Designs, Codes and Cryptography (2023) 91:1845–1872
https://doi.org/10.1007/s10623-022-01163-8

Short attribute-based signatures for arbitrary Turing machines from standard assumptions

Pratish Datta[1] · **Ratna Dutta**[2] · **Sourav Mukhopadhyay**[2]

Received: 26 June 2021 / Revised: 28 May 2022 / Accepted: 6 December 2022 /
Published online: 17 January 2023

Abstract
This paper presents the *first attribute-based signature* (ABS) scheme supporting signing policies representable by *Turing machines* (TM), based on *well-studied computational assumptions*. Our work supports *arbitrary* TMs as signing policies in the sense that the TMs can accept signing attribute strings of *unbounded* polynomial length and there is *no limit* on their running time, description size, or space complexity. Moreover, we are able to achieve *input-specific* running time for the signing algorithm. All other known expressive ABS schemes could at most support signing policies realizable by either arbitrary polynomial-size circuits or TMs having a pre-determined upper bound on the running time. Consequently, those schemes can only deal with signing attribute strings whose lengths are a priori bounded, as well as suffers from the worst-case running time problem. On a more positive note, for the *first* time in the literature, the signature size of our ABS scheme only depends on the size of the signed message and is completely *independent* of the size of the signing policy under which the signature is generated. This is a significant achievement from the point of view of communication efficiency. Our ABS construction makes use of indistinguishability obfuscation (IO) for polynomial-size circuits and certain IO-compatible cryptographic tools. Note that, all of these building blocks including IO for polynomial-size circuits are currently known to be realizable under well-studied computational assumptions.

Keywords Attribute-based signatures · Turing machines · Standard assumptions · Indistinguishability obfuscation

Mathematics Subject Classification 94A60

Communicated by R. Steinfeld.

✉ Pratish Datta
pratish.datta@ntt-research.com

Ratna Dutta
ratna@maths.iitkgp.ernet.in

Sourav Mukhopadhyay
sourav@maths.iitkgp.ernet.in

[1] CIS Lab, NTT Research, Inc., Sunnyvale, CA, USA

[2] Department of Mathematics, IIT Kharagpur, Kharagpur, West Bengal, India

1 Introduction

In a traditional digital signature scheme, each signer possesses a secret signing key and publishes its corresponding verification key. A signature on some message issued by a certain signer is verified with respect to the public verification key of the respective signer, and hence during the verification process, the explicit signer gets identified. In other words, standard digital signatures can guarantee no privacy in the relationship between signers and claims attested by signatures due to the tight correspondence between the signing and verification keys.

Attribute-based signatures (ABS), introduced by Maji et al. [19], aims to relax such a firm relationship between signers and signatures issued by them, thereby ensuring some form of *signer privacy*. ABS comes in two flavors, namely, *key-policy* and *signature-policy*. In a key-policy ABS scheme, a setup authority holds a master signing key and publishes system public parameters. Using its master signing key, the authority can give out restricted signing keys corresponding to specific signing policies. Such a constrained signing key enables a signer to sign messages with respect to only those signing attributes which are accepted by the signing policy embedded within the signing key. The signatures are verifiable by anyone using solely the public parameters. By verifying a signature on some message with respect to some signing attributes, a verifier gets convinced that the signature is indeed generated by a signer possessing a signing key corresponding to some signing policy that accepts the signing attributes. However, the verifier cannot trace the exact signer or signing policy used to generate the signature. The signature-policy variant interchanges the roles of signing attributes and signing policies. Other than being an exciting primitive in its own right, ABS has countless practical applications such as attribute-based messaging, attribute-based authentication, anonymous credential systems, trust negotiation, and leaking secrets.

A central theme of research in the field of ABS has been to expand the class of admissible signing policies in view of implementing ABS in scenarios where the correspondence between signers and signatures is more and more sophisticated. Starting with the initial work of Maji et al. [19], which supports signing policies representable by monotone span programs, the family of supported signing policies has been progressively enlarged by Okamoto and Takashima [20] to admit non-monotone span programs, by Datta et al. [6] to support arithmetic branching programs, and further by Tang et al. [25], Sakai et al. [23], Tsabary [26], as well as El Kaafarani and Katsumata [16] to realize arbitrary polynomial-size circuits. On the other hand, Bellare and Fuchsbauer [4] have put forth a versatile cryptographic primitive termed as policy-based signatures (PBS) and have exhibited a generic transformation from PBS to ABS. Their generic conversion can be used in conjunction with their proposed PBS construction to build an ABS scheme for general polynomial-size circuits as well.

While the circuit model is already powerful enough to capture arbitrary computations, an important *bottleneck* of this model is that it is *non-uniform* in nature and thus ABS schemes supporting circuit-realizable signing policies can withstand only signing attribute strings of *bounded length*, where the bound is determined during setup. Another drawback of representing signing policies as circuits is that generating a signature with respect to some signing attribute string using a signing key corresponding to certain signing policy is at least as slow as the *worst-case running time* of that policy circuit on all possible signing attribute strings. These are serious limitations not only for ABS itself, but also for all the aforementioned applications of ABS

In this paper, we aim to express signing policies in a *uniform* computational model, namely, the *Turing machine* (TM) model, which is the most natural direction to overcome the above problems. First, we would like to mention that concurrently and independently to our work, Sakai et al. [24] have developed an ABS scheme which can withstand TM-realizable signing policies under the symmetric external Diffie–Hellman (SXDH) assumption. Unfortunately however, in their ABS scheme, the size of a signature scales with the running time of the signing policy TM used to generate it on the signing attribute string with respect to which it is created. As a result, for ensuring signer privacy, their scheme should impose a universal upper bound on the running times of the signing policy TMs, and should enforce the size of the signatures to scale with that system-wide upper bound. Evidently, such a universal running-time bound in turn induces a bound on the lengths of the allowable signing attribute strings. Moreover, it implies that the signing algorithm should also have running time proportional to that universal time bound, i.e., incurs the worst-case running time in order to generate the signatures. Consequently, it is clear that their scheme actually *fails to achieve* both the advanced properties which are the sole utility of considering the richer TM model over the circuit model, namely, unbounded-length signing attribute strings and input-specific running time of the signing algorithm. Further, the failure to achieve these rich properties is in fact the result of their approach that involves giving out non-interactive zero-knowledge (NIZK) proofs for each of the evaluation steps of the signing policy TM on the signing attribute string considered in a manner analogous to how an NIZK proof is issued for each gate of the signing policy circuit in [23]. In contrast, our goal in this paper is to devise techniques to accomplish both the rich properties expected from the TM model and thereby truly expand the state of the art in the field of ABS beyond the essential barriers of the circuit model. Additionally, we aim at making the signature size as small as that of an ordinary digital signature scheme, that is, dependent only on the size of the signed message—a feature that has remained elusive despite the tremendous progress in the field of ABS so far.

1.1 Our contribution

In this paper, we present the *first* ever key-policy ABS scheme supporting signing policies representable as *Turing machines* (TM) which can handle signing attribute strings of *unbounded polynomial length*, as well as have *arbitrary* (polynomial) running time, description size, and space complexity. Thus, our work captures the *most general* form of signing policies possible. Moreover, in our ABS scheme, generating a signing key corresponding to a signing policy takes time polynomial in the description size of that signing policy, which may be much shorter compared to the worst-case running time of that signing policy. Also, the signature generation time only depends on the time the used signing policy takes to run on the signing attribute string with respect to which the signature is being generated, rather than its worst-case running time. These features were *beyond* the reach of any other known ABS construction. On a more positive note, for the *first* time in the literature, the signature size of our ABS scheme only depends on the size of the signed message and is completely *independent* of the associated signing policy. This is a significant achievement from the point of view of communication efficiency. Further, using the technique of universal TM, our key-policy ABS construction can be readily converted into a signature-policy variant while preserving the same level of expressiveness as the key-policy version. Table 1presents a comparison of our work and prior works in the area.

Our ABS construction is shown to possess *perfect signer privacy* and *existential unforgeability against selective attribute adaptive chosen message attacks* under *well-studied computational assumptions*. The construction makes use of indistinguishability obfuscation

Table 1 State of the art in ABS

Scheme	Supported signing policies	Policy assignment	Complexity assumption	Attribute length	Signature size	Signing time
SAH16 [23]	Generic Boolean formulas	Key/signature	SXDH	Bounded	$\mathsf{poly}(\lvert\mathsf{msg}\rvert, \lvert\mathbb{A}\rvert)$	Worst-case
SKAH18 [24]	TM	Key/signature	SXDH	Bounded	$\mathsf{poly}(\lvert\mathsf{msg}\rvert, \lvert\mathbb{A}\rvert)$	Worst-case
KK18 [16]	Arithmetic formulas	Signature	SIS and LWE	Bounded	$\mathsf{poly}(\lvert\mathsf{msg}\rvert, \lvert\mathbb{A}\rvert)$	Worst-case
DOT19 [6]	ABP	Signature	DLIN	Bounded	$\mathsf{poly}(\lvert\mathsf{msg}\rvert, \lvert\mathbb{A}\rvert)$	Worst-case
OT11 [20]	Non-Monotone Boolean formulas	Signature	DLIN	Bounded	$\mathsf{poly}(\lvert\mathsf{msg}\rvert, \mathsf{inp}(\mathbb{A}))$	Worst-case
MPR11 [19]	MSP	Signature	GGM	Bounded	$\mathsf{poly}(\lvert\mathsf{msg}\rvert, \lvert\mathbb{A}\rvert)$	Worst-case
TLL14 [25]	Monotone Boolean formulas	Key	MCDH	Bounded	$\mathsf{poly}(\lvert\mathsf{msg}\rvert, \mathsf{inp}(\mathbb{A}) + \mathsf{dep}(\mathbb{A}))$	Worst-case
Tsa18 [26]	Generic Boolean formulas	Key	SIS	Bounded	$\mathsf{poly}(\lvert\mathsf{msg}\rvert, \mathsf{inp}(\mathbb{A}), \mathsf{dep}(\mathbb{A}))$	Worst-case
This work	TM	Key/signature	$\overbrace{[\mathsf{LPN} + \mathsf{DLIN} + \mathsf{PRG}]}^{\mathsf{IO}} + \mathsf{DDH/DCR} + \mathsf{OWF}$	Unbounded	$\mathsf{poly}(\lvert\mathsf{msg}\rvert)$	Input-specific

The notations used in this table have the following meanings: ABP: Arithmetic Branching Programs; MSP: Monotone Span Programs; TM: turing machines; SXDH: symmetric external Diffie–Hellman; SIS: short integer solution problem; LWE: learning with errors problem; DLIN: decisional linear assumption; GGM: generic group model; MCDH: multilinear computational Diffie–Hellman; IO: indistinguishability obfuscation; LPN: learning parity with noise assumption; PRG: Boolean pseudo-random generator in NC^0; DDH: decisional Diffie–Hellman assumption; DCR: decisional composite residuocity assumption; OWF: one-way functions; poly: arbitrary polynomial; msg: message; $\mathbb{A}$: access policy; $\lvert\mathsf{x}\rvert$: size of x; $\mathsf{inp}(x)$: input length of x; $\mathsf{dep}(x)$: depth of x

(IO) for polynomial-size circuits. Other than IO, we make use of standard digital signatures (SIG), injective pseudorandom generators (PRG), and certain additional IO-compatible cryptographic tools, namely, puncturable pseudorandom functions, somewhere statistically binding (SSB) hash functions, positional accumulators, cryptographic iterators, and splittable signatures. Among the cryptographic building blocks used in our ABS construction in addition to IO, iterators and splittable signatures are realizable using IO itself in conjunction with one-way functions, whereas all the others have efficient instantiations based on standard number theoretic assumptions or one-way functions. Very recently, a series of works [2, 9, 12–15] have finally provided an IO candidate based on the sub-exponential security of three well-studied computational assumptions, namely, learning parity with noise (LPN), existence of boolean pseudorandom generators (PRG) in NC^0, and Decisional Linear (DLIN).

We note that while the proposed ABS scheme demonstrates asymptotically better performance compared to existing schemes as described above, the concrete computational overhead might not outweigh that of existing schemes at this point primarily due to fact that the current realization of IO is highly inefficient. However, IO research has so far advanced a long way over the last two decade since its inception by Barak et al. [3], and we believe efficient IO candidates would be discovered in the future improving the concrete overhead of the proposed ABS scheme. On the other hand, the existing ABS schemes, especially those supporting comparably expressive signing policies [16, 23–26] would still continue to suffer from the asymptotic worst-case efficiency bottleneck. Another limitation of the proposed ABS scheme is that it only achieves selective unforgeability as mentioned above. We leave it as an interesting open problem to construct an ABS scheme for the same class of access policies and with the same asymptotic efficiency as ours while achieving adaptive security at the same time.

To achieve our result, we *extend* the techniques employed by Koppula et al. [17] for designing message-hiding encoding schemes for TMs, or by Deshpande et al. [7] for designing constrained pseudorandom functions (CPRF) for TMs secure in the selective challenge selective constraints model to withstand *adaptive* signing key queries of the adversary. We give an overview of our techniques in the next section.

2 Preliminaries

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, $\upsilon \xleftarrow{\$} S$ represents the uniform random variable on S. For a randomized algorithm $\mathcal{R}$, we denote by $\psi = \mathcal{R}(\upsilon; \rho)$ the random variable defined by the output of $\mathcal{R}$ on input υ and randomness ρ, while $\psi \xleftarrow{\$} \mathcal{R}(\upsilon)$ has the same meaning with the randomness suppressed. Also, if $\mathcal{R}$ is a deterministic algorithm $\psi = \mathcal{R}(\upsilon)$ denotes the output of $\mathcal{R}$ on input υ. We will use the alternative notation $\mathcal{R}(\upsilon) \rightarrow \psi$ as well to represent the output of the algorithm $\mathcal{R}$, whether randomized or deterministic, on input υ. 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'. A function negl is *negligible* if for every integer c, there exists an integer k such that for all $\lambda > k$, $|\mathsf{negl}(\lambda)| < 1/\lambda^c$.

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

At each time step, the TM 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, $\text{ST}_{M,t} = q$, $\text{SYM}^{(\text{WRITE})}_{M,t} = \zeta$, 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 [3]) An indistinguishability obfuscator (IO) $\mathcal{IO}$ for a 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

$$\mathsf{Pr}\big[(C_0, C_1, \xi) \xleftarrow{\$} \mathcal{D}_1(1^\lambda) \ : \ \forall \upsilon, C_0(\upsilon) = C_1(\upsilon)\big] \geq 1 - \mathsf{negl}(\lambda),$$
$$\text{then } \big|\mathsf{Pr}\big[\mathcal{D}_2(\xi, \mathcal{IO}(1^\lambda, C_0)) = 1\big] - \mathsf{Pr}\big[\mathcal{D}_2(\xi, \mathcal{IO}(1^\lambda, C_1)) = 1\big]\big| \leq \mathsf{negl}(\lambda).$$

- ▶ **Efficiency**: For any security parameter λ and any circuit $C \in \mathbb{C}_\lambda$, the size of the obfuscated circuit $\mathcal{IO}(1^\lambda, C)$ is polynomial in λ and the size of C.

We remark that the two distinct algorithms $\mathcal{D}_1$ and $\mathcal{D}_2$, which pass state ξ, can be viewed equivalently as a single stateful algorithm $\mathcal{D}$. In this paper we employ the latter approach, although here we present the definition as it appears in [3]. When clear from the context, we will drop 1^λ as an input to $\mathcal{IO}$.

The circuit class we are interested in are polynomial-size circuits, i.e., when $\mathbb{C}_\lambda$ is the collection of all circuits of size at most λ. This circuit class is denoted by $\mathsf{P/poly}$. The first candidate construction of IO for $\mathsf{P/poly}$ was presented by Garg et al. [8] in 2013. Their construction uses nonstandard instance dependent assumption on graded multilinear encodings. Since then, there has been a rapid progress towards designing IO from better understood cryptographic tools and complexity assumptions. Very recently, a series of exciting works [1, 2, 9, 12, 13, 15, 18] have finally provided an IO candidate based on the sub-exponential security of four well-studied computational assumptions, namely, learning with errors (LWE), learning parity with noise (LPN), existence of boolean pseudorandom generators (PRG) in NC^0, and symmetric external Diffie–Hellman (SXDH).

2.3 IO-compatible cryptographic primitives

In this section, we present the syntax and efficiency considerations of certain IO-friendly cryptographic tools which we use in our ABS construction. The security properties these primitives are described in Online Appendix A.

2.3.1 Puncturable pseudorandom function

Pseudorandom functions (PRF) [10] are a fundamental tool of 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 puncturable pseudorandom function (PPRF), first introduced by Sahai and Waters [22], is an augmentation of a PRF with an additional puncturing algorithm which enables a party holding a PRF key to derive punctured keys that allow the evaluation of the PRF over all points of the input domain except one. However, given a punctured key, the PRF evaluation still remains indistinguishable from random on the input at which the key is punctured.

Puncturable pseudorandom function PPRF [22]: A puncturable pseudorandom function (PPRF) $\mathcal{F} : \mathcal{K}_{\text{PPRF}} \times \mathcal{X}_{\text{PPRF}} \to \mathcal{Y}_{\text{PPRF}}$ 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 ($\mathcal{F}.\mathsf{Setup}, \mathcal{F}.\mathsf{Eval}, \mathcal{F}.\mathsf{Puncture}, \mathcal{F}.\mathsf{Eval\text{-}Punctured}$) 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,

$\mathcal{F}.\mathsf{Setup}(1^\lambda) \to K$: The setup authority takes as input the security parameter 1^λ and uniformly samples a PPRF key $K \in \mathcal{K}_{\text{PPRF}}$.

$\mathcal{F}.\mathsf{Eval}(K, x) \to r$: 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.

$\mathcal{F}.\mathsf{Puncture}(K, x) \to K\{x\}$: 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 $K\{x\} \in \mathcal{K}_{\text{PPRF-PUNC}}$.

$\mathcal{F}.\mathsf{Eval\text{-}Puncured}(K\{x\}, x') \to r$ 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

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

▶ **Efficiency**: The $\mathcal{F}$.Setup algorithm runs in time polynomial in the security parameter λ, while the algorithms $\mathcal{F}$.Eval and $\mathcal{F}$.Punctured run in time polynomial in λ and the input size $\ell_{\text{PPRF-INP}}$. Moreover, the size of the full PPRF keys is polynomial in λ, whereas that of the punctured keys is polynomial in λ and $\ell_{\text{PPRF-INP}}$. Hence, the algorithm $\mathcal{F}$.Eval-Punctured also runs in time polynomial in λ and $\ell_{\text{PPRF-INP}}$. Boneh and Waters [5], have shown that the tree-based PRF constructed by Goldreich et al. [10] can be readily modified to build a PPRF from one-way functions.

2.3.2 Somewhere statistically binding hash function

We provide the definition of somewhere statistically binding hash function as defined by Hubacek et al. [11]. A somewhere statistically binding hash can be used to create a short digest of some long string. A hashing key is created by specifying a special binding index and the generated hashing key gets the property that the hash value of some string created with the hashing key is statistically binding for the specified index, meaning that the hash value completely determines the symbol of the hashed input at that index. In other words, even if some hash value has several pre-images, all of those pre-images agree in the symbol at the specified index. The index on which the hash is statistically binding should remain computationally hidden given the hashing key. Moreover, it is possible to prove that the input string underlying a given hash value contains a specific symbol at a particular index, by providing a short opening value.

Somewhere statistically binding hash function SSB *Hash* [11]: 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:

$\mathsf{SSB.Gen}(1^\lambda, n_{\text{SSB-BLK}}, i^*) \to \text{HK}$: 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 SSB.Gen and SSB.Open are randomized, while the algorithm 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 $\mathsf{SSB.Verify}(\text{HK}, \mathcal{H}_{\text{HK}}(x), i, x_i, \pi_{\text{SSB}}) = 1$.
- **Efficiency**: The SSB.Gen algorithm runs in time polynomial in the security parameter λ and $\log n_{\text{SSB- BLK}}$. Moreover, the hash and opening values have size polynomial in λ and $\log n_{\text{SSB- BLK}}$. Hence, the SSB.Verify algorithm also runs in time polynomial in λ and $\log n_{\text{SSB- BLK}}$. On the other hand, the algorithms $\mathcal{H}_{\text{HK}}$ and SSB.Open run in time polynomial in λ and $n_{\text{SSB- BLK}}$ in the worst case. The first construction of an SSB hash is presented by Hubacek et al. [11]. Their construction is based on fully homomorphic encryption (FHE). Recently, Okamoto et al.[21] provides alternative constructions of SSB hash based on various standard number theoretic assumptions such as the Decisional Diffie-Hellman assumption. In this paper, we consider $\ell_{\text{SSB- BLK}} = 1$ and $n_{\text{SSB- BLK}} = 2^{\lambda}$.

2.3.3 Positional accumulator

We will now present the notion of a positional accumulator as defined by Koppula et al. [17]. Intuitively, a positional accumulator is a cryptographic data structure that maintains two values, namely, a storage value and an accumulator value. The storage value is allowed to grow comparatively large, while the accumulator value is constrained to be short. Message symbols can be written to various positions in the underlying storage, and new accumulated values can be computed as a string, knowing only the previous accumulator value and the newly written symbol together with its position in the data structure. Since the accumulator values are small, one cannot hope to recover everything written in the storage from the accumulator value alone. However, there are additional helper algorithms which essentially allow a party who is maintaining the full storage to help a more restricted party maintaining only the accumulator value recover the data currently written at an arbitrary location. The helper is not necessarily trusted, so the party maintaining the accumulator value performs a verification procedure in order to be convinced that it is indeed reading the correct symbols.
Positional accumulator [17]: A positional accumulator consists of PPT algorithms (ACC.Setup, ACC.Setup-Enforce-Read, ACC.Setup-Enforce-Write, ACC.Prep-Read, ACC. Prep-Write, ACC.Verify-Read, ACC.Write-Store, 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:

$\mathsf{ACC.Setup}(1^{\lambda}, n_{\text{ACC- BLK}}) \rightarrow (\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0)$: 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.

$\mathsf{ACC.Setup\text{-}Enforce\text{-}Read}(1^{\lambda}, n_{\text{ACC- BLK}}, ((x_1, i_1), \ldots, (x_{\kappa}, i_{\kappa})), i^*) \rightarrow (\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0)$: 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.

$\mathsf{ACC.Setup\text{-}Enforce\text{-}Write}(1^{\lambda}, n_{\text{ACC- BLK}}, ((x_1, i_1), \ldots, x_{\kappa}, i_{\kappa}))) \rightarrow (\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0)$: 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.

ACC.Prep-Read$(\text{PP}_{\text{ACC}}, \text{STORE}_{\text{IN}}, i_{\text{IN}}) \rightarrow (x_{\text{OUT}}, \pi_{\text{ACC}})$: 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 $\pi_{\text{ACC}} \in \Pi_{\text{ACC}}$.

ACC.Prep-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.

ACC.Verify-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 $\hat{\beta} \in \{0, 1\}$.

ACC.Write-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 $\text{STORE}_{\text{OUT}}$.

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 [7, 17], 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 $(\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0) \xleftarrow{\$} \text{ACC.Setup}(1^\lambda, n_{\text{ACC-BLK}})$. For $j = 1, \ldots, \kappa$, iteratively define the following:

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

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 $(\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0) \xleftarrow{\$} \text{ACC.Setup}(1^\lambda, n_{\text{ACC-BLK}})$, if STORE_κ is computed as above, then $\text{ACC.Prep-Read}(\text{PP}_{\text{ACC}}, \text{STORE}_\kappa, i^*)$ 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 $(\text{PP}_{\text{ACC}}, w_0, \text{STORE}_0) \xleftarrow{\$} \text{ACC.Setup}(1^\lambda, n_{\text{ACC-BLK}})$, if STORE_κ and w_κ are computed as above and $(x_{\text{OUT}}, \pi_{\text{ACC}}) = \text{ACC.Prep-Read}(\text{PP}_{\text{ACC}}, \text{STORE}_\kappa, i^*)$, then $\text{ACC.Verify-Read}(\text{PP}_{\text{ACC}}, w_\kappa, x_{\text{OUT}}, i^*, \pi_{\text{ACC}}) = 1$

▶ **Efficiency**: The ACC.Setup algorithm runs in time polynomial in the security parameter λ and $\log n_{\text{ACC-BLK}}$. Moreover, the accumulator values and proofs have size polynomial in λ and $\log n_{\text{ACC-BLK}}$. Thus, the algorithms ACC.Verify-Read and ACC.Update also run in time polynomial in λ and $\log n_{\text{ACC-BLK}}$. The storage has size polynomial in the number of values stored so far. Hence, the algorithms $\mathsf{ACC.Write\text{-}Store}$, $\mathsf{ACC.Prep\text{-}Read}$, and $\mathsf{ACC.Prep\text{-}Write}$ run in time polynomial in λ and $n_{\text{ACC-BLK}}$ in the worst case. The first construction of a positional accumulator is presented by Koppula et al. [17] based on IO and one-way function. Recently, Okamoto et al. [21] provided an alternative construction of positional accumulator from standard number theoretic assumptions. Such as the Decisional Diffie–Hellman assumption.

2.3.4 Iterator

Here, we define cryptographic iterators again following [17]. Informally speaking, a cryptographic iterator consists of a small state that is updated in an iterative fashion as messages are received. An update to incorporate a new message given the current state is performed with the help of some public parameters. Since, states are relatively small regardless of the number of messages that have been iteratively incorporated, there is in general many sequences of messages that lead to the same state. However, the security property requires that the normal public parameters should be computationally indistinguishable from specially constructed enforcing parameters which ensure that a particular state can only be obtained as the outcome of an update to precisely one other state-message pair. Note that this enforcement is a very localized property to a specific state and hence can be achieved information-theoretically when it is fixed ahead of time where exactly this enforcement is desired.

Iterator [17]: A cryptographic iterator consists of PPT algorithms ($\mathsf{ITR.Setup}$, $\mathsf{ITR.Setup\text{-}Enforce}$, $\mathsf{ITR.Iterate}$) 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:

$\mathsf{ITR.Setup}(1^\lambda, n_{\text{ITR}}) \rightarrow (\text{PP}_{\text{ITR}}, v_0)$: 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 $v_0 \in \{0,1\}^{\ell_{\text{ITR-ST}}}$.

$\mathsf{ITR.Setup\text{-}Enforce}(1^\lambda, n_{\text{ITR}}, (\mu_1, \ldots, \mu_\kappa)) \rightarrow (\text{PP}_{\text{ITR}}, v_0)$: 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 $v_0 \in \{0,1\}^{\ell_{\text{ITR-ST}}}$.

$\mathsf{ITR.Iterate}(\text{PP}_{\text{ITR}}, v_{\text{IN}} \in \{0,1\}^{\ell_{\text{ITR-ST}}}, \mu) \rightarrow v_{\text{OUT}}$: 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.

► **Efficiency**: The algorithms $\mathsf{ITR.Setup}$ and $\mathsf{ITR.Iterate}$ run in time polynomial in the security parameter λ and $\log n_{\text{ITR}}$. Also, the state has size polynomial in λ and $\log n_{\text{ITR}}$. Koppula et al. [17] have presented a construction of cryptographic iterators from IO and one-way function.

2.3.5 Splittable signature

The following background on splittable signatures is taken verbatim from Koppula et al. [17] as well. A splittable signature scheme is essentially a normal signature scheme augmented by some additional algorithms that produce alternative signing and verification keys with different capabilities. More precisely, there are "all-but-one" signing and verification keys which work correctly for all messages except for a specific one, as well as there are "one" signing and verification keys which work only for a particular message. Additionally, there are "reject" verification keys which always reject signatures.

Splittable signature SPS [17]: 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 (SPS.Setup, SPS.Sign, SPS.Verify, SPS.Split, SPS.Sign-ABO) which are described below:

$\mathsf{SPS.Setup}(1^\lambda) \rightarrow (\text{SK}_{\text{SPS}}, \text{VK}_{\text{SPS}}, \text{VK}_{\text{SPS-REJ}})$: 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 $\text{VK}_{\text{SPS-REJ}}$.

$\mathsf{SPS.Sign}(\text{SK}_{\text{SPS}}, m) \rightarrow \sigma_{\text{SPS}}$: A signer given a signing key SK_{SPS} along with a message $m \in \mathcal{M}_{\text{SPS}}$, produces a signature $\sigma_{\text{SPS}} \in \mathcal{S}_{\text{SPS}}$.

$\mathsf{SPS.Verify}(\text{VK}_{\text{SPS}}, m, \sigma_{\text{SPS}}) \rightarrow \hat{\beta} \in \{0,1\}$: 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 $\hat{\beta} \in \{0,1\}$.

$\mathsf{SPS.Split}(\text{SK}_{\text{SPS}}, m^*) \rightarrow (\sigma_{\text{SPS-ONE},m^*}, \text{VK}_{\text{SPS-ONE}}, \text{SK}_{\text{SPS-ABO}}, \text{VK}_{\text{SPS-ABO}})$: 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 $(\text{SK}_{\text{SPS-ABO}}, \text{VK}_{\text{SPS-ABO}})$.

$\mathsf{SPS.Sign\text{-}ABO}(\text{SK}_{\text{SPS-ABO}}, m) \rightarrow \sigma_{\text{SPS}}$ 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) $\forall m \in \mathcal{M}_{\text{SPS}},\ \sigma_{\text{SPS}} \in \mathcal{S}_{\text{SPS}},\ \mathsf{SPS.Verify}(\text{VK}_{\text{SPS-REJ}}, m, \sigma_{\text{SPS}}) = 0$.

► **Efficiency**: The algorithms SPS.Setup, SPS.Sign, and SPS.Verify run in time polynomial in the security parameter λ and the supported message length $\ell_{\text{SPS-MSG}}$. Koppula et al. [17] have constructed a splittable signature scheme using IO and one-way function.

3 Our attribute-based signature for TMs

3.1 Notion of attribute-based signatures for TMs

First, we will formally define the notion of an attribute-based signature scheme where signing policies are associated with TMs. This definition is similar to that defined in [23, 25] for circuits.

Definition 3.1 (*Attribute-based signature for turing machines:* ABS) Let $\mathbb{M}_\lambda$ be a class of TMs, the members of which have (worst-case) running time bounded by $T = 2^\lambda$. An attribute-based signature (ABS) scheme for signing policies associated with the TMs in $\mathbb{M}_\lambda$ comprises of an attribute universe $\mathcal{U}_{\text{ABS}} \subset \{0, 1\}^*$, a message space $\mathcal{M}_{\text{ABS}} = \{0, 1\}^{\ell_{\text{ABS-MSG}}}$, a signature space $\mathcal{S}_{\text{ABS}} = \{0, 1\}^{\ell_{\text{ABS-SIG}}}$, where $\ell_{\text{ABS-MSG}}, \ell_{\text{ABS-SIG}}$ are some polynomials in the security parameter λ, and PPT algorithms (ABS.Setup, ABS.KeyGen, ABS.Sign, ABS.Verify) described below:

$\mathsf{ABS.Setup}(1^\lambda) \rightarrow (\text{PP}_{\text{ABS}}, \text{MSK}_{\text{ABS}})$: The setup authority takes as input the security parameter 1^λ. It publishes the public parameters PP_{ABS} while generates a master secret key MSK_{ABS} for itself.

$\mathsf{ABS.KeyGen}(\text{MSK}_{\text{ABS}}, M) \rightarrow \text{SK}_{\text{ABS}}(M)$: Taking as input the master secret key MSK_{ABS} and a signing policy TM $M \in \mathbb{M}_\lambda$ of a signer, the setup authority provides the corresponding signing key $\text{SK}_{\text{ABS}}(M)$ to the legitimate signer.

$\mathsf{ABS.Sign}(\text{SK}_{\text{ABS}}(M), x, \mathsf{msg}) \rightarrow \sigma_{\text{ABS}}$ or $\perp$: On input the signing key $\text{SK}_{\text{ABS}}(M)$ corresponding to the legitimate signing policy TM $M \in \mathbb{M}_\lambda$, a signing attribute string $x \in \mathcal{U}_{\text{ABS}}$, and a message $\mathsf{msg} \in \mathcal{M}_{\text{ABS}}$, a signer outputs either a signature $\sigma_{\text{ABS}} \in \mathcal{S}_{\text{ABS}}$ or $\perp$ indicating failure.

$\mathsf{ABS.Verify}(\text{PP}_{\text{ABS}}, x, \mathsf{msg}, \sigma_{\text{ABS}}) \rightarrow \hat{\beta} \in \{0, 1\}$: A verifier takes as input the public parameters PP_{ABS}, a signing attribute string $x \in \mathcal{U}_{\text{ABS}}$, a message $\mathsf{msg} \in \mathcal{M}_{\text{ABS}}$, and a purported signature $\sigma_{\text{ABS}} \in \mathcal{S}_{\text{ABS}}$. It outputs a bit $\hat{\beta} \in \{0, 1\}$.

We note that all the algorithms described above except ABS.Verify are randomized. The algorithms satisfy the following properties:

► **Correctness**: For any security parameter λ, $(\text{PP}_{\text{ABS}}, \text{MSK}_{\text{ABS}}) \xleftarrow{\$} \mathsf{ABS.Setup}(1^\lambda)$, $M \in \mathbb{M}_\lambda$, $\text{SK}_{\text{ABS}}(M) \xleftarrow{\$} \mathsf{ABS.KeyGen}(\text{MSK}_{\text{ABS}}, M)$, $x \in \mathcal{U}_{\text{ABS}}$, and $\mathsf{msg} \in \mathcal{M}_{\text{ABS}}$, if $M(x) = 1$, then $\mathsf{ABS.Sign}(\text{SK}_{\text{ABS}}(M), x, \mathsf{msg})$ outputs $\sigma_{\text{ABS}} \in \mathcal{S}_{\text{ABS}}$ such that ABS.Verify $(\text{PP}_{\text{ABS}}, x, \mathsf{msg}, \sigma_{\text{ABS}}) = 1$.

► **Signer privacy**: An ABS scheme is said to provide signer privacy if for any security parameter λ, message $\mathsf{msg} \in \mathcal{M}_{\text{ABS}}$, $(\text{PP}_{\text{ABS}}, \text{MSK}_{\text{ABS}}) \xleftarrow{\$} \mathsf{ABS.Setup}(1^\lambda)$, signing policies $M, M' \in \mathbb{M}_\lambda$, signing keys $\text{SK}_{\text{ABS}}(M) \xleftarrow{\$} \mathsf{ABS.KeyGen}(\text{MSK}_{\text{ABS}}, M)$, $\text{SK}_{\text{ABS}}(M') \xleftarrow{\$} \mathsf{ABS.KeyGen}(\text{MSK}_{\text{ABS}}, M')$, $x \in \mathcal{U}_{\text{ABS}}$ such that $M(x) = 1 = M'(x)$, the distributions of the signatures outputted by $\mathsf{ABS.Sign}(\text{SK}_{\text{ABS}}(M), x, \mathsf{msg})$ and $\mathsf{ABS.Sign}(\text{SK}_{\text{ABS}}(M'), x, \mathsf{msg})$ are identical.

► **Existential unforgeability against selective attribute adaptive chosen message attack**: This property of an ABS scheme is defined through the following experiment between an adversary $\mathcal{A}$ and a challenger $\mathcal{B}$:

- $\mathcal{A}$ submits a challenge attribute string $x^* \in \mathcal{U}_{\text{ABS}}$ to $\mathcal{B}$.
- $\mathcal{B}$ generates $(\text{PP}_{\text{ABS}}, \text{MSK}_{\text{ABS}}) \xleftarrow{\$} \mathsf{ABS.Setup}(1^\lambda)$ and provides $\mathcal{A}$ with PP_{ABS}.

- $\mathcal{A}$ may adaptively make a polynomial number of queries of the following types:
  - **Signing key query**: When $\mathcal{A}$ queries a signing key corresponding to a signing policy TM $M \in \mathbb{M}_\lambda$ subject to the restriction that $M(x^*) = 0$, $\mathcal{B}$ gives back $\text{SK}_{\text{ABS}}(M) \xleftarrow{\$} \mathsf{ABS.KeyGen}(\text{MSK}_{\text{ABS}}, M)$ to $\mathcal{A}$.
  - **Signature query**: When $\mathcal{A}$ queries a signature on a message $\mathsf{msg} \in \mathcal{M}_{\text{ABS}}$ under an attribute string $x \in \mathcal{U}_{\text{ABS}}$, $\mathcal{B}$ samples a signing policy TM $M \in \mathbb{M}_\lambda$ such that $M(x) = 1$, creates a signing key $\text{SK}_{\text{ABS}}(M) \xleftarrow{\$} \mathsf{ABS.KeyGen}(\text{MSK}_{\text{ABS}}, M)$, and generates a signature $\sigma_{\text{ABS}} \xleftarrow{\$} \mathsf{ABS.Sign}(\text{SK}_{\text{ABS}}(M), x, \mathsf{msg})$, which $\mathcal{B}$ returns to $\mathcal{A}$.
- At the end of interaction $\mathcal{A}$ outputs a message-signature pair $(\mathsf{msg}^*, \sigma^*_{\text{ABS}})$. $\mathcal{A}$ wins if the following hold simultaneously:
  (i) $\mathsf{ABS.Verify}(\text{PP}_{\text{ABS}}, x^*, \mathsf{msg}^*, \sigma^*_{\text{ABS}}) = 1$.
  (ii) $\mathcal{A}$ has not made any signature query on msg^* under x^*.

The ABS scheme is said to be existentially unforgeable against selective attribute adaptive chosen message attack if for any PPT adversary $\mathcal{A}$, for any security parameter λ,

$$\mathsf{Adv}_{\mathcal{A}}^{\text{ABS,UF-CMA}}(\lambda) = \Pr[\mathcal{A} \text{ wins}] \leq \mathsf{negl}(\lambda),$$

for some negligible function negl.

Remark 3.1 Note that in the existential unforgeability experiment above without loss of generality, we can consider signature queries on messages only under the challenge attribute string x^*. This is because any signature query under some attribute string $x \neq x^*$ can be replaced by a signing key query for a signing policy TM $M_x \in \mathbb{M}_\lambda$ that accepts only the string x. Since $x \neq x^*$, $M_x(x^*) = 0$, and thus M_x forms a valid signing key query. We use this simplification in our proof.

3.2 Overview of the proposed ABS scheme

Here we explain the high level technical ideas underlying our ABS scheme. We start by adapting the techniques of Deshpande et al. [7] and Koppula et al. [17] to the ABS setting. Our master signing key consists of a PPRF key K and a set of public parameters PP_{ACC} of a positional accumulator. We assign a different signing key-verification key pair of a standard SIG scheme to each of the possible signing attribute strings and we define such an SIG signing key-verification key pair associated with a signing attribute string x as those obtained by running the setup algorithm of SIG using the randomness obtained by evaluating the underlying PPRF with key K on w_{INP}, where w_{INP} is the accumulation of the bits of x using PP_{ACC}. Our signing key corresponds to some TM M would comprise of two IO-obfuscated programs $\mathcal{P}_1$ and $\mathcal{P}_{\text{ABS}}$. 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, $\mathcal{P}_{\text{ABS}}$, also known as the next step program, has the PPRF key K hardwired in it. It 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 SIG signing-verification key pair, if M reaches the accepting state. More precisely, in case of reaching to the accepting state, $\mathcal{P}_{\text{ABS}}$ first computes the PPRF with key K on input w_{INP}. After that, it generates and outputs a SIG signing key-verification key pair by running the setup algorithm of SIG using the computed pseudorandom string.

The program $\mathcal{P}_{\text{ABS}}$ also performs certain authenticity checks before computing the next step function of M in order to prevent illegal inputs. For this purpose, $\mathcal{P}_{\text{ABS}}$ 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 $\mathcal{P}_{\text{ABS}}$ verifies the signature in order to ensure authenticity, 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 $\mathcal{P}_{\text{ABS}}$ 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 $\mathcal{P}_{\text{ABS}}$ 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.

Our ABS public parameters would contain PP_{ACC} along with the IO-obfuscated verifying program $\mathcal{V}_{\text{ABS}}$, which has the PPRF key K hardwired in it. It takes as input an accumulator value and performs the following steps: First, it runs the PPRF with key K on the accumulator value to generate a pseudorandom string. Next, it runs the setup algorithm of SIG using that pseudorandom string to generate and output the SIG verification key associated with the accumulator value.

In order to sign a message under some signing attribute string accepted by the TM embedded in its signing key, a signer first computes the accumulation w_{INP} of the bits of x using PP_{ACC} which are also included in the PPRF key K, 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 signer then repeatedly runs the program $\mathcal{P}_{\text{ABS}}$, 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 signer also feeds w_{INP} to $\mathcal{P}_{\text{ABS}}$. The iteration is continued until the program $\mathcal{P}_{\text{ABS}}$ either outputs the proper signing-verification key pair or the designated symbol $\perp$ indicating failure. After that, the signer signs the message using the obtained SIG signing key. The ABS signature consists of the generated SIG verification key-signature pair. The signature verification process on some message under some claimed signing attribute string requires first generating the SIG verification key associated with the claimed signing attribute string by accumulating the bits of the claimed signing attribute string using PP_{ACC} and inputting the accumulated value to $\mathcal{V}_{\text{ABS}}$. It then checks whether the generated SIG verification key matches the one included within the ABS signature, as well as whether the SIG signature included within the ABS signature verifies under that SIG verification key.

While the above strategy appears to be sound, there still remain some subtle issues. Observe that to handle the positional accumulator related verifications and updations, the program $\mathcal{P}_{\text{ABS}}$ must have PP_{ACC} hardwired in it. During the course of the selective unforgeability proof, we have to modify the signing keys given to the adversary $\mathcal{A}$ to embed the punctured PPRF key $\text{K}\{w^*_{\text{INP}}\}$ punctured at w^*_{INP} instead of the full PPRF key K. Here, w^*_{INP} is the accumulation of the bits of the challenge signing attribute string x^*, committed by $\mathcal{A}$ at the beginning of the experiment, using PP_{ACC} included within the public parameters given to the adversary $\mathcal{A}$. In order to make this substitution, it is to be ensured that the programs $\mathcal{P}_{\text{ABS}}$ included in those signing keys always outputs $\perp$ for signing attribute strings corresponding to w^*_{INP} even if reaching the accepting state. As usual, we would carry out the transformation one signing key at a time through multiple hybrid steps. Now, suppose for transforming the signing keys we attempt to follow a strategy similar to that of [7, 17]. Let the total number of signing keys queried by $\mathcal{A}$ be $\hat{q}_{\text{KEY}}$. Consider the transformation of the νth signing key ($1 \leq \nu \leq \hat{q}_{\text{KEY}}$) corresponding to the TM $M^{(\nu)}$ that runs on the challenge signing attribute string x^* for $t^{*(\nu)}$ steps and reaches the rejecting state. In the course of transformation, the program $\mathcal{P}^{(\nu)}_{\text{ABS}}$

contained in the νth signing key would first be altered to one that always outputs $\perp$ for inputs corresponding to w^*_{INP} within the first $t^{*(\nu)}$ steps. Towards accomplishing this transition, in successive hybrids, the steps of execution of $M^{(\nu)}$ on x^* would be repeatedly programmed and unprogrammed within $\mathcal{P}^{(\nu)}_{\text{ABS}}$ taking the help of IO. In order to perform this operation using IO, at various stages, we need to guarantee program functional equivalence, and for that we need to generate PP_{ACC} in read/write enforcing mode, certain special statistically binding modes indistinguishable from the normal setup mode. However, in the prefixed version of positional accumulator employed in [7] or in [17], to setup PP_{ACC} in read/write-enforcing mode, we require the entire sequence of symbol-position pairs arising from iteratively running $M^{(\nu)}$ on x^* up to the step we are programming in. This was not a problem for [7, 17] since in their security model the adversary $\mathcal{A}$ was bounded to declare the TM queries prior to setting up the system. On the contrary, in our unforgeability experiment, $\mathcal{A}$ is allowed to adaptively submit signing key queries corresponding to signing policy TMs of its choice throughout the experiment. In such a case, we would be able to determine those symbol-position pairs *only after receiving* the νth queried TM $M^{(\nu)}$ from $\mathcal{A}$. However, we would require PP_{ACC} while creating the signing keys queried by $\mathcal{A}$ before making the νth signing key query, and even for preparing the public parameters. Thus, it is immediate that we must generate PP_{ACC} *prior to receiving* the νth signing key query from $\mathcal{A}$. This is clearly *impossible* as setting PP_{ACC} in read/write enforcing mode requires the knowledge of the TM $M^{(\nu)}$, which is *not available* before the νth signing key query of $\mathcal{A}$.

Observe that the root cause of the problem discussed above is the use of a single set of public parameters PP_{ACC} of the positional accumulator throughout the system. Therefore, we attempt to assign a fresh set of public parameters of the positional accumulator to each signing key. However, for compressing the signing attribute strings to a fixed length, on which the PPRF can be applied to produce the pseudorandom strings specifying the SIG signing key-verification key pairs associated with those signing attribute strings, we need a system-wide compressing tool. We employ a somewhere statistically-binding (SSB) hash function for this purpose.

Our idea is that while signing a message under some signing attribute string x using its legitimate signing key for some TM M, the signer first computes the hash value h by hashing x using the system-wide SSB hash key, which is part of the ABS public parameters. The signer also computes the accumulator value w_{INP} by accumulating the bits of x using the public parameters of positional accumulator, specific to its signing key. Then, using the obfuscated initial signing program $\mathcal{P}_1$ included in its signing key, the signer will obtain a signature on w_{INP} along with the initial state and header position of M. Finally, the signer will repeatedly run the obfuscated next step program $\mathcal{P}_{\text{ABS}}$ included in its signing key, each time giving as input all the quantities as earlier, except that it would now have to feed the SSB hash value h in place of w_{INP} in each iteration. This is because, in case $\mathcal{P}_{\text{ABS}}$ reaches the accepting state, it would require h to apply the PPRF for producing the SIG signing key-verification key pair associated with x. The same change would also apply to the public verifying program $\mathcal{V}_{\text{ABS}}$, namely, it would also take as input the SSB hash value of a signing attribute string in place of the accumulator value obtained by accumulating its bits.

However, this approach is not completely sound yet. Observe that, a possibly malicious signer can compute the SSB hash value h on the signing attribute string x, with respect to which it wishes to generate a signature despite of the fact that its signing policy TM M does not accepts it, but initiates the computation by accumulating the bits of only a substring of x or some entirely different signing attribute string, which is accepted by M. To prevent such malicious behavior, we include another IO-obfuscated program $\mathcal{P}_2$ within the signing key, we call the accumulating program, whose purpose is to restrict the signer from accumulating the bits of a different signing attribute string 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 ith position of the accumulator storage, and signs the updated accumulator value (along with the initial state and header position of M). The signature used by $\mathcal{P}_2$ is also a splittable signature that facilitates the security proof. The obfuscated initial signing program $\mathcal{P}_1$ included in the signing key is also modified to take as input a hash value, and output a signature on the accumulator value corresponding to the empty accumulator storage together with the initial state and header position of M.

Moreover, for forbidding the signer from performing the computation by accumulating an M-accepted substring of the hashed input, we define the SIG signing key-verification key pair associated with a signing attribute string as the output of the setup algorithm of SIG using the pseudorandom string generated by applying the PPRF on the pair (hash value, length) of the signing attribute string instead of just the hash value. Note that, without loss of generality, we can set the upper bound of the length of signing attribute strings to be 2^{λ}, where λ is the underlying security parameter, in view of the fact that by suitably choosing λ we can accommodate signing attribute strings of any polynomial length. Since the lengths of the attribute strings are bounded by 2^{λ}, the lengths can be expressed as bit strings of size λ. Hence, the total size of the hash value-length pair corresponding to a signing attribute string would still be bounded. Also, the obfuscated next step programs $\mathcal{P}_{\text{ABS}}$ included in our signing keys, must also take as input the length of the signing attribute strings for applying the PPRF if reaching to the accepting state.

Thus, the signing procedure of our ABS scheme becomes the following: to sign a message under some signing attribute string using its legitimate signing key corresponding to some TM M, a signer first hash the signing attribute string with the system-wide SSB hash key. The signer also obtains a signature on the empty accumulator value, 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 authentically accumulate the bits of the hashed signing attribute string. Finally, it runs the obfuscated next step program $\mathcal{P}_{\text{ABS}}$ iteratively on the current accumulator value along with other legitimate inputs, until it obtains either the SIG signing key-verification key pair associated with the signing attribute string under consideration or $\perp$. Once it obtains the SIG signing key-verification key pair associated with the signing attribute string, it simply signs the message using the SIG signing key, and outputs the SIG verification key-signature pair as the ABS signature on the message.

Notice that the problem with enforcing the public parameters of the positional accumulator while transforming the adaptively queried signing keys will not appear in our case as we have assigned a separate set of public parameters of positional accumulator to each signing key. However, our actual proof of selective unforgeability 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, another IO-obfuscated program $\mathcal{P}_3$ is also included within the signing keys, we call the signature changing program, that changes the splittable signature obtained from $\mathcal{P}_2$ on the accumulation of the bits of the signing attribute string, before starting the iterative computation with the obfuscated next step program $\mathcal{P}_{\text{ABS}}$.

```
Constants:  PPRF key $K$
   Inputs:  SSB hash value $h$, Length $\ell_{\text{INP}}$
   Output:  SIG verification key $\widehat{\text{VK}}_{\text{SIG}}$
1. Compute $\hat{r}_{\text{SIG}} = \mathcal{F}(K, (h, \ell_{\text{INP}}))$, $(\widehat{\text{SK}}_{\text{SIG}}, \widehat{\text{VK}}_{\text{SIG}}) = \mathsf{SIG.Setup}(1^\lambda; \hat{r}_{\text{SIG}})$.
2. Output $\widehat{\text{VK}}_{\text{SIG}}$.
```

Fig. 1 Verify.Prog$_{\text{ABS}}$

3.3 The proposed ABS scheme

We now formally describe our ABS scheme. Let λ be the underlying security parameter. Let $\mathbb{M}_\lambda$ denote a family of TMs, 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 ABS construction supporting signing attribute universe $\mathcal{U}_{\text{ABS}} \subset \{0, 1\}^*$, signing policies representable by TMs in $\mathbb{M}_\lambda$, and message space $\mathcal{M}_{\text{ABS}} = \{0, 1\}^{\ell_{\text{ABS-MSG}}}$ 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 block alphabet $\Sigma_{\text{SSB-BLK}} = \{0, 1\}$.
(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 block alphabet $\Sigma_{\text{ACC-BLK}} = \{0, 1, \_\}$.
(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 : $\{0, 1\}^\lambda \rightarrow \{0, 1\}^{2\lambda}$: A length-doubling pseudorandom generator.
(vii) $\mathcal{F} = (\mathcal{F}.\mathsf{Setup}, \mathcal{F}.\mathsf{Puncture}, \mathcal{F}.\mathsf{Eval})$: A puncturable pseudorandom function whose domain and range are chosen appropriately.
(viii) SIG = (SIG.Setup, SIG.Sign, SIG.Verify): A digital signature scheme with associated message space $\mathcal{M}_{\text{ABS}} = \{0, 1\}^{\ell_{\text{ABS}}}$ that is existentially unforgeable against chosen message attack (CMA).

Our ABS scheme is described below:

ABS.Setup$(1^\lambda) \rightarrow (\text{PP}_{\text{ABS}} = (\text{HK}, \mathcal{V}_{\text{ABS}}), \text{MSK}_{\text{ABS}} = (K, \text{HK}))$: The setup authority takes as input the security parameter 1^λ and proceeds as follows:

1. It first chooses a PPRF key $K \xleftarrow{\$} \mathcal{F}.\mathsf{Setup}(1^\lambda)$.
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 $\mathcal{V}_{\text{ABS}} = \mathcal{IO}(\mathsf{Verify.Prog}_{\text{ABS}}[K])$, where the program Verify.Prog$_{\text{ABS}}$ is described in Fig. 1.
4. It keeps the master secret key $\text{MSK}_{\text{ABS}} = (K, \text{HK})$ and publishes the public parameters $\text{PP}_{\text{ABS}} = (\text{HK}, \mathcal{V}_{\text{ABS}})$.

ABS.KeyGen$(\text{MSK}_{\text{ABS}}, M) \rightarrow \text{SK}_{\text{ABS}}(M) = (\text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0, \text{PP}_{\text{ITR}}, v_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{ABS}})$: On input the master secret key $\text{MSK}_{\text{ABS}} = (K, \text{HK})$ and a signing policy 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 $K_1, \ldots, K_\lambda, K_{\text{SPS},A}, K_{\text{SPS},E} \xleftarrow{\$} \mathcal{F}.\mathsf{Setup}(1^\lambda)$.

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

Fig. 2 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 $\text{PP}_{\text{ACC}}$, Public parameters for iterator $\text{PP}_{\text{ITR}}$,
           PPRF key $K_{\text{SPS},E}$
Inputs: Index $i$, Symbol $\text{SYM}_{\text{IN}}$, TM state ST, Accumulator value $w_{\text{IN}}$, Auxiliary value AUX,
        Iterator value $v_{\text{IN}}$, Signature $\sigma_{\text{SPS,IN}}$, SSB hash value $h$, SSB opening value $\pi_{\text{SSB}}$
Output: (Accumulator value $w_{\text{OUT}}$, Iterator value $v_{\text{OUT}}$, Signature $\sigma_{\text{SPS-OUT}}$), or $\perp$
1. (a) Compute $r_{\text{SPS},E} = \mathcal{F}(K_{\text{SPS},E}, (h, i))$, $(\text{SK}_{\text{SPS},E}, \text{VK}_{\text{SPS},E}, \text{VK}_{\text{SPS-REJ},E}) = \mathsf{SPS.Setup}(1^\lambda; r_{\text{SPS},E})$.
   (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 $r'_{\text{SPS},E} = \mathcal{F}(K_{\text{SPS},E}, (h, i+1))$, $(\text{SK}'_{\text{SPS},E}, \text{VK}'_{\text{SPS},E}, \text{VK}'_{\text{SPS-REJ},E}) = \mathsf{SPS.Setup}(1^\lambda; r'_{\text{SPS},E})$.
   (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 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
        $\ell_{\text{INP}}$, Signature $\sigma_{\text{SPS,IN}}$
Output: Signature $\sigma_{\text{SPS,OUT}}$, or $\perp$
1. (a) Compute $r_{\text{SPS},E} = \mathcal{F}(K_{\text{SPS},E}, (h, \ell_{\text{INP}})$, $(\text{SK}_{\text{SPS},E}, \text{VK}_{\text{SPS},E}, \text{VK}_{\text{SPS-REJ},E})) = \mathsf{SPS.Setup}(1^\lambda; r_{\text{SPS},E})$.
   (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 $r_{\text{SPS},A} = \mathcal{F}(K_{\text{SPS},A}, (h, \ell_{\text{INP}}, 0))$, $(\text{SK}_{\text{SPS},A}, \text{VK}_{\text{SPS},A}, \text{VK}_{\text{SPS-REJ},A}) =$
       $\mathsf{SPS.Setup}(1^\lambda; r_{\text{SPS},A})$.
   (b) Output $\sigma_{\text{SPS,OUT}} = \mathsf{SPS.Sign}(\text{SK}_{\text{SPS},A}, m)$.
```

Fig. 4 Change-SPS.Prog

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

   where the programs Init-SPS.Prog, Accumulate.Prog, Change-SPS.Prog and $\mathsf{Constrained\text{-}Key.Prog}_{\text{ABS}}$ are shown respectively in Figs. 2, 3, 4 and 5.
4. It provides the constrained key $\text{SK}_{\text{ABS}}(M) = (\text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0, \text{PP}_{\text{ITR}}, v_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{ABS}})$ to a legitimate signer.

$\mathsf{ABS.Sign}(\text{SK}_{\text{ABS}}(M), x, \mathsf{msg}) \to \sigma_{\text{ABS}} = (\text{VK}_{\text{SIG}}, \sigma_{\text{SIG}})$ or $\perp$: A signer takes as input its signing key $\text{SK}_{\text{ABS}}(M) = (\text{HK}, \text{PP}_{\text{ACC}}, w_0, \text{STORE}_0, \text{PP}_{\text{ITR}}, v_0, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_{\text{ABS}})$, corresponding to its legitimate signing policy TM $M = \langle Q, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \delta, q_0, q_{\text{AC}}, q_{\text{REJ}} \rangle \in$

```
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 $\text{PP}_{\text{ACC}}$, Public parameters for iterator $\text{PP}_{\text{ITR}}$, PPRF keys $K, K_1, \ldots, K_{\lambda}, K_{\text{SPS},A}$
Inputs: Time $t$, String $\text{SEED}_{\text{IN}}$, Header position $\text{POS}_{\text{IN}}$, Symbol $\text{SYM}_{\text{IN}}$, TM state $\text{ST}_{\text{IN}}$, Accumulator value $w_{\text{IN}}$, Accumulator proof $\pi_{\text{ACC}}$, Auxiliary value $\text{AUX}$, Iterator value $v_{\text{IN}}$, SSB hash value $h$, Length $\ell_{\text{INP}}$, Signature $\sigma_{\text{SPS,IN}}$
Output: (SIG signing key $\text{SK}_{\text{SIG}}$, SIG verification key $\text{VK}_{\text{SIG}}$), or (Header Position $\text{POS}_{\text{OUT}}$, Symbol $\text{SYM}_{\text{OUT}}$, TM state $\text{ST}_{\text{OUT}}$, Accumulator value $w_{\text{OUT}}$, Iterator value $v_{\text{OUT}}$, Signature $\sigma_{\text{SPS,OUT}}$, String $\text{SEED}_{\text{OUT}}$), or $\perp$
1. Identify an integer $\tau$ 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 $r_{\text{SPS},A} = \mathcal{F}(K_{\text{SPS},A}, (h, \ell_{\text{INP}}, t-1))$, $(\text{SK}_{\text{SPS},A}, \text{VK}_{\text{SPS},A}, \text{VK}_{\text{SPS-REJ},A}) = \mathsf{SPS.Setup}(1^{\lambda}; r_{\text{SPS},A})$.
   (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 $\text{ST}_{\text{OUT}} = q_{\text{AC}}$, perform the following:
         (I) Compute $r_{\text{SIG}} = \mathcal{F}(K, (h, \ell_{\text{INP}}))$, $(\text{SK}_{\text{SIG}}, \text{VK}_{\text{SIG}}) = \mathsf{SIG.Setup}(1^{\lambda}; r_{\text{SIG}})$.
         (II) Output $(\text{SK}_{\text{SIG}}, \text{VK}_{\text{SIG}})$.
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 $r'_{\text{SPS},A} = \mathcal{F}(K_{\text{SPS},A}, (h, \ell_{\text{INP}}, t))$, $(\text{SK}'_{\text{SPS},A}, \text{VK}'_{\text{SPS},A}, \text{VK}'_{\text{SPS-REJ},A}) = \mathsf{SPS.Setup}(1^{\lambda}; r'_{\text{SPS},A})$.
   (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 $t + 1 = 2^{\tau'}$, for some integer $\tau'$, 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. 5 Constrained-Key.Prog$_{\text{ABS}}$

$\mathbb{M}_{\lambda}$, an attribute string $x = x_0 \ldots x_{\ell_x - 1} \in \mathcal{U}_{\text{ABS}}$ with $|x| = \ell_x$, and a message $\mathsf{msg} \in \mathcal{M}_{\text{ABS}}$. If $M(x) = 0$, it outputs $\perp$. Otherwise, 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 $j = 1, \ldots, \ell_x$, 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 $\text{STORE}_j = \mathsf{ACC.Write\text{-}Store}\left(\text{PP}_{\text{ACC}}, \text{STORE}_{j-1}, j-1, x_{j-1}\right)$.
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 accepts x in t_x steps. For $t = 1, \ldots, t_x$, 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 $\text{OUT} = \mathcal{P}_{\text{ABS}}(t, \text{SEED}_{t-1}, \text{POS}_{M,t-1}, \text{SYM}_{M,t-1}, \text{ST}_{M,t-1}, w_{\ell_x+t-1}, \pi_{\text{ACC},t-1}, \text{AUX}_{\ell_x+t}, v_{\ell_x+t-1}, h, \ell_x, \sigma_{\text{SPS},t-1})$.
   (d) If $t = t_x$, it parses OUT as $\text{OUT} = (\text{SK}_{\text{SIG}}, \text{VK}_{\text{SIG}})$. 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 $\text{STORE}_{\ell_x+t} = \mathsf{ACC.Write\text{-}Store}(\text{PP}_{\text{ACC}}, \text{STORE}_{\ell_x+t-1}, \text{POS}_{M,t-1}, \text{SYM}^{(\text{WRITE})}_{M,t})$.

7. Finally, it computes $\sigma_{\text{SIG}} \xleftarrow{\$} \mathsf{SIG.Sign}(\text{SK}_{\text{SIG}}, \mathsf{msg})$.
8. It outputs the signature $\sigma_{\text{ABS}} = (\text{VK}_{\text{SIG}}, \sigma_{\text{SIG}}) \in \mathcal{S}_{\text{ABS}}$.

$\mathsf{ABS.Verify}(\text{PP}_{\text{ABS}}, x, \mathsf{msg}, \sigma_{\text{ABS}}) \to \hat{\beta} \in \{0, 1\}$: A verifier takes as input the public parameters $\text{PP}_{\text{ABS}} = (\text{HK}, \mathcal{V}_{\text{ABS}})$, an attribute string $x = x_0 \ldots x_{\ell_x - 1} \in \mathcal{U}_{\text{ABS}}$, where $|x| = \ell_x$, a message $\mathsf{msg} \in \mathcal{M}_{\text{ABS}}$, together with a signature $\sigma_{\text{ABS}} = (\text{VK}_{\text{SIG}}, \sigma_{\text{SIG}}) \in \mathcal{S}_{\text{ABS}}$. It executes the following:

1. It first computes $h = \mathcal{H}_{\text{HK}}(x)$.
2. Next, it computes $\widehat{\text{VK}}_{\text{SIG}} = \mathcal{V}_{\text{ABS}}(h, \ell_x)$.
3. If $[\text{VK}_{\text{SIG}} = \widehat{\text{VK}}_{\text{SIG}}] \wedge [\mathsf{SIG.Verify}(\text{VK}_{\text{SIG}}, \mathsf{msg}, \sigma_{\text{SIG}}) = 1]$, it outputs 1. Otherwise, it outputs 0.

Remark 3.2 (Efficiency of the proposed ABS scheme) From the efficiency of the underlying building blocks discussed in Sect. 2.3, it follows that our ABS.Setup algorithm runs in time polynomial in the security parameter λ and $\log T$, while the ABS.KeyGen algorithm runs in time polynomial in λ, the size of the TM M for which the signing key is being generated, and $\log T$, where T is the upper bound on the worst-case running time of the supported TM family. On the other hand, the algorithm ABS.Verify runs in time polynomial in λ, the size of the signing attribute string x considered, the size of the signed message msg, and $\log T$, whereas the algorithm ABS.Sign additionally depends on the description size of the TM M embedded within the used signing key, and the running time t_x of M on the signing attribute string x. In all prior ABS constructions, the ABS.Sign algorithm depends polynomially on the worst-case running time T of the supported class of signing policies (in addition to the sizes of the signing policy, the signing attribute string, and the message considered), which in general is much larger compared to the actual running time of the used signing policy on the signing attribute string considered. Moreover, the size of signatures of our ABS scheme depends only on λ and the size of the signed message msg, in particular independent of the size of the signing attribute string x under which the signature is generated. This is evidently an important achievement from the point of view of communication efficiency that was beyond the reach of all prior ABS constructions.

4 Security analysis

Theorem 4.1 (Security of the proposed *ABS* scheme) *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* SIG *is existentially unforgeable against chosen message attack, the proposed* ABS *scheme satisfies signer privacy and existential unforgeability against selective attribute adaptive chosen message attack.*

4.1 Proof overview of Theorem 4.1

Signer privacy: Observe that the ABS scheme described in Sect. 3.3 clearly preserves signer privacy since the signature on some message with respect to some signing attribute string only contains the SIG verification key obtained from hashing the signing attribute string with the system wide SSB hash function and its length together with an SIG signature on the message verifiable under that SIG verification key. In particular, the ABS signatures do not depend on the signing keys used to generate them.

Existential unforgeability: We provide here a bird's eye view of the proof of existential unforgeability of the ABS scheme proposed in Sect. 3.3. We avoid many subtle technical details which are actually not easy to describe at an informal level and can hinder the intuitive blueprint of proof. Recall that in the selective unforgeability experiment, the adversary $\mathcal{A}$ has to commit to some signing attribute string x^*, under which it wishes to output a forgery, at the beginning of the experiment, and then is supplied with the public parameters, and is allowed to adaptively request any polynomial number of signatures and signing keys associated with signing policies that do not accept x^*. At the end, $\mathcal{A}$ outputs a forged signature on some message msg^* under x^*, and is declared to be the winner if it has not queried any signature on msg^* under x^*.

To argue selective unforgeability of the above ABS construction, we first change the original unforgeability experiment into one in which we hardwire the punctured PPRF key $\mathrm{K}\{x^*\}$ punctured at x^* within the verifying program $\mathcal{V}_{\mathrm{ABS}}$ included within the public parameters given to $\mathcal{A}$, as well as in the signing programs $\mathcal{P}_{\mathrm{ABS}}$ included within all the signing keys provided to $\mathcal{A}$. More precisely, we modify the program $\mathcal{V}_{\mathrm{ABS}}$ into a new program $\mathcal{V}'_{\mathrm{ABS}}$ as follows: When run on some signing attribute string $x \neq x^*$, the program $\mathcal{V}'_{\mathrm{ABS}}$ runs identically to $\mathcal{V}_{\mathrm{ABS}}$, but it uses the punctured PPRF key $\mathrm{K}\{x^*\}$ in place of the full PPRF key K. On the other hand, when run on x^*, it uses a hardwired string $\hat{r}^*_{\mathrm{SIG}}$ as the randomness for generating the SIG verification key corresponding to x^*. We set $\hat{r}^*_{\mathrm{SIG}}$ to be the evaluation of the PPRF with key K on x^*. We similarly modify the signing programs $\mathcal{P}_{\mathrm{ABS}}$ into new programs $\mathcal{P}'_{\mathrm{ABS}}$ as follows: When run on some signing attribute string $x \neq x^*$, $\mathcal{P}'_{\mathrm{ABS}}$ runs identically to $\mathcal{P}_{\mathrm{ABS}}$ except that it uses the punctured PPRF key $\mathrm{K}\{x^*\}$ in place of the full PPRF key K. On the other hand, when run on input x^*, $\mathcal{P}'_{\mathrm{ABS}}$ outputs $\perp$.

Observe that the programs $\mathcal{V}_{\mathrm{ABS}}$ and $\mathcal{V}'_{\mathrm{ABS}}$ are clearly functionally identical since the punctured PPRF key behaves identically to the full PPRF key on all inputs $x \neq x^*$. For the same reason, for all the signing keys given to $\mathcal{A}$, the programs $\mathcal{P}_{\mathrm{ABS}}$ and $\mathcal{P}'_{\mathrm{ABS}}$ are also functionally identical since $\mathcal{A}$ is allowed to request signing keys for only those signing policies that does not accept x^*. Thus, by the security of IO, which stipulates that obfuscations of functionally identical programs are computationally indistinguishable, the modified experiment is computationally indistinguishable from the original one. After that, we apply the pseudorandomness at punctured point property of PPRF to change the pseudorandom string $\hat{r}_{\mathrm{SIG}}$ hardwired within $\mathcal{V}'_{\mathrm{ABS}}$ to a uniformly random one. This modification essentially ensures that a perfectly distributed SIG signing key-verification key pair gets associated to x^*. Note that once this alteration is made, we can directly prove the unforgeability of our ABS scheme relying on the unforgeability property of SIG.

We follow the same technique introduced in [7] for handling the tail hybrids in the final stage of transformation of the signing keys in our unforgeability experiment. Note that as in [7], we consider TMs which run for at most $T = 2^\lambda$ steps on any input. Unlike [17], Deshpande et al. [7] have devised an 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 unforgeability proof. Please refer to [7] for a high level overview of the approach.

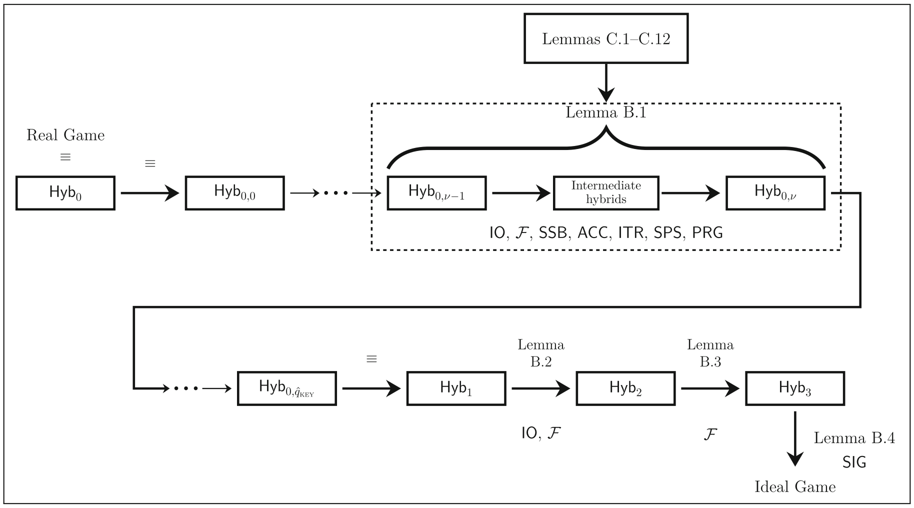

Fig. 6 Structure of the hybrid reduction proving unforgeability of our ABS scheme

4.2 Formal Proof of Theorem 4.1

▶ **Signer privacy**: Observe that for any message $\mathsf{msg} \in \mathcal{M}_{\text{ABS}}$, $(\text{PP}_{\text{ABS}} = (\text{HK}, \mathcal{IO}(\mathsf{Verify.Prog}_{\text{ABS}}[K])), \text{MSK}_{\text{ABS}} = (K, \text{HK})) \xleftarrow{\$} \mathsf{ABS.Setup}(1^\lambda)$, and $x \in \mathcal{U}_{\text{ABS}}$ with $|x| = \ell_x$, a signature on msg under x is of the form $\sigma_{\text{ABS}} = (\text{VK}_{\text{SIG}}, \sigma_{\text{SIG}})$, where $(\text{SK}_{\text{SIG}}, \text{VK}_{\text{SIG}}) = \mathsf{SIG.Setup}(1^\lambda; \mathcal{F}(K, (\mathcal{H}_{\text{HK}}(x), \ell_x))), \sigma_{\text{SIG}} = \mathsf{SIG.Sign}(\text{SK}_{\text{SIG}}, \mathsf{msg})$. Here, $\text{HK} \xleftarrow{\$} \mathsf{SSB.Gen}(1^\lambda, n_{\text{SSB-BLK}} = 2^\lambda, i^* = 0)$ and $K \xleftarrow{\$} \mathcal{F}.\mathsf{Setup}(1^\lambda)$. Thus, the distribution of the signature σ_{ABS} is clearly the same regardless of the signing key $\text{SK}_{\text{ABS}}(M)$ that is used to compute it.

▶ **Existential unforgeability**: We will prove the existential unforgeability of the ABS construction of Sect. 3.3 against selective attribute adaptive chosen message attack by means of a sequence of hybrid experiments. We will demonstrate based on the security of various primitives that the advantage of any PPT adversary $\mathcal{A}$ in consecutive hybrid experiments differs only negligibly as well as that in the final hybrid experiment is negligible. We note that due to the selective attribute setting, the challenger $\mathcal{B}$ knows the challenge attribute string $x^* = x_0^* \ldots x_{\ell^*-1}^* \in \mathcal{U}_{\text{ABS}}$ and the SSB hash value $h^* = \mathcal{H}_{\text{HK}}(x^*)$ before receiving any signing key or signature query from the adversary $\mathcal{A}$. Suppose, the total number of signing key query and signature query made by the adversary $\mathcal{A}$ be $\hat{q}_{\text{KEY}}$ and $\hat{q}_{\text{SIGN}}$ respectively. As noted in Remark 3.1, without loss of generality we will assume that $\mathcal{A}$ only queries signatures on messages under the challenge attribute string x^*. Please refer to Fig. 6 for an overview of the hybrid transitions and their analysis. The description of the hybrid experiments follows:

4.3 Sequence of hybrid experiments

Hyb$_0$: This experiment corresponds to the real selective attribute adaptive chosen message unforgeability experiment described in Sect. 3.1. More precisely, this experiment proceeds as follows:

- $\mathcal{A}$ submits a challenge attribute string $x^* = x_0^* \dots x_{\ell^*-1}^* \in \mathcal{U}_{\text{ABS}}$ with $|x^*| = \ell^*$ to $\mathcal{B}$.
- $\mathcal{B}$ generates $(\text{PP}_{\text{ABS}} = (\text{HK}, \mathcal{IO}(\mathsf{Verify.Prog}_{\text{ABS}}[K])),\ \text{MSK}_{\text{ABS}} = (\text{HK}, K)) \xleftarrow{\$} \mathsf{ABS.Setup}(1^\lambda)$, as described in Sect. 3.3, and provides PP_{ABS} to $\mathcal{A}$.
- For $\eta = 1, \dots, \hat{q}_{\text{KEY}}$, in response to the ηth signing key query corresponding to signing policy TM $M^{(\eta)} = \langle Q^{(\eta)}, \Sigma_{\text{INP}}, \Sigma_{\text{TAPE}}, \delta^{(\eta)}, q_0^{(\eta)}, q_{\text{AC}}^{(\eta)}, q_{\text{REJ}}^{(\eta)} \rangle \in \mathbb{M}_\lambda$ with $M^{(\eta)}(x^*) = 0$, $\mathcal{B}$ creates

$$\text{SK}_{\text{ABS}}(M^{(\eta)}) = \left(\begin{array}{l} \text{HK}, \text{PP}_{\text{ACC}}^{(\eta)}, w_0^{(\eta)}, \text{STORE}_0^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, v_0^{(\eta)}, \\ \mathcal{IO}(\mathsf{Init\text{-}SPS.Prog}[q_0^{(\eta)}, w_0^{(\eta)}, v_0^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Accumulate.Prog}[n_{\text{SSB-BLK}} = 2^\lambda, \text{HK}, \text{PP}_{\text{ACC}}^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Change\text{-}SPS.Prog}[K_{\text{SPS},A}^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Constrained\text{-}Key.Prog}_{\text{ABS}}[M^{(\eta)}, T = 2^\lambda, \text{PP}_{\text{ACC}}^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, K, \\ \quad K_1^{(\eta)}, \dots, K_\lambda^{(\eta)}, K_{\text{SPS},A}^{(\eta)}]) \end{array} \right)$$
$$\xleftarrow{\$} \mathsf{ABS.KeyGen}(\text{MSK}_{\text{ABS}}, M^{(\eta)}),$$

as described in Sect. 3.3 and returns $\text{SK}_{\text{ABS}}(M^{(\eta)})$ to $\mathcal{A}$.
- For $\theta = 1, \dots, \hat{q}_{\text{SIGN}}$, in reply to the θth signature query on message $\mathsf{msg}^{(\theta)}$ under attribute string x^*, $\mathcal{B}$ identifies some TM $M^* \in \mathbb{M}_\lambda$ such that $M^*(x^*) = 1$, generates $\text{SK}_{\text{ABS}}(M^*) \xleftarrow{\$} \mathsf{ABS.KeyGen}(\text{MSK}_{\text{ABS}}, M^*)$, and computes $\sigma_{\text{ABS}}^{(\theta)} = (\text{VK}_{\text{SIG}}^*, \sigma_{\text{SIG}}^{(\theta)}) \xleftarrow{\$} \mathsf{ABS.Sign}(\text{SK}_{\text{ABS}}(M^*), x^*, \mathsf{msg}^{(\theta)})$ as described in Sect. 3.3. $\mathcal{B}$ gives back $\sigma_{\text{ABS}}^{(\theta)}$ to $\mathcal{A}$.
- Finally, $\mathcal{A}$ outputs a forged signature σ_{ABS}^* on some message msg^* under attribute string x^*.

$\mathbf{Hyb}_{0,\nu}$ **($\nu = 1, \dots, \hat{q}_{\text{KEY}}$)**: This experiment is similar to Hyb_0 except that for $\eta \in [\hat{q}_{\text{KEY}}]$, in reply to the ηth signing key query of $\mathcal{A}$ corresponding to signing policy TM $M^{(\eta)} \in \mathbb{M}_\lambda$ with $M^{(\eta)}(x^*) = 0$, $\mathcal{B}$ returns the signing key

$$\text{SK}_{\text{ABS}}(M^{(\eta)}) = \left(\begin{array}{l} \text{HK}, \text{PP}_{\text{ACC}}^{(\eta)}, w_0^{(\eta)}, \text{STORE}_0^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, v_0^{(\eta)}, \\ \mathcal{IO}(\mathsf{Init\text{-}SPS.Prog}[q_0^{(\eta)}, w_0^{(\eta)}, v_0^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Accumulate.Prog}[n_{\text{SSB-BLK}} = 2^\lambda, \text{HK}, \text{PP}_{\text{ACC}}^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Change\text{-}SPS.Prog}[K_{\text{SPS},A}^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Constrained\text{-}Key.Prog}'_{\text{ABS}}[M^{(\eta)}, T = 2^\lambda, \text{PP}_{\text{ACC}}^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, K, \\ \quad K_1^{(\eta)}, \dots, K_\lambda^{(\eta)}, K_{\text{SPS},A}^{(\eta)}, h^*, \ell^*]) \end{array} \right),$$

if $\eta \leq \nu$, where the program $\mathsf{Constrained\text{-}Key.Prog}'_{\text{ABS}}$ is an alteration of the program $\mathsf{Constrained\text{-}Key.Prog}_{\text{ABS}}$ (Fig. 5) and is described in Fig. 7, while it returns the signing key

$$\text{SK}_{\text{ABS}}(M^{(\eta)}) = \left(\begin{array}{l} \text{HK}, \text{PP}_{\text{ACC}}^{(\eta)}, w_0^{(\eta)}, \text{STORE}_0^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, v_0^{(\eta)}, \\ \mathcal{IO}(\mathsf{Init\text{-}SPS.Prog}[q_0^{(\eta)}, w_0^{(\eta)}, v_0^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Accumulate.Prog}[n_{\text{SSB-BLK}} = 2^\lambda, \text{HK}, \text{PP}_{\text{ACC}}^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Change\text{-}SPS.Prog}[K_{\text{SPS},A}^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Constrained\text{-}Key.Prog}_{\text{ABS}}[M^{(\eta)}, T = 2^\lambda, \text{PP}_{\text{ACC}}^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, K, \\ \quad K_1^{(\eta)}, \dots, K_\lambda^{(\eta)}, K_{\text{SPS},A}^{(\eta)}]) \end{array} \right),$$

```
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 $\text{PP}_{\text{ACC}}$, Public parameters for iterator $\text{PP}_{\text{ITR}}$, PPRF keys
           $K, K_1, \ldots, K_\lambda, K_{\text{SPS},A}$, $\underline{\text{SSB hash value of challenge input } h^*\text{, Length of challenge input } \ell^*}$
Inputs:    Time $t$, String $\text{SEED}_{\text{IN}}$, Header position $\text{POS}_{\text{IN}}$, Symbol $\text{SYM}_{\text{IN}}$, TM state $\text{ST}_{\text{IN}}$, Accu-
           mulator value $w_{\text{IN}}$, Accumulator proof $\pi_{\text{ACC}}$, Auxiliary value $\text{AUX}$, Iterator value
           $v_{\text{IN}}$, SSB hash value $h$, Length $\ell_{\text{INP}}$, Signature $\sigma_{\text{SPS,IN}}$
Output:    (SIG signing key $\text{SK}_{\text{SIG}}$, SIG verification key $\text{VK}_{\text{SIG}}$), or (Header Position $\text{POS}_{\text{OUT}}$,
           Symbol $\text{SYM}_{\text{OUT}}$, TM state $\text{ST}_{\text{OUT}}$, Accumulator value $w_{\text{OUT}}$, Iterator value $v_{\text{OUT}}$,
           Signature $\sigma_{\text{SPS,OUT}}$, String $\text{SEED}_{\text{OUT}}$), or $\perp$
1. Identify an integer $\tau$ 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 $r_{\text{SPS},A} = \mathcal{F}(K_{\text{SPS},A}, (h, \ell_{\text{INP}}, t-1))$, $(\text{SK}_{\text{SPS},A}, \text{VK}_{\text{SPS},A}, \text{VK}_{\text{SPS-REJ},A}) =$
       $\mathsf{SSB.Setup}(1^\lambda; r_{\text{SPS},A})$.
   (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 $[\text{ST}_{\text{OUT}} = q_{\text{AC}}] \wedge [(h, \ell_{\text{INP}}) \neq (h^*, \ell^*)]$, perform the following:
         (I) Compute $r_{\text{SIG}} = \mathcal{F}(K, (h, \ell_{\text{INP}}))$, $(\text{SK}_{\text{SIG}}, \text{VK}_{\text{SIG}}) = \mathsf{SIG.Setup}(1^\lambda; r_{\text{SIG}})$.
         (II) Output $(\text{SK}_{\text{SIG}}, \text{VK}_{\text{SIG}})$.
       Else if $\text{ST}_{\text{OUT}} = q_{\text{AC}}$, output $\perp$.
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 $r'_{\text{SPS},A} = \mathcal{F}(K_{\text{SPS},A}, (h, \ell_{\text{INP}}, t))$, $(\text{SK}'_{\text{SPS},A}, \text{VK}'_{\text{SPS},A}, \text{VK}'_{\text{SPS-REJ},A}) =$
       $\mathsf{SPS.Setup}(1^\lambda; r'_{\text{SPS},A})$.
   (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 $t + 1 = 2^{\tau'}$, for some integer $\tau'$, 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. 7 Constrained-Key.Prog$'_{\text{ABS}}$

if $\eta > \nu$. Observe that $\mathsf{Hyb}_{0,0}$ coincides with Hyb_0.
Hyb_1: This experiment coincides with $\mathsf{Hyb}_{0,\hat{q}_{\text{KEY}}}$. More formally, in this experiment for $\eta = 1, \ldots, \hat{q}_{\text{KEY}}$, in reply to the ηth signing key query of $\mathcal{A}$ corresponding to signing policy TM $M^{(\eta)} \in \mathbb{M}_\lambda$ with $M^{(\eta)}(x^*) = 0$, $\mathcal{B}$ generates all the components of the signing key as in Hyb_0, however, it returns the signing key

$$\text{SK}_{\text{ABS}}(M^{(\eta)}) = \begin{pmatrix} \text{HK}, \text{PP}_{\text{ACC}}^{(\eta)}, w_0^{(\eta)}, \text{STORE}_0^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, v_0^{(\eta)}, \\ \mathcal{IO}(\mathsf{Init\text{-}SPS.Prog}[q_0^{(\eta)}, w_0^{(\eta)}, v_0^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Accumulate.Prog}[n_{\text{SSB-BLK}} = 2^\lambda, \text{HK}, \text{PP}_{\text{ACC}}^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Change\text{-}SPS.Prog}[K_{\text{SPS},A}^{(\eta)}, K_{\text{SPS},E}^{(\eta)}]) \\ \mathcal{IO}(\mathsf{Constrained\text{-}Key.Prog}'_{\text{ABS}}[M^{(\eta)}, T = 2^\lambda, \text{PP}_{\text{ACC}}^{(\eta)}, \text{PP}_{\text{ITR}}^{(\eta)}, K, \\ \quad K_1^{(\eta)}, \ldots, K_\lambda^{(\eta)}, K_{\text{SPS},A}^{(\eta)}, h^*, \ell^*]) \end{pmatrix}.$$

The rest of the experiment is analogous to Hyb_0.
Hyb_2: This experiment is identical to Hyb_1 other than the following exceptions:

(I) Upon receiving the challenge attribute string x^*, $\mathcal{B}$ proceeds as follows:

1. It selects a PPRF key $K \xleftarrow{\$} \mathcal{F}.\mathsf{Setup}(1^\lambda)$ and generates $\text{HK} \xleftarrow{\$} \mathsf{SSB.Gen}(1^\lambda, n_{\text{SSB-BLK}} = 2^\lambda, i^* = 0)$ just as in Hyb_1,

```
Constants:  Punctured PPRF key $K\{(h^*, \ell^*)\}$, SIG verification key $\widehat{\mathrm{VK}}^*_{\mathrm{SIG}}$, SSB hash value of
            challenge input $h^*$, Length of challenge input $\ell^*$
   Inputs:  SSB hash value $h$, Length $\ell_{\mathrm{INP}}$
   Output:  SIG verification key $\widehat{\mathrm{VK}}_{\mathrm{SIG}}$
(a) If $(h, \ell_{\mathrm{INP}}) = (h^*, \ell^*)$, output $\widehat{\mathrm{VK}}^*_{\mathrm{SIG}}$.
    Else compute $\hat{r}_{\mathrm{SIG}} = \mathcal{F}(K\{(h^*, \ell^*)\}, (h, \ell_{\mathrm{INP}}))$, $(\widehat{\mathrm{SK}}_{\mathrm{SIG}}, \widehat{\mathrm{VK}}_{\mathrm{SIG}}) = \mathsf{SIG.Setup}(1^\lambda; \hat{r}_{\mathrm{SIG}})$.
(b) Output $\widehat{\mathrm{VK}}_{\mathrm{SIG}}$.
```

Fig. 8 $\mathsf{Verify.Prog}'_{\mathrm{ABS}}$

2. It computes $h^* = \mathcal{H}_{\mathrm{HK}}(x^*)$ and creates the punctured PPRF key $K\{(h^*, \ell^*)\} \xleftarrow{\$} \mathcal{F}.\mathsf{Puncture}(K, (h^*, \ell^*))$,
3. It computes $\widehat{r}^*_{\mathrm{SIG}} = \mathcal{F}(K, (h^*, \ell^*))$, forms $(\widehat{\mathrm{SK}}^*_{\mathrm{SIG}}, \widehat{\mathrm{VK}}^*_{\mathrm{SIG}}) = \mathsf{SIG.Setup}(1^\lambda; \widehat{r}^*_{\mathrm{SIG}})$,
4. It sets the public parameters $\mathrm{PP}_{\mathrm{ABS}}$ to be given to $\mathcal{A}$ as $\mathrm{PP}_{\mathrm{ABS}} = (\mathrm{HK}, \mathcal{IO}(\mathsf{Verify.Prog}'_{\mathrm{ABS}}[K\{(h^*, \ell^*)\}, \widehat{\mathrm{VK}}^*_{\mathrm{SIG}}, h^*, \ell^*]))$, where the program $\mathsf{Verify.Prog}'_{\mathrm{ABS}}$ is an alteration of the program $\mathsf{Verify.Prog}_{\mathrm{ABS}}$ (Fig. 1) and is depicted in Fig. 8.

(II) For $\eta = 1, \ldots, \hat{q}_{\mathrm{KEY}}$, in response to the ηth signing key query of $\mathcal{A}$ corresponding to signing policy TM $M^{(\eta)} \in \mathbb{M}_\lambda$ with $M^{(\eta)}(x^*) = 0$, $\mathcal{B}$ provides $\mathcal{A}$ with the signing key

$$\mathrm{SK}_{\mathrm{ABS}}(M^{(\eta)}) = \left(\begin{array}{l} \mathrm{HK}, \mathrm{PP}^{(\eta)}_{\mathrm{ACC}}, w^{(\eta)}_0, \mathrm{STORE}^{(\eta)}_0, \mathrm{PP}^{(\eta)}_{\mathrm{ITR}}, v^{(\eta)}_0, \\ \mathcal{IO}(\mathsf{Init\text{-}SPS.Prog}[q^{(\eta)}_0, w^{(\eta)}_0, v^{(\eta)}_0, K^{(\eta)}_{\mathrm{SPS},E}]) \\ \mathcal{IO}(\mathsf{Accumulate.Prog}[n_{\mathrm{SSB\text{-}BLK}} = 2^\lambda, \mathrm{HK}, \mathrm{PP}^{(\eta)}_{\mathrm{ACC}}, \mathrm{PP}^{(\eta)}_{\mathrm{ITR}}, K^{(\eta)}_{\mathrm{SPS},E}]) \\ \mathcal{IO}(\mathsf{Change\text{-}SPS.Prog}[K^{(\eta)}_{\mathrm{SPS},A}, K^{(\eta)}_{\mathrm{SPS},E}]) \\ \mathcal{IO}(\mathsf{Constrained\text{-}Key.Prog}'_{\mathrm{ABS}}[M^{(\eta)}, T = 2^\lambda, \mathrm{PP}^{(\eta)}_{\mathrm{ACC}}, \mathrm{PP}^{(\eta)}_{\mathrm{ITR}}, K\{(h^*, \ell^*)\}, \\ \quad K^{(\eta)}_1, \ldots, K^{(\eta)}_\lambda, K^{(\eta)}_{\mathrm{SPS},A}, h^*, \ell^*]) \end{array}\right).$$

Hyb_3: This experiment is similar to Hyb_2 with the only exception that $\mathcal{B}$ selects $\hat{r}^*_{\mathrm{SIG}} \xleftarrow{\$} \mathcal{Y}_{\mathrm{PPRF}}$. More formally, this experiment has the following deviations from Hyb_2:

(I) In this experiment $\mathcal{B}$ creates the punctured PPRF key $K\{(h^*, \ell^*)\}$ as in Hyb_2, however, it generates $(\widehat{\mathrm{SK}}^*_{\mathrm{SIG}}, \widehat{\mathrm{VK}}^*_{\mathrm{SIG}}) \xleftarrow{\$} \mathsf{SIG.Setup}(1^\lambda)$. It includes the obfuscated program $\mathcal{IO}(\mathsf{Verify.Prog}'_{\mathrm{ABS}}[K\{(h^*, \ell^*)\}, \widehat{\mathrm{VK}}^*_{\mathrm{SIG}}, h^*, \ell^*])$ within the public parameters $\mathrm{PP}_{\mathrm{ABS}}$ to be provided to $\mathcal{A}$ as earlier.

(II) Also, for $\theta = 1, \ldots, \hat{q}_{\mathrm{SIGN}}$, to answer the θth signature query of $\mathcal{A}$ on message $\mathsf{msg}^{(\theta)} \in \mathcal{M}_{\mathrm{ABS}}$ under attribute string x^*, $\mathcal{B}$ computes $\sigma^{(\theta)}_{\mathrm{SIG}} \xleftarrow{\$} \mathsf{SIG.Sign}(\widehat{\mathrm{SK}}^*_{\mathrm{SIG}}, \mathsf{msg}^{(\theta)})$ and returns $\sigma^{(\theta)}_{\mathrm{ABS}} = (\widehat{\mathrm{VK}}^*_{\mathrm{SIG}}, \sigma^{(\theta)}_{\mathrm{SIG}})$ to $\mathcal{A}$.

4.4 Analysis

Let $\mathsf{Adv}^{(0)}_{\mathcal{A}}(\lambda)$, $\mathsf{Adv}^{(0,\nu)}_{\mathcal{A}}(\lambda)$ $(\nu = 1, \ldots, \hat{q}_{\mathrm{KEY}})$, $\mathsf{Adv}^{(1)}_{\mathcal{A}}(\lambda)$, $\mathsf{Adv}^{(2)}_{\mathcal{A}}(\lambda)$, and $\mathsf{Adv}^{(3)}_{\mathcal{A}}(\lambda)$ represent respectively the advantage of the adversary $\mathcal{A}$, i.e., $\mathcal{A}$'s probability of successfully outputting a valid forgery, in Hyb_0, $\mathsf{Hyb}_{0,\nu}$ $(\nu = 1, \ldots, \hat{q}_{\mathrm{KEY}})$, Hyb_1, Hyb_2, and Hyb_3 respectively. Then, by the description of the hybrid experiments it follows that $\mathsf{Adv}^{\mathrm{ABS,UF\text{-}CMA}}_{\mathcal{A}}(\lambda) \equiv \mathsf{Adv}^{(0)}_{\mathcal{A}}(\lambda) \equiv$

$\mathsf{Adv}_{\mathcal{A}}^{(0,0)}(\lambda)$ and $\mathsf{Adv}_{\mathcal{A}}^{(1)}(\lambda) \equiv \mathsf{Adv}_{\mathcal{A}}^{(0,\hat{q}_{\mathrm{KEY}})}(\lambda)$. Hence, we have

$$\mathsf{Adv}_{\mathcal{A}}^{\mathrm{ABS,UF\text{-}CMA}}(\lambda) \leq \sum_{\nu=1}^{\hat{q}_{\mathrm{KEY}}} |\mathsf{Adv}_{\mathcal{A}}^{(0,\nu-1)}(\lambda) - \mathsf{Adv}_{\mathcal{A}}^{(0,\nu)}(\lambda)| + \sum_{j=1}^{2} |\mathsf{Adv}_{\mathcal{A}}^{(j)}(\lambda) - \mathsf{Adv}_{\mathcal{A}}^{(j+1)}(\lambda)| + \mathsf{Adv}_{\mathcal{A}}^{(3)}(\lambda). \tag{4.1}$$

Lemmas B.1–B.4 presented in Online Appendix B will show that the RHS of Eq. (4.1) is negligible and thus the existential unforgeability of the ABS construction of Sect. 3.3 follows. □

5 Conclusion

In this paper, we construct the *first* ABS scheme supporting signing policies expressible as *Turing machines* (TM) which can handle signing attribute strings of *arbitrary polynomial length* and at the same time features input-specific running time for the signing algorithm. On the technical side, we devise new ideas to empower the techniques of [7, 17] to deal with adaptive key queries.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10623-022-01163-8.

Data availability Data sharing not applicable to this article as no datasets were generated or analyzed during the current study.

References

1. Ananth P., Jain A., Sahai A.: Indistinguishability obfuscation without multilinear maps: io from lwe, bilinear maps, and weak pseudorandomness. In: Cryptology ePrint Archive, Report 2018/615 (2018).
2. Ananth P., Jain A., Lin H., Matt C., Sahai A.: Indistinguishability obfuscation without multilinear maps: new paradigms via low degree weak pseudorandomness and security amplification. In: CRYPTO 2019, pp. 284–332. Springer (2019).
3. Barak B., Goldreich O., Impagliazzo R., Rudich S., Sahai A., Vadhan S., Yan, K.: On the (im) possibility of obfuscating programs. In: CRYPTO 2001, pp. 1–18. Springer (2001).
4. Bellare M., Fuchsbauer G.: Policy-based signatures. In: PKC 2014, pp. 520–537. Springer (2014).
5. Boneh D., Waters B.: Constrained pseudorandom functions and their applications. In: ASIACRYPT 2013, pp. 280–300. Springer (2013).
6. Datta P., Okamoto T., Takashima K.: Efficient attribute-based signatures for unbounded arithmetic branching programs. In: PKC 2019, pp. 127–158. Springer (2019).
7. Deshpande A., Koppula V., Waters B.: Constrained pseudorandom functions for unconstrained inputs. In: EUROCRYPT 2016, pp. 124–153. Springer (2016).

8. Garg S., Gentry C., Halevi S., Raykova M., Sahai A., Waters B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: FOCS 2013, pp. 40–49. IEEE (2013).
9. Gay R., Jain A., Lin H., Sahai A.: Indistinguishability obfuscation from simple-to-state hard problems: new assumptions, new techniques, and simplification. In: EUROCRYPT 2021, pp. 97–126. Springer (2021).
10. Goldreich O., Goldwasser S., Micali S.: How to construct random functions. J. ACM **33**(4), 792–807 (1986).
11. Hubacek P., Wichs D.: On the communication complexity of secure function evaluation with long output. In: ITCS 2015, pp. 163–172. ACM (2015).
12. Jain A., Lin H., Matt C., Sahai A.: How to leverage hardness of constant-degree expanding polynomials over $\mathbb{R}$ to build $i\mathcal{O}$. In: EUROCRYPT 2019, pp. 251–281. Springer (2019).
13. Jain A., Lin H., Sahai A.: Simplifying constructions and assumptions for $i\mathcal{O}$. In: Cryptology ePrint Archive, Report 2019/1252 (2019).
14. Jain A., Lin H., Sahai A.: Indistinguishability obfuscation from LPN over $\mathbb{F}_p$, DLIN, and PRGs in NC^0. In: Cryptology ePrint Archive, Report 2021/1334 (2021).
15. Jain A., Lin H., Sahai A.: Indistinguishability obfuscation from well-founded assumptions. In: STOC 2021, pp. 60–73. ACM (2021).
16. Kaafarani A.E., Katsumata S.: Attribute-based signatures for unbounded circuits in the rom and efficient instantiations from lattices. In: PKC 2018, pp. 89–119. Springer (2018).
17. Koppula V., Lewko A.B., Waters B.: Indistinguishability obfuscation for turing machines with unbounded memory. In: STOC 2015, pp. 419–428. ACM (2015).
18. Lin H., Matt C.: Pseudo flawed-smudging generators and their application to indistinguishability obfuscation. In: Cryptology ePrint Archive, Report 2018/646 (2018).
19. Maji H.K., Prabhakaran M., Rosulek M.: Attribute-based signatures. In: CT-RSA 2011, pp. 376–392. Springer (2011).
20. Okamoto T., Takashima K.: Efficient attribute-based signatures for non-monotone predicates in the standard model. In: PKC 2011, pp. 35–52. Springer (2011).
21. Okamoto T., Pietrzak K., Waters B., Wichs D.: New realizations of somewhere statistically binding hashing and positional accumulators. In: ASIACRYPT 2015, pp. 121–145. Springer (2015).
22. Sahai A., Waters B.: How to use indistinguishability obfuscation: deniable encryption, and more. In: STOC 2014, pp. 475–484. ACM (2014).
23. Sakai Y., Attrapadung N., Hanaoka G.: Attribute-based signatures for circuits from bilinear map. In: PKC 2016, pp. 283–300. Springer (2016).
24. Sakai Y., Katsumata S., Attrapadung N., Hanaoka G.: Attribute-based signatures for unbounded languages from standard assumptions. In: ASIACRYPT 2018, pp. 493–522. Springer (2018).
25. Tang F., Li H., Liang B.: Attribute-based signatures for circuits from multilinear maps. In: ISC 2014, pp. 54–71. Springer (2014).
26. Tsabary R.: An equivalence between attribute-based signatures and homomorphic signatures, and new constructions for both. In: TCC 2018, pp. 489–518. Springer (2018).