. 445–463. Springer (2016)
2. Abusalah, H., Fuchsbauer, G., Pietrzak, K.: Constrained prfs for unbounded inputs. In: Topics in Cryptology–CT-RSA 2016. pp. 413–428. Springer (2016)
3. Ananth, P., Chen, Y.C., Chung, K.M., Lin, H., Lin, W.K.: Delegating ram computations with adaptive soundness and privacy. In: Theory of Cryptography–TCC 2016-B. pp. 3–30. Springer (2016)
4. Banerjee, A., Fuchsbauer, G., Peikert, C., Pietrzak, K., Stevens, S.: Key-homomorphic constrained pseudorandom functions. In: Theory of Cryptography–TCC 2015. pp. 31–60. Springer (2015)
5. Bellare, M., Fuchsbauer, G.: Policy-based signatures. In: Public Key Cryptography–PKC 2014. pp. 520–537. Springer (2014)
6. Boneh, D., Waters, B.: Constrained pseudorandom functions and their applications. In: Advances in Cryptology–ASIACRYPT 2013. pp. 280–300. Springer (2013)
7. Boyle, E., Goldwasser, S., Ivan, I.: Functional signatures and pseudorandom functions. In: Public Key Cryptography–PKC 2014. pp. 501–519. Springer (2014)

8. Brakerski, Z., Vaikuntanathan, V.: Constrained key-homomorphic prfs from standard lattice assumptions. In: Theory of Cryptography–TCC 2015. pp. 1–30. Springer (2015)
9. Chandran, N., Raghuraman, S., Vinayagamurthy, D.: Constrained pseudorandom functions: Verifiable and delegatable. Cryptology ePrint Archive, Report 2014/522 (2014)
10. Deshpande, A., Koppula, V., Waters, B.: Constrained pseudorandom functions for unconstrained inputs. In: Advances in Cryptology–EUROCRYPT 2016. pp. 124–153. Springer (2016)
11. Deshpande, A., Koppula, V., Waters, B.: Constrained pseudorandom functions for unconstrained inputs. Cryptology ePrint Archive, Report 2016/301, Version 20160819:153952 (2016)
12. Fuchsbauer, G.: Constrained verifiable random functions. In: Security and Cryptography for Networks–SCN 2014. pp. 95–114. Springer (2014)
13. Fuchsbauer, G., Konstantinov, M., Pietrzak, K., Rao, V.: Adaptive security of constrained prfs. In: Advances in Cryptology–ASIACRYPT 2014. pp. 82–101. Springer (2014)
14. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on. pp. 40–49. IEEE (2013)
15. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. Journal of the ACM (JACM) 33(4), 792–807 (1986)
16. Hofheinz, D., Kamath, A., Koppula, V., Waters, B.: Adaptively secure constrained pseudorandom functions. Cryptology ePrint Archive, Report 2014/720 (2014)
17. Hohenberger, S., Koppula, V., Waters, B.: Adaptively secure puncturable pseudorandom functions in the standard model. In: Advances in Cryptology–ASIACRYPT 2015. pp. 79–102. Springer (2015)
18. Hubacek, P., Wichs, D.: On the communication complexity of secure function evaluation with long output. In: The 2015 Conference on Innovations in Theoretical Computer Science. pp. 163–172. ACM (2015)
19. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: The 2013 ACM SIGSAC conference on Computer & communications security. pp. 669–684. ACM (2013)
20. Koppula, V., Lewko, A.B., Waters, B.: Indistinguishability obfuscation for turing machines with unbounded memory. In: The 47th Annual ACM on Symposium on Theory of Computing. pp. 419–428. ACM (2015)
21. Micali, S., Rabin, M., Vadhan, S.: Verifiable random functions. In: Foundations of Computer Science, 1999. 40th Annual Symposium on. pp. 120–130. IEEE (1999)
22. Okamoto, T., Pietrzak, K., Waters, B., Wichs, D.: New realizations of somewhere statistically binding hashing and positional accumulators. In: Advances in Cryptology–ASIACRYPT 2015. pp. 121–145. Springer (2015)
23. Sahai, A., Waters, B.: How to use indistinguishability obfuscation: deniable encryption, and more. In: The 46th Annual ACM Symposium on Theory of Computing. pp. 475–484. ACM (2014)