Security Analysis of Coconut, an Attribute-Based Credential Scheme with Threshold Issuance

Alfredo Rial and Ania M. Piotrowska

Nym Technologies
{alfredo, ania}@nymtech.net

Abstract. Coconut [NDSS 2019] is an attribute-based credential scheme with threshold issuance. We analyze its security properties. To this end, we define an ideal functionality $\mathcal{F}_{\mathrm{AC}}$ for attribute-based access control with threshold issuance. We describe a construction Π_{AC} that realizes $\mathcal{F}_{\mathrm{AC}}$. Π_{AC} follows Coconut with a few changes. In particular, Π_{AC} modifies the protocols for blind issuance of credentials and for credential show so that user privacy holds against computationally unbounded adversaries. The modified protocols are slightly more efficient than those of Coconut. Π_{AC} also extends the public key, which seems necessary to prove unforgeability.

1 Introduction

Attribute-based credentials (ABC) [1–10], also known as anonymous credentials, selective-disclosure credentials or minimal disclosure tokens, are cryptographic schemes used to provide privacy-preserving access control. A credential σ is a digital signature on one or more user attributes a signed by an authority $\mathcal{V}$. Through an issuance protocol, a user $\mathcal{U}$ obtains a credential σ from $\mathcal{V}$, which is in charge of verifying that $\mathcal{U}$ indeed possesses a. Usually, ABC schemes provide a blind issuance protocol, which allows $\mathcal{U}$ to obtain σ on a without revealing a to $\mathcal{V}$, while still proving that a fulfills some statements ϕ. In the show protocol, $\mathcal{U}$ uses her credentials to prove to a service provider $\mathcal{P}$ that her attributes satisfy an access control policy φ. The show protocol does not reveal to $\mathcal{P}$ more information about the user attributes than the fact that they satisfy φ. ABC schemes provide unlinkability, i.e. the show of a credential σ by a user cannot be linked to other shows of σ or to its issuance, even if an adversarial authority and service providers collude.

Typically, ABC schemes consider a single authority $\mathcal{V}$, which is trusted with the task of verifying that users possess attributes. In [11], an ABC scheme with threshold issuance was proposed. This scheme, named Coconut, considers n authorities $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$. In the issuance protocol, $\mathcal{U}$ must obtain partial credentials from at least t authorities to be able to compute a credential σ on an attribute a that can be used in the show protocol. Therefore, the task of verifying that $\mathcal{U}$ possesses a is distributed among t authorities. Threshold issuance is desirable

when ABC schemes are used to provide access control in blockchains, because blockchains guarantee integrity whenever the number of dishonest authorities is below a threshold (Byzantine fault tolerance).

Our Contribution. In [11], the security properties satisfied by Coconut are analyzed informally. We provide a security proof in the ideal-world/real-world paradigm.

First, in Section 5 we give a security definition in the form of an ideal functionality $\mathcal{F}_{\mathrm{AC}}$ for attribute-based access control with threshold issuance. $\mathcal{F}_{\mathrm{AC}}$ follows the universal composability (UC) framework [12], and can be used to analyze the security of ABC schemes with threshold issuance that are universally composable. We remark that Coconut is not UC secure, but $\mathcal{F}_{\mathrm{AC}}$ can be used to analyze the security of Coconut under sequential composition. We also remark that the ideal-world protocol defined by $\mathcal{F}_{\mathrm{AC}}$ provides the unforgeability, blindness and unlinkability properties described in [11].

Second, in Section 7 we describe a construction Π_{AC} that realizes $\mathcal{F}_{\mathrm{AC}}$. Π_{AC} follows Coconut and extends it with the use of three ideal functionalities $\mathcal{F}_{\mathrm{NYM}}$, $\mathcal{F}_{\mathrm{KG}}$ and $\mathcal{F}_{\mathrm{RO}}$:

- In [11], the communication channel between parties is not described. Π_{AC} uses an ideal functionality $\mathcal{F}_{\mathrm{NYM}}$ for a pseudonymous channel. $\mathcal{F}_{\mathrm{NYM}}$ models a channel where parties have optional unlikability by using pseudonyms.
- In [11], public and private keys of authorities are generated by an algorithm, which should be executed by a trusted party. The way each authority retrieves its keys is not described, but it is mentioned that the algorithm can be replaced by a distributed key generation protocol. Π_{AC} uses a functionality $\mathcal{F}_{\mathrm{KG}}$ for key generation, which allows authorities to retrieve their secret keys and any party to retrieve the public keys.
- Coconut uses the random oracle (RO) model, which we model by using a functionality $\mathcal{F}_{\mathrm{RO}}$ for random oracles.

In addition to using $\mathcal{F}_{\mathrm{NYM}}$, $\mathcal{F}_{\mathrm{KG}}$ and $\mathcal{F}_{\mathrm{RO}}$, we remark that some other changes are introduced in Π_{AC} in comparison to Coconut because they were needed or desirable.

- In order to prove that a user cannot forge credentials, it seems necessary to add additional elements to the public key.
- The issuance protocol in Coconut provides privacy for the user (blindness) under the XDH assumption. After adding additional elements to the public key, it is possible to design an issuance protocol that is both more efficient and that provides privacy for the user against computationally unbounded adversaries.
- The show protocol in Coconut does not provide privacy for the user (unlinkability) against computationally unbounded adversaries. Moreover, it is necessary that credentials contain at least one random attribute so that unlinkability holds. Π_{AC} uses a modified version of the show protocol that provides user privacy against computationally unbounded adversaries and that avoids the need of including a random attribute in each credential.

We describe more in detail the reason for these changes in Section 3.1.

In Section 8, we show that Π_{AC} realizes $\mathcal{F}_{\mathrm{AC}}$. We use static corruptions. Coconut uses implicitly a modified version of the Pointcheval-Sanders signature scheme [13] in the random oracle model. We formalize this modified scheme and its security definition in Section 6.4, and we prove that Π_{AC} provides unforgeability thanks to this scheme in Section 8.

Outline of the paper. In Section 2, we describe the building blocks used in Coconut. In Section 3, we recall the description of Coconut in [11] and discuss the differences between Coconut and Π_{AC}. In Section 4, we describe the ideal-world/real-word paradigm and the notation we use to describe ideal functionalities. The ideal functionality $\mathcal{F}_{\mathrm{AC}}$ is depicted in Section 5. In Section 6, we describe the functionalities $\mathcal{F}_{\mathrm{NYM}}$, $\mathcal{F}_{\mathrm{KG}}$ and $\mathcal{F}_{\mathrm{RO}}$, and the Pointcheval-Sanders signature scheme in the RO model. Section 7 describes the construction Π_{AC} and Section 8 analyzes the security of Π_{AC}. We conclude in Section 9.

2 Building Blocks of Coconut

2.1 Bilinear Maps and Assumptions

Let $\mathbb{G}$, $\tilde{\mathbb{G}}$ and $\mathbb{G}_t$ be groups of prime order p. A map $e : \mathbb{G} \times \tilde{\mathbb{G}} \rightarrow \mathbb{G}_t$ must satisfy bilinearity, i.e., $e(g^x, \tilde{g}^y) = e(g, \tilde{g})^{xy}$; non-degeneracy, i.e., for all generators $g \in \mathbb{G}$ and $\tilde{g} \in \tilde{\mathbb{G}}$, $e(g, \tilde{g})$ generates $\mathbb{G}_t$; and efficiency, i.e., there exists an efficient algorithm $\mathcal{G}(1^k)$ that outputs the pairing group setup $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ and an efficient algorithm to compute $e(a, b)$ for any $a \in \mathbb{G}$, $b \in \tilde{\mathbb{G}}$. In type 3 pairings, $\mathbb{G} \neq \tilde{\mathbb{G}}$ and there exists no efficiently computable homomorphism $f : \tilde{\mathbb{G}} \rightarrow \mathbb{G}$.

Definition 1. *[Assumption 1 [13]] Let $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ be a bilinear group setting of type 3, with g (respectively $\tilde{g}$) a generator of $\mathbb{G}$ (respectively $\tilde{\mathbb{G}}$). For (g^x, g^y) and $(\tilde{g}^x, \tilde{g}^y)$, where $x, y \in \mathbb{Z}_p$ are random, we define the oracle $\mathcal{O}(m)$ on input $m \leftarrow \mathbb{Z}_p$ that chooses a random $h \in \mathbb{G}$ and outputs the pair $P = (h, h^{x+my})$. Given $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, g^y, \tilde{g}^x, \tilde{g}^y)$ and unlimited access to this oracle, no adversary can efficiently generate such a pair, with $h \neq 1_{\mathbb{G}}$, for a new scalar m^* not asked to $\mathcal{O}$.*

Definition 2. *[Assumption 2 [13]] Assumption 2 is a weaker version of Assumption 1. Let $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$ be a bilinear group setting of type 3, with g (respectively $\tilde{g}$) a generator of $\mathbb{G}$ (respectively $\tilde{\mathbb{G}}$). For $(\tilde{g}^x, \tilde{g}^y)$, where $x, y \in \mathbb{Z}_p$ are random, we define the oracle $\mathcal{O}(m)$ on input $m \leftarrow \mathbb{Z}_p$ that chooses a random $h \in \mathbb{G}$ and outputs the pair $P = (h, h^{x+my})$. Given $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, \tilde{g}^x, \tilde{g}^y)$ and unlimited access to this oracle, no adversary can efficiently generate such a pair, with $h \neq 1_{\mathbb{G}}$, for a new scalar m^* not asked to $\mathcal{O}$.*

Definition 3. *[XDH Assumption [14]] Given $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$, the external Diffie-Hellman assumption states that the decisional Diffie-Hellman problem is intractable in $\mathbb{G}$.*

2.2 Zero-Knowledge Proofs of Knowledge

Informally speaking, a zero-knowledge proof of knowledge is a two-party protocol between a prover and a verifier with two properties. First, it should be a proof of knowledge, i.e., there should exist a knowledge extractor that extracts the secret input from a successful prover with all but negligible probability. Second, it should be zero-knowledge, i.e., for all possible verifiers there exists a simulator that, without knowledge of the secret input, yields a distribution that cannot be distinguished from the interaction with a real prover.

To express a zero-knowledge proof of knowledge, we follow the notation introduced by Camenisch and Stadler [15]. For example, $\mathsf{ZK}\{(s) : y = f(s)\}$ denotes a "*zero-knowledge proof of knowledge of the secret input s such that $y = f(s)$*". Letters in the parenthesis, in this example s, denote the secret input, while y and the function f are also known to the verifier.

Σ-protocol and Fiat-Shamir transform. Let $\mathcal{L}$ be a language in NP. We can associate to any NP-language $\mathcal{L}$ a polynomial time recognizable relation $\mathcal{R}_{\mathcal{L}}$ defining $\mathcal{L}$ as $\mathcal{L} = \{x : \exists w \text{ s.t. } (x, w) \in \mathcal{R}_{\mathcal{L}}\}$, where $|w| \leq \text{poly}(|x|)$. The string w is called a witness for membership of $x \in \mathcal{L}$.

A protocol $\Sigma = (\mathcal{P}, \mathcal{V})$ for an NP-language $\mathcal{L}$ is an interactive proof system. The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ know an instance x of the language $\mathcal{L}$. The prover $\mathcal{P}$ also knows a witness w for membership of $x \in \mathcal{L}$. Σ-protocols have a 3-move shape where the first message α, called *commitment*, is sent by the prover. The second message β, called *challenge*, is chosen randomly and sent by the verifier. The last message γ, called *response*, is sent by the prover. A Σ-protocol fulfills the properties of completeness, honest-verifier zero-knowledge, and special soundness defined in Faust et al. [16].

In Coconut, zero-knowledge arguments of knowledge based on the Fiat-Shamir transform [17] are used. The Fiat-Shamir transform removes the interaction between the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ of a Σ protocol by replacing the challenge with a hash value $H(\alpha, x)$ computed by the prover, where H is modeled as a random oracle. An argument π consists of $(\alpha, H(\alpha, x), \gamma)$. The Fiat-Shamir system is denoted by $(\mathcal{P}^H, \mathcal{V}^H)$ and fulfills the properties of zero-knowledge and weak simulation extractability defined in Faust et al. [16], which we recall in Appendix A.1.

2.3 Commitment Schemes

A commitment scheme consists of algorithms CSetup, Com and VfCom. The algorithm $\mathsf{CSetup}(1^k)$ generates the parameters of the commitment scheme par_c, which include a description of the message space $\mathcal{M}$. $\mathsf{Com}(par_c, x)$ outputs a commitment com to x and auxiliary information $open$. A commitment is opened by revealing $(x, open)$ and checking whether $\mathsf{VfCom}(par_c, com, x, open)$ outputs 1 or 0.

A commitment scheme should fulfill the *correctness*, *hiding* and *binding* properties. We recall the definitions of those properties in Appendix A.2.

Coconut uses the commitment scheme by Pedersen [18] to commit to elements $x \in \mathbb{Z}_p$, where p is a prime. This commitment scheme is perfectly hiding and computationally binding under the discrete logarithm assumption. The Pedersen commitment scheme consists of the following algorithms.

$\mathsf{CSetup}(1^k)$**.** On input the security parameter 1^k, pick random generators g, h of a group $\mathbb{G}_p$ of prime order p. Output $par_c = (g, h, \mathcal{M})$, where $\mathcal{M} = \mathbb{Z}_p$.

$\mathsf{Com}(par_c, x)$**.** Check that $x \in \mathcal{M}$. Pick random $open \in \mathbb{Z}_p$, compute $com = g^{open} h^x$ and output com.

$\mathsf{VfCom}(par_c, com, x', open')$**.** Compute $com' = g^{open'} h^{x'}$. If $com = com'$ then output 1 else 0.

The Pedersen commitment scheme can be extended to allow the computation of commitments to more than one message.

$\mathsf{CSetup}(1^k, l)$**.** On input the security parameter 1^k and an upper bound l on the number of elements to be committed, pick $l + 1$ random generators $h_1, \ldots, h_l, g$ of a group $\mathbb{G}_p$ of prime order p. Output $par_c = (h_1, \ldots, h_l, g, \mathcal{M})$, where $\mathcal{M} = \mathbb{Z}_p^l$.

$\mathsf{Com}(par_c, \langle x_1, \ldots, x_l \rangle)$**.** Pick random $open \leftarrow \mathbb{Z}_p$, compute $com = g^{open} \prod_{i=1}^{l} h_i^{x_i}$ and output com.

$\mathsf{VfCom}(par_c, com, \langle x'_1, \ldots, x'_l \rangle, open')$**.** Compute $com' = g^{open'} \prod_{i=1}^{l} h_i^{x'_i}$. If $com = com'$ then output 1 else 0.

2.4 Public-Key Encryption

An IND-CPA (or semantically) secure Public-Key Encryption (PKE) scheme consists of three algorithms ($\mathsf{Setup}, \mathsf{Encrypt}, \mathsf{Decrypt}$) defined as follows.

$\mathsf{Setup}(1^k)$**.** On input the security parameter 1^k, it outputs a public key pk_{enc} and a secret key sk_{enc}.

$\mathsf{Encrypt}(m, pk_{enc})$**.** On input a message m and a public key pk_{enc}, it outputs a ciphertext c.

$\mathsf{Decrypt}(c, sk_{enc})$**.** On input a ciphertext c and a secret key sk_{enc}, it outputs m.

A PKE scheme must be correct and satisfy the IND-CPA property. We recall the correctness and IND-CPA definitions in Appendix A.3.

Coconut uses the ElGamal public-key encryption scheme [19], which is IND-CPA secure under the Decisional Diffie-Hellman (DDH) assumption. The ElGamal PKE scheme works as follows.

$\mathsf{Setup}(1^k)$**.** On input 1^k, take a generator g of a cyclic group $\mathbb{G}$ of primer order p. Pick random $x \leftarrow \mathbb{Z}_p$ and set the public key $pk_{enc} \leftarrow g^x$ and the secret key $sk_{enc} \leftarrow x$.

$\mathsf{Encrypt}(m, pk_{enc})$**.** On input a message m and a public key pk_{enc}, pick random $r \in \mathbb{Z}_p$ and output the ciphertext $c = (g^r, m \cdot (pk_{enc})^r)$.

$\mathsf{Decrypt}(c, sk_{enc})$**.** On input a ciphertext $c = (a, b)$ and a decryption key sk_{enc}, output $m = b \cdot a^{-sk_{enc}}$.

Coconut uses the additively homomorphic ElGamal PKE scheme. A PKE scheme is additvely homomorphic if there exist two operations $\oplus$ and $\odot$ as follows. Given two ciphertexts $c_1 \leftarrow \mathsf{Encrypt}(m_1, pk_{enc})$ and $c_2 \leftarrow \mathsf{Encrypt}(m_2, pk_{enc})$, the operation $c \leftarrow c_1 \oplus c_2$ produces a ciphertext such that $m_1 + m_2 \leftarrow \mathsf{Decrypt}(c, sk_{enc})$. Given $c_1 \leftarrow \mathsf{Encrypt}(m_1, pk_{enc})$ and m_2, the operation $c \leftarrow c_1 \odot m_2$ produces a ciphertext such that $m_1 \cdot m_2 \leftarrow \mathsf{Decrypt}(c, sk_{enc})$.

2.5 Signature Schemes

A signature scheme consists of the algorithms KeyGen, Sign, and VfSig. Algorithm $\mathsf{KeyGen}(1^k)$ outputs a secret key sk and a public key pk, which include a description of the message space $\mathcal{M}$. $\mathsf{Sign}(sk, m)$ outputs a signature σ on message $m \in \mathcal{M}$. $\mathsf{VfSig}(pk, \sigma, m)$ outputs 1 if σ is a valid signature on m and 0 otherwise. This definition can be extended to blocks of messages $(m_1, \ldots, m_q)$. In this case, $\mathsf{KeyGen}(1^k, q)$ receives the maximum number of messages as input.

A signature scheme must fulfill the following correctness and existential unforgeability properties [20]. We recall the definitions of those properties in Appendix A.4.

Coconut uses the Pointcheval-Sanders signature scheme [13]. This scheme is existentially unforgeable under Assumption 2 (see Definition 2). It works as follows.

$\mathsf{KeyGen}(1^k, q)$**.** Run $\mathcal{G}(1^k)$ to obtain a pairing group setup $\theta = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g})$. Pick random secret key $(x, y_1, \ldots, y_q) \leftarrow \mathbb{Z}_p^{q+1}$. Output the secret key $sk = (\theta, x, y_1, \ldots y_q)$ and the public key $pk = (\theta, \alpha, \beta_1, \ldots, \beta_q) \leftarrow (\theta, \tilde{g}^x, \tilde{g}^{y_1}, \ldots, \tilde{g}^{y_q})$.

$\mathsf{Sign}(sk, m_1, \ldots, m_q)$**.** Parse sk as $(\theta, x, y_1, \ldots, y_q)$. Pick random $r \leftarrow \mathbb{Z}_p$ and set $h \leftarrow g^r$. Output the signature $\sigma = (h, s) \leftarrow (h, h^{x+y_1 m_1+\ldots+y_q m_q})$.

$\mathsf{VfSig}(pk, \sigma, m_1, \ldots, m_q)$**.** Parse the public key pk as $(\theta, \alpha, \beta_1, \ldots, \beta_q)$ and the signature σ as (h, s). Output 1 if $h \neq 1$ and $e(h, \alpha \prod_{j=1}^{q} \beta_j^{m_j}) = e(s, \tilde{g})$. Otherwise output 0.

This signature scheme is randomizable. To randomize a signature $\sigma = (h, s)$, pick random $r' \leftarrow \mathbb{Z}_p$ and compute $\sigma' = (h^{r'}, s^{r'})$.

3 The Coconut Scheme

We recall the description of the Coconut scheme in Appendix B in [11]. The parties in the protocol are the user $\mathcal{U}$, the authorities $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$ and the provider $\mathcal{P}$.

Setup. Algorithm $\mathsf{Setup}(1^\lambda, q)$ is run by a trusted party. 1^λ is the security parameter in unary notation and q is the number of messages that are signed by a signature.

- Run $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$.
- Compute q generators $(h_1, \ldots, h_q)$ of $\mathbb{G}$.
- Output the system parameters $params = (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, g, \tilde{g}, h_1, \ldots, h_q)$.

Key Generation. Algorithm $\mathsf{TTPKeyGen}(params, t, n, q)$ is run by a trusted party. *params* are the parameters output by Setup, t is the threshold, n is the number of authorities, and q is the number of messages that are signed by a signature.

- Choose $(q+1)$ polynomials $(v, w_1, \ldots, w_q)$ of degree $(t-1)$ with random coefficients in $\mathbb{Z}_p$.
- Set $(x, y_1, \ldots, y_q) \leftarrow (v(0), w_1(0), \ldots, w_q(0))$.
- For $i = 1$ to n, set the secret key sk_i of each authority $\mathcal{V}_i$ as $sk_i = (x_i, y_{i,1}, \ldots, y_{i,q}) \leftarrow (v(i), w_1(i), \ldots, w_q(i))$.
- For $i = 1$ to n, set the verification key pk_i of each authority $\mathcal{V}_i$ as $pk_i = (\alpha_i, \beta_{i,1}, \ldots, \beta_{i,q}) \leftarrow (\tilde{g}^{x_i}, \tilde{g}^{y_{i,1}}, \ldots, \tilde{g}^{y_{i,q}})$.
- Compute the verification key $pk = (\alpha, \beta_1, \ldots, \beta_q) \leftarrow (\tilde{g}^x, \tilde{g}^{y_1}, \ldots, \tilde{g}^{y_q})$.
- Output $(pk, sk_1, pk_1, \ldots, sk_n, pk_n)$.

Protocol for Issuing a Signature. This protocol consists of four algorithms. First, the user runs $\mathsf{PrepareBlindSign}(params, m_1, \ldots, m_q, \phi)$. *params* are the parameters output by Setup, $(m_1, \ldots, m_q)$ are the messages to be signed in the signature, and ϕ are statements to be proven about the messages.

- Generate an ElGamal key pair (d, γ). Pick $d \leftarrow \mathbb{Z}_p$ and compute $\gamma \leftarrow g^d$.
- Pick random $o \leftarrow \mathbb{Z}_p$ and compute the commitment $c_m \leftarrow g^o \prod_{j=1}^{q} h_j^{m_j}$.
- Compute $h \leftarrow H(c_m)$. h is the generator used to compute the signature.
- Compute ElGamal encryptions of each of the messages. For $j = 1$ to q, pick random $k_j \leftarrow \mathbb{Z}_p$ and set $c_j = (a_j, b_j) = (g^{k_j}, \gamma^{k_j} h^{m_j})$.
- Compute a ZK argument of knowledge π_s via the Fiat-Shamir heuristic for the relation

$$
\begin{aligned}
\pi_s = \mathrm{NIZK}\{&(d, m_1, \ldots, m_q, o, k_1, \ldots, k_q) : \\
&\gamma = g^d \wedge \\
&c_m = g^o \prod_{j=1}^{q} h_j^{m_j} \wedge \\
&\{a_j = g^{k_j} \wedge \ b_j = \gamma^{k_j} h^{m_j}\}_{\forall j \in [1,\ldots,q]} \wedge \\
&\phi(m_1, \ldots, m_q) = 1\}
\end{aligned}
$$

- Output $(d, \Lambda = (\gamma, c_m, c_1, \ldots, c_q, h, \pi_s), \phi)$.

The user sends Λ and ϕ to t authorities. Each authority $\mathcal{V}_i$ runs algorithm $\mathsf{BlindSign}(params, sk_i, \Lambda, \phi)$. *params* are the parameters output by Setup, Λ is the second output of $\mathsf{PrepareBlindSign}$ and ϕ are statements proven about the messages.

- Parse Λ as $(\gamma, c_m, c_1, \ldots, c_q, h, \pi_s)$.
- Compute $h' \leftarrow H(c_m)$. Abort if $h \neq h'$.
- Verify π_s by using ϕ, *params* and $(\gamma, c_m, c_1, \ldots, c_q, h)$. Abort if the proof is not correct.

– For $j=1$ to q, parse c_j as (a_j, b_j).
– Compute $\tilde{c} = (\prod_{j=1}^{q} a_j^{y_j}, h^x \prod_{j=1}^{q} b_j^{y_j})$.
– Output the blinded signature share $\tilde{\sigma}_i = (h, \tilde{c})$.

Each authority $\mathcal{V}_i$ sends $\tilde{\sigma}_i$ to the user. For all the signature shares received, the user then executes $\mathsf{Unblind}(params, d, \Lambda, \tilde{\sigma}_i, pk_i, m_1, \ldots, m_q)$. *params* are the parameters output by Setup, d is the secret key of ElGamal output by $\mathsf{PrepareBlindSign}$, Λ is the tuple output by $\mathsf{PrepareBlindSign}$, $\tilde{\sigma}_i$ is the signature share output by $\mathsf{BlindSign}$, and $(m_1, \ldots, m_q)$ are the signed messages.

– Parse Λ as $(\gamma, c_m, c_1, \ldots, c_q, h, \pi_s)$.
– Parse $\tilde{\sigma}_i$ as $(h', \tilde{c})$ and $\tilde{c}$ as $(\tilde{a}, \tilde{b})$. Abort if $h \neq h'$.
– Compute $\sigma_i = (h, s_i) \leftarrow (h, \tilde{b}(\tilde{a})^{-d})$.
– Parse pk_i as $(\alpha_i, \beta_{i,1}, \ldots, \beta_{i,q})$ and verify the signature share. Abort if the equality $e(h, \alpha_i \prod_{j=1}^{q} \beta_{i,j}^{m_j}) = e(s_i, \tilde{g})$ does not hold.
– Output the signature share σ_i.

Let $\mathbb{S} \in [1, n]$ be the set of indices of authorities that provided signature shares. The user executes $\mathsf{AggCred}(params, \langle\sigma_i\rangle_{i\in\mathbb{S}}, pk, m_1, \ldots, m_q)$. *params* are the parameters output by Setup, $\langle\sigma_i\rangle_{i\in\mathbb{S}}$ are the t signature shares output by t executions of $\mathsf{Unblind}$, pk is the verification key output by $\mathsf{TTPKeyGen}$, and $(m_1, \ldots, m_q)$ are the signed messages.

– For all $i \in \mathbb{S}$, parse σ_i as (h, s_i).
– For all $i \in \mathbb{S}$, evaluate at 0 the Lagrange basis polynomials

$$l_i = [\prod_{j\in\mathbb{S}, j\neq i} (0-j)][\prod_{j\in\mathbb{S}, j\neq i} (i-j)]^{-1} \bmod p$$

– Compute the signature $\sigma = (h, s) \leftarrow (h, \prod_{i\in\mathbb{S}} s_i^{l_i})$.
– Parse pk as $(\alpha, \beta_1, \ldots, \beta_q)$ and verify the signature. Abort if the equality $e(h, \alpha \prod_{j=1}^{q} \beta_j^{m_j}) = e(s, \tilde{g})$ does not hold.
– Output σ.

Protocol for Proving Signature Possession. This protocol consists of two algorithms. First, the user runs $\mathsf{ProveCred}(params, pk, m_1, \ldots, m_q, \sigma, \phi')$. *params* are the parameters output by Setup, pk is the verification key output by algorithm $\mathsf{TTPKeyGen}$, $(m_1, \ldots, m_q)$ are the signed messages, σ is the signature output by $\mathsf{AggCred}$ and ϕ' are statements to be proven about the messages.

– Parse σ as (h, s).
– Parse pk as $(\alpha, \beta_1, \ldots, \beta_q)$.
– Pick random $r \leftarrow \mathbb{Z}_p$ and $r' \leftarrow \mathbb{Z}_p$.
– Compute $h' \leftarrow h^{r'}$ and $s' \leftarrow s^{r'}$.
– Compute $\kappa \leftarrow \alpha \prod_{j=1}^{q} \beta_j^{m_j} \tilde{g}^r$.
– Compute $\nu \leftarrow (h')^r$.

– Compute the ZK argument of knowledge π_v via the Fiat-Shamir heuristic.

$$\begin{aligned}\pi_v = &\text{NIZK}\{(m_1, \ldots, m_q, r) : \\ &\kappa = \alpha \prod_{j=1}^{q} \beta_j^{m_j} \tilde{g}^r \ \wedge \\ &\nu = (h')^r \ \wedge \\ &\phi'(m_1, \ldots, m_q) = 1\}\end{aligned}$$

– Set $\Theta = (\kappa, \nu, h', s', \pi_v)$.
– Output (Θ, ϕ').

The user sends (Θ, ϕ') to the provider. The provider runs the algorithm $\mathsf{VerifyCred}(params, pk, \Theta, \phi')$. $params$ are the parameters output by Setup, pk is the verification key output by $\mathsf{TTPKeyGen}$, and (Θ, ϕ') is the output of $\mathsf{ProveCred}$.

– Parse Θ as $(\kappa, \nu, h', s', \pi_v)$.
– Verify π_v by using ϕ', $params$, pk, h', s', κ and ν. Output *false* if the proof is not correct.
– Output *false* if $h' = 1$.
– Output *false* if $e(h', \kappa) = e(s'\nu, \tilde{g})$ does not hold.
– Output *true*.

3.1 Discussion of Differences between Coconut and $\Pi_{\mathbf{AC}}$

In order to prove the security of Π_{AC}, we have modified Coconut as follows:

– Coconut uses as building block the Pointcheval-Sanders signature scheme. A natural way to prove that Coconut fulfills the unforgeability property would be to describe a reduction to the unforgeability of Pointcheval-Sanders signatures, i.e. to show that, if an adversarial user is able to show a signature that signs attributes that the adversary was not issued, then that means that we can break the unforgeability property of Pointcheval-Sanders signatures. Such a reduction would receive the public key from the challenger of the unforgeability game and assign that public key to an authority. The reduction would then interact with an adversarial user by simulating both the authority during the issuance phase and the provider during the show phase. In order to simulate the authority, the reduction would extract the attributes whose issuance the adversary requests by running the Fiat-Shamir extractor for proof π_s. The reduction would submit them to the signing oracle provided by the challenger of the unforgeability game. Then the reduction would use the signature sent by the oracle to produce the blinded signature share $\tilde{\sigma}_i = (h, \tilde{c})$, where $\tilde{c} = (\tilde{a}, \tilde{b}) = (\prod_{j=1}^{q} a_j^{y_j}, h^x \prod_{j=1}^{q} b_j^{y_j})$, that needs to be sent to the adversary.
However, it does not seem possible for the reduction to compute the value $\tilde{a} = \prod_{j=1}^{q} a_j^{y_j}$. The reduction does not know the values y_j of the secret

key. The signature received from the oracle is of the form $\sigma = (h, s) \leftarrow (h, h^{x+y_1 m_1+\ldots+y_q m_q})$, and seemingly cannot be used to compute $\tilde{a}$ either. To solve this issue, Π_{AC} extends the public key $pk = (\alpha, \beta_1, \ldots, \beta_q) \leftarrow (\tilde{g}, \tilde{g}^x, \tilde{g}^{y_1}, \ldots, \tilde{g}^{y_q})$ by adding the values $(g^{y_1}, \ldots, g^{y_q})$. This implies that unforgeability in Π_{AC} holds under Assumption 1 in Definition 1 rather than the weaker Assumption 2 in Definition 2. We remark that Pointcheval and Sanders [13] also propose an ABC scheme with a single authority based on their signature scheme, and unforgeability for that scheme holds under Assumption 1, despite the fact that unforgeability for the signature scheme holds under Assumption 2.

- In the issuance protocol in Coconut, blindness holds under the XDH assumption (Definition 3) because of the use of the ElGamal PKE scheme. After extending the public key as described above, it is possible to modify the issuance protocol so that blindness holds against unbounded adversaries. The modified protocol, used in Π_{AC}, is also more efficient.
- In the show protocol in Cococut, a valid signature is revealed to the verifier. Therefore, unlinkability requires that signatures sign at least one random attribute. Additionally, unlinkability cannot be proven against unbounded adversaries. As can be seen, an adversary able to solve the discrete logarithm problem can compute the value r from ν, and later compute $\alpha \prod_{j=1}^{q} \beta_j^{m_j}$ from κ. The value $\alpha \prod_{j=1}^{q} \beta_j^{m_j}$ can be used to trace all the shows of the same credential.
  In Π_{AC}, a modified version of the show protocol is used. This show protocol provides unlinkability against unbounded adversaries and removes the need of having a random attribute signed in each credential. Additionally, it is slightly more efficient.

In addition to the changes mentioned above, Π_{AC} uses the functionalities $\mathcal{F}_{\mathrm{NYM}}$ to model the communication channel between parties, $\mathcal{F}_{\mathrm{KG}}$ to model key generation and distribution, and $\mathcal{F}_{\mathrm{RO}}$ to model hash functions. Those functionalities are defined in Section 6.

4 Security Framework

We prove our protocol secure in the ideal-world/real-world paradigm [12]. This paradigm allows one to define and analyze the security of cryptographic protocols so that security is retained under arbitrary composition with other protocols. The security of a protocol is defined by means of an ideal protocol that carries out the desired task. In the ideal protocol, all parties send their inputs to an ideal functionality $\mathcal{F}$ for the task. $\mathcal{F}$ locally computes the outputs of the parties and provides each party with its prescribed output.

The security of a protocol φ is analyzed by comparing the view of an environment $\mathcal{Z}$ in a real execution of φ against that of $\mathcal{Z}$ in the ideal protocol defined in $\mathcal{F}_{\varphi}$. $\mathcal{Z}$ chooses the inputs of the parties and collects their outputs. In the real world, $\mathcal{Z}$ can communicate freely with an adversary $\mathcal{A}$ who controls both the

network and any corrupt parties. In the ideal world, $\mathcal{Z}$ interacts with dummy parties, who simply relay inputs and outputs between $\mathcal{Z}$ and $\mathcal{F}_\varphi$, and a simulator $\mathcal{S}$. We say that a protocol φ securely realizes $\mathcal{F}_\varphi$ if $\mathcal{Z}$ cannot distinguish the real world from the ideal world, i.e., $\mathcal{Z}$ cannot distinguish whether it is interacting with $\mathcal{A}$ and parties running protocol φ or with $\mathcal{S}$ and dummy parties relaying to $\mathcal{F}_\varphi$.

More formally, indistinguishability is defined as follows. Two binary distribution ensembles $X = \{X(k,a)\}_{k\in\mathbb{N},a\in\{0,1\}^*}$ and $Y = \{Y(k,a)\}_{k\in\mathbb{N},a\in\{0,1\}^*}$ are indistinguishable ($X \approx Y$) if for any $c, d \in \mathbb{N}$ there exists $k_0 \in \mathbb{N}$ such that for all $k > k_0$ and all $a \in \cup_{\kappa\le k^d}\{0,1\}^\kappa$, $|\Pr[X(k,a)=1] - \Pr[Y(k,a)=1]| < k^{-c}$. Let $\mathrm{REAL}_{\varphi,\mathcal{A},\mathcal{Z}}(k,a)$ denote the distribution given by the output of $\mathcal{Z}$ when executed on input a with $\mathcal{A}$ and parties running φ, and let $\mathrm{IDEAL}_{\mathcal{F}_\varphi,\mathcal{S},\mathcal{Z}}(k,a)$ denote the output distribution of $\mathcal{Z}$ when executed on input a with $\mathcal{S}$ and dummy parties relaying to $\mathcal{F}_\varphi$. We say that the protocol φ securely realizes $\mathcal{F}_\varphi$ if, for all polynomial-time $\mathcal{A}$, there exists a polynomial-time $\mathcal{S}$ such that, for all polynomial-time $\mathcal{Z}$, $\mathrm{REAL}_{\varphi,\mathcal{A},\mathcal{Z}} \approx \mathrm{IDEAL}_{\mathcal{F}_\varphi,\mathcal{S},\mathcal{Z}}$.

A protocol $\varphi^{\mathcal{G}}$ securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model when φ is allowed to invoke the ideal functionality $\mathcal{G}$. Therefore, for any protocol ψ that securely realizes $\mathcal{G}$, the composed protocol φ^ψ, which is obtained by replacing each invocation of an instance of $\mathcal{G}$ with an invocation of an instance of ψ, securely realizes $\mathcal{F}$.

In the ideal functionalities described in this paper, we consider static corruptions. When describing ideal functionalities, we use the following conventions as in [21].

Interface Naming Convention. An ideal functionality can be invoked by using one or more interfaces. The name of a message in an interface consists of three fields separated by dots, e.g., ac.setup.ini in $\mathcal{F}_{\mathrm{AC}}$ in Section 5. The first field indicates the name of the functionality and is the same in all interfaces of the functionality. This field is useful for distinguishing between invocations of different functionalities in a hybrid protocol that uses two or more different functionalities. The second field indicates the kind of action performed by the functionality and is the same in all messages that the functionality exchanges within the same interface. The third field distinguishes between the messages that belong to the same interface, and can take the following different values. A message ac.setup.ini is the incoming message received by the functionality, i.e., the message through which the interface is invoked. A message ac.setup.end is the outgoing message sent by the functionality, i.e., the message that ends the execution of the interface. The message ac.setup.sim is used by the functionality to send a message to $\mathcal{S}$, and the message ac.setup.rep is used to receive a message from $\mathcal{S}$.

Network vs local communication. The identity of an interactive Turing machine instance (ITI) consists of a party identifier *pid* and a session identifier *sid*. A set of parties in an execution of a system of interactive Turing machines is a protocol instance if they have the same session identifier *sid*. ITIs can pass direct inputs to and outputs from "local" ITIs that have the

same *pid*. An ideal functionality $\mathcal{F}$ has $pid = \perp$ and is considered local to all parties. An instance of $\mathcal{F}$ with the session identifier *sid* only accepts inputs from and passes outputs to machines with the same session identifier *sid*. Some functionalities require the session identifier to have some structure. Those functionalities check whether the session identifier possesses the required structure in the first message that invokes the functionality. For the subsequent messages, the functionality implicitly checks that the session identifier equals the session identifier used in the first message. Communication between ITIs with different party identifiers must take place over the network. The network is controlled by $\mathcal{A}$, meaning that he can arbitrarily delay, modify, drop, or insert messages.

Query identifiers. Some interfaces in a functionality can be invoked more than once. When the functionality sends a message ac.setup.sim to $\mathcal{S}$ in such an interface, a query identifier *qid* is included in the message. The query identifier must also be included in the response ac.setup.rep sent by $\mathcal{S}$. The query identifier is used to identify the message ac.setup.sim to which $\mathcal{S}$ replies with a message ac.setup.rep. We note that, typically, $\mathcal{S}$ in the security proof may not be able to provide an immediate answer to the functionality after receiving a message ac.setup.sim. The reason is that $\mathcal{S}$ typically needs to interact with the copy of $\mathcal{A}$ it runs in order to produce the message ac.setup.rep, but $\mathcal{A}$ may not provide the desired answer or may provide a delayed answer. In such cases, when the functionality sends more than one message ac.setup.sim to $\mathcal{S}$, $\mathcal{S}$ may provide delayed replies, and the order of those replies may not follow the order of the messages received.

Aborts. When an ideal functionality $\mathcal{F}$ aborts after being activated with a message sent by a party, we mean that $\mathcal{F}$ halts the execution of its program and sends a special abortion message to the party that invoked the functionality. When an ideal functionality $\mathcal{F}$ aborts after being activated with a message sent by $\mathcal{S}$, we mean that $\mathcal{F}$ halts the execution of its program and sends a special abortion message to the party that receives the outgoing message from $\mathcal{F}$ after $\mathcal{F}$ is activated by $\mathcal{S}$.

5 Ideal Functionality $\mathcal{F}_{\mathrm{AC}}$

We depict our functionality $\mathcal{F}_{\mathrm{AC}}$ for attribute-based access control with threshold issuance in Figure 1 and Figure 2. $\mathcal{F}_{\mathrm{AC}}$ interacts with authorities $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$, any number of users $\mathcal{U}_j$ and any number of providers $\mathcal{P}_k$. $\mathcal{F}_{\mathrm{AC}}$ consists of the following interfaces:

1. $\mathcal{V}_i$ uses the ac.setup interface to set up $\mathcal{F}_{\mathrm{AC}}$. $\mathcal{F}_{\mathrm{AC}}$ stores the fact that $\mathcal{V}_i$ has run the setup interface. $\mathcal{F}_{\mathrm{AC}}$ enforces that each authority runs the setup interface only once. The simulator $\mathcal{S}$ is allowed to learn that $\mathcal{V}_i$ has run the setup interface.
2. $\mathcal{U}_j$ uses the ac.request interface to send an attribute a, a statement ϕ, the identifier of an authority $\mathcal{V}_i$ and a pseudonym P to $\mathcal{F}_{\mathrm{AC}}$. $\mathcal{F}_{\mathrm{AC}}$ checks that the attribute a fulfills the statement ϕ, which we represent by $1 = \phi(a)$. $\mathcal{F}_{\mathrm{AC}}$

also checks that $\mathcal{V}_i$ has run the setup interface. $\mathcal{F}_{\mathrm{AC}}$ stores the request under a request identifier *reqid* and sends ϕ, P and *reqid* to $\mathcal{V}_i$.

3. $\mathcal{V}_i$ uses the ac.issue interface to send a request identifier *reqid* to $\mathcal{F}_{\mathrm{AC}}$. $\mathcal{F}_{\mathrm{AC}}$ checks that there is a request pending for *reqid*. Then $\mathcal{F}_{\mathrm{AC}}$ records that the authority $\mathcal{V}_i$ issues the attribute a to the user $\mathcal{U}_j$ and informs the user that $\mathcal{V}_i$ issued the attribute a.
4. $\mathcal{U}_j$ uses the ac.show interface to send a statement φ, a pseudonym P, and the identifier of a provider $\mathcal{P}_k$ to $\mathcal{F}_{\mathrm{AC}}$. By using an algorithm Find, $\mathcal{F}_{\mathrm{AC}}$ checks that there is a set $\mathbb{A}$ of attributes issued to $\mathcal{U}_j$ that fulfills φ, which we represent by $1 = \varphi(\mathbb{A})$. Each of the attributes in the set must have been issued by at least t authorities. Then $\mathcal{F}_{\mathrm{AC}}$ sends φ and P to the provider $\mathcal{P}_k$.

The session identifier *sid* has the structure $(\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$. This allows any authorities $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$ to create an instance of $\mathcal{F}_{\mathrm{AC}}$. After the first invocation of $\mathcal{F}_{\mathrm{AC}}$, $\mathcal{F}_{\mathrm{AC}}$ implicitly checks that the session identifier in a message is equal to the one received in the first invocation.

When invoked by an authority $\mathcal{V}_i$ or a user $\mathcal{U}_j$, $\mathcal{F}_{\mathrm{AC}}$ first checks the correctness of the input. $\mathcal{F}_{\mathrm{AC}}$ aborts if any of the inputs does not belong to the correct domain. $\mathcal{F}_{\mathrm{AC}}$ also aborts if an interface is invoked at an incorrect moment in the protocol. For example, an authority $\mathcal{V}_i$ cannot invoke the ac.issue interface on input a request identifier *reqid* if that authority did not receive a request associated with *reqid*. Similar abortion conditions are listed when $\mathcal{F}_{\mathrm{AC}}$ receives a message from the simulator $\mathcal{S}$.

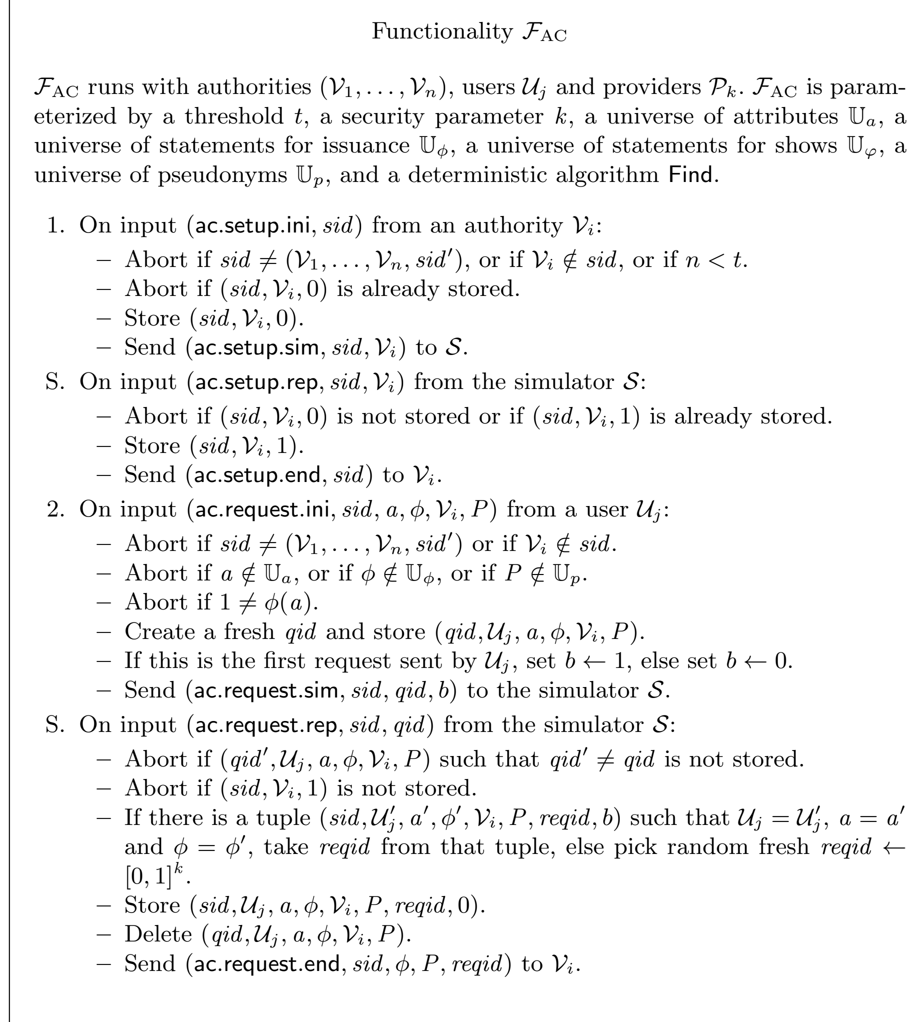
Before $\mathcal{F}_{\mathrm{AC}}$ queries $\mathcal{S}$, $\mathcal{F}_{\mathrm{AC}}$ saves its state, which is recovered when receiving a response from $\mathcal{S}$. When an interface, e.g. ac.request, can be invoked by a party more than once, $\mathcal{F}_{\mathrm{AC}}$ creates a query identifier *qid*, which allows $\mathcal{F}_{\mathrm{AC}}$ to match a query to $\mathcal{S}$ to a response from $\mathcal{S}$. Creating *qid* is not necessary if an interface, such as ac.setup, can be invoked only once by each authority, and the authority identifier is revealed to $\mathcal{S}$.

The information that $\mathcal{F}_{\mathrm{AC}}$ reveals to $\mathcal{S}$ is information that an adversary that controls the network but does not corrupt any party is allowed to learn. For example, in the ac.setup interface, $\mathcal{S}$ learns the authority identifier $\mathcal{V}_i$, and thus a construction for $\mathcal{F}_{\mathrm{AC}}$ does not need to hide that from the adversary.

Compared to other ideal functionalities, $\mathcal{F}_{\mathrm{AC}}$ looks more complex. The reason is that we list all the conditions for abortion and that $\mathcal{F}_{\mathrm{AC}}$ saves state information before querying $\mathcal{S}$ and recovers it after receiving a response from $\mathcal{S}$. These operations are also required but have frequently been omitted in the description of ideal functionalities in the literature.

Below we describe and discuss $\mathcal{F}_{\mathrm{AC}}$ more in detail.

1. The ac.setup interface is invoked by an authority $\mathcal{V}_i$. $\mathcal{F}_{\mathrm{AC}}$ aborts if the session identifier *sid* does not have the correct structure. $\mathcal{F}_{\mathrm{AC}}$ also aborts if $\mathcal{V}_i$ already invoked the ac.setup interface. If $\mathcal{F}_{\mathrm{AC}}$ does not abort, $\mathcal{F}_{\mathrm{AC}}$ records that $\mathcal{V}_i$ invoked the setup interface and queries the simulator $\mathcal{S}$ on input the authority identifier $\mathcal{V}_i$. When prompted by $\mathcal{S}$, $\mathcal{F}_{\mathrm{AC}}$ aborts if $\mathcal{V}_i$ did not invoke the setup interface or if the setup interface was already run for authority $\mathcal{V}_i$.



Functionality $\mathcal{F}_{\mathrm{AC}}$

$\mathcal{F}_{\mathrm{AC}}$ runs with authorities $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$, users $\mathcal{U}_j$ and providers $\mathcal{P}_k$. $\mathcal{F}_{\mathrm{AC}}$ is parameterized by a threshold t, a security parameter k, a universe of attributes $\mathbb{U}_a$, a universe of statements for issuance $\mathbb{U}_\phi$, a universe of statements for shows $\mathbb{U}_\varphi$, a universe of pseudonyms $\mathbb{U}_p$, and a deterministic algorithm Find.

1. On input $(\mathsf{ac.setup.ini}, sid)$ from an authority $\mathcal{V}_i$:
   - Abort if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$, or if $\mathcal{V}_i \notin sid$, or if $n < t$.
   - Abort if $(sid, \mathcal{V}_i, 0)$ is already stored.
   - Store $(sid, \mathcal{V}_i, 0)$.
   - Send $(\mathsf{ac.setup.sim}, sid, \mathcal{V}_i)$ to $\mathcal{S}$.

S. On input $(\mathsf{ac.setup.rep}, sid, \mathcal{V}_i)$ from the simulator $\mathcal{S}$:
   - Abort if $(sid, \mathcal{V}_i, 0)$ is not stored or if $(sid, \mathcal{V}_i, 1)$ is already stored.
   - Store $(sid, \mathcal{V}_i, 1)$.
   - Send $(\mathsf{ac.setup.end}, sid)$ to $\mathcal{V}_i$.

2. On input $(\mathsf{ac.request.ini}, sid, a, \phi, \mathcal{V}_i, P)$ from a user $\mathcal{U}_j$:
   - Abort if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$ or if $\mathcal{V}_i \notin sid$.
   - Abort if $a \notin \mathbb{U}_a$, or if $\phi \notin \mathbb{U}_\phi$, or if $P \notin \mathbb{U}_p$.
   - Abort if $1 \neq \phi(a)$.
   - Create a fresh qid and store $(qid, \mathcal{U}_j, a, \phi, \mathcal{V}_i, P)$.
   - If this is the first request sent by $\mathcal{U}_j$, set $b \leftarrow 1$, else set $b \leftarrow 0$.
   - Send $(\mathsf{ac.request.sim}, sid, qid, b)$ to the simulator $\mathcal{S}$.

S. On input $(\mathsf{ac.request.rep}, sid, qid)$ from the simulator $\mathcal{S}$:
   - Abort if $(qid', \mathcal{U}_j, a, \phi, \mathcal{V}_i, P)$ such that $qid' \neq qid$ is not stored.
   - Abort if $(sid, \mathcal{V}_i, 1)$ is not stored.
   - If there is a tuple $(sid, \mathcal{U}'_j, a', \phi', \mathcal{V}_i, P, reqid, b)$ such that $\mathcal{U}_j = \mathcal{U}'_j$, $a = a'$ and $\phi = \phi'$, take $reqid$ from that tuple, else pick random fresh $reqid \leftarrow [0,1]^k$.
   - Store $(sid, \mathcal{U}_j, a, \phi, \mathcal{V}_i, P, reqid, 0)$.
   - Delete $(qid, \mathcal{U}_j, a, \phi, \mathcal{V}_i, P)$.
   - Send $(\mathsf{ac.request.end}, sid, \phi, P, reqid)$ to $\mathcal{V}_i$.

Fig. 1. Description of functionality $\mathcal{F}_{\mathrm{AC}}$: interfaces ac.setup and ac.request

Functionality $\mathcal{F}_{\mathrm{AC}}$

3. On input ($\mathsf{ac.issue.ini}$, sid, $reqid$) from an authority $\mathcal{V}_i$:
   - Abort if a tuple $(sid, \mathcal{U}_j, a, \phi, \mathcal{V}'_i, P, reqid', 0)$ such that $reqid' = reqid$ and $\mathcal{V}'_i = \mathcal{V}_i$ is not stored.
   - Create a fresh qid and store $(qid, \mathcal{U}_j, a, \mathcal{V}_i, P, \phi, reqid)$.
   - Store $(sid, \mathcal{U}_j, a, \phi, \mathcal{V}_i, P, reqid, 1)$ and delete $(sid, \mathcal{U}_j, a, \phi, \mathcal{V}_i, P, reqid, 0)$.
   - Send ($\mathsf{ac.issue.sim}$, sid, qid) to the simulator $\mathcal{S}$.

S. On input ($\mathsf{ac.issue.rep}$, sid, qid) from the simulator $\mathcal{S}$:
   - Abort if $(qid', \mathcal{U}_j, a, \mathcal{V}_i, P, \phi, reqid)$ such that $qid' \neq qid$ is not stored.
   - If $\mathcal{U}_j$ is honest, set $\mathcal{U}'_j \leftarrow \mathcal{U}_j$, else set $\mathcal{U}'_j \leftarrow \mathcal{A}$.
   - If a tuple $(sid, \mathcal{U}'_j, \mathsf{DB})$ is not stored, store $(sid, \mathcal{U}'_j, \mathsf{DB})$, where DB is a (initially empty) database with entries of the form $[reqid, a, \mathbb{V}]$.
   - If there is not an entry $[reqid, a, \mathbb{V}]$ in DB such that $reqid' = reqid$, add an entry $[reqid, a, \emptyset]$ to DB.
   - If $\mathcal{U}_j$ or $\mathcal{V}_i$ are honest, replace the entry $[reqid, a, \mathbb{V}]$ in DB by $[reqid, a, \mathbb{V} \cup \{\mathcal{V}_i\}]$.
   - Delete $(qid, \mathcal{U}_j, a, \mathcal{V}_i, P, \phi, reqid)$.
   - Send ($\mathsf{ac.issue.end}$, sid, a, ϕ, $\mathcal{V}_i$) to $\mathcal{U}_j$.

4. On input ($\mathsf{ac.show.ini}$, sid, φ, P, $\mathcal{P}_k$) from a user $\mathcal{U}_j$:
   - Abort if $\varphi \notin \mathbb{U}_\varphi$, or if $P \notin \mathbb{U}_p$.
   - If $\mathcal{U}_j$ is honest, set $\mathcal{U}'_j \leftarrow \mathcal{U}_j$, else set $\mathcal{U}'_j \leftarrow \mathcal{A}$.
   - If a tuple $(sid, \mathcal{U}'_j, \mathsf{DB})$ is not stored, store $(sid, \mathcal{U}'_j, \mathsf{DB})$, where DB is a (initially empty) database with entries of the form $[reqid, a, \mathbb{V}]$.
   - Evaluate $\mathbb{A} \leftarrow \mathsf{Find}(\varphi, \mathsf{DB})$. The algorithm Find outputs a set $\mathbb{A}$ of attributes such that both $1 = \varphi(\mathbb{A})$ and, for each $a \in \mathbb{A}$, the entry $[reqid, a, \mathbb{V}] \in \mathsf{DB}$ is such that $|\mathbb{V}| \geq t'$, where $t' = t$, if $\mathcal{U}_j$ is honest, and $t' = t - \tilde{t}$, if $\mathcal{U}_j$ is corrupt. ($\tilde{t} < t$ is the number of corrupt authorities.)
   - Abort if $\mathbb{A} = \emptyset$.
   - Create a fresh qid and store $(qid, \varphi, P, \mathcal{P}_k)$.
   - If this is the first show received by $\mathcal{P}_k$, set $b \leftarrow 1$, else set $b \leftarrow 0$.
   - Send ($\mathsf{ac.show.sim}$, sid, qid, b) to $\mathcal{S}$.

S. On input ($\mathsf{ac.show.rep}$, sid, qid) from $\mathcal{S}$:
   - Abort if a tuple $(qid', \varphi, P, \mathcal{P}_k)$ such that $qid \neq qid'$ is not stored.
   - Delete $(qid, \varphi, P, \mathcal{P}_k)$.
   - Send ($\mathsf{ac.show.end}$, φ, P) to $\mathcal{P}_k$.

Fig. 2. Description of functionality $\mathcal{F}_{\mathrm{AC}}$: interfaces ac.issue and ac.show

If $\mathcal{F}_{\text{AC}}$ does not abort, $\mathcal{F}_{\text{AC}}$ records that the setup interface is already run for $\mathcal{V}_i$ and sends a message to $\mathcal{V}_i$.
The fact that $\mathcal{F}_{\text{AC}}$ queries the simulator $\mathcal{S}$ and only concludes the execution of the setup interface when prompted by the simulator allows an adversary in the real world to delay the execution of the setup interface or prevent it from being finalized. We recall that, in the real world, the adversary controls the network, and thus it can arbitrarily delay or drop messages, thereby delaying or preventing the finalization of the execution. Therefore, $\mathcal{F}_{\text{AC}}$ must allow $\mathcal{S}$ to do that in the setup interface and in other interfaces. The only exception, which does not happen in $\mathcal{F}_{\text{AC}}$, would be an interface whose execution is intended to be local, i.e., an interface where a party is invoked by the environment on some input and produces an output to the environment without sending any message.
$\mathcal{F}_{\text{AC}}$ allows each authority $\mathcal{V}_i$ to run the setup interface independently of other authorities, i.e. the execution of the setup interface for one authority can be finalized without involvement of other authorities. Therefore, $\mathcal{F}_{\text{AC}}$ is realizable by protocols where authorities run the setup interface independently of each other. For example, protocols where each authority creates its own keys, or protocols where authorities obtain their keys from a trusted third party that generates them. $\mathcal{F}_{\text{AC}}$ can be modified so that it is realizable by protocols in which the setup interface requires interaction between authorities, e.g. protocols that use a distributed key generation protocol as building block.

2. The ac.request interface is invoked by a user $\mathcal{U}_j$ on input an attribute a, a statement ϕ, an authority identifier $\mathcal{V}_i$, and a pseudonym P. $\mathcal{F}_{\text{AC}}$ aborts if $\mathcal{V}_i$ is not included in the session identifier sid. $\mathcal{F}_{\text{AC}}$ also aborts if a, ϕ or $\mathcal{V}_i$ do not belong to their respective domains, or if the attribute a does not fulfill the statement ϕ. If $\mathcal{F}_{\text{AC}}$ does not abort, $\mathcal{F}_{\text{AC}}$ saves the request under a query identifier and queries the simulator. The bit b sent to the simulator reveals whether this is the first request by $\mathcal{U}_j$ or not. Construction Π_{AC} reveals this information because, in the first request, the user needs to obtain the keys. When the simulator replies, $\mathcal{F}_{\text{AC}}$ aborts if the query identifier sent by the simulator is not stored. $\mathcal{F}_{\text{AC}}$ also aborts if the authority $\mathcal{V}_i$ has not run the setup interface. Otherwise $\mathcal{F}_{\text{AC}}$ records the request under a request identifier $reqid$ and sends ϕ, P and $reqid$ to $\mathcal{V}_i$.
We remark that $\mathcal{F}_{\text{AC}}$ enforces that requests from the same user for the same attribute and statement to different authorities are linkable through $reqid$. The reason is that such linkability is required in the Coconut protocol and in construction Π_{AC}. Namely, in Coconut the commitment c_m sent to different authorities to request the issuance of an attribute needs to be equal, or otherwise the user would not be able to aggregate the signature shares from different authorities. For other cases, users can choose whether their requests are linkable or unlinkable by choosing appropriate pseudonyms.
We also remark that $\mathcal{F}_{\text{AC}}$ only considers the case in which users prove the same statement to different authorities to request the issuance of a certain attribute. $\mathcal{F}_{\text{AC}}$ and Π_{AC} can easily be generalized to allow users to prove

different statements to each of the authorities to request the issuance of the same attribute.

3. The $\mathsf{ac.issue}$ interface is invoked by an authority $\mathcal{V}_i$ on input a request identifier *reqid*. $\mathcal{F}_{\mathrm{AC}}$ aborts if a request $(sid, \mathcal{U}_j, a, \phi, \mathcal{V}_i, P, reqid')$ such that $reqid' = reqid$ was not sent to $\mathcal{V}_i$. If $\mathcal{F}_{\mathrm{AC}}$ does not abort, $\mathcal{F}_{\mathrm{AC}}$ records that $\mathcal{V}_i$ wishes to issue the attribute corresponding to that request. $\mathcal{F}_{\mathrm{AC}}$ queries the simulator. After being prompted by the simulator, $\mathcal{F}_{\mathrm{AC}}$ updates a database DB for user $\mathcal{U}_j$ to record that $\mathcal{V}_i$ issued the attribute a to $\mathcal{U}_j$. We remark that, if $\mathcal{V}_i$ had already issued a to $\mathcal{U}_j$ for a certain *reqid*, then the database is not updated. The database is not updated either when both $\mathcal{V}_i$ and $\mathcal{U}_j$ are corrupt. In the $\mathsf{ac.show}$ interface, it is assumed that corrupt authorities have issued any attributes to corrupt users.
4. The $\mathsf{ac.show}$ interface is invoked by a user $\mathcal{U}_j$ on input a statement φ, a pseudonym P and the identifier of a provider $\mathcal{P}_k$. $\mathcal{F}_{\mathrm{AC}}$ aborts if φ or P do not belong to their respective universes. We remark that φ differs from ϕ in that φ may involve more than one attribute. $\mathcal{F}_{\mathrm{AC}}$ finds a set $\mathbb{A}$ of attributes that were issued to $\mathcal{U}_j$ and that satisfy φ. If $\mathcal{U}_j$ is honest, the attributes need to be issued by t authorities. If $\mathcal{U}_j$ is corrupt and there are $\tilde{t}$ corrupt authorities, the attributes need to be issued by $t - \tilde{t}$ honest authorities. $\mathcal{F}_{\mathrm{AC}}$ aborts if a set $\mathbb{A}$ cannot be found. Otherwise $\mathcal{F}_{\mathrm{AC}}$ queries the simulator. The bit b sent to the simulator reveals whether this is the first show received by $\mathcal{P}_k$. Construction Π_{AC} reveals this information because, in the first show received, $\mathcal{P}_k$ retrieves the keys. After being prompted by the simulator, $\mathcal{F}_{\mathrm{AC}}$ sends φ and P to the provider $\mathcal{P}_k$.
   We remark that, $\mathcal{F}_{\mathrm{AC}}$ does not enforce that pseudonyms are unique. Therefore, users can optionally link executions of the show interface or an execution of the show interface with an execution of the request interface.
   We also remark that $\mathcal{F}_{\mathrm{AC}}$ does not take into account the case where t or more authorities are corrupt. We will analyze the security of construction Π_{AC} under the assumption that at most $t - 1$ authorities are corrupt.

$\mathcal{F}_{\mathrm{AC}}$ guarantees the following security properties, which are described in [11]:

Unforgeability. $\mathcal{F}_{\mathrm{AC}}$ ensures that any attribute that is used to prove that the user's attributes satisfy the statement φ needs to be issued at least $t - \tilde{t}$ times, where $\tilde{t}$ is the number of corrupt authorities. Therefore, under the assumption that $\tilde{t} \leq t - 1$, at least one honest authority needs to issue the attribute to the user so that the user can employ it to satisfy φ.

Blindness. $\mathcal{F}_{\mathrm{AC}}$ ensures that, in the request interface, an authority learns that the attribute a satisfies the statement ϕ, but no further information about a is revealed to the authority.

Unlinkability. In the request and show interfaces, a user receives a pseudonym as input. This pseudonym will determine whether the requests are linkable with each other, and whether shows are linkable with each other or with requests. Generally, $\mathcal{F}_{\mathrm{AC}}$ does not impose requirements on the value of the pseudonyms, thereby providing optional unlinkability. The only exception is that requests for the same attribute and statement by the same user to

different authorities need to be linkable. This linkability is established by using the same request identifier *reqid*.

6 Building Blocks of $\Pi_{\mathbf{AC}}$

6.1 Ideal Functionality $\mathcal{F}_{\mathbf{KG}}$

In Coconut, key generation involves creating a key pair for the Pointcheval-Sanders signature scheme in such a way that the shares of the secret key are given to n authorities, so that $t \leq n$ authorities are needed to produce a signature. Key generation could be conducted by either a trusted party, which computes the keys and gives each authority its share, or via a distributed key generation protocol [22, 23], which avoids the need of a trusted party. In Π_{AC}, the authorities obtain their secret keys through the ideal functionality for key generation $\mathcal{F}_{\mathrm{KG}}$ in Figure 3. Alternatively, we could replace $\mathcal{F}_{\mathrm{KG}}$ by an ideal functionality for distributed key generation. With that replacement, Π_{AC} would realize a modified version of $\mathcal{F}_{\mathrm{AC}}$ where authorities cannot finalize the execution of the setup phase without involvement of other authorities, as discussed in Section 5.

$\mathcal{F}_{\mathrm{KG}}$ interacts with n authorities $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$. $\mathcal{F}_{\mathrm{KG}}$ consists of two interfaces $\mathsf{kg.getkey}$ and $\mathsf{kg.retrieve}$. The interface $\mathsf{kg.getkey}$ is used by $\mathcal{V}_i$ to obtain its public key pk_i and secret key sk_i, as well as the public key pk. The interface $\mathsf{kg.retrieve}$ is used by any party $\mathcal{P}$ to obtain the public key pk and the public keys $\langle pk_i \rangle_{i=1}^{n}$ of each of the authorities.

6.2 Ideal Functionality $\mathcal{F}_{\mathbf{NYM}}$

Π_{AC} uses the functionality $\mathcal{F}_{\mathrm{NYM}}$ for a secure idealized pseudonymous channel. We use $\mathcal{F}_{\mathrm{NYM}}$ to describe Π_{AC} for simplicity, in order to hide the details of real-world pseudonymous channels. $\mathcal{F}_{\mathrm{NYM}}$ is similar to the functionality for anonymous secure message transmission in [24]. $\mathcal{F}_{\mathrm{NYM}}$ interacts with senders $\mathcal{T}$ and receivers $\mathcal{R}$. $\mathcal{F}_{\mathrm{NYM}}$ is parameterized by a message space $\mathcal{M}$, a security parameter k, a universe of pseudonyms $\mathbb{U}_p$, and a leakage function l, which leaks the message length. $\mathcal{F}_{\mathrm{NYM}}$ consists of two interfaces $\mathsf{nym.send}$ and $\mathsf{nym.reply}$.

1. $\mathcal{T}$ uses the $\mathsf{nym.send}$ interface to send a message $m \in \mathcal{M}$, a pseudonym $P \in \mathbb{U}_p$ and a receiver identifier $\mathcal{R}$ to $\mathcal{F}_{\mathrm{NYM}}$. $\mathcal{F}_{\mathrm{NYM}}$ sends $l(m)$ to the simulator $\mathcal{S}$. After receiving a response from $\mathcal{S}$, $\mathcal{F}_{\mathrm{NYM}}$ creates a send identifier *sendid* and sends m, P and *sendid* to $\mathcal{R}$.
2. $\mathcal{R}$ uses the $\mathsf{nym.reply}$ interface to send a message $m \in \mathcal{M}$ and a send identifier *sendid* to $\mathcal{F}_{\mathrm{NYM}}$. $\mathcal{F}_{\mathrm{NYM}}$ checks if there is a reply pending for send identifier *sendid*. In that case, $\mathcal{F}_{\mathrm{NYM}}$ sends $l(m)$ to $\mathcal{S}$. After receiving a response from $\mathcal{S}$, $\mathcal{F}_{\mathrm{NYM}}$ sends m and P to $\mathcal{T}$.

$\mathcal{R}$ does not learn the identifier $\mathcal{T}$. $\mathcal{R}$ learns a pseudonym P chosen by $\mathcal{T}$ and can reply to messages received from $\mathcal{T}$. $\mathcal{T}$ can choose different pseudonyms to make the messages sent unlinkable towards $\mathcal{R}$.

Functionality $\mathcal{F}_{\mathrm{KG}}$

$\mathcal{F}_{\mathrm{KG}}$ is parameterized by a probabilistic algorithm KeyGen and a security parameter 1^k and a maximum number q of messages to be signed.

1. On input $(\mathsf{kg.getkey.ini}, sid)$ from an authority $\mathcal{V}_i$:
   - Abort if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$ or if $\mathcal{V}_i \notin sid$.
   - If $(sid, pk, \langle pk_i, sk_i \rangle_{i=1}^n)$ is not stored, run algorithm $(pk, \langle pk_i, sk_i \rangle_{i=1}^n) \leftarrow \mathsf{KeyGen}(1^k, q)$ and store $(sid, pk, \langle pk_i, sk_i \rangle_{i=1}^n)$.
   - Create a fresh qid and store $(qid, \mathcal{V}_i)$.
   - Send $(\mathsf{kg.getkey.sim}, sid, qid, pk, \langle pk_i \rangle_{i=1}^n)$ to $\mathcal{S}$.

S. On input $(\mathsf{kg.getkey.rep}, sid, qid)$ from the simulator $\mathcal{S}$:
   - Abort if $(qid, \mathcal{V}_i)$ such that $qid \neq qid'$ is not stored.
   - Delete $(qid, \mathcal{V}_i)$.
   - Send $(\mathsf{kg.getkey.end}, sid, pk, pk_i, sk_i)$ to $\mathcal{V}_i$.

2. On input $(\mathsf{kg.retrieve.ini}, sid)$ from any party $\mathcal{P}$:
   - Abort if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$.
   - If $(sid, pk, \langle pk_i, sk_i \rangle_{i=1}^n)$ is stored, set $v \leftarrow (pk, \langle pk_i \rangle_{i=1}^n)$, else set $v \leftarrow \perp$.
   - Create a fresh qid and store $(qid, \mathcal{P}, v)$.
   - Send $(\mathsf{kg.retrieve.sim}, sid, qid, v)$ to $\mathcal{S}$.

S. On input $(\mathsf{kg.retrieve.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', \mathcal{P}, v)$ such that $qid' \neq qid$ is not stored.
   - Delete the record $(qid, \mathcal{P}, v)$.
   - Send $(\mathsf{kg.retrieve.end}, sid, v)$ to $\mathcal{P}$.

Fig. 3. Description of functionality $\mathcal{F}_{\mathrm{KG}}$

Functionality $\mathcal{F}_{\text{NYM}}$

$\mathcal{F}_{\text{NYM}}$ is parameterized by a message space $\mathcal{M}$, a security parameter k, a universe of pseudonyms $\mathbb{U}_p$, and a leakage function l, which leaks the message length.

1. On input $(\mathsf{nym.send.ini}, sid, m, P, \mathcal{R})$ from $\mathcal{T}$:
   - Abort if $m \notin \mathcal{M}$, or if $P \notin \mathbb{U}_p$.
   - Create a fresh qid and store $(qid, P, \mathcal{T}, m, \mathcal{R})$.
   - Send $(\mathsf{nym.send.sim}, sid, qid, l(m))$ to $\mathcal{S}$.

S. On input $(\mathsf{nym.send.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', P, \mathcal{T}, m, \mathcal{R})$ such that $qid = qid'$ is not stored.
   - Create random fresh $sendid \leftarrow [0,1]^k$ and store $(sid, \mathcal{T}, P, sendid)$.
   - Delete the record $(qid, P, \mathcal{T}, m, \mathcal{R})$.
   - Send $(\mathsf{nym.send.end}, sid, m, P, sendid)$ to $\mathcal{R}$.

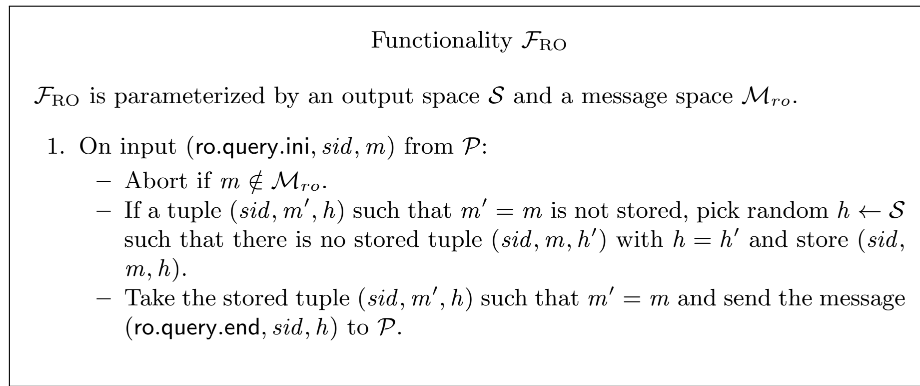
2. On input $(\mathsf{nym.reply.ini}, sid, m, sendid)$ from $\mathcal{R}$:
   - Abort if $m \notin \mathcal{M}$.
   - Abort if there is not a tuple $(sid, \mathcal{T}, P, sendid')$ stored such that $sendid = sendid'$.
   - Create a fresh qid and store $(qid, P, \mathcal{T}, m)$.
   - Delete the tuple $(sid, \mathcal{T}, P, sendid)$.
   - Send $(\mathsf{nym.reply.sim}, sid, qid, l(m))$ to $\mathcal{S}$.

S. On input $(\mathsf{nym.reply.rep}, sid, qid)$ from $\mathcal{S}$:
   - Abort if $(qid', P, \mathcal{T}, m)$ such that $qid = qid'$ is not stored.
   - Delete the record $(qid, P, \mathcal{T}, m)$.
   - Send $(\mathsf{nym.reply.end}, sid, m, P)$ to $\mathcal{T}$.

Fig. 4. Description of functionality $\mathcal{F}_{\text{NYM}}$

6.3 Ideal Functionality $\mathcal{F}_{\mathrm{RO}}$

Π_{AC} uses the functionality $\mathcal{F}_{\mathrm{RO}}$ for a random oracle [25]. $\mathcal{F}_{\mathrm{RO}}$ models an idealized hash function. $\mathcal{F}_{\mathrm{RO}}$ interacts with parties $\mathcal{P}$ and is parameterized by an output space $\mathcal{S}$ and a message space $\mathcal{M}_{ro}$. $\mathcal{F}_{\mathrm{RO}}$ consists of an interface $\mathsf{ro.query}$. A party $\mathcal{P}$ uses the $\mathsf{ro.query}$ interface on input a message m. $\mathcal{F}_{\mathrm{RO}}$ checks whether m has been received before. If that is not the case, $\mathcal{F}_{\mathrm{RO}}$ generates a random response h, records (m, h) and sends h to $\mathcal{P}$. Otherwise $\mathcal{F}_{\mathrm{RO}}$ sends the recorded h to $\mathcal{P}$.



Functionality $\mathcal{F}_{\mathrm{RO}}$

$\mathcal{F}_{\mathrm{RO}}$ is parameterized by an output space $\mathcal{S}$ and a message space $\mathcal{M}_{ro}$.

1. On input $(\mathsf{ro.query.ini}, sid, m)$ from $\mathcal{P}$:
   - Abort if $m \notin \mathcal{M}_{ro}$.
   - If a tuple (sid, m', h) such that $m' = m$ is not stored, pick random $h \leftarrow \mathcal{S}$ such that there is no stored tuple (sid, m, h') with $h = h'$ and store (sid, m, h).
   - Take the stored tuple (sid, m', h) such that $m' = m$ and send the message $(\mathsf{ro.query.end}, sid, h)$ to $\mathcal{P}$.

Fig. 5. Description of functionality $\mathcal{F}_{\mathrm{RO}}$

6.4 Pointcheval-Sanders Signatures in the Random Oracle Model

In the Pointcheval-Sanders signature scheme (see Section 2.5), to compute a signature, the signer picks a random generator $h \in \mathbb{G}$. In Coconut, this random generator is computed by evaluating a hash function on input a commitment to the messages to be signed (see Section 3). This is necessary so that every authority uses the same random generator h to compute a signature share, which allows the user to aggregate those shares.

Construction Π_{AC} uses the same idea as Coconut. In order to formalize this idea and analyze the security of Π_{AC}, we describe a modified version of Pointcheval-Sanders signatures in the random oracle model.

First, we modify the syntax of algorithm Sign as follows. Sign uses a random oracle $H : \mathcal{M}_{ro} \to \mathcal{S}$. $\mathsf{Sign}(sk, m_1, \ldots m_q, r, st)$ receives as input a secret key sk, a tuple of messages $(m_1, \ldots, m_q)$, a value $r \in \mathcal{M}_{ro}$ and state information st, which stores tuples of the form $(m_1, \ldots, m_q, r)$. Sign outputs a signature σ on $(m_1, \ldots, m_q)$ if st does not contain a tuple $(m'_1, \ldots, m'_q, r')$ such that $(m_1, \ldots, m_q) \neq (m'_1, \ldots, m'_q)$ and $r = r'$. Sign also outputs updated state information st'.

The modified Pointcheval-Sanders signature scheme works as follows. Algorithms KeyGen and VfSig remain unmodified.

$\mathsf{Sign}(sk, m_1, \ldots, m_q, r, st)$**.** Parse sk as $(\theta, x, y_1, \ldots, y_q)$. If st contains a tuple $(m'_1, \ldots, m'_q, r')$ such that $(m_1, \ldots, m_q) \neq (m'_1, \ldots, m'_q)$ and $r = r'$, output $\sigma = \perp$ and $st' = st$. Otherwise compute $h \leftarrow H(r)$ and output the signature $\sigma = (h, s) \leftarrow (h, h^{x+y_1 m_1+\ldots+y_q m_q})$ and the updated state information $st' = st \cup \{(m_1, \ldots, m_q, r)\}$.

Definition 4 (Existential Unforgeability in the RO). *For any ppt adversary $\mathcal{A}$, existential unforgeability is the RO is defined as follows.*

$$Pr\left[\begin{array}{l}(sk, pk) \leftarrow \mathsf{KeyGen}(1^k);\ (m, \sigma) \leftarrow \mathcal{A}(pk)^{\mathcal{O}_s(sk,\cdot,\cdot), H(\cdot)} : \\ 1 = \mathsf{VfSig}(pk, \sigma, m)\ \wedge\ m \in \mathcal{M}\ \wedge\ m \notin S_s\end{array}\right] \leq \epsilon(k)$$

$\mathcal{O}_s(sk, \cdot, \cdot)$ works as follows. On input sk, a message $m = (m_1, \ldots, m_q)$ and the value r, $\mathcal{O}_s$ runs $(\sigma, st') \leftarrow \mathsf{Sign}(sk, m_1, \ldots, m_q, r, st)$. $\mathcal{O}_s$ replaces st by st' and returns (σ, st') to $\mathcal{A}$. (st is empty in the first invocation of $\mathcal{O}_s$.) S_s is a set that contains the messages sent to $\mathcal{O}_s$.

In comparison to the definition of existential unforgeability (see Definition 11), $\mathcal{A}$ has access to the random oracle H, and the signing oracle is modified to follow the new syntax.

The modified Pointcheval-Sanders scheme is existentially unforgeable under Assumption 2 (see Definition 2), like the original scheme. It suffices to observe that the value h output by the random oracle is random, and that it is different for each signed message tuple $(m_1, \ldots, m_q)$. The latter is ensured by checking in st that the input to H is different for different message tuples.

In Coconut and in Π_{AC}, the authorities that sign messages do not receive as input the messages to be signed and thus cannot keep state information st. However, H receives as input a commitment to a tuple of messages, and the binding property of the commitment scheme ensures that users are not able to open a commitment to two different tuples of messages, which ensures that the value h is different for each tuple of messages signed (see the security analysis in Section 8.3).

7 Construction Π_{AC}

We describe construction Π_{AC} for $\mathcal{F}_{\mathrm{AC}}$. Π_{AC} runs with authorities $(\mathcal{V}_1, \ldots, \mathcal{V}_n)$, any number of users $\mathcal{U}_j$ and any number of providers $\mathcal{P}_k$. Π_{AC} is parameterized by a threshold t, a security parameter k, a universe of attributes $\mathbb{U}_a = \mathbb{Z}_p^q$, a universe of statements for issuance $\mathbb{U}_\phi$ and a universe of statements for shows $\mathbb{U}_\varphi$ that are CNF/DNF formulae for proofs of knowledge of discrete logarithms and representations, a universe of pseudonyms $\mathbb{U}_p$, and an algorithm Find.

Π_{AC} uses the ideal functionalities $\mathcal{F}_{\mathrm{NYM}}$, $\mathcal{F}_{\mathrm{KG}}$ and $\mathcal{F}_{\mathrm{RO}}$. It also uses the Pedersen commitment scheme. Π_{AC} uses the signature scheme by Pointcheval and Sanders [13], and non-interactive zero-knowledge proofs of knowledge computed via the Fiat-Shamir heuristic. We describe those proofs by using the notation in Section 2.2. We remark that the computation and verification of those proofs involve calls to $\mathcal{F}_{\mathrm{RO}}$, but they are not depicted in our description of Π_{AC}.

1. On input $(\mathsf{ac.setup.ini}, sid)$, $\mathcal{V}_i$ does the following:
   - Abort if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$, or if $\mathcal{V}_i \notin sid$, or if $n < t$.
   - Send $(\mathsf{kg.getkey.ini}, sid)$ to $\mathcal{F}_{\mathrm{KG}}$. $\mathcal{F}_{\mathrm{KG}}$ executes an algorithm $(pk, \langle pk_i, sk_i \rangle_{i=1}^{n}) \leftarrow \mathsf{KeyGen}(1^k, q)$, which works as follows:
     - Run $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}) \leftarrow \mathcal{G}(1^k)$.
     - Pick q random generators $(h_1, \ldots, h_q) \leftarrow \mathbb{G}$.
     - Set the parameters $par \leftarrow (p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, h_1, \ldots, h_q)$.
     - Choose $(q+1)$ polynomials $(v, w_1, \ldots, w_q)$ of degree $(t-1)$ with random coefficients in $\mathbb{Z}_p$.
     - Set $(x, y_1, \ldots, y_q) \leftarrow (v(0), w_1(0), \ldots, w_q(0))$.
     - For $i = 1$ to n, set the secret key sk_i of each authority $\mathcal{V}_i$ as $sk_i = (x_i, y_{i,1}, \ldots, y_{i,q}) \leftarrow (v(i), w_1(i), \ldots, w_q(i))$.
     - For $i = 1$ to n, set the verification key pk_i of each authority $\mathcal{V}_i$ as $pk_i = (\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \ldots, \beta_{i,q}, \tilde{\beta}_{i,q}) \leftarrow (\tilde{g}^{x_i}, g^{y_{i,1}}, \tilde{g}^{y_{i,1}}, \ldots, g^{y_{i,q}}, \tilde{g}^{y_{i,q}})$. (In comparison to Coconut, Π_{AC} extends the public key to provide an issuance protocol with unconditional privacy for the users and to make the protocol provably unforgeable.)
     - Compute the verification key $pk = (par, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \ldots, \beta_q, \tilde{\beta}_q) \leftarrow (par, \tilde{g}^{x}, g^{y_1}, \tilde{g}^{y_1}, \ldots, g^{y_q}, \tilde{g}^{y_q})$.
     - Output $(pk, \langle pk_i, sk_i \rangle_{i=1}^{n})$.
   - Receive $(\mathsf{kg.getkey.end}, sid, pk, pk_i, sk_i)$ from $\mathcal{F}_{\mathrm{KG}}$.
   - Store (sid, pk, pk_i, sk_i).
   - Output $(\mathsf{ac.setup.end}, sid)$.
2. On input $(\mathsf{ac.request.ini}, sid, a, \phi, \mathcal{V}_i, P)$, user $\mathcal{U}_j$ and authority $\mathcal{V}_i$ do the following:
   - $\mathcal{U}_j$ aborts if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$ or if $\mathcal{V}_i \notin sid$.
   - Abort if $a \notin \mathbb{U}_a$, or if $\phi \notin \mathbb{U}_\phi$, or if $P \notin \mathbb{U}_p$.
   - Abort if $\phi(a) \neq 1$.
   - If $(sid, a', o_1, \ldots, o_q, P, \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi', reqid)$ such that $a' = a$ and $\phi' = \phi$ is not stored, do the following:
     - If $(sid, pk, \langle pk_i \rangle_{i=1}^{n})$ is not stored, do the following:
       * Send $(\mathsf{kg.retrieve.ini}, sid)$ to $\mathcal{F}_{\mathrm{KG}}$.
       * Receive $(\mathsf{kg.retrieve.end}, sid, v)$ from $\mathcal{F}_{\mathrm{KG}}$.
       * Abort if $v = \perp$.
       * Parse v as $(pk, \langle pk_i \rangle_{i=1}^{n})$ and store $(sid, pk, \langle pk_i \rangle_{i=1}^{n})$.
     - Parse pk as $(par, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \ldots, \beta_q, \tilde{\beta}_q)$ and par as $(p, \mathbb{G}, \tilde{\mathbb{G}}, \mathbb{G}_t, e, g, \tilde{g}, h_1, \ldots, h_q)$.
     - Parse a as $(m_1, \ldots, m_q)$. Pick random $open = o \leftarrow \mathbb{Z}_p$ and compute $com = g^{o} \prod_{j=1}^{q} h_j^{m_j}$.
     - Send $(\mathsf{ro.query.ini}, sid, com)$ to $\mathcal{F}_{\mathrm{RO}}$ and receive $(\mathsf{ro.query.end}, sid, h)$ from $\mathcal{F}_{\mathrm{RO}}$.
     - Compute commitments to each of the messages. For $j = 1$ to q, pick random $o_j \leftarrow \mathbb{Z}_p$ and set $com_j = g^{o_j} h^{m_j}$. (These commitments replace the ElGamal ciphertexts used in Coconut to provide unconditional privacy for the users.)

- Compute a ZK argument of knowledge π_s via the Fiat-Shamir heuristic for the following relation:

$$\begin{aligned}\pi_s = &\text{NIZK}\{(m_1,\ldots,m_q,o,o_1,\ldots,o_q):\\ &com = g^o \prod_{j=1}^{q} h_j^{m_j} \wedge\\ &\{com_j = g^{o_j} h^{m_j}\}_{\forall j\in[1,q]} \wedge\\ &\phi(m_1,\ldots,m_q)=1\}\end{aligned}$$

- Pick random fresh $reqid \leftarrow [0,1]^k$.
- Store $(sid, a, o_1, \ldots, o_q, P, \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid)$.

– Store $(sid, reqid, \mathcal{V}_i)$.
– Send (nym.send.ini, $sid, \langle \Lambda, \phi, reqid\rangle, P, \mathcal{V}_i$) to $\mathcal{F}_{\mathrm{NYM}}$. (We recall that P is a pseudonym.)
– $\mathcal{V}_i$ receives (nym.send.end, $sid, \langle \Lambda, \phi, reqid\rangle, P, sendid$) from $\mathcal{F}_{\mathrm{NYM}}$.
– Abort if (sid, pk, pk_i, sk_i) is not stored.
– Send (ro.query.ini, sid, com) to $\mathcal{F}_{\mathrm{RO}}$ and receive (ro.query.end, sid, h) from $\mathcal{F}_{\mathrm{RO}}$. Abort if $h \neq h'$.
– Verify π_s by using ϕ, pk and $(com, com_1, \ldots, com_q, h)$. Abort if the proof π_s is not correct.
– Store $(sid, P, reqid, com, com_1, \ldots, com_q, h, sendid)$.
– Output (ac.request.end, $sid, \phi, P, reqid$).

3. On input (ac.issue.ini, $sid, reqid$), authority $\mathcal{V}_i$ and user $\mathcal{U}_j$ do the following:
– $\mathcal{V}_i$ aborts if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$ or if $\mathcal{V}_i \notin sid$.
– Abort if a tuple $(sid, P, reqid', com, com_1, \ldots, com_q, h, sendid)$ such that $reqid' = reqid$ is not stored.
– Abort if (sid, pk, pk_i, sk_i) is not stored.
– Parse sk_i as $(x_i, y_{i,1}, \ldots, y_{i,q})$.
– Compute $c = h^{x_i} \prod_{j=1}^{q} com_j^{y_{i,j}}$.
– Set the blinded signature share $\hat{\sigma}_i = (h, c)$.
– Send (nym.reply.ini, $sid, \langle \mathcal{V}_i, \hat{\sigma}_i, reqid\rangle, sendid$) to $\mathcal{F}_{\mathrm{NYM}}$.
– $\mathcal{U}_j$ receives (nym.reply.end, $sid, \langle \mathcal{V}_i, \hat{\sigma}_i, reqid\rangle, P$) from $\mathcal{F}_{\mathrm{NYM}}$.
– Abort if $(sid, reqid', \mathcal{V}_i')$ such that $reqid = reqid'$ and $\mathcal{V}_i = \mathcal{V}_i'$ is not stored.
– Take the tuple $(sid, a, o_1, \ldots, o_q, P, \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid')$ such that $reqid = reqid'$.
– Parse $\hat{\sigma}_i$ as (h', c). Abort if $h \neq h'$.
– Parse pk_i as $(\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \ldots, \beta_{i,q}, \tilde{\beta}_{i,q})$.
– Compute $\sigma_i = (h, s_i) \leftarrow (h, c \prod_{j=1}^{q} \beta_{i,j}^{-o_j})$.
– Parse a as $(m_1, \ldots, m_q)$. Abort if $e(h, \tilde{\alpha}_i \prod_{j=1}^{q} \tilde{\beta}_{i,j}^{m_j}) = e(s_i, \tilde{g})$ does not hold.
– If a tuple (sid, DB) is not stored, store (sid, DB), where DB is a (initially empty) database with entries of the form $[reqid, a, \mathbb{V}, \sigma]$.
– If there is not an entry $[reqid', a, \mathbb{V}, \sigma]$ in DB such that $reqid' = reqid$, add an entry $[reqid, a, \emptyset, \perp]$ to DB.

  - Replace the entry $[reqid, a, \mathbb{V}, \sigma]$ in DB by $[reqid, a, \mathbb{V} \cup \{\mathcal{V}_i, \sigma_i\}, \sigma]$. If $\sigma = \perp$ and $|\mathbb{V}| \geq t$, do the following:
    - Let $\mathbb{S} \in [1, n]$ be a set of t indices of authorities in $\mathbb{V}$.
    - For all $i \in \mathbb{S}$, evaluate at 0 the Lagrange basis polynomials

$$l_i = [\prod_{j \in \mathbb{S}, j \neq i} (0 - j)][\prod_{j \in \mathbb{S}, j \neq i} (i - j)]^{-1} \bmod p$$

    - For all $i \in \mathbb{S}$, take $\sigma_i = (h, s_i)$ from $\mathbb{V}$ and compute the signature $\sigma = (h, s) \leftarrow (h, \prod_{i \in \mathbb{S}} s_i^{l_i})$.
    - Parse pk as $pk = (par, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \ldots, \beta_q, \tilde{\beta}_q)$ and abort if the equality $e(h, \tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_j}) = e(s, \tilde{g})$ does not hold, else replace $[reqid, a, \mathbb{V}, \perp]$ by $[reqid, a, \mathbb{V}, \sigma]$.
  - Output $(\mathsf{ac.issue.end}, sid, a, \phi, \mathcal{V}_i)$.
4. On input $(\mathsf{ac.show.ini}, sid, \varphi, P, \mathcal{P}_k)$, user $\mathcal{U}_j$ and provider $\mathcal{P}_k$ do the following:
  - $\mathcal{U}_j$ aborts if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$.
  - Abort if $\varphi \notin \mathbb{U}_\varphi$, or if $P \notin \mathbb{U}_p$.
  - Abort if a tuple (sid, DB) is not stored.
  - Evaluate $\mathbb{A} \leftarrow \mathsf{Find}(\varphi, \mathsf{DB})$. The algorithm Find outputs a set $\mathbb{A} = [a_l, \sigma_l]_{l=1}^{L}$ of attributes-signature pairs such that the attributes satisfy φ.
  - Abort if $\mathbb{A} = \emptyset$.
  - For $l = 1$ to L, do the following:
    - Parse σ_l as (h_l, s_l).
    - Retrieve the stored tuple $(sid, pk, \langle pk_i \rangle_{i=1}^{n})$ and parse pk as $(par, \tilde{\alpha}, \beta_1, \tilde{\beta}_1, \ldots, \beta_q, \tilde{\beta}_q)$.
    - Pick random $r_l \leftarrow \mathbb{Z}_p$ and $r'_l \leftarrow \mathbb{Z}_p$.
    - Compute $\sigma'_l = (h'_l, s'_l) \leftarrow (h_l^{r'_l}, s_l^{r'_l} (h'_l)^{r_l})$. (The value $\nu = (h'_l)^{r_l}$ in Coconut is not present in Π_{AC}. Instead, it is used to compute s'_l. This modification allows the design of a show protocol that provides unconditional privacy for the users.)
    - Parse a_l as $(m_{l,1}, \ldots, m_{l,q})$. Compute $\kappa_l \leftarrow \tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_{l,j}} \tilde{g}^{r_l}$.
  - Compute the ZK argument of knowledge π_v via the Fiat-Shamir heuristic.

$$\begin{aligned} \pi_v = &\mathrm{NIZK}\{(\langle m_{l,1}, \ldots, m_{l,q}, r_l \rangle_{l=1}^{L}) : \\ &\{\kappa_l = \tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_{l,j}} \tilde{g}^{r_l}\}_{l=1}^{L} \wedge \\ &\varphi(\langle m_{l,1}, \ldots, m_{l,q} \rangle_{l=1}^{L}) = 1\} \end{aligned}$$

  - Send $(\mathsf{nym.send.ini}, sid, \langle \{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi \rangle, P, \mathcal{P}_k)$ to $\mathcal{F}_{\mathrm{NYM}}$.
  - $\mathcal{P}_k$ receives $(\mathsf{nym.send.end}, sid, \langle \{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi \rangle, P, sendid)$ from $\mathcal{F}_{\mathrm{NYM}}$.
  - $\mathcal{P}_k$ aborts if $sid \neq (\mathcal{V}_1, \ldots, \mathcal{V}_n, sid')$.
  - If (sid, pk) is not stored, do the following:

  - Send $(\mathsf{kg.retrieve.ini}, sid)$ to $\mathcal{F}_{\mathrm{KG}}$.
  - Receive $(\mathsf{kg.retrieve.end}, sid, v)$ from $\mathcal{F}_{\mathrm{KG}}$.
  - Abort if $v = \perp$.
  - Parse v as $(pk, \langle pk_i \rangle_{i=1}^{n})$ and store (sid, pk).
- For $l = 1$ to L, parse σ'_l as (h'_l, s'_l) and abort if $h'_l = 1$ or if $e(h'_l, \kappa_l) = e(s'_l, \tilde{g})$ does not hold.
- Verify π_v by using φ, pk and $\{\kappa_l\}_{l=1}^{L}$. Abort if the proof is not correct.
- Output $(\mathsf{ac.show.end}, \varphi, P)$.

In the following, we explain the interfaces of Π_{AC}.

1. In the ac.setup interface, an authority $\mathcal{V}_i$ queries $\mathcal{F}_{\mathrm{KG}}$ to obtain a key pk for signature verification, and (pk_i, sk_i), which is a signing key pair for authority $\mathcal{V}_i$.
   $\mathcal{F}_{\mathrm{KG}}$ computes the keys. The main idea is that $\mathcal{F}_{\mathrm{KG}}$ needs to generate the keys (pk_i, sk_i) of authorities in such a way that, if a user possesses at least $t \leq n$ signatures σ_i from t different authorities $\mathcal{V}_i$ (each authority uses his key sk_i to compute the signature σ_i), then those signatures σ_i can be aggregated by the user to obtain a signature σ that can be verified with the key pk.
   Π_{AC} considers attributes a that consist of at most q fields $(m_1, \ldots, m_q)$. In the Pointcheval-Sanders signature scheme, a secret key for signing q messages consists of $q + 1$ elements $(x_i, y_{i,1}, \ldots, y_{i,q})$ in $\mathbb{Z}_p$.
   $\mathcal{F}_{\mathrm{KG}}$ uses $q + 1$ random polynomials of degree $t - 1$ to generate the keys. The key pk_i of each authority $\mathcal{V}_i$ is the evaluation of those $q + 1$ polynomials on input i, while the secret key sk corresponding to the verification key pk is the evaluation of those polynomials on input 0. Thanks to polynomial interpolation, it is possible to reconstruct a polynomial of degree $t - 1$ with t evaluations of that polynomial. Therefore, t secret keys sk_i are enough to compute the key sk corresponding to pk.
   In the ac.issue interface, the user first obtains t signatures σ_i computed on input sk_i by t different authorities $\mathcal{V}_i$. Then the user uses polynomial interpolation in order to aggregate those t signatures σ_i into a signature σ that is verifiable with the verification key pk, i.e., as if σ was signed with sk.
2. In the ac.request interface, the user $\mathcal{U}_j$ sends a request to an authority $\mathcal{V}_i$ in order to obtain a signature σ_i on an attribute $a = (m_1, \ldots, m_q)$. $\mathcal{U}_j$ would need to run the ac.request interface at least t times with t different authorities to be able to obtain a signature σ on $(m_1, \ldots, m_q)$ verifiable with the key pk.
   In each request, $\mathcal{U}_j$ may receive as input a description ϕ of statements that the attribute $(m_1, \ldots, m_q)$ must fulfill, which we denote as $\phi(m_1, \ldots, m_q) = 1$. (If $\phi = \perp$, then no statements need to be proven.) Intuitively, $\mathcal{U}_j$ needs to compute a request in such a way that $\mathcal{V}_i$ will be able to sign $(m_1, \ldots, m_q)$ without learning $(m_1, \ldots, m_q)$ and, if required, to verify that $(m_1, \ldots, m_q)$ fulfills $\phi(m_1, \ldots, m_q) = 1$. $\mathcal{U}_j$ proceeds as follows:
   - $\mathcal{U}_j$ computes a commitment com to $(m_1, \ldots, m_q)$ using a random opening $open$. The hiding property ensures that com does not reveal any information about $(m_1, \ldots, m_q)$. The binding property ensures that com cannot be opened to a different message.

- $\mathcal{U}_j$ queries the random oracle H on input com to obtain an output h. The reason why this is done is the following. In the Pointcheval-Sanders signature scheme [13], computing a signature on a message requires the signer to generate a random value h that is part of the signature. In Π_{AC}, $\mathcal{U}_j$ needs that each authority produces a signature σ_i on $(m_1, \ldots, m_q)$ signed with his secret key sk_i. After obtaining t signatures from t authorities, $\mathcal{U}_j$ needs to aggregate those signatures σ_i to get a signature σ verifiable with verification key pk. For that aggregation to be possible, it is necessary that all the authorities use the same random value h to compute σ_i. However, it is undesirable that authorities need to communicate with each other to agree on a common random value. To solve this issue, a random oracle is used instead.

  Therefore, requests for the same attribute to different authorities need to use the same commitment com, and thus they are linkable. $\mathcal{F}_{\text{AC}}$ allows this linkability.
- $\mathcal{U}_j$ computes q commitments to the messages $(m_1, \ldots, m_q)$. The hiding property ensures that authorities do not learn any information about the committed message beyond its length. The reason why $\mathcal{U}_j$ computes these commitments is the following. $\mathcal{U}_j$ needs that each authority computes a signature σ_i on $(m_1, \ldots, m_q)$ without learning $(m_1, \ldots, m_q)$. A Pointcheval-Sanders signature has the form $(h, h^{x_i + y_{1,i} m_1 + \ldots + y_{q,i} m_q})$. To allow an authority to sign $(m_1, \ldots, m_q)$, $\mathcal{U}_j$ commits to m_i, for $i \in [1, q]$, and uses h as part of the commitment parameters. Then the authority will, roughly speaking, sign the committed messages.
- $\mathcal{U}_j$ sends com and $(com_1, \ldots, com_q)$ to $\mathcal{V}_i$, along with a non-interactive zero-knowledge proof of knowledge π_s. π_s proves that com and $(com_1, \ldots, com_q)$ are well-formed. It also proves that $(m_1, \ldots, m_q)$ committed in com are the same as the messages committed in $(com_1, \ldots, com_q)$. Finally, it proves that $(m_1, \ldots, m_q)$ satisfy ϕ.

We remark that, if there is already a request computed for attribute a and statement ϕ, $\mathcal{U}_j$ reuses the request to send it to other authorities. We note that $\mathcal{F}_{\text{AC}}$ and Π_{AC} only consider the case in which the user needs to prove the same statement to different authorities to be issued an attribute. Therefore, if ϕ changes, a new request is computed even if the attribute was the same in a previous request. $\mathcal{F}_{\text{AC}}$ and Π_{AC} can easily be modified to consider the case in which users need to prove different statements to different authorities to request the issuance of an attribute.

$\mathcal{V}_i$ receives through $\mathcal{F}_{\text{NYM}}$ a message com, $(com_1, \ldots, com_q)$, h, π_s, ϕ and $reqid$. The request identifier $reqid$ is included so that the user can link this request to the reply sent later by $\mathcal{V}_i$ in the ac.issue interface.

After receiving the message, $\mathcal{V}_i$ recomputes the value h by evaluating the random oracle on input com. $\mathcal{V}_i$ verifies the proof π_s by using ϕ, pk and $(com, com_1, \ldots, com_q, h)$. If the proof is correct, $\mathcal{V}_i$ outputs ϕ, P and $reqid$. The pseudonym P and the request identifier $reqid$ are received from $\mathcal{F}_{\text{NYM}}$. $reqid$ is needed by $\mathcal{V}_i$ to reply to $\mathcal{U}_j$ in the ac.issue interface.

3. In the $\mathsf{ac.issue}$ interface, an authority $\mathcal{V}_i$ receives as input *reqid*. $\mathcal{V}_i$ retrieves the values $(com, com_1, \dots, com_q)$, h and *sendid* associated with the request *reqid*.
   $\mathcal{V}_i$ takes his signing secret key $sk_i = (x_i, y_{1,i}, \dots, y_{q,i})$ and turns $(com_1, \dots, com_q)$ into a value c that can be seen as a commitment to $x_i + y_{1,i}m_1 + \dots + y_{q,i}m_q$. To do that, $\mathcal{V}_i$ uses the homomorphic properties of the commitment scheme.
   $\mathcal{V}_i$ sets $\hat{\sigma}_i = (h, c)$ and sends pk_i, $\hat{\sigma}_i$ and *reqid* to $\mathcal{U}_j$ through $\mathcal{F}_{\mathrm{NYM}}$. $\mathcal{U}_j$ retrieves the data for the request associated with *reqid* and the authority associated with pk_i, which contains the openings $(o_1, \dots, o_j)$ for the commitments. Then $\mathcal{U}_j$ uses the openings to obtain $h^{x_i+y_{1,i}m_1+\dots+y_{q,i}m_q}$ from c. $\mathcal{U}_j$ sets the signature $\sigma_i = (h, h^{x_i+y_{1,i}m_1+\dots+y_{q,i}m_q})$, which is a Pointcheval-Sanders signature on $(m_1, \dots, m_q)$ verifiable by the verification key pk_i.
   If $\mathcal{U}_j$ possesses t signatures σ_i from t different authorities, $\mathcal{U}_j$ aggregates them into a signature σ verifiable with verification key pk. For this purpose, the user uses polynomial interpolation. $\mathcal{U}_j$ verifies σ on input the verification key pk and the signed attribute $(m_1, \dots, m_q)$.
   Finally, $\mathcal{U}_j$ outputs a, ϕ and $\mathcal{V}_i$. This means that $\mathcal{V}_i$ issued the attribute a to $\mathcal{U}_j$.
4. In the $\mathsf{ac.show}$ interface, $\mathcal{U}_j$ receives as input statements φ, a pseudonym P and the identity of a provider $\mathcal{P}_k$. First $\mathcal{U}_j$ runs an algorithm Find to find out whether she has been issued a set of attributes $\mathbb{A}$ that satisfies φ, i.e., $\mathcal{U}_j$ checks whether he possesses signatures σ_l verifiable with pk that sign all the attributes a_l in the set $\mathbb{A}$.
   $\mathcal{U}_j$ then proves to $\mathcal{P}_k$ that she has been issued such a set of attributes and that the attributes satisfy φ. For privacy reasons, $\mathcal{P}_k$ should not be able to link the use of a signature σ_l in two executions of the $\mathsf{ac.show}$ interface, or to the executions of the $\mathsf{ac.issue}$ interface in which the signature was issued. (We remark that $\mathcal{U}_j$ can optionally link those executions by choosing P appropriately.) To this end, $\mathcal{U}_j$ proceeds as follows:
   – $\mathcal{U}_j$ picks random values r'_l, r_l and uses them to compute $\sigma'_l = (h'_l, s'_l)$. σ'_l can be seen as a signature that signs an additional random attribute under a dummy public key $\tilde{g}$.
   – $\mathcal{U}_j$ computes the values κ_l on input the public key pk, the attributes $(m_{l,1}, \dots, m_{l,q})$, and the random values r_l. The reason κ_l are necessary is the following. $\mathcal{U}_j$ sends σ'_l to the provider $\mathcal{P}_k$. However, $\mathcal{U}_j$ does not send the attributes $(m_{l,1}, \dots, m_{l,q})$ to $\mathcal{P}_k$ (to preserve unlinkability). To allow $\mathcal{P}_k$ to verify σ'_l without knowing $(m_{l,1}, \dots, m_{l,q})$, $\mathcal{U}_j$ computes the values κ_l. We note that κ_l is computed on input $(m_{l,1}, \dots, m_{l,q})$, but thanks to using the random value r_l, κ_l does not reveal any information about $(m_{l,1}, \dots, m_{l,q})$.
   – Additionally, $\mathcal{U}_j$ computes a non-interactive zero-knowledge proof of knowledge π_v that proves that the values κ_l are correctly computed. I.e., π_v proves that $\mathcal{U}_j$ knows the attributes $(m_{l,1}, \dots, m_{l,q})$ and that κ_l is well-formed and computed on input $(m_{l,1}, \dots, m_{l,q})$ and r_l. π_v also proves that $(m_{l,1}, \dots, m_{l,q})$ used to compute κ_l fulfill the statements φ.

$\mathcal{U}_j$ sends a message $\langle\{\kappa_l, \sigma_l'\}_{l=1}^{L}, \pi_v, \varphi\rangle$ to $\mathcal{P}_k$ through $\mathcal{F}_{\mathrm{NYM}}$. $\mathcal{P}_k$ verifies the proof π_v. Then $\mathcal{P}_k$ verifies σ_l'. To this end, $\mathcal{P}_k$ uses a modified version of the verification equation of Pointcheval-Sanders signatures in which the signed attributes $(m_{l,1}, \ldots, m_{l,q})$ and the verification key pk are replaced by the values κ_l. This modified verification equation, along with the proof π_v that κ_l are correctly computed, guarantee that the user possesses a valid signature verifiable with pk. We recall that pk is used to compute κ_l. If both the verification of π_v and of each σ_l' are successful, $\mathcal{P}_k$ outputs φ and a pseudonym P received from $\mathcal{F}_{\mathrm{NYM}}$.

8 Security Analysis of Coconut

To prove that construction Π_{AC} securely realizes the ideal functionality $\mathcal{F}_{\mathrm{AC}}$, we have to show that for any environment $\mathcal{Z}$ and any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that $\mathcal{Z}$ cannot distinguish between whether it is interacting with $\mathcal{A}$ and the protocol in the real world or with $\mathcal{S}$ and $\mathcal{F}_{\mathrm{AC}}$. The simulator thereby plays the role of all honest parties in the real world and interacts with $\mathcal{F}_{\mathrm{AC}}$ for all corrupt parties in the ideal world.

$\mathcal{S}$ runs a copy of any adversary $\mathcal{A}$, which is used to provide to $\mathcal{Z}$ a view that is indistinguishable from the view given by $\mathcal{A}$ in the real world. To achieve that, $\mathcal{S}$ must simulate the real-world protocol towards the copy of $\mathcal{A}$, in such a way that $\mathcal{A}$ cannot distinguish an interaction with $\mathcal{S}$ from an interaction with the real-world protocol. $\mathcal{S}$ uses the information provided by $\mathcal{F}_{\mathrm{AC}}$ to provide a simulation of the real-world protocol.

Our simulator $\mathcal{S}$ runs copies of the functionalities $\mathcal{F}_{\mathrm{NYM}}$, $\mathcal{F}_{\mathrm{KG}}$ and $\mathcal{F}_{\mathrm{RO}}$. When any of the copies of these functionalities aborts, $\mathcal{S}$ implicitly forwards the abortion message to the adversary if the functionality sends the abortion message to a corrupt party.

$\mathcal{S}$ also runs copies of the extractors $\mathcal{E}_s$ and $\mathcal{E}_v$ and simulators $\mathcal{S}_s$ and $\mathcal{S}_v$ for the non-interactive zero-knowledge arguments of knowledge π_s and π_v, which are computed through the Fiat-Shamir transform. We remark that they involve calls to the random oracle and rewinding of the adversary. For simplicity, we omit those details.

In Section 8.1, we analyze the security of construction Π_{AC} when up to $t-1$ authorities $\mathcal{V}_i$ are corrupt. In Section 8.2, we analyze the security of construction Π_{AC} when (a subset of) service providers $\mathcal{P}_k$ are corrupt. In Section 8.3, we analyze the security of construction Π_{AC} when (a subset of) users $\mathcal{U}_j$ are corrupt. Finally, in Section 8.4, we analyze the security of Π_{AC} when up to $t-1$ authorities $\mathcal{V}_i$, (a subset of) service providers $\mathcal{P}_k$ and (a subset of) users $\mathcal{U}_j$ are corrupt.

8.1 Security Analysis of Π_{AC} when $\mathcal{V}_i$ is Corrupt

Intuition. When up to $t-1$ authorities $\mathcal{V}_i$ are corrupt, we need to construct a simulator $\mathcal{S}$ that simulates the honest users towards a copy of any adversary that controls the corrupt authorities. $\mathcal{S}$ needs to simulate the request messages that

honest users send to authorities with the information given by $\mathcal{F}_{\mathrm{AC}}$. $\mathcal{F}_{\mathrm{AC}}$ does not disclose to authorities the identity of honest users and the attribute used to compute a request. Therefore, $\mathcal{S}$ must be able to compute request messages that are indistinguishable from those computed by honest users in the real protocol, without knowing the identity of honest users or the attributes.

We show that the honest user identities are hidden from authorities thanks to the use of $\mathcal{F}_{\mathrm{KG}}$ and $\mathcal{F}_{\mathrm{NYM}}$. $\mathcal{F}_{\mathrm{NYM}}$ guarantees that messages sent by users to authorities do not reveal the user identifier. $\mathcal{F}_{\mathrm{KG}}$ guarantees that all users receive the same public keys and public parameters.

We also show that attributes used to compute requests are hidden from authorities thanks to the hiding property of the commitment scheme and the zero-knowledge property of the non-interactive proof of knowledge scheme.

Simulator. We describe the simulator $\mathcal{S}$ for the case in which up to $t-1$ authorities $\mathcal{V}_i$ are corrupt. $\mathcal{S}$ simulates the protocol by running the users' side of protocol Π_{AC} and copies of the ideal functionalities involved.

Honest authority $\mathcal{V}_i$ starts setup. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.setup.sim}, sid, \mathcal{V}_i)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{KG}}$ on input $(\mathsf{kg.getkey.ini}, sid)$. When $\mathcal{F}_{\mathrm{KG}}$ sends the message $(\mathsf{kg.getkey.sim}, sid, qid, pk, \langle pk_i \rangle_{i=1}^n)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest authority $\mathcal{V}_i$ ends setup. When $\mathcal{A}$ sends $(\mathsf{kg.getkey.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on that input. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.getkey.end}, sid, pk, pk_i, sk_i)$, $\mathcal{S}$ sends $(\mathsf{ac.setup.rep}, sid, \mathcal{V}_i)$ to $\mathcal{F}_{\mathrm{AC}}$.

$\mathcal{A}$ requests keys. When a corrupt authority $\tilde{\mathcal{V}}_i$ sends $(\mathsf{kg.getkey.ini}, sid)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.getkey.sim}, sid, qid, pk, \langle pk_i \rangle_{i=1}^n)$, $\mathcal{S}$ forwards that message to $\tilde{\mathcal{V}}_i$.

$\mathcal{A}$ receives keys. When $\tilde{\mathcal{V}}_i$ sends $(\mathsf{kg.getkey.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.getkey.end}, sid, pk, pk_i, sk_i)$, $\mathcal{S}$ sends $(\mathsf{ac.setup.ini}, sid)$ to $\mathcal{F}_{\mathrm{AC}}$. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.setup.sim}, sid, \tilde{\mathcal{V}}_i)$, $\mathcal{S}$ sends $(\mathsf{ac.setup.rep}, sid, \tilde{\mathcal{V}}_i)$ to $\mathcal{F}_{\mathrm{AC}}$. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.setup.end}, sid)$, $\mathcal{S}$ sends $(\mathsf{kg.getkey.end}, sid, pk, pk_i, sk_i)$ to $\tilde{\mathcal{V}}_i$.

Honest user requests keys. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.request.sim}, sid, qid, b)$, if $b = 1$, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input $(\mathsf{kg.retrieve.ini}, sid)$. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.sim}, sid, qid, v)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest user receives keys. When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.end}, sid, v)$, $\mathcal{S}$ proceeds with the "Honest user sends request" item.

Honest user sends request. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.request.sim}, sid, qid, b)$, if $b = 0$ or otherwise if the "Honest user requests keys" item has been run, the simulator $\mathcal{S}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$ to $\mathcal{A}$, where $l(m)$ is equal to the length of the message that the honest user sends to an authority in the $\mathsf{ac.request}$ interface. When $\mathcal{A}$ sends $(\mathsf{nym.send.rep}, sid, qid)$, $\mathcal{S}$ sends $(\mathsf{ac.request.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

$\mathcal{A}$ receives request. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.request.end}, sid, \phi, P, reqid)$ to a corrupt authority $\tilde{\mathcal{V}}_i$, $\mathcal{S}$ runs a copy of a user on input $(\mathsf{ac.request.ini}, sid, a', \phi, \tilde{\mathcal{V}}_i, P)$, where, if the copy of the user stores a tuple $(sid, a, o_1, \ldots, o_q, P, \Lambda =$

$(com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid')$ such that $reqid' = reqid$, then $a' = a$, else a' is a random attribute. (The copy of the user sets $reqid$ to the value received from $\mathcal{F}_{\mathrm{AC}}$.) When running the copy of the user, $\mathcal{S}$ uses the simulator $\mathcal{S}_s$ to compute a simulated proof π_s. When the copy of the user sends the message $(\mathsf{nym.send.ini}, sid, \langle \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid\rangle, P, \tilde{\mathcal{V}}_i)$ to $\mathcal{F}_{\mathrm{NYM}}$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input $(\mathsf{nym.send.rep}, sid, qid)$. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.send.end}, sid, \langle \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid\rangle, P, sendid)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ queries random oracle. When $\mathcal{A}$ sends $(\mathsf{ro.query.ini}, sid, com)$, $\mathcal{S}$ runs the functionality $\mathcal{F}_{\mathrm{RO}}$ on that input. When $\mathcal{F}_{\mathrm{RO}}$ sends $(\mathsf{ro.query.end}, sid, h)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest authority issues attribute. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.issue.sim}, sid, qid)$, $\mathcal{S}$ sends $(\mathsf{nym.reply.sim}, sid, qid, l(m))$ to $\mathcal{A}$, where $l(m)$ is the length of the message that an honest authority sends in the $\mathsf{ac.issue}$ interface.

Honest user receives issuance from honest authority. When the adversary $\mathcal{A}$ sends $(\mathsf{nym.reply.rep}, sid, qid)$, $\mathcal{S}$ sends $(\mathsf{ac.issue.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

$\mathcal{A}$ issues attribute. When a corrupt authority $\tilde{\mathcal{V}}_i$ sends $(\mathsf{nym.reply.ini}, sid, \langle \tilde{\mathcal{V}}_i, \tilde{\sigma}_i, reqid\rangle, sendid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.reply.sim}, sid, qid, l(m))$, $\mathcal{S}$ sends that message to $\mathcal{A}$.

Honest user receives issuance from $\mathcal{A}$. When $\mathcal{A}$ sends $(\mathsf{nym.reply.rep}, sid, qid)$, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When the functionality $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.reply.end}, sid, \langle \tilde{\mathcal{V}}_i, \hat{\sigma}_i, reqid\rangle, P)$, $\mathcal{S}$ runs the copy of the user on input that message. We remark that the copy of the user finds the request identifier $reqid$ associated with this issuance message, or aborts if it is not found. When the copy of the user outputs $(\mathsf{ac.issue.end}, sid, a, \phi, \mathcal{V}_i)$, $\mathcal{S}$ sends $(\mathsf{ac.issue.ini}, sid, reqid)$ to $\mathcal{F}_{\mathrm{AC}}$. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.issue.sim}, sid, qid)$, $\mathcal{S}$ sends $(\mathsf{ac.issue.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

Honest user shows credential. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.show.sim}, sid, qid, b)$, $\mathcal{S}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$ to $\mathcal{A}$, where $l(m)$ is the length of the message $\langle \{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi\rangle$ sent by honest users.

Honest provider receives credential show. When $\mathcal{A}$ sends $(\mathsf{nym.send.rep}, sid, qid)$, if $b = 0$ in the message $(\mathsf{ac.show.sim}, sid, qid, b)$ received from $\mathcal{F}_{\mathrm{AC}}$, $\mathcal{S}$ sends $(\mathsf{ac.show.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$, else $\mathcal{S}$ proceeds with the "Honest provider requests keys" item.

Honest provider requests keys. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.show.sim}, sid, qid, b)$, if $b = 1$, after running the "Honest user shows credential" item, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input $(\mathsf{kg.retrieve.ini}, sid)$. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.sim}, sid, qid, v)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest provider receives keys. When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.end}, sid, v)$, $\mathcal{S}$ sends $(\mathsf{ac.show.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

Theorem 1. *When up to $t-1$ authorities $\mathcal{V}_i$ are corrupt, the construction Π_{AC} securely realizes $\mathcal{F}_{\mathrm{AC}}$ in the $(\mathcal{F}_{\mathrm{KG}}, \mathcal{F}_{\mathrm{NYM}}, \mathcal{F}_{\mathrm{RO}})$-hybrid model if the commitment scheme is hiding and the non-interactive proof of knowledge scheme is zero-knowledge.*

Proof of Theorem 1. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\mathrm{REAL}_{\Pi_{\mathrm{AC}},\mathcal{A},\mathcal{Z}}$ and the ensemble $\mathrm{IDEAL}_{\mathcal{F}_{\mathrm{AC}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by $\Pr[\mathbf{Game}\ i]$ the probability that the environment distinguishes **Game** i from the real-world protocol.

Game 0**:** This game corresponds to the execution of the real-world protocol. Therefore, $\Pr[\mathbf{Game}\ 0] = 0$.

Game 1**:** This game proceeds as **Game** 0, except that in **Game** 1 the non-interactive proofs of knowledge π_s are replaced by simulated proofs computed by the simulator $\mathcal{S}_s$. Under the zero-knowledge property of the proof system (see Definition 5), we have that $|\Pr[\mathbf{Game}\ 1] - \Pr[\mathbf{Game}\ 0]| \leq \mathrm{Adv}^{\mathsf{zk}}_{\mathcal{A}}$.

Game 2**:** This game proceeds as **Game** 1, except that in **Game** 2 the values $(com_1, \ldots, com_q)$ in each request are replaced by random values in $\mathbb{G}$. At this point, the proofs π_s are simulated proofs of false statements. Since the values $(com_1, \ldots, com_q)$ are randomly distributed, this change does not alter the view of the environment and we have that $|\Pr[\mathbf{Game}\ 2] - \Pr[\mathbf{Game}\ 1]| = 0$.

Game 3**:** This game proceeds as **Game** 2, except that in **Game** 3 the commitments to attributes $a = (m_1, \ldots, m_q)$ are replaced by commitments to random messages. Under the hiding property of the commitment scheme (see Definition 7), we have that $|\Pr[\mathbf{Game}\ 3] - \Pr[\mathbf{Game}\ 2]| \leq \mathrm{Adv}^{\mathsf{hid}}_{\mathcal{A}} \cdot N_{req}$, where N_{req} is the number of different requests computed by honest users. Since the Pedersen commitment scheme is information theoretically hiding, we have that $|\Pr[\mathbf{Game}\ 3] - \Pr[\mathbf{Game}\ 2]| = 0$.

Proof sketch. The proof uses a sequence of hybrid games **Game** $2.i$, for $i = 0$ to $N_{req} + 1$. In **Game** $2.i$, the parameters of the commitment scheme par_c are received from the challenger of the hiding game and they are used to set up the values $(g, h_1, \ldots, h_q)$ in the parameters par in $\mathcal{F}_{\mathrm{KG}}$. The commitments for the first $i - 1$ requests are computed on input random messages, whereas the commitments for the requests from $i + 1$ to N_{req} are computed on input the attribute values. The commitment for request i is the challenge commitment computed by the challenger after receiving the correct attribute value. Therefore, **Game** 2.0 is equal to **Game** 2, whereas **Game** $2.(N_{req}+1)$ is equal to **Game** 3. Given the advantage $\mathrm{Adv}^{\mathsf{hid}}_{\mathcal{A}}$ of the adversary against the hiding property, the probability of distinguishing between **Game** $2.i$ and **Game** $2.(i+1)$ is bound by $\mathrm{Adv}^{\mathsf{hid}}_{\mathcal{A}}$. Therefore, the probability of distinguishing between **Game** 2 and **Game** 3 is bound by $\mathrm{Adv}^{\mathsf{hid}}_{\mathcal{A}} \cdot N_{req}$.

The distribution of **Game** 3 is identical to that of our simulation. In **Game** 3, the request of an honest user is computed on input a random attribute instead of the correct attribute received as input from the environment. The overall advantage of the environment to distinguish between the real and the ideal protocol is $|\Pr[\mathbf{Game}\ 3] - \Pr[\mathbf{Game}\ 0]| \leq \mathrm{Adv}^{\mathsf{zk}}_{\mathcal{A}}$. This concludes the proof of Theorem 1.

8.2 Security Analysis of $\Pi_{\mathbf{AC}}$ when $\mathcal{P}_k$ is Corrupt

Intuition. When (a subset of) providers $\mathcal{P}_k$ are corrupt, we need to construct a simulator $\mathcal{S}$ that simulates the honest users towards a copy of any adversary that controls the corrupt providers. $\mathcal{S}$ needs to simulate the credential show messages that honest users send to providers with the information given by $\mathcal{F}_{\mathrm{AC}}$. $\mathcal{F}_{\mathrm{AC}}$ does not disclose to providers the identity of honest users or the attributes used to compute a credential show. Therefore, $\mathcal{S}$ must be able to compute credential show messages that are indistinguishable from those computed by honest users in the real protocol, without knowing the identity of honest users or the attributes.

We show that the honest user identities are hidden from providers thanks to the use of $\mathcal{F}_{\mathrm{KG}}$ and $\mathcal{F}_{\mathrm{NYM}}$. $\mathcal{F}_{\mathrm{NYM}}$ guarantees that messages sent by users to providers do not reveal the user identifier. $\mathcal{F}_{\mathrm{KG}}$ guarantees that all users receive the same public keys and public parameters.

We also show that the attributes used to compute credential show messages are hidden from providers thanks to the zero-knowledge property of the non-interactive proof of knowledge scheme.

Simulator. We describe the simulator $\mathcal{S}$ for the case in which (a subset of) providers $\mathcal{P}_k$ are corrupt. $\mathcal{S}$ simulates the protocol by running the users' side of protocol Π_{AC} and copies of the ideal functionalities involved.

Honest authority $\mathcal{V}_i$ starts setup. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.setup.sim}, sid, \mathcal{V}_i)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{KG}}$ on input $(\mathsf{kg.getkey.ini}, sid)$. When $\mathcal{F}_{\mathrm{KG}}$ sends the message $(\mathsf{kg.getkey.sim}, sid, qid, pk, \langle pk_i \rangle_{i=1}^{n})$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest authority $\mathcal{V}_i$ ends setup. When $\mathcal{A}$ sends $(\mathsf{kg.getkey.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on that input. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.getkey.end}, sid, pk, pk_i, sk_i)$, $\mathcal{S}$ sends $(\mathsf{ac.setup.rep}, sid, \mathcal{V}_i)$ to $\mathcal{F}_{\mathrm{AC}}$.

Honest user requests keys. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.request.sim}, sid, qid, b)$, if $b = 1$, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input $(\mathsf{kg.retrieve.ini}, sid)$. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.sim}, sid, qid, v)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest user receives keys. When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.end}, sid, v)$, $\mathcal{S}$ proceeds with the "Honest user sends request" item.

Honest user sends request. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.request.sim}, sid, qid, b)$, if $b = 0$ or otherwise if the "Honest user requests keys" item has been run, the simulator $\mathcal{S}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$ to $\mathcal{A}$, where $l(m)$ is equal to the length of the message that the honest user sends to an authority in the $\mathsf{ac.request}$ interface.

Honest authority receives request. When $\mathcal{A}$ sends $(\mathsf{nym.send.rep}, sid, qid)$, $\mathcal{S}$ sends $(\mathsf{ac.request.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

Honest authority issues credential. When the functionality $\mathcal{F}_{\mathrm{AC}}$ sends the message $(\mathsf{ac.issue.sim}, sid, qid)$, $\mathcal{S}$ sends $(\mathsf{nym.reply.sim}, sid, qid, l(m))$ to $\mathcal{A}$, where $l(m)$ is equal to the length of the message that the honest authority sends to a user in the $\mathsf{ac.issue}$ interface.

Honest user receives issuance. When $\mathcal{A}$ sends $(\mathsf{nym.reply.rep}, sid, qid)$, the simulator $\mathcal{S}$ sends $(\mathsf{ac.issue.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

Honest user shows credential. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.show.sim}, sid, qid, b)$, $\mathcal{S}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$ to $\mathcal{A}$, where $l(m)$ is the length of the message $\langle \{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi \rangle$ sent by honest users. When $\mathcal{A}$ sends $(\mathsf{nym.send.rep}, sid, qid)$, if $b = 0$ in the message $(\mathsf{ac.show.sim}, sid, qid, b)$ received from $\mathcal{F}_{\mathrm{AC}}$, $\mathcal{S}$ sends $(\mathsf{ac.show.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$, else $\mathcal{S}$ proceeds with the "Honest provider requests keys" item.

Honest provider requests keys. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.show.sim}, sid, qid, b)$, if $b = 1$, after running the "Honest user shows credential" item, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input $(\mathsf{kg.retrieve.ini}, sid)$. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.sim}, sid, qid, v)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest provider receives keys. When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.end}, sid, v)$, $\mathcal{S}$ sends $(\mathsf{ac.show.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

$\mathcal{A}$ **receives credential show.** When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.show.end}, \varphi, P)$ to a corrupt provider $\tilde{\mathcal{P}}_k$, $\mathcal{S}$ sets the message to be sent to the adversary as follows. For $l = 1$ to L:

- Pick random $r_l \leftarrow \mathbb{Z}_p$ and $r'_l \leftarrow \mathbb{Z}_p$.
- Compute $\sigma'_l = (h'_l, s'_l) \leftarrow (g^{r'_l}, g^{r_l r'_l})$.
- Compute $\kappa_l \leftarrow \tilde{g}^{r_l}$.

$\mathcal{S}$ runs the simulator $\mathcal{S}_v$ to compute a simulated proof π_v. $\mathcal{S}$ sends the message $(\mathsf{nym.send.end}, sid, \langle \{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi \rangle, P, sendid)$ to $\mathcal{A}$.

$\mathcal{A}$ **requests keys.** When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.ini}, sid)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.sim}, sid, qid, v)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ **receives keys.** When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.end}, sid, v)$, $\mathcal{S}$ sends that message to $\mathcal{A}$.

Theorem 2. *When (a subset of) providers $\mathcal{P}_k$ are corrupt, Π_{AC} securely realizes $\mathcal{F}_{\mathrm{AC}}$ in the $(\mathcal{F}_{\mathrm{KG}}, \mathcal{F}_{\mathrm{NYM}}, \mathcal{F}_{\mathrm{RO}})$-hybrid model if the non-interactive proof of knowledge scheme is zero-knowledge.*

Proof of Theorem 2. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\mathrm{REAL}_{\Pi_{\mathrm{AC}},\mathcal{A},\mathcal{Z}}$ and the ensemble $\mathrm{IDEAL}_{\mathcal{F}_{\mathrm{AC}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by $\Pr[\textbf{Game } i]$ the probability that the environment distinguishes **Game** i from the real-world protocol.

Game 0**:** This game corresponds to the execution of the real-world protocol. Therefore, $\Pr[\textbf{Game } 0] = 0$.

Game 1**:** This game proceeds as **Game** 0, except that in **Game** 1 the non-interactive proofs of knowledge π_v are replaced by simulated proofs computed by the simulator $\mathcal{S}_v$. Under the zero-knowledge property of the proof system (Definition 5), we have that $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| \le \mathrm{Adv}^{\mathsf{zk}}_{\mathcal{A}}$.

Game 2**:** This game proceeds as **Game** 1, except that in **Game** 2, for $l = 1$ to L, the values σ'_l and κ_l are computed as follows:

- Pick random $t_l \leftarrow \mathbb{Z}_p$ and $t'_l \leftarrow \mathbb{Z}_p$.
- Set $\sigma'_l = (h'_l, s'_l) \leftarrow (g^{t'_l}, g^{t_l t'_l})$.
- Set $\kappa_l \leftarrow \tilde{g}^{t_l}$.

Those values follow the same distribution as the ones computed by the honest user in the real-world protocol. Observe that the honest user computes the following:
- Pick random $r_l \leftarrow \mathbb{Z}_p$ and $r'_l \leftarrow \mathbb{Z}_p$.
- Set $\sigma'_l = (h'_l, s'_l) \leftarrow (h_l^{r'_l}, s_l^{r'_l}(h'_l)^{r_l}) = (h_l^{r'_l}, h_l^{(x+m_{l,1}y_1+\ldots+m_{l,q}y_q+r_l)r'_l}) = (g^{u_l r'_l}, g^{(x+m_{l,1}y_1+\ldots+m_{l,q}y_q+r_l)u_l r'_l})$.
- Set $\kappa_l \leftarrow \alpha \prod_{j=1}^{q} \beta_j^{m_{l,j}} \tilde{g}^{r_l} = \tilde{g}^{x+m_{l,1}y_1+\ldots+m_{l,q}y_q+r_l}$.

Therefore, t_l corresponds to $(x+m_{l,1}y_1+\ldots+m_{l,q}y_q+r_l)$ and t'_l corresponds to $u_l r'_l$, where u_l is a random value such that $h_l = g^{u_l}$. Both $(x + m_{l,1}y_1 + \ldots + m_{l,q}y_q + r_l)$ and $u_l r'_l$ are random. Observe as well that the verification equation $e(h'_l, \kappa_l) = e(s'_l, \tilde{g})$ still holds because $e(g^{t'_l}, \tilde{g}^{t_l}) = e(g^{t_l t'_l}, \tilde{g})$. Since the values are distributed identically, we have that $|\Pr[\textbf{Game } 2] - \Pr[\textbf{Game } 1]| = 0$.

The distribution of **Game** 2 is identical to that of our simulation. In **Game** 2, the credential show is computed without knowledge of the signature or the attributes shown by the honest user. The overall advantage of the environment to distinguish between the real and the ideal protocol is $|\Pr[\textbf{Game } 2] - \Pr[\textbf{Game } 0]| \leq \mathrm{Adv}^{\mathsf{zk}}_{\mathcal{A}}$. This concludes the proof of Theorem 2.

8.3 Security Analysis of $\Pi_{\mathbf{AC}}$ when $\mathcal{U}_j$ is Corrupt

Intuition. When (a subset of) users $\mathcal{U}_j$ are corrupt, we need to construct a simulator $\mathcal{S}$ that simulates the honest authorities and providers towards a copy of any adversary that controls the corrupt users. $\mathcal{S}$ needs to simulate the issue messages that honest authorities send to users with the information given by $\mathcal{F}_{\mathrm{AC}}$.

$\mathcal{S}$ must be able to extract from corrupt users the attributes for which they request credentials during the issuance phase, and the attributes they use to prove statements during the show phase, in order to be able to send them to $\mathcal{F}_{\mathrm{AC}}$. To this end, $\mathcal{S}$ uses the extractability property of the Fiat-Shamir transform.

Additionally, $\mathcal{S}$ must ensure that corrupt users are not able to show attributes if they did not obtain a credential for those attributes from at least t authorities. For this purpose, we prove that corrupt users that are able to do that can be used against the existential unforgeability of Pointcheval-Sanders signatures. This reduction also relies on the binding property of the commitment scheme, which is necessary to ensure that the random generators h, which are output by functionality $\mathcal{F}_{\mathrm{RO}}$ on input commitments, and which are used to compute signatures, are random and unique for each message tuple signed.

Simulator. We describe the simulator $\mathcal{S}$ for the case in which (a subset of) users $\mathcal{U}_j$ are corrupt. $\mathcal{S}$ simulates the protocol by running the authority and provider sides of protocol Π_{AC} and copies of the ideal functionalities involved.

Honest authority $\mathcal{V}_i$ starts setup. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.setup.sim}, sid, \mathcal{V}_i)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{KG}}$ on input $(\mathsf{kg.getkey.ini}, sid)$. When $\mathcal{F}_{\mathrm{KG}}$ sends the message $(\mathsf{kg.getkey.sim}, sid, qid, pk, \langle pk_i \rangle_{i=1}^{n})$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest authority $\mathcal{V}_i$ ends setup. When $\mathcal{A}$ sends $(\mathsf{kg.getkey.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on that input. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.getkey.end}, sid, pk, pk_i, sk_i)$, $\mathcal{S}$ sends $(\mathsf{ac.setup.rep}, sid, \mathcal{V}_i)$ to $\mathcal{F}_{\mathrm{AC}}$.

Honest user requests keys. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.request.sim}, sid, qid, b)$, if $b = 1$, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input $(\mathsf{kg.retrieve.ini}, sid)$. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.sim}, sid, qid, v)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest user receives keys. When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.end}, sid, v)$, $\mathcal{S}$ proceeds with the "Honest user sends request" item.

Honest user sends request. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.request.sim}, sid, qid, b)$, if $b = 0$ or otherwise if the "Honest user requests keys" item has been run, the simulator $\mathcal{S}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$ to $\mathcal{A}$, where $l(m)$ is equal to the length of the message that the honest user sends to an authority in the $\mathsf{ac.request}$ interface.

Honest authority receives request. When $\mathcal{A}$ sends $(\mathsf{nym.send.rep}, sid, qid)$, $\mathcal{S}$ sends $(\mathsf{ac.request.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

$\mathcal{A}$ requests keys. When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.ini}, sid)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.sim}, sid, qid, v)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ receives keys. When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.end}, sid, v)$, $\mathcal{S}$ sends that message to $\mathcal{A}$.

$\mathcal{A}$ queries random oracle. When $\mathcal{A}$ sends $(\mathsf{ro.query.ini}, sid, com)$, $\mathcal{S}$ runs the functionality $\mathcal{F}_{\mathrm{RO}}$ on that input. When $\mathcal{F}_{\mathrm{RO}}$ sends $(\mathsf{ro.query.end}, sid, h)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

$\mathcal{A}$ requests credential. When $\mathcal{A}$ sends $(\mathsf{nym.send.ini}, sid, \langle \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid \rangle, P, \mathcal{V}_i)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$, $\mathcal{S}$ sends that message to $\mathcal{A}$.

Honest authority receives request from $\mathcal{A}$. When $\mathcal{A}$ sends $(\mathsf{nym.send.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When $\mathcal{F}_{\mathrm{NYM}}$ sends the message $(\mathsf{nym.send.end}, sid, m, P, sendid)$, $\mathcal{S}$ parses the message m as $\langle \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid \rangle$ and does the following:

- Abort if the authority $\mathcal{V}_i$ did not end the setup.
- If $\mathcal{A}$ had already sent that message before, then retrieve the stored tuple $(\langle \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid, P, \mathcal{V}_i \rangle, \langle m_1, \ldots, m_q, o, o_1, \ldots, o_q \rangle)$. Else do the following:
  - Run $\mathcal{F}_{\mathrm{RO}}$ on input $(\mathsf{ro.query.ini}, sid, com)$ and receive $(\mathsf{ro.query.end}, sid, \hat{h})$ from $\mathcal{F}_{\mathrm{RO}}$. If $\hat{h} \neq h$ abort.
  - Verify π_s by using ϕ, pk and $(com, com_1, \ldots, com_q, h)$. Abort if the proof π_s is not correct.
  - Run the extractor $\mathcal{E}_s$ to extract the witness $\langle m_1, \ldots, m_q, o, o_1, \ldots, o_q \rangle$ from π_s.

- If there is a tuple $(\langle \Lambda = (com', com'_1, \ldots, com'_q, h', \pi'_s), \phi', reqid', P', \mathcal{V}'_i \rangle, \langle m'_1, \ldots, m'_q, o', k'_1, \ldots, k'_q \rangle)$ stored such that $com' = com$ but $(m'_1, \ldots, m'_q) \neq (m_1, \ldots, m_q)$, $\mathcal{S}$ outputs failure.
- Store the tuple $(\langle \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid, P, \mathcal{V}_i \rangle, \langle m_1, \ldots, m_q, o, o_1, \ldots, o_q \rangle)$.

$\mathcal{S}$ sets $a \leftarrow (m_1, \ldots, m_q)$ and sends $(\textsf{ac.request.ini}, sid, a, \phi, \mathcal{V}_i, P)$ to $\mathcal{F}_{\mathrm{AC}}$. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\textsf{ac.request.sim}, sid, qid, b)$, $\mathcal{S}$ sends $(\textsf{ac.request.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

Honest authority issues attribute. When the functionality $\mathcal{F}_{\mathrm{AC}}$ sends the message $(\textsf{ac.issue.sim}, sid, qid)$, $\mathcal{S}$ sends $(\textsf{nym.reply.sim}, sid, qid, l(m))$ to $\mathcal{A}$, where $l(m)$ is equal to the length of the message that the honest authority sends to a user in the ac.issue interface. When $\mathcal{A}$ sends $(\textsf{nym.reply.rep}, sid, qid)$, the simulator $\mathcal{S}$ sends $(\textsf{ac.issue.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

$\mathcal{A}$ receives issuance. When $\mathcal{F}_{\mathrm{AC}}$ sends the message $(\textsf{ac.issue.end}, sid, a, \phi, \mathcal{V}_i)$, $\mathcal{S}$ finds the stored tuple $(\langle \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi', reqid, P, \mathcal{V}'_i \rangle, \langle m_1, \ldots, m_q, o, o_1, \ldots, o_q \rangle)$ such that $a = (m_1, \ldots, m_q)$, $\phi' = \phi$ and $\mathcal{V}'_i = \mathcal{V}_i$. $\mathcal{S}$ uses the secret key $sk_i = (x_i, y_{i,1}, \ldots, y_{i,q})$ to compute $c = h^{x_i} \prod_{j=1}^{q} com_j^{y_{i,j}}$ and set the blinded signature share $\hat{\sigma}_i = (h, c)$ as in Π_{AC}. The simulator $\mathcal{S}$ stores a tuple $(sid, m_1, \ldots, m_q, \phi, \mathcal{V}_i)$ and sends $(\textsf{nym.reply.end}, sid, \langle \mathcal{V}_i, \hat{\sigma}_i, reqid \rangle, P)$ to $\mathcal{A}$.

Honest user shows credential. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\textsf{ac.show.sim}, sid, qid, b)$, $\mathcal{S}$ sends $(\textsf{nym.send.sim}, sid, qid, l(m))$ to $\mathcal{A}$, where $l(m)$ is the length of the message $\langle \{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi \rangle$ sent by honest users.

Honest provider receives credential show. When $\mathcal{A}$ sends $(\textsf{nym.send.rep}, sid, qid)$, if $b = 0$ in the message $(\textsf{ac.show.sim}, sid, qid, b)$ received from $\mathcal{F}_{\mathrm{AC}}$, $\mathcal{S}$ sends $(\textsf{ac.show.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$, else $\mathcal{S}$ proceeds with the "Honest provider requests keys" item.

Honest provider requests keys. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\textsf{ac.show.sim}, sid, qid, b)$, if $b = 1$, after running the "Honest user shows credential" item, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input $(\textsf{kg.retrieve.ini}, sid)$. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\textsf{kg.retrieve.sim}, sid, qid, v)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest provider receives keys. When $\mathcal{A}$ sends $(\textsf{kg.retrieve.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\textsf{kg.retrieve.end}, sid, v)$, $\mathcal{S}$ sends $(\textsf{ac.show.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

$\mathcal{A}$ initiates credential show. When the adversary $\mathcal{A}$ sends $(\textsf{nym.send.ini}, sid, \langle \{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi \rangle, P, \mathcal{P}_k)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When $\mathcal{F}_{\mathrm{NYM}}$ sends the message $(\textsf{nym.reply.sim}, sid, qid, l(m))$, $\mathcal{S}$ sends that message to $\mathcal{A}$.

Honest provider receives show from $\mathcal{A}$. When the adversary $\mathcal{A}$ sends the message $(\textsf{nym.reply.rep}, sid, qid)$, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When the functionality $\mathcal{F}_{\mathrm{NYM}}$ sends $(\textsf{nym.send.end}, sid, \langle \{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi \rangle, P, sendid)$, $\mathcal{S}$ follows construction Π_{AC} to verify the values σ'_l (for $l = 1$ to L) and the proof π_v. Then $\mathcal{S}$ proceeds as follows:
  - $\mathcal{S}$ runs the extractor $\mathcal{E}_v$ to extract the witness $(\langle m_{l,1}, \ldots, m_{l,q}, r_l \rangle_{l=1}^{L})$ from the proof π_v.
  - For $l = 1$ to L, $\mathcal{S}$ parses σ'_l as (h'_l, s'_l) and computes $\hat{\sigma}_l = (\hat{h}_l, \hat{s}_l) = (h'_l, s'_l (h'_l)^{-r_l})$.

- For $l = 1$ to L, $\mathcal{S}$ runs the verification equation $e(\hat{h}_l, \tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_j}) = e(\hat{s}_l, \tilde{g})$ of the Pointcheval-Sanders signature scheme. If for any signature $\hat{\sigma}_l$ the verification equation does not hold, $\mathcal{S}$ outputs failure.
- For $l = 1$ to L, $\mathcal{S}$ checks that there are at least t tuples $(sid, m_{l,1}, \ldots, m_{l,q}, \phi, \mathcal{V}_i)$ stored for t different authorities. If for any l that is not the case, $\mathcal{S}$ outputs failure.

$\mathcal{S}$ sends $(\mathsf{ac.show.ini}, sid, \varphi, P, \mathcal{P}_k)$ to $\mathcal{F}_{\mathrm{AC}}$. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.show.sim}, sid, qid, b)$, if $b = 1$, $\mathcal{S}$ proceeds with the "Honest provider requests keys" item, else $\mathcal{S}$ sends $(\mathsf{ac.show.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

Theorem 3. *When (a subset of) users $\mathcal{U}_j$ are corrupt, Π_{AC} securely realizes $\mathcal{F}_{\mathrm{AC}}$ in the $(\mathcal{F}_{\mathrm{KG}}, \mathcal{F}_{\mathrm{NYM}}, \mathcal{F}_{\mathrm{RO}})$-hybrid model if the non-interactive proof of knowledge scheme is extractable, the commmitment scheme is binding, and the signature scheme by Pointcheval-Sanders is unforgeable.*

Proof of Theorem 3. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\mathrm{REAL}_{\Pi_{\mathrm{AC}},\mathcal{A},\mathcal{Z}}$ and the ensemble $\mathrm{IDEAL}_{\mathcal{F}_{\mathrm{AC}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by $\Pr[\textbf{Game } i]$ the probability that the environment distinguishes **Game** i from the real-world protocol.

Game 0**:** This game corresponds to the execution of the real-world protocol. Therefore, $\Pr[\textbf{Game } 0] = 0$.

Game 1**:** This game proceeds as **Game** 0, except that **Game** 1 runs the extractors $\mathcal{E}_s$ and $\mathcal{E}_v$ for the the non-interactive proofs of knowledge π_s and π_v. Under the weak simulation extractability property of the proof system (Definition 6), we have that $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{ext}}$.

Game 2**:** This game proceeds as **Game** 1, except that **Game** 2 outputs failure if two request messages were received with commitments com' and com and proofs π'_s and π_s such that $com' = com$ but, after extraction of the witnesses from π'_s and π_s, $(m'_1, \ldots, m'_q) \neq (m_1, \ldots, m_q)$. Under the binding property of the commitment scheme, we have that $|\Pr[\textbf{Game } 2] - \Pr[\textbf{Game } 1]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{bin}}$. We omit a formal proof of this claim.

Game 3**:** This game proceeds as **Game** 2, except that, after extracting the witness $(\langle m_{l,1}, \ldots, m_{l,q}, r_l \rangle_{l=1}^{L})$ from the proof π_v, for $l = 1$ to L, **Game** 3 parses σ'_l received in the request as (h'_l, s'_l) and computes $\hat{\sigma}_l = (\hat{h}_l, \hat{s}_l) = (h'_l, s'_l(h'_l)^{-r_l})$. Then **Game** 3 outputs failure if any $\hat{\sigma}_l$ is not a valid signature. We observe that, after extraction from π_v is successful, each value κ_l is of the form $\kappa_l \leftarrow \tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_{l,j}} \tilde{g}^{r_l}$. We also know that the following equality holds

$$e(h'_l, \kappa_l) = e(s'_l, \tilde{g})$$

If we replace κ_l by $\tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_{l,j}} \tilde{g}^{r_l}$, we have that

$$e(h'_l, \tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_{l,j}} \tilde{g}^{r_l}) = e(s'_l, \tilde{g})$$

If we now multiply the two sides of the equality by $e(h'_l, \tilde{g}^{-r_l})$, we have that

$$e(h'_l, \tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_{l,j}} \tilde{g}^{r_l}) e(h'_l, \tilde{g}^{-r_l}) = e(s'_l, \tilde{g}) e(h'_l, \tilde{g}^{-r_l})$$

and this gives us

$$e(h'_l, \tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_{l,j}}) = e(s'_l (h'_l)^{-r_l}, \tilde{g})$$

which is the verification equation of the Pointcheval-Sanders signature scheme for the signature $(h'_l, s'_l (h'_l)^{-r_l})$. Therefore, the computation $\hat{\sigma}_l = (\hat{h}_l, \hat{s}_l) = (h'_l, s'_l (h'_l)^{-r_l})$ always produces a valid signature, and thus $|\Pr[\textbf{Game } 3] - \Pr[\textbf{Game } 2]| = 0$.

Game 4**:** This game proceeds as **Game** 3, except that **Game** 4 outputs failure if, after computing the signatures $\hat{\sigma}_l = (\hat{h}_l, \hat{s}_l) = (h'_l, s'_l (h'_l)^{-r_l})$ on $(m_{l,1}, \ldots, m_{l,q} \rangle_{l=1}^{L})$, it is the case that, for at least one index l, the adversary was not issued at least t signatures from t different authorities on the messages $(m_{l,1}, \ldots, m_{l,q})$. Under the unforgeability property of Pointcheval-Sanders signatures in the random oracle model, we have that $|\Pr[\textbf{Game } 4] - \Pr[\textbf{Game } 3]| \leq \mathrm{Adv}^{\mathsf{unf}}_{\mathcal{A}} \cdot (n!/((t-1)!(n-t+1)!))$, where n is the number of authorities.

Proof. We construct an algorithm B that interacts with the challenger of the existential unforgeability game in the RO model (Definition 4) and the adversary $\mathcal{A}$ and that shows that, if $\mathcal{A}$ makes **Game** 4 output failure with non-negligible probability, then $\mathcal{A}$ can be used by B to break the existential unforgeability property in the RO model of Pointcheval-Sanders signatures. B receives a public key $(\theta, \tilde{\alpha}, \beta_1, \tilde{\beta}_1 \ldots, \beta_q, \tilde{\beta}_q)$ from the challenger. To set up the keys when running functionality $\mathcal{F}_{\mathrm{KG}}$, B proceeds as follows.

- B picks a random index $i' \in [1, n]$ and assigns that public key to authority $\mathcal{V}_{i'}$, i.e. $pk_{i'} \leftarrow (\tilde{\alpha}, \beta_1, \tilde{\beta}_1 \ldots, \beta_q, \tilde{\beta}_q)$.
- To compute the secret keys and public keys of the first $t-1$ authorities (if $i' \geq t$) or of authorities $\mathcal{V}_i$ such that $i \in [1, t]$ and $i' \neq i$ (if $i' \in [1, t-1]$), B picks random $(x_i, y_{i,1}, \ldots, y_{i,q}) \leftarrow \mathbb{Z}_p$ and computes $pk_i = (\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \ldots, \beta_{i,q}, \tilde{\beta}_{i,q}) \leftarrow (\tilde{g}^{x_i}, g^{y_{i,1}}, \tilde{g}^{y_{i,1}}, \ldots, g^{y_{i,q}}, \tilde{g}^{y_{i,q}})$.
- Let $\mathbb{S}$ be the set of indices of authorities whose public key has already been computed. We note that $|\mathbb{S}| = t$. To compute the public key $pk = (\tilde{\alpha}, \beta_1, \tilde{\beta}_1, \ldots, \beta_q, \tilde{\beta}_q)$, B does the following.
  - For all $i \in \mathbb{S}$, evaluate at 0 the Lagrange basis polynomials

$$l_i = [\prod_{j \in \mathbb{S}, j \neq i} (0 - j)][\prod_{j \in \mathbb{S}, j \neq i} (i - j)]^{-1} \bmod p$$

  - For all $i \in \mathbb{S}$, take $(\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \ldots, \beta_{i,q}, \tilde{\beta}_{i,q})$ and then compute $(\tilde{\alpha}, \beta_1, \tilde{\beta}_1, \ldots, \beta_q, \tilde{\beta}_q) = (\prod_{i \in \mathbb{S}} \tilde{\alpha}_i^{l_i}, \prod_{i \in \mathbb{S}} \beta_{i,1}^{l_i}, \prod_{i \in \mathbb{S}} \tilde{\beta}_{i,1}^{l_i}, \ldots, \prod_{i \in \mathbb{S}} \beta_{i,q}^{l_i}, \prod_{i \in \mathbb{S}} \tilde{\beta}_{i,q}^{l_i})$.

- To compute the public keys of the remaining authorities, i.e. the authorities in the set $\mathbb{D} = [1, n] \setminus \mathbb{S}$, B does the following. For all $d \in \mathbb{D}$:
  - For all $i \in \mathbb{S}$, evaluate at d the Lagrange basis polynomials

$$l_i = [\prod_{j \in \mathbb{S}, j \neq i} (d - j)][\prod_{j \in \mathbb{S}, j \neq i} (i - j)]^{-1} \bmod p$$

  - For all $i \in \mathbb{S}$, take $(\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \ldots, \beta_{i,q}, \tilde{\beta}_{i,q})$ and then compute $pk_d = (\tilde{\alpha}_d, \beta_{d,1}, \tilde{\beta}_{d,1}, \ldots, \beta_{d,q}, \tilde{\beta}_{d,q}) = (\prod_{i \in \mathbb{S}} \tilde{\alpha}_i^{l_i}, \prod_{i \in \mathbb{S}} \beta_{i,1}^{l_i}, \prod_{i \in \mathbb{S}} \tilde{\beta}_{i,1}^{l_i}, \ldots, \prod_{i \in \mathbb{S}} \beta_{i,q}^{l_i}, \prod_{i \in \mathbb{S}} \tilde{\beta}_{i,q}^{l_i})$.

  We remark that the public keys computed this way are identically distributed to the ones computed in Π_{AC}.

To reply the random oracle queries of $\mathcal{A}$, B forwards the query *com* to the random oracle provided by the challenger and sends $\mathcal{A}$ the response h given by the challenger.
When $\mathcal{A}$ sends a valid request $\langle \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid \rangle$, B runs the extractor $\mathcal{E}_s$ to extract the witness $\langle m_1, \ldots, m_q, o, o_1, \ldots, o_q \rangle$ from π_s. B outputs failure if two request messages were received with commitments com' and com and proofs π'_s and π_s such that $com' = com$ but, after extraction of the witnesses from π'_s and π_s, $(m'_1, \ldots, m'_q) \neq (m_1, \ldots, m_q)$. Like in **Game** 2, the probability that B fails is negligible if the commitment scheme is binding.
If the request $\langle \Lambda = (com, com_1, \ldots, com_q, h', \pi_s), \phi, reqid \rangle$ is sent to an authority $\mathcal{V}_i$ such that $i \in \mathbb{S}$ and $i \neq i'$, B computes an issuance message by following Π_{AC} and stores $(m_1, \ldots, m_q, \mathcal{V}_i)$. (We note that in this case B knows the secret key of the authority.) If the request is sent to $\mathcal{V}_{i'}$ or to $\mathcal{V}_d$ such that $d \in \mathbb{D}$, B proceeds as follows:

- B submits the message tuple $(m_1, \ldots, m_q)$ that was extracted by $\mathcal{E}_s$ and the commitment *com* to the signing oracle provided by the challenger. The challenger sends a signature $\sigma_{i'} = (h, s_{i'})$ and state information st'.
- If the request was sent to authority $\mathcal{V}_{i'}$, B takes $\sigma_{i'} = (h, s_{i'})$ sent by the challenger, computes $\hat{\sigma}_{i'} = (h, s_{i'} \prod_{j=1}^{q} \beta_{i',j}^{o_j})$ and includes $\hat{\sigma}_{i'}$ in the issuance message sent to $\mathcal{A}$. B stores $(m_1, \ldots, m_q, \mathcal{V}_{i'})$.
- If the request was sent to an authority $\mathcal{V}_d$ such that $d \in \mathbb{D}$, B proceeds as follows:
  - For all $i \in \mathbb{S}$ such that $i \neq i'$, B computes a signature $\sigma_i = (h, s_i)$ by using the secret keys of authorities in $\mathbb{S}$.
  - For all $i \in \mathbb{S}$, B evaluates at d the Lagrange basis polynomials

$$l_i = [\prod_{j \in \mathbb{S}, j \neq i} (d - j)][\prod_{j \in \mathbb{S}, j \neq i} (i - j)]^{-1} \bmod p$$

  - B computes the signature $\sigma_d = (h, s_d) \leftarrow (h, \prod_{i \in \mathbb{S}} s_i^{l_i})$. We note that in this computation the signature sent by the challenger is used.
  - B computes $\hat{\sigma}_d = (h, s_d \prod_{j=1}^{q} \beta_{d,j}^{o_j})$ and includes it in the issuance message sent to $\mathcal{A}$. B stores $(m_1, \ldots, m_q, \mathcal{V}_d)$.

After $\mathcal{A}$ sends a valid credential show $\langle \{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi \rangle$, B proceeds as follows.

- B runs the extractor $\mathcal{E}_v$ to extract the witness $(\langle m_{l,1}, \ldots, m_{l,q}, r_l \rangle_{l=1}^{L})$ from the proof π_v.
- For $l = 1$ to L, B parses σ'_l as (h'_l, s'_l) and computes $\hat{\sigma}_l = (\hat{h}_l, \hat{s}_l) = (h'_l, s'_l(h'_l)^{-r_l})$.
- For $l = 1$ to L, B runs the verification equation $e(\hat{h}_l, \tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_j}) = e(\hat{s}_l, \tilde{g})$ of the Pointcheval-Sanders signature scheme. If for any signature $\hat{\sigma}_l$ the verification equation does not hold, B outputs failure. As shown in **Game** 3, the probability that B outputs failure is 0.
- For $l = 1$ to L, B checks that there are at least t tuples $(m_{l,1}, \ldots, m_{l,q}, \mathcal{V}_i)$ stored for t different authorities. If that is the case for any l, B does nothing because $\mathcal{A}$ was issued enough signatures to compute the credential show. Else, if for some l $\mathcal{A}$ received less than t signatures from different authorities, but $\mathcal{A}$ did receive a signature from $\mathcal{V}_{i'}$ or from an authority $\mathcal{V}_d$ such that $d \in \mathbb{D}$, B fails. However, if for some l $\mathcal{A}$ received less than t signatures from different authorities, and all those authorities $\mathcal{V}_i$ are such that $i \neq i'$ and $i \in \mathbb{S}$, B does the following.
  - For all $i \in \mathbb{S}$ such that $i \neq i'$, B computes $t-1$ signatures $\sigma_i = (h, s_i) = (\hat{h}_l, s_i)$ using the secret keys of authorities $\mathcal{V}_i$. B also sets $\sigma_0 = (h, s_0) = (\hat{h}_l, \hat{s}_l)$.
  - Set $\mathbb{S}' = (\mathbb{S} \setminus \{i'\}) \cup \{0\}$. For all $i \in \mathbb{S}'$, B evaluates at i' the Lagrange basis polynomials
  $$l_i = [\prod_{j \in \mathbb{S}, j \neq i} (i' - j)][\prod_{j \in \mathbb{S}, j \neq i} (i - j)]^{-1} \bmod p$$
  - B computes
  $$\sigma_{i'} = (h, \prod_{i \in \mathbb{S}'} s_i^{l_i})$$
  We note that the signature $\hat{\sigma}_l$ is used in this computation.
  - B sends $\sigma_{i'}$ to the challenger to win the existential unforgeability game.

The probability that B fails can be bound as follows. B needs to query the signing oracle of the challenger whenever $\mathcal{A}$ requests a signature from authority $\mathcal{V}_{i'}$ or from an authority $\mathcal{V}_d$ such that $d \in \mathbb{D}$. Therefore, when $\mathcal{A}$ is able to show a signature without receiving t signatures shares from t different authorities, B fails whenever $\mathcal{A}$ did request a signature from $\mathcal{V}_{i'}$ or from an authority $\mathcal{V}_d$ such that $d \in \mathbb{D}$. In the worst case, $\mathcal{A}$ received $t-1$ signatures from $t-1$ authorities. In that worst case, B only succeeds when those $t-1$ authorities are those authorities $\mathcal{V}_i$ such that $i \in \mathbb{S}$ and $i \neq i'$. The probability that B succeeds, i.e. the probability that $\mathcal{A}$ picks those $t-1$ authorities from the set of n authorities is given by the inverse of the number of $(t-1)$-element combinations of n objects taken without repetition

$$\frac{(t-1)!(n-t+1)!}{n!}$$

We remark that, in the frequent case in which $t = n$, then B succeeds with probability $1/t$.

The distribution of **Game** 4 is identical to that of our simulation. In **Game** 4, the user attributes are extracted from the request and show messages and can be sent to $\mathcal{F}_{\mathrm{AC}}$. Additionally, it is guaranteed that users cannot show attributes for which they did not receive credentials from at least t authorities. The overall advantage of the environment to distinguish between the real and the ideal protocol is $|\Pr[\textbf{Game } 4] - \Pr[\textbf{Game } 0]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{ext}} + \mathrm{Adv}_{\mathcal{A}}^{\mathsf{bin}} + \mathrm{Adv}_{\mathcal{A}}^{\mathsf{unf}} \cdot (n!/((t-1)!(n-t+1)!))$. This concludes the proof of Theorem 3.

8.4 Security Analysis of Π_{AC} when $\mathcal{V}_i$, $\mathcal{P}_k$ and $\mathcal{U}_j$ are Corrupt

Intuition. When a subset of users $\mathcal{U}_j$, a subset of providers $\mathcal{P}_k$ and up to $t-1$ authorities are corrupt, we need to construct a simulator $\mathcal{S}$ that simulates the honest parties towards a copy of any adversary that controls the corrupt parties, by using the information given by $\mathcal{F}_{\mathrm{AC}}$.

$\mathcal{S}$ must be able to compute request messages from honest users to corrupt authorities that are indistinguishable from those computed by honest users in the real protocol, without knowing the identity of honest users or the attributes that they request. Similarly, $\mathcal{S}$ must be able to compute credential show messages from honest users to corrupt providers that are indistinguishable from those computed by honest users in the real protocol, without knowing the identity of honest users or the attributes used to prove statements. The use of $\mathcal{F}_{\mathrm{NYM}}$, the zero-knowledge property of the Fiat-Shamir transform and the hiding property of commitments allow the simulator to do that.

Additionally, $\mathcal{S}$ must be able to extract from corrupt users the attributes for which they request credentials during the issuance phase, and the attributes they use to prove statements during the show phase, in order to be able to send them to $\mathcal{F}_{\mathrm{AC}}$. To this end, $\mathcal{S}$ uses the weak simulation extractability property of the Fiat-Shamir transform.

Finally, $\mathcal{S}$ outputs failure if a corrupt user is able to show an attribute but the user did not obtain signatures shares for that attribute from at least $t - |\mathbb{T}|$ authorities, where $|\mathbb{T}|$ is the number of corrupt authorities. The binding property of the commitment scheme and the existential unforgeability of Pointcheval-Sanders signatures in the RO model allows us to prove that $\mathcal{S}$ fails with negligible probability.

Simulator. We describe the simulator $\mathcal{S}$ for the case in which a subset of users $\mathcal{U}_j$, a subset of providers $\mathcal{P}_k$ and up to $t-1$ authorities are corrupt. $\mathcal{S}$ simulates the honest parties in the protocol Π_{AC} and runs copies of the ideal functionalities involved.

Honest authority $\mathcal{V}_i$ starts setup. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.setup.sim}, sid, \mathcal{V}_i)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{KG}}$ on input $(\mathsf{kg.getkey.ini}, sid)$. When $\mathcal{F}_{\mathrm{KG}}$ sends the message $(\mathsf{kg.getkey.sim}, sid, qid, pk, \langle pk_i \rangle_{i=1}^{n})$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest authority $\mathcal{V}_i$ ends setup. When $\mathcal{A}$ sends $(\mathsf{kg.getkey.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on that input. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.getkey.end}, sid, pk, pk_i, sk_i)$, $\mathcal{S}$ sends $(\mathsf{ac.setup.rep}, sid, \mathcal{V}_i)$ to $\mathcal{F}_{\mathrm{AC}}$.

Corrupt authority $\tilde{\mathcal{V}}_i$ starts setup. When a corrupt authority $\tilde{\mathcal{V}}_i$ sends the message $(\mathsf{kg.getkey.ini}, sid)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends the message $(\mathsf{kg.getkey.sim}, sid, qid, pk, \langle pk_i \rangle_{i=1}^{n})$, $\mathcal{S}$ forwards that message to $\tilde{\mathcal{V}}_i$.

Corrupt authority $\tilde{\mathcal{V}}_i$ ends setup. When $\tilde{\mathcal{V}}_i$ sends $(\mathsf{kg.getkey.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.getkey.end}, sid, pk, pk_i, sk_i)$, $\mathcal{S}$ sends $(\mathsf{ac.setup.ini}, sid)$ to $\mathcal{F}_{\mathrm{AC}}$. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.setup.sim}, sid, \tilde{\mathcal{V}}_i)$, $\mathcal{S}$ sends $(\mathsf{ac.setup.rep}, sid, \tilde{\mathcal{V}}_i)$ to $\mathcal{F}_{\mathrm{AC}}$. When $\mathcal{F}_{\mathrm{AC}}$ sends the message $(\mathsf{ac.setup.end}, sid)$, $\mathcal{S}$ sends $(\mathsf{kg.getkey.end}, sid, pk, pk_i, sk_i)$ to $\tilde{\mathcal{V}}_i$.

Honest user requests keys. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.request.sim}, sid, qid, b)$, if $b = 1$, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input $(\mathsf{kg.retrieve.ini}, sid)$. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.sim}, sid, qid, v)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest user receives keys. When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.end}, sid, v)$, $\mathcal{S}$ proceeds with the "Honest user sends request" item.

Honest user sends request. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.request.sim}, sid, qid, b)$, if $b = 0$ or otherwise if the "Honest user requests keys" item has been run, the simulator $\mathcal{S}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$ to $\mathcal{A}$, where $l(m)$ is equal to the length of the message that the honest user sends to an authority in the $\mathsf{ac.request}$ interface.

Honest authority receives request from honest user. When $\mathcal{A}$ sends the message $(\mathsf{nym.send.rep}, sid, qid)$, $\mathcal{S}$ sends $(\mathsf{ac.request.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

Corrupt authority receives request from honest user. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.request.end}, sid, \phi, P, reqid)$ to a corrupt authority $\tilde{\mathcal{V}}_i$, $\mathcal{S}$ runs a copy of a user on input $(\mathsf{ac.request.ini}, sid, a', \phi, \tilde{\mathcal{V}}_i, P)$, where, if the copy of the user stores a tuple $(sid, a, o_1, \ldots, o_q, P, \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid')$ such that $reqid' = reqid$, then $a' = a$, else a' is a random attribute. (The copy of the user sets $reqid$ to the value received from $\mathcal{F}_{\mathrm{AC}}$.) When running the copy of the user, $\mathcal{S}$ uses the simulator $\mathcal{S}_s$ to compute a simulated proof π_s. When the copy of the user sends the message $(\mathsf{nym.send.ini}, sid, \langle \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid\rangle, P, \tilde{\mathcal{V}}_i)$ to $\mathcal{F}_{\mathrm{NYM}}$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.send.sim}, sid, qid, l(m))$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input $(\mathsf{nym.send.rep}, sid, qid)$. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.send.end}, sid, \langle \Lambda = (com, com_1, \ldots, com_q, h, \pi_s), \phi, reqid\rangle, P, sendid)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Corrupt user requests keys. When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.ini}, sid)$, $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.sim}, sid, qid, v)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Corrupt user receives keys. When $\mathcal{A}$ sends $(\mathsf{kg.retrieve.rep}, sid, qid)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends $(\mathsf{kg.retrieve.end}, sid, v)$, $\mathcal{S}$ sends that message to $\mathcal{A}$.

$\mathcal{A}$ queries random oracle. When $\mathcal{A}$ sends $(\mathsf{ro.query.ini}, \mathit{sid}, \mathit{com})$, $\mathcal{S}$ runs functionality $\mathcal{F}_{\mathrm{RO}}$ on that input. When $\mathcal{F}_{\mathrm{RO}}$ sends $(\mathsf{ro.query.end}, \mathit{sid}, h)$, $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Corrupt user requests credential. When $\mathcal{A}$ sends $(\mathsf{nym.send.ini}, \mathit{sid}, \langle \Lambda = (\mathit{com}, \mathit{com}_1, \ldots, \mathit{com}_q, h, \pi_s), \phi, \mathit{reqid}\rangle, P, \mathcal{V}_i)$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.send.sim}, \mathit{sid}, \mathit{qid}, l(m))$, $\mathcal{S}$ sends that message to $\mathcal{A}$.

Honest authority receives request from corrupt user. When the adversary $\mathcal{A}$ sends $(\mathsf{nym.send.rep}, \mathit{sid}, \mathit{qid})$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When $\mathcal{F}_{\mathrm{NYM}}$ sends the message $(\mathsf{nym.send.end}, \mathit{sid}, m, P, \mathit{sendid})$, $\mathcal{S}$ parses the message m as $\langle \Lambda = (\mathit{com}, \mathit{com}_1, \ldots, \mathit{com}_q, h, \pi_s), \phi, \mathit{reqid}\rangle$ and does the following:

- Abort if the authority $\mathcal{V}_i$ did not end the setup.
- If $\mathcal{A}$ had already sent that message before, then retrieve the stored tuple $(\langle \Lambda = (\mathit{com}, \mathit{com}_1, \ldots, \mathit{com}_q, h, \pi_s), \phi, \mathit{reqid}, P, \mathcal{V}_i\rangle, \langle m_1, \ldots, m_q, o, o_1, \ldots, o_q\rangle)$. Else do the following:
  - Run $\mathcal{F}_{\mathrm{RO}}$ on input $(\mathsf{ro.query.ini}, \mathit{sid}, \mathit{com})$ and receive $(\mathsf{ro.query.end}, \mathit{sid}, \hat{h})$ from $\mathcal{F}_{\mathrm{RO}}$. If $\hat{h} \neq h$ abort.
  - Verify π_s by using ϕ, pk and $(\mathit{com}, \mathit{com}_1, \ldots, \mathit{com}_q, h)$. Abort if the proof π_s is not correct.
  - Run the extractor $\mathcal{E}_s$ to extract the witness $\langle m_1, \ldots, m_q, o, o_1, \ldots, o_q\rangle$ from π_s.
  - If there is a tuple $(\langle \Lambda = (\mathit{com}', \mathit{com}'_1, \ldots, \mathit{com}'_q, h', \pi'_s), \phi', \mathit{reqid}', P', \mathcal{V}'_i\rangle, \langle m'_1, \ldots, m'_q, o', k'_1, \ldots, k'_q\rangle)$ stored such that $\mathit{com}' = \mathit{com}$ but $(m'_1, \ldots, m'_q) \neq (m_1, \ldots, m_q)$, $\mathcal{S}$ outputs failure.
  - Store the tuple $(\langle \Lambda = (\mathit{com}, \mathit{com}_1, \ldots, \mathit{com}_q, h, \pi_s), \phi, \mathit{reqid}, P, \mathcal{V}_i\rangle, \langle m_1, \ldots, m_q, o, o_1, \ldots, o_q\rangle)$.

$\mathcal{S}$ sets $a \leftarrow (m_1, \ldots, m_q)$ and sends $(\mathsf{ac.request.ini}, \mathit{sid}, a, \phi, \mathcal{V}_i, P)$ to $\mathcal{F}_{\mathrm{AC}}$. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.request.sim}, \mathit{sid}, \mathit{qid}, b)$, $\mathcal{S}$ sends $(\mathsf{ac.request.rep}, \mathit{sid}, \mathit{qid})$ to $\mathcal{F}_{\mathrm{AC}}$.

Honest authority issues attribute. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.issue.sim}, \mathit{sid}, \mathit{qid})$, $\mathcal{S}$ sends $(\mathsf{nym.reply.sim}, \mathit{sid}, \mathit{qid}, l(m))$ to $\mathcal{A}$, where $l(m)$ is the length of the message that an honest authority sends in the $\mathsf{ac.issue}$ interface.

Honest user receives issuance from honest authority. When the adversary $\mathcal{A}$ sends $(\mathsf{nym.reply.rep}, \mathit{sid}, \mathit{qid})$, $\mathcal{S}$ sends $(\mathsf{ac.issue.rep}, \mathit{sid}, \mathit{qid})$ to $\mathcal{F}_{\mathrm{AC}}$.

Corrupt user receives issuance. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.issue.end}, \mathit{sid}, a, \phi, \mathcal{V}_i)$, $\mathcal{S}$ finds the stored tuple $(\langle \Lambda = (\mathit{com}, \mathit{com}_1, \ldots, \mathit{com}_q, h, \pi_s), \phi', \mathit{reqid}, P, \mathcal{V}'_i\rangle, \langle m_1, \ldots, m_q, o, o_1, \ldots, o_q\rangle)$ such that $a = (m_1, \ldots, m_q)$, $\phi' = \phi$ and $\mathcal{V}'_i = \mathcal{V}_i$. $\mathcal{S}$ uses the secret key $\mathit{sk}_i = (x_i, y_{i,1}, \ldots, y_{i,q})$ to compute $c = h^{x_i} \prod_{j=1}^{q} \mathit{com}_j^{y_{i,j}}$ and set the blinded signature share $\hat{\sigma}_i = (h, c)$ as in construction Π_{AC}. $\mathcal{S}$ stores a tuple $(\mathit{sid}, m_1, \ldots, m_q, \phi, \mathcal{V}_i)$ and sends the message $(\mathsf{nym.reply.end}, \mathit{sid}, \langle \mathcal{V}_i, \hat{\sigma}_i, \mathit{reqid}\rangle, P)$ to $\mathcal{A}$.

Corrupt authority issues attribute. When a corrupt authority $\tilde{\mathcal{V}}_i$ sends the message $(\mathsf{nym.reply.ini}, \mathit{sid}, \langle \tilde{\mathcal{V}}_i, \tilde{\sigma}_i, \mathit{reqid}\rangle, \mathit{sendid})$, $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When $\mathcal{F}_{\mathrm{NYM}}$ sends $(\mathsf{nym.reply.sim}, \mathit{sid}, \mathit{qid}, l(m))$, $\mathcal{S}$ sends that message to $\mathcal{A}$.

Honest user receives issuance from corrupt authority. When the adversary $\mathcal{A}$ sends the message (nym.reply.rep, sid, qid), the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When $\mathcal{F}_{\mathrm{NYM}}$ sends (nym.reply.end, $sid, \langle\tilde{\mathcal{V}}_i, \hat{\sigma}_i, reqid\rangle, P$), $\mathcal{S}$ runs the copy of the user on input that message. We remark that the copy of the user finds the request identifier $reqid$ associated with this issuance message, or aborts if it is not found. When the copy of the user outputs (ac.issue.end, $sid, a, \phi, \mathcal{V}_i$), $\mathcal{S}$ sends (ac.issue.ini, $sid, reqid$) to $\mathcal{F}_{\mathrm{AC}}$. When $\mathcal{F}_{\mathrm{AC}}$ sends (ac.issue.sim, sid, qid), $\mathcal{S}$ sends (ac.issue.rep, sid, qid) to $\mathcal{F}_{\mathrm{AC}}$.

Honest user shows credential. When $\mathcal{F}_{\mathrm{AC}}$ sends (ac.show.sim, sid, qid, b), $\mathcal{S}$ sends (nym.send.sim, $sid, qid, l(m)$) to $\mathcal{A}$, where $l(m)$ is the length of the message $\langle\{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi\rangle$ sent by honest users.

Honest provider receives credential show. When $\mathcal{A}$ sends (nym.send.rep, sid, qid), if $b = 0$ in the message (ac.show.sim, sid, qid, b) received from $\mathcal{F}_{\mathrm{AC}}$, $\mathcal{S}$ sends (ac.show.rep, sid, qid) to $\mathcal{F}_{\mathrm{AC}}$, else $\mathcal{S}$ proceeds with the "Honest provider requests keys" item.

Honest provider requests keys. When $\mathcal{F}_{\mathrm{AC}}$ sends (ac.show.sim, sid, qid, b), if $b = 1$, after running the "Honest user shows credential" item, the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input (kg.retrieve.ini, sid). When $\mathcal{F}_{\mathrm{KG}}$ sends (kg.retrieve.sim, sid, qid, v), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Honest provider receives keys. When $\mathcal{A}$ sends (kg.retrieve.rep, sid, qid), $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends (kg.retrieve.end, sid, v), $\mathcal{S}$ sends (ac.show.rep, sid, qid) to $\mathcal{F}_{\mathrm{AC}}$.

Corrupt provider receives credential show. When $\mathcal{F}_{\mathrm{AC}}$ sends (ac.show.end, φ, P) to a corrupt provider $\tilde{\mathcal{P}}_k$, $\mathcal{S}$ sets the message to be sent to the adversary as follows. For $l = 1$ to L:
- Pick random $r_l \leftarrow \mathbb{Z}_p$ and $r'_l \leftarrow \mathbb{Z}_p$.
- Compute $\sigma'_l = (h'_l, s'_l) \leftarrow (g^{r'_l}, g^{r_l r'_l})$.
- Compute $\kappa_l \leftarrow \tilde{g}^{r_l}$.

$\mathcal{S}$ runs the simulator $\mathcal{S}_v$ to compute a simulated proof π_v. $\mathcal{S}$ sends the message (nym.send.end, $sid, \langle\{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi\rangle, P, sendid$) to $\mathcal{A}$.

Corrupt provider requests keys. When $\mathcal{A}$ sends (kg.retrieve.ini, sid), $\mathcal{S}$ runs a copy of $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends (kg.retrieve.sim, sid, qid, v), $\mathcal{S}$ forwards that message to $\mathcal{A}$.

Corrupt provider receives keys. When $\mathcal{A}$ sends (kg.retrieve.rep, sid, qid), $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{KG}}$ on input that message. When $\mathcal{F}_{\mathrm{KG}}$ sends (kg.retrieve.end, sid, v), $\mathcal{S}$ sends that message to $\mathcal{A}$.

Corrupt user initiates credential show. When the adversary $\mathcal{A}$ sends the message (nym.send.ini, $sid, \langle\{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi\rangle, P, \mathcal{P}_k$), $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When $\mathcal{F}_{\mathrm{NYM}}$ sends the message (nym.reply.sim, $sid, qid, l(m)$), $\mathcal{S}$ sends that message to $\mathcal{A}$.

Honest provider receives show from corrupt user. When the adversary $\mathcal{A}$ sends the message (nym.reply.rep, sid, qid), the simulator $\mathcal{S}$ runs $\mathcal{F}_{\mathrm{NYM}}$ on input that message. When the functionality $\mathcal{F}_{\mathrm{NYM}}$ sends (nym.send.end, $sid, \langle\{\kappa_l, \sigma'_l\}_{l=1}^{L}, \pi_v, \varphi\rangle, P, sendid$), $\mathcal{S}$ follows construction Π_{AC} to verify the values σ'_l (for $l = 1$ to L) and the proof π_v. Then $\mathcal{S}$ proceeds as follows:

- $\mathcal{S}$ runs the extractor $\mathcal{E}_v$ to extract the witness $(\langle m_{l,1}, \ldots, m_{l,q}, r_l \rangle_{l=1}^{L})$ from the proof π_v.
- For $l = 1$ to L, $\mathcal{S}$ parses σ'_l as (h'_l, s'_l) and computes $\hat{\sigma}_l = (\hat{h}_l, \hat{s}_l) = (h'_l, s'_l(h'_l)^{-r_l})$.
- For $l = 1$ to L, $\mathcal{S}$ runs the verification equation $e(\hat{h}_l, \tilde{\alpha} \prod_{j=1}^{q} \tilde{\beta}_j^{m_j}) = e(\hat{s}_l, \tilde{g})$ of the Pointcheval-Sanders signature scheme. If for any signature $\hat{\sigma}_l$ the verification equation does not hold, $\mathcal{S}$ outputs failure.
- For $l = 1$ to L, $\mathcal{S}$ checks that there are at least $t - |\mathbb{T}|$ tuples $(sid, m_{l,1}, \ldots, m_{l,q}, \phi, \mathcal{V}_i)$ stored for $t - |\mathbb{T}|$ different authorities, where $|\mathbb{T}|$ is the number of corrupt authorities. If for any l that is not the case, $\mathcal{S}$ outputs failure.

$\mathcal{S}$ sends $(\mathsf{ac.show.ini}, sid, \varphi, P, \mathcal{P}_k)$ to $\mathcal{F}_{\mathrm{AC}}$. When $\mathcal{F}_{\mathrm{AC}}$ sends $(\mathsf{ac.show.sim}, sid, qid, b)$, if $b = 1$, $\mathcal{S}$ proceeds with the "Honest provider requests keys" item, else $\mathcal{S}$ sends $(\mathsf{ac.show.rep}, sid, qid)$ to $\mathcal{F}_{\mathrm{AC}}$.

Theorem 4. *When a subset of users $\mathcal{U}_j$, a subset of providers $\mathcal{P}_k$ and up to $t-1$ authorities are corrupt, Π_{AC} securely realizes $\mathcal{F}_{\mathrm{AC}}$ in the $(\mathcal{F}_{\mathrm{KG}}, \mathcal{F}_{\mathrm{NYM}}, \mathcal{F}_{\mathrm{RO}})$-hybrid model if the non-interactive proof of knowledge scheme is zero-knowledge and provides weak simulation extractability, the signature scheme by Pointcheval-Sanders is unforgeable, and the commmitment scheme is hiding and binding.*

Proof of Theorem 4. We show by means of a series of hybrid games that the environment $\mathcal{Z}$ cannot distinguish between the ensemble $\mathrm{REAL}_{\Pi_{\mathrm{AC}},\mathcal{A},\mathcal{Z}}$ and the ensemble $\mathrm{IDEAL}_{\mathcal{F}_{\mathrm{AC}},\mathcal{S},\mathcal{Z}}$ with non-negligible probability. We denote by $\Pr[\textbf{Game } i]$ the probability that the environment distinguishes **Game** i from the real-world protocol.

Game 0**:** This game corresponds to the execution of the real-world protocol. Therefore, $\Pr[\textbf{Game } 0] = 0$.

Game 1**:** This game proceeds as **Game** 0, except that **Game** 1 runs the extractors $\mathcal{E}_s$ and $\mathcal{E}_v$ for the the non-interactive proofs of knowledge π_s and π_v. Under the weak simulation extractability property of the proof system (Definition 6), we have that $|\Pr[\textbf{Game } 1] - \Pr[\textbf{Game } 0]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{ext}}$.

Game 2**:** This game proceeds as **Game** 1, except that **Game** 2 outputs failure if two request messages were received with commitments com' and com and proofs π'_s and π_s such that $com' = com$ but, after extraction of the witnesses from π'_s and π_s, $(m'_1, \ldots, m'_q) \neq (m_1, \ldots, m_q)$. Under the binding property of the commitment scheme, we have that $|\Pr[\textbf{Game } 2] - \Pr[\textbf{Game } 1]| \leq \mathrm{Adv}_{\mathcal{A}}^{\mathsf{bin}}$. We omit a formal proof of this claim.

Game 3**:** This game proceeds as **Game** 2, except that, after extracting the witness $(\langle m_{l,1}, \ldots, m_{l,q}, r_l \rangle_{l=1}^{L})$ from the proof π_v, for $l = 1$ to L, **Game** 3 parses σ'_l received in the request as (h'_l, s'_l) and computes $\hat{\sigma}_l = (\hat{h}_l, \hat{s}_l) = (h'_l, s'_l(h'_l)^{-r_l})$. Then **Game** 3 outputs failure if any $\hat{\sigma}_l$ is not a valid signature. As shown in the proof of Theorem 3, the computation $\hat{\sigma}_l = (\hat{h}_l, \hat{s}_l) = (h'_l, s'_l(h'_l)^{-r_l})$ always produces a valid signature, and thus $|\Pr[\textbf{Game } 3] - \Pr[\textbf{Game } 2]| = 0$.

Game 4**:** This game proceeds as **Game** 3, except that **Game** 4 outputs failure if, after computing the signatures $\hat{\sigma}_l = (\hat{h}_l, \hat{s}_l) = (h'_l, s'_l(h'_l)^{-r_l})$ on $(m_{l,1}, \ldots, m_{l,q}\rangle_{l=1}^{L})$, it is the case that, for at least one index l, the adversary was not issued at least $t - |\mathbb{T}|$ signatures from $t - |\mathbb{T}|$ different authorities on the messages $(m_{l,1}, \ldots, m_{l,q})$. Under the unforgeability property of Pointcheval-Sanders signatures in the random oracle model, we have that $|\Pr[\textbf{Game } 4] - \Pr[\textbf{Game } 3]| \leq \text{Adv}_{\mathcal{A}}^{\mathsf{unf}} \cdot ((n - |\mathbb{T}|)!/((t-1-|\mathbb{T}|)!(n-t+1)!))$, where n is the number of authorities.

Proof. This proof follows the proof of unforgeability given in the proof of Theorem 3 with a few changes. B receives a public key $(\theta, \tilde{\alpha}, \beta_1, \tilde{\beta}_1 \ldots, \beta_q, \tilde{\beta}_q)$ from the challenger. To set up the keys when running functionality $\mathcal{F}_{\text{KG}}$, B proceeds as follows.

- Let $\mathbb{T}$ be the set of indices of corrupt authorities. B picks a random index $i' \in [1, n] \setminus \mathbb{T}$ and assigns that public key to authority $\mathcal{V}_{i'}$, i.e. $pk_{i'} \leftarrow (\tilde{\alpha}, \beta_1, \tilde{\beta}_1 \ldots, \beta_q, \tilde{\beta}_q)$.
- Let $\mathbb{U}$ be a set of indices of size $t-1-|\mathbb{T}|$ picked at random from $[1, n] \setminus \mathbb{T}$. To compute the secret keys and public keys of authorities $\mathcal{V}_i$ in $\mathbb{T} \cup \mathbb{U}$, B picks random $sk_i = (x_i, y_{i,1}, \ldots, y_{i,q}) \leftarrow \mathbb{Z}_p$ and computes $pk_i = (\tilde{\alpha}_i, \beta_{i,1}, \tilde{\beta}_{i,1}, \ldots, \beta_{i,q}, \tilde{\beta}_{i,q}) \leftarrow (\tilde{g}^{x_i}, g^{y_{i,1}}, \tilde{g}^{y_{i,1}}, \ldots, g^{y_{i,q}}, \tilde{g}^{y_{i,q}})$.
- Let $\mathbb{S} \leftarrow \mathbb{T} \cup \mathbb{U} \cup \{i'\}$ and let $\mathbb{D} = [1, n] \setminus \mathbb{S}$. From this point on, the verification key and the public keys of authorities $\mathcal{V}_d$ such that $d \in \mathbb{D}$ are set as in the proof of Theorem 3. We remark that, because the maximum number of corrupt authorities is $t - 1$, B always knows the secret keys of corrupt authorities.

The rest of the proof works very similarly as the proof in Theorem 3, by using the new definitions of $\mathbb{D}$ and $\mathbb{S}$. The only changes are the following. B sends the secret key sk_i to an authority $\mathcal{V}_i$ such that $i \in \mathbb{T}$ (a corrupt authority) when the adversary requests it. After that, B does not need to reply to attribute issuance requests to corrupt authorities from the adversary. Later, when the adversary shows credentials and B checks if the adversary was issued enough signature shares, B checks that the adversary received at least $t - \mathbb{T}$ shares, instead of t shares.

Finally, the probability that B fails can be bound as follows. B needs to query the signing oracle of the challenger whenever $\mathcal{A}$ requests a signature from authority $\mathcal{V}_{i'}$ or from an authority $\mathcal{V}_d$ such that $d \in \mathbb{D}$. Therefore, when $\mathcal{A}$ is able to show a signature without receiving $t - |\mathbb{T}|$ signatures shares from $t - |\mathbb{T}|$ different authorities, B fails whenever $\mathcal{A}$ did request a signature from $\mathcal{V}_{i'}$ or from an authority $\mathcal{V}_d$ such that $d \in \mathbb{D}$. In the worst case, $\mathcal{A}$ received $t - 1 - |\mathbb{T}|$ signatures from $t - 1 - |\mathbb{T}|$ authorities. In that worst case, B only succeeds when those $t - 1 - |\mathbb{T}|$ authorities are those authorities $\mathcal{V}_i$ such that $i \in \mathbb{S}$ and $i \neq i'$. The probability that B succeeds, i.e. the probability that $\mathcal{A}$ picks those $t - 1 - |\mathbb{T}|$ authorities from the set of $n - |\mathbb{T}|$ authorities is given by the inverse of the number of $(t - 1 - |\mathbb{T}|)$-element combinations of $n - |\mathbb{T}|$

objects taken without repetition

$$\frac{(t-1-|\mathbb{T}|)!(n-t+1!)}{(n-|\mathbb{T}|)!}$$

We remark that, in the frequent case in which $t = n$, then B succeeds with probability $1/(t - |\mathbb{T}|)$.

Game 5**:** This game proceeds as **Game** 4, except that in **Game** 5 the non-interactive proofs of knowledge π_s and π_v that are sent to the adversary are replaced by simulated proofs computed by the simulator $\mathcal{S}_s$. Under the zero-knowledge property of the proof system (see Definition 5), we have that $|\Pr[\textbf{Game } 5] - \Pr[\textbf{Game } 4]| \le \mathrm{Adv}^{\mathsf{zk}}_{\mathcal{A}}$.

Game 6**:** This game proceeds as **Game** 5, except that in **Game** 6 the values $(com_1, \ldots, com_q)$ in each request are replaced by random values in $\mathbb{G}$. At this point, the proofs π_s are simulated proofs of false statements. Since the values $(com_1, \ldots, com_q)$ are randomly distributed, this change does not alter the view of the environment and we have that $|\Pr[\textbf{Game } 6]-\Pr[\textbf{Game } 5]| = 0$.

Game 7**:** This game proceeds as **Game** 6, except that in **Game** 7 the commitments to attributes $a = (m_1, \ldots, m_q)$ are replaced by commitments to random messages. Under the hiding property of the commitment scheme (see Definition 7), we have that $|\Pr[\textbf{Game } 7] - \Pr[\textbf{Game } 6]| \le \mathrm{Adv}^{\mathsf{hid}}_{\mathcal{A}} \cdot N_{req}$, where N_{req} is the number of different requests computed by honest users. Since the Pedersen commitment scheme is information theoretically hiding, we have that $|\Pr[\textbf{Game } 7] - \Pr[\textbf{Game } 6]| = 0$.

Game 8**:** This game proceeds as **Game** 7, except that in **Game** 8, for $l = 1$ to L, the values σ'_l and κ_l are computed as follows:
- Pick random $t_l \leftarrow \mathbb{Z}_p$ and $t'_l \leftarrow \mathbb{Z}_p$.
- Set $\sigma'_l = (h'_l, s'_l) \leftarrow (g^{t'_l}, g^{t_l t'_l})$.
- Set $\kappa_l \leftarrow \tilde{g}^{t_l}$.

Those values follow the same distribution as the ones computed by the honest user in the real-world protocol, as explained in the proof of Theorem 2. Since the values are distributed identically, we have that $|\Pr[\textbf{Game } 8] - \Pr[\textbf{Game } 7]| = 0$.

The distribution of **Game** 8 is identical to that of our simulation. In **Game** 8, the request message is computed without knowledge of the attributes requested by the honest user. The credential show is computed without knowledge of the signatures or the attributes shown by the honest user. Additionally, it is guaranteed that corrupt users cannot show attributes unless they obtained enough signatures shares from different authorities. The overall advantage of the environment to distinguish between the real and the ideal protocol is $|\Pr[\textbf{Game } 8]-\Pr[\textbf{Game } 0]| \le \mathrm{Adv}^{\mathsf{ext}}_{\mathcal{A}} + \mathrm{Adv}^{\mathsf{bin}}_{\mathcal{A}} + \mathrm{Adv}^{\mathsf{unf}}_{\mathcal{A}} \cdot ((n-|\mathbb{T}|)!/((t-1-|\mathbb{T}|)!(n-t+1)!)) + \mathrm{Adv}^{\mathsf{zk}}_{\mathcal{A}}$. This concludes the proof of Theorem 4.

9 Conclusion and Future Work

We have described and ideal functionality $\mathcal{F}_{\mathrm{AC}}$ for attributed-based credentials (ABC) with threshold issuance and a construction Π_{AC}, based on Coconut [11],

that realizes $\mathcal{F}_{\mathrm{AC}}$. Future work could extend ABC with threshold issuance with functionalities such as revocation, inspection and limited spending, or could also improve the efficiency of Π_{AC} by, e.g., providing a credential show protocol whose complexity does not grow linearly with the number of messages signed in a credential.

Acknowledgements. We thank George Danezis and Alberto Sonnino for their valuable comments.

References

1. Camenisch, J., Lysyanskaya, A.: An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In Pfitzmann, B., ed.: Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding. Volume 2045 of Lecture Notes in Computer Science., Springer (2001) 93–118
2. Camenisch, J., Lysyanskaya, A.: Signature schemes and anonymous credentials from bilinear maps. In Franklin, M.K., ed.: Advances in Cryptology - CRYPTO 2004, 24th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings. Volume 3152 of Lecture Notes in Computer Science., Springer (2004) 56–72
3. Belenkiy, M., Chase, M., Kohlweiss, M., Lysyanskaya, A.: P-signatures and noninteractive anonymous credentials. In Canetti, R., ed.: Theory of Cryptography, Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Volume 4948 of Lecture Notes in Computer Science., Springer (2008) 356–374
4. Belenkiy, M., Camenisch, J., Chase, M., Kohlweiss, M., Lysyanskaya, A., Shacham, H.: Randomizable proofs and delegatable anonymous credentials. In Halevi, S., ed.: Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings. Volume 5677 of Lecture Notes in Computer Science., Springer (2009) 108–125
5. Camenisch, J., Groß, T.: Efficient attributes for anonymous credentials. ACM Trans. Inf. Syst. Secur. **15**(1) (2012) 4:1–4:30
6. Baldimtsi, F., Lysyanskaya, A.: Anonymous credentials light. In Sadeghi, A., Gligor, V.D., Yung, M., eds.: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013, ACM (2013) 1087–1098
7. Chase, M., Meiklejohn, S., Zaverucha, G.: Algebraic macs and keyed-verification anonymous credentials. In Ahn, G., Yung, M., Li, N., eds.: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014, ACM (2014) 1205–1216
8. Camenisch, J., Dubovitskaya, M., Haralambiev, K., Kohlweiss, M.: Composable and modular anonymous credentials: Definitions and practical constructions. In Iwata, T., Cheon, J.H., eds.: Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II. Volume 9453 of Lecture Notes in Computer Science., Springer (2015) 262–288

9. Libert, B., Ling, S., Mouhartem, F., Nguyen, K., Wang, H.: Signature schemes with efficient protocols and dynamic group signatures from lattice assumptions. In Cheon, J.H., Takagi, T., eds.: Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II. Volume 10032 of Lecture Notes in Computer Science. (2016) 373–403
10. Fuchsbauer, G., Hanser, C., Slamanig, D.: Structure-preserving signatures on equivalence classes and constant-size anonymous credentials. J. Cryptol. **32**(2) (2019) 498–546
11. Sonnino, A., Al-Bassam, M., Bano, S., Meiklejohn, S., Danezis, G.: Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019, The Internet Society (2019)
12. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA. (2001) 136–145
13. Pointcheval, D., Sanders, O.: Short randomizable signatures. In Sako, K., ed.: Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings. Volume 9610 of Lecture Notes in Computer Science., Springer (2016) 111–126
14. Ballard, L., Green, M., de Medeiros, B., Monrose, F.: Correlation-resistant storage via keyword-searchable encryption. IACR Cryptol. ePrint Arch. (2005) 417
15. Camenisch, J., Stadler, M.: Proof systems for general statements about discrete logarithms. Technical Report TR 260, Institute for Theoretical Computer Science, ETH Zürich (March 1997)
16. Faust, S., Kohlweiss, M., Marson, G.A., Venturi, D.: On the non-malleability of the fiat-shamir transform. In: Progress in Cryptology-INDOCRYPT 2012. Springer (2012) 60–79
17. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In Odlyzko, A.M., ed.: CRYPTO '86. Volume 263., Springer Verlag (1987) 186–194
18. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In Feigenbaum, J., ed.: CRYPTO. Volume 576 of Lecture Notes in Computer Science., Springer (1991) 129–140
19. Gamal, T.E.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Trans. Inf. Theory **31**(4) (1985) 469–472
20. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM J. Comput. **17**(2) (1988) 281–308
21. Camenisch, J., Dubovitskaya, M., Rial, A.: UC commitments for modular protocol design and applications to revocation and attribute tokens. In: Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III. (2016) 208–239
22. Kate, A., Huang, Y., Goldberg, I.: Distributed key generation in the wild. IACR Cryptol. ePrint Arch. **2012** (2012) 377
23. Groth, J.: Non-interactive distributed key generation and key resharing. Cryptology ePrint Archive, Report 2021/339 (2021) `https://eprint.iacr.org/2021/339`.
24. Camenisch, J., Lehmann, A., Neven, G., Rial, A.: Privacy-preserving auditing for attribute-based credentials. In Kutylowski, M., Vaidya, J., eds.: Computer Security

- ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II. Volume 8713 of Lecture Notes in Computer Science., Springer (2014) 109–127
25. Camenisch, J., Drijvers, M., Gagliardoni, T., Lehmann, A., Neven, G.: The wonderful world of global random oracles. Cryptology ePrint Archive, Report 2018/165 (2018) https://ia.cr/2018/165.

A Security Definitions of Coconut Building Blocks

A.1 Fiat-Shamir Transform

Definition 5 (Zero-Knowledge). *Define the zero knowledge simulator $\mathcal{S}$ as follows. $\mathcal{S}$ is a stateful algorithm that can operate in two modes: $(h_i, st) \leftarrow \mathcal{S}(1, st, q_i)$ answers random oracle queries q_i, while $(\pi, st) \leftarrow \mathcal{S}(2, st, x)$ outputs a simulated proof π for an instance x. $\mathcal{S}(1, \cdots)$ and $\mathcal{S}(2, \cdots)$ share the state st that is updated after each operation.*

Let $\mathcal{L}$ be a language in NP. Denote with $(\mathcal{S}_1, \mathcal{S}_2)$ the oracles such that $\mathcal{S}_1(q_i)$ returns the first output of $(h_i, st) \leftarrow \mathcal{S}(1, st, q_i)$ and $\mathcal{S}_2(x, w)$ returns the first output of $(\pi, st) \leftarrow \mathcal{S}(2, st, x)$ if $(x, w) \in \mathcal{R}_\mathcal{L}$. A protocol $(\mathcal{P}^H, \mathcal{V}^H)$ is a non-interactive zero-knowledge proof for the language $\mathcal{L}$ in the random oracle model if there exists a ppt simulator $\mathcal{S}$ such that for all ppt distinguishers $\mathcal{D}$ we have

$$Pr[\mathcal{D}^{H(\cdot), \mathcal{P}^H(\cdot,\cdot)}(1^k) = 1] \approx Pr[\mathcal{D}^{\mathcal{S}_1(\cdot), \mathcal{S}_2(\cdot,\cdot)}(1^k) = 1],$$

where both $\mathcal{P}$ and $\mathcal{S}_2$ oracles output $\perp$ if $(x, w) \notin \mathcal{R}_\mathcal{L}$.

Definition 6 (Weak Simulation Extractability). *Let $\mathcal{L}$ be a language in NP. Consider a non-interactive zero-knowledge proof system $(\mathcal{P}^H, \mathcal{V}^H)$ for $\mathcal{L}$ with zero-knowledge simulator $\mathcal{S}$. Let $(\mathcal{S}_1, \mathcal{S}'_2)$ be oracles returning the first output of $(h_i, st) \leftarrow \mathcal{S}(1, st, q_i)$ and $(\pi, st) \leftarrow \mathcal{S}(2, st, x)$ respectively. $(\mathcal{P}^H, \mathcal{V}^H)$ is weakly simulation extractable with extraction error ν and with respect to $\mathcal{S}$ in the random oracle model, if for all ppt adversaries $\mathcal{A}$ there exists an efficient algorithm $\mathcal{E}_\mathcal{A}$ with access to the answers $(\mathcal{T}_H, \mathcal{T})$ of $(\mathcal{S}_1, \mathcal{S}'_2)$ respectively such that the following holds. Let*

$$\begin{aligned} \mathrm{acc} &= Pr[(x^*, \pi^*) \leftarrow \mathcal{A}^{\mathcal{S}_1(\cdot), \mathcal{S}'_2(\cdot)}(1^k; \rho) : (x^*, \pi^*) \notin \mathcal{T}; \mathcal{V}^{\mathcal{S}_1}(x^*, \pi^*) = 1] \\ \mathrm{ext} &= Pr[(x^*, \pi^*) \leftarrow \mathcal{A}^{\mathcal{S}_1(\cdot), \mathcal{S}'_2(\cdot)}(1^k; \rho); \\ &\quad w^* \leftarrow \mathcal{E}_\mathcal{A}(x^*, \pi^*; \rho, \mathcal{T}_H, \mathcal{T}) : (x^*, \pi^*) \notin \mathcal{T}; (x^*, w^*) \in \mathcal{R}_\mathcal{L}], \end{aligned}$$

where the probability space in both cases is over the random choices of $\mathcal{S}$ and the adversary's random tape ρ. Then, there exists a constant $d > 0$ and a polynomial p such that whenever $\mathrm{acc} \geq \nu$, *we have* $\mathrm{ext} \geq (1/p)(\mathrm{acc} - \nu)^d$.

A.2 Commitment Schemes

Correctness requires that VfCom accepts all commitments created by algorithm Com, i.e., for all $x \in \mathcal{M}$

$$\Pr\left[\begin{array}{c} par_c \leftarrow \mathsf{CSetup}(1^k);\ (com, open) \leftarrow \mathsf{Com}(par_c, x) : \\ 1 = \mathsf{VfCom}(par_c, com, x, open) \end{array}\right] = 1\ .$$

The *hiding* property ensures that a commitment *com* to x does not reveal any information about x, whereas the *binding* property ensures that *com* cannot be opened to another value x'.

Definition 7 (Hiding Property). *For any ppt adversary $\mathcal{A}$, the hiding property is defined as follows:*

$$Pr\left[\begin{array}{l} par_c \leftarrow \mathsf{CSetup}(1^k); \\ (x_0, st) \leftarrow \mathcal{A}(par_c); \\ x_1 \leftarrow \mathcal{M}; \\ b \leftarrow \{0,1\};\ (com, open) \leftarrow \mathsf{Com}(par_c, x_b); \\ b' \leftarrow \mathcal{A}(st, com): \\ x_0 \in \mathcal{M}\ \wedge\ x_1 \in \mathcal{M}\ \wedge\ b = b' \end{array}\right] \leq \frac{1}{2} + \epsilon(k)\ .$$

Definition 8 (Binding Property). *For any ppt adversary $\mathcal{A}$, the binding property is defined as follows:*

$$Pr\left[\begin{array}{l} par_c \leftarrow \mathsf{CSetup}(1^k);\ (com, x, open, x', open') \leftarrow \mathcal{A}(par_c): \\ x \in \mathcal{M}\ \wedge\ x' \in \mathcal{M}\ \wedge x \neq x' \wedge\ 1 = \mathsf{VfCom}(par_c, com, x, open) \\ \wedge\ 1 = \mathsf{VfCom}(par_c, com, x', open') \end{array}\right] \leq \epsilon(k)\ .$$

A.3 Public-Key Encryption Schemes

Correctness requires that, for all pairs $(pk_{enc}, sk_{enc}) \in \mathsf{Setup}$, for all messages m in the message space and all ciphertexts c output by $\mathsf{Encrypt}(pk_{enc}, m)$, the algorithm $\mathsf{Decrypt}(c, sk_{enc})$ outputs m with overwhelming probability.

Definition 9 (IND-CPA). *The PKE scheme is said to be IND-CPA (or semantically) secure if for any ppt adversary $\mathcal{A}$, there exists a negligible function $\nu(\cdot)$ such that the following is satisfied for any two messages m_0, m_1 in the message space and for $b \in \{0,1\}$:*

$$|\Pr\left[\mathcal{A}(1^k, \mathsf{Encrypt}(m_0, pk_{enc})) = b\right] - \Pr\left[\mathcal{A}(1^k, \mathsf{Encrypt}(m_1, pk_{enc})) = b\right]| \leq \nu(k).$$

A.4 Signature Schemes

Definition 10 (Correctness). *Correctness ensures that the algorithm* VfSig *accepts the signatures created by the algorithm* Sign *on input a secret key computed by algorithm* KeyGen*. More formally, correctness is defined as follows.*

$$Pr\left[\begin{array}{l} (sk, pk) \leftarrow \mathsf{KeyGen}(1^k);\ m \leftarrow \mathcal{M}; \\ \sigma \leftarrow \mathsf{Sign}(sk, m):\ 1 = \mathsf{VfSig}(pk, \sigma, m) \end{array}\right] = 1$$

Definition 11 (Existential Unforgeability). *The property of existential unforgeability ensures that it is not feasible to output a signature on a message without knowledge of the secret key or of another signature on that message. Let*

$\mathcal{O}_s$ be an oracle that, on input sk and a message $m \in \mathcal{M}$, outputs $\mathsf{Sign}(sk, m)$*, and let S_s be a set that contains the messages sent to $\mathcal{O}_s$. More formally, for any ppt adversary $\mathcal{A}$, existential unforgeability is defined as follows.*

$$Pr\left[\begin{array}{l}(sk, pk) \leftarrow \mathsf{KeyGen}(1^k);\ (m, \sigma) \leftarrow \mathcal{A}(pk)^{\mathcal{O}_s(sk,\cdot)} : \\ 1 = \mathsf{VfSig}(pk, \sigma, m)\ \wedge\ m \in \mathcal{M}\ \wedge\ m \notin S_s\end{array}\right] \leq \epsilon(k)$$