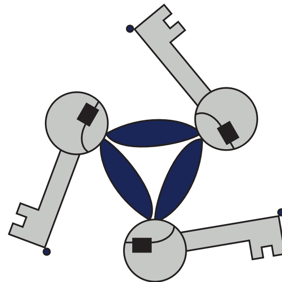
PADERBORN UNIVERSITY
The University for the Information Society

Final Documentation

Project Group:

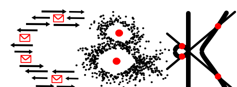
Re(AC)^{t}
Reputation and Anonymous Credentials for Access Control (t=2)



Supervisors:

Prof. Dr. Johannes Blömer
Jan Bobolz
Fabian Eidens

Research Group
Codes and Cryptography



Participants of the Project Group

Kai Bemmann

Henrik Bröcher

Denis Diemert

Lukas Eilers

Jan Haltermann

Burhan Otour

Laurens Porzenheim

Simon Pukrop

Erik Schilling

Michael Schlichtig

Marcel Stienemeier

Contents

1 Introduction 1

I Theory Documentation 5

2 Notation 7

3 Building Blocks 9
3.1 Bilinear Groups . . . 10
3.2 Computational Assumptions . . . 10
3.3 Hash Functions . . . 13
3.4 Digital Signatures . . . 14
3.5 Commitment Schemes . . . 22
3.6 Secret-Sharing Schemes . . . 27
3.7 Zero-Knowledge Arguments of Knowledge . . . 29
3.8 Σ-Protocols . . . 33
3.9 Damgård's Technique . . . 37
3.10 Fiat-Shamir Heuristic . . . 41
3.11 Proofs of Partial Knowledge . . . 42
3.12 Accumulators . . . 45

4 Anonymous Credential and Reputation System 53
4.1 Preliminaries . . . 54
4.2 Basic Anonymous Credential System . . . 68
4.3 Extended Anonymous Credential System . . . 83
4.4 Attribute-Based Anonymous Credential and Reputation System . . . 100
4.5 Further Extensions . . . 107

II Practical Realization 111

5 From Theory to Practice 113

6 Implementation 117
6.1 Introduction . . . 118
6.2 Architecture . . . 118
6.3 Building Blocks . . . 119
6.4 Zero-Knowledge Component . . . 131
6.5 Reputation System . . . 141
6.6 API . . . 142
6.7 Example Application . . . 148

7 Conclusion 155

Bibliography 157

1 Introduction

The usage of the Internet brings along many advantages that make our everyday life easier. But apart from that, it can be abused to collect private data of users. Important to note is that the user often is not in control which data is revealed. Consider the example of authentication. Here, the user often is asked to share some sensitive data to get access to a service, even though some of the information might not be needed in the context of the service. This point is quite critical because the services can abuse this information for their advantage, e.g. sell parts of the information.

The standard methods of authentication are either using a combination of login name and password or authenticating through a third party. The first option requires the user to remember all different combinations for all services, while in the second case the third party can track the user's activities. Note that in both cases the service provider can identify and track the user.

A better alternative is to introduce an *anonymous credential system (ACS)*. Instead of authenticating through a third party, one uses credentials, which are a certificate over some attributes. The certificate is issued by some organization that the user sent a request to. Then, the user can show the credential to some service provider, which can verify the validity of the credential to allow the user to access the service. Obviously, the service provider should only grant access if the user truly was issued credentials that allow her access to the service. This means that a user can neither create valid credentials herself nor show others' credentials to gain access. One important thing for the verification is that it is possible with only public information, meaning that the issuing organization is not needed in this step. This allows the user to not reveal to the issuing organization which services she is using. Furthermore, a user wants to stay anonymous which is achieved by using pseudonyms. By allowing the user to choose new pseudonyms whenever she wants to, her activities can only be tracked if she chooses to do so. Moreover, a user can decide what part of a credential is shown to a verifying party. Thus, a service provider only learns necessary information and not additional information she does not need to know. Sometimes, credentials should not be permanently valid. Thus, one may allow the issuer to revoke credentials at any point in time. Additionally, a trusted third party may be added to the credential system having the power to punish misuse by revealing the identity behind a pseudonym.

Furthermore, a credential system can be used to implement a *reputation system (RS)*. If allowed by a credential, a reputation system grants a user the ability to rate other sellers or their products. Through the use of credentials, one could allow a user to rate certain products that he for example paid for. A user should be able to rate products anonymously to not fear any backlash. Still, there should exist a trusted third party, which is the only one that can link a rating to its (misbehaving) rater. In such a case, the user should be punishable, e. g. through revoking the user from the reputation system. Although a rating should not be linkable to a user, everyone should be able to see if two ratings were published by the same user, so that she cannot falsify the overall rating of some product. Hence, users can only rate a product once while remaining anonymous.

A good example for the usage of an anonymous credential system together with a reputation system is an online shop. In this setup, the store manager can issue buyer permits, seller permits and maybe some other credentials. Then, a user can buy an item from a seller, and additionally receives a credential from the seller. This credential can then be used by the buyer to rate the seller and/or the item she bought, but only once. The buyer can stay anonymous while only revealing to the seller that she is allowed to purchase the item and some extra information

necessary for the transaction, but not more. If the user does not pay after purchasing, there is some trusted third party, for example the store manager, that can reveal the identity of the user to force her to pay.

Related Work One approach to construct an anonymous credential system is the framework used by Camenisch and Lysyanskaya in [CL04]. In their proposal, a secure signature scheme, a commitment scheme, efficient protocols for signing a committed value, proving knowledge of a signature and showing the equality of two committed values suffice to construct a very basic anonymous credential system. In 2016, Pointcheval and Sanders [PS16] proposed a signature scheme which allows for a more efficient construction of the framework above. Since the security requirements and functionalities vary depending on the context, there is no unified model of an anonymous credential system. Though, there are two major approaches of defining security itself, i. e. via ideal worlds like Camenisch and Lysyanskaya [CL04] and game-based like Pashalidis and Mitchell [PM04]. A basic framework like in [CL04] limits users to prove possession of a credential, whereas in practice services often require more complex access policies than only possessing a single credential. To overcome this restriction, the technique of proofs of partial knowledge proposed by Cramer, Damgård, and Schoenmakers [CDS94] is frequently used. An example can be found in the work of Anada, Arita, and Sakurai [AAS16], which supports monotone predicates over a single relation. Furthermore, some anonymous credential systems include the option for a trusted party which can identify users in case of misuse, for example. This identity escrow can, for example, be realized using public key encryption as noted in [CL04].

In practice, systems with components like policies and identity escrow have been realized. One notable example is ABC4Trust[1]. ABC4Trust is an EU project aiming at building a common architectural framework and providing unified definitions for attributed-based credential systems. However, the policies supported by ABC4Trust are limited. In their framework specification, the policy only consists of a single conjunction or disjunction, respectively. Another example for attribute-based credential system is Identity Mixer (idemix)[2] which is an anonymous credential system developed at IBM Research. It implements many desirable features of a credential system, but lacks of a dedicated rating mechanism we want to integrate.

In combination with the credential system we provide features of an anonymous reputation system which, to the best of our knowledge, has not been done before. However, an anonymous and publicly linkable reputation system has been defined and constructed by Blömer, Juhnke, and Kolb [BJK15]. Its key feature is that raters can anonymously rate items as long as they rate the same item at most once.

Outcome of the Project Group In this work, we present a combination of an attribute-based anonymous credential and an anonymous reputation system, which we call an *attribute-based anonymous credential and reputation system (ACRS)*. We formally define the syntax of such a system and provide a game-based security model for the anonymous credential system. Our system includes a trusted party which has the capability of identifying users who have shown a credential or rated an item, for example, in case of misuse. Moreover, we integrate the features of an anonymous reputation system to rate services and items anonymously. As result we construct a system offering identification-free access control in combination with a rating mechanism which, to the best of our knowledge, has neither been done theoretically nor practically before. Our main contribution is that we let services require almost arbitrary predicates as an access policy. In this context, we allow services to equip their policies with (in)equalities of attributes and membership of attributes in a set or a range. These relations can be concatenated via conjunctions and disjunctions within the policies which are represented by predicates within our formal models. In addition to the theoretical treatment of the used building blocks and the system itself, we

[1] `https://www.abc4trust.eu/`
[2] `https://www.zurich.ibm.com/identity_mixer/`

provide an implementation in form of a Java library. Here, we provide implementations for all used building blocks and a compiler that translates a statement into a suitable protocol.

Part I

Theory Documentation

2 Notation

In this chapter, we introduce notation used throughout this document.

- Assigning a value y to variable x is denoted by $x \coloneqq y$.
- For a finite set S, we denote by $x \leftarrow S$ the operation of choosing an element uniformly at random from set S and assigning it to variable x.
- If A is a deterministic algorithm, we write $x \coloneqq A(y_1, y_2, \dots)$ to denote that algorithm A on inputs $y_1, y_2, \dots$ outputs value x.
- If A is a probabilistic algorithm, we write $x \leftarrow A(y_1, y_2, \dots)$ to denote that algorithm A on inputs $y_1, y_2, \dots$ outputs value x. Note that the operator "$\leftarrow$" is used differently in this case. Here, it indicates that x is a random variable taking on values according to the distribution of outputs generated by A on the given inputs.
- By writing $A(x; k)$ we describe starting a probabilistic algorithm A with input x and random coins k. This means that the algorithm uses the coins k to do all its random choices for which we use the notation $k \leftarrow \mathsf{Coins}(A)$. Note that the output of a probabilistic algorithm becomes deterministic if the random coins are fixed.
- We use the abbreviation ppt to talk about probabilistic algorithms with polynomial run time. Analogously, if we talk about expected ppt, we mean a probabilistic algorithms with *expected* polynomial run time.
- If $x \leftarrow A(y_1, y_2, \dots)$, we denote the set of all possible outputs of algorithm A on inputs $y_1, y_2, \dots$ by $[A(y_1, y_2, \dots)] \coloneqq \{z \mid \Pr[x = z] > 0\}$.
- For two interactive algorithms A and B, we denote the set of all possible outputs by $[A(y_1, y_2, \dots)] \times [B(z_1, z_2, \dots)]$.
- Running an algorithm A on inputs $y_1, y_2, \dots$ and with access to the oracles $\mathcal{O}_1, \mathcal{O}_2, \dots$ is denoted by either $A(y_1, y_2, \dots : \mathcal{O}_1, \mathcal{O}_2, \dots)$ or $A^{\mathcal{O}_1, \mathcal{O}_2, \dots}(y_1, y_2, \dots)$. We mainly use the latter notation. Still, sometimes an algorithm is provided with lots of oracles, where the former notation is more suitable due to readability.
- By $x_A \leftarrow A(y_1, y_2, \dots) \leftrightarrow B(z_1, z_2, \dots) \rightarrow x_B$ we denote that A and B interact on the stated inputs, where eventually A outputs x_A, and B outputs x_B.
- We use $\mathrm{output}_A[A(y_1, y_2, \dots) \leftrightarrow B(z_1, z_2, \dots)]$ to denote the random variable of A's output after interacting with B on the particular inputs.
- A variable followed by brackets indicates a vector of elements. For example, $\mathsf{upk}[\cdot]$ describes a vector of *user public keys*, where the i-th component, write $\mathsf{upk}[i]$ denotes the public key of user i.

3 Building Blocks

Contents

3.1 Bilinear Groups **10**
3.2 Computational Assumptions **10**
3.2.1 Discrete Logarithm 11
3.2.2 Symmetric Discrete Logarithm Assumption 11
3.2.3 Decisional Diffie-Hellman 11
3.2.4 Pointcheval-Sanders Assumptions 12
3.2.5 Modified Strong Diffie-Hellman Assumption 13
3.3 Hash Functions **13**
3.4 Digital Signatures **14**
3.4.1 Pointcheval-Sanders Signature Scheme 15
3.4.1.1 Single-Message Signatures 16
3.4.1.2 Multi-Message Signatures 18
3.5 Commitment Schemes **22**
3.5.1 Generalized Pedersen Commitment Scheme 23
3.5.2 Trapdoor Commitment Schemes 25
3.5.3 Hash-Then-Commit 26
3.6 Secret-Sharing Schemes **27**
3.6.1 Smooth Secret-Sharing Schemes 29
3.7 Zero-Knowledge Arguments of Knowledge **29**
3.7.1 Zero-Knowledge Arguments 31
3.7.2 Arguments of Knowledge 31
3.7.3 Camenisch-Stadler Notation 32
3.7.4 Non-interactive Arguments 32
3.8 Σ-Protocols **33**
3.8.1 Schnorr Protocol 34
3.8.2 Generalized Schnorr Protocol 35
3.9 Damgård's Technique **37**
3.10 Fiat-Shamir Heuristic **41**
3.10.1 Non-Interactive Arguments via the Fiat-Shamir Heuristic 42
3.10.2 Signature Schemes via the Fiat-Shamir Heuristic 42
3.11 Proofs of Partial Knowledge **42**
3.12 Accumulators **45**
3.12.1 Static Accumulators 45
3.12.2 Dynamic Accumulators 46
3.12.3 Nguyen Accumulator 47

We start our theoretical treatment of our anonymous credential and reputation system by formally defining all building blocks used in our constructions given in Chapter 4. Apart from formal definitions of the building blocks, we give rigorous security proofs for most of the schemes presented.

3.1 Bilinear Groups

The signature scheme and therefore our construction of the anonymous credential and reputation system will be based on *bilinear groups.* The next definition is an adaption of the one used by Pointcheval and Sanders [PS16].

Definition 3.1 (Bilinear Group)**.** A *bilinear group* is a tuple $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ such that

- $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ (written multiplicatively) are cyclic groups with prime order $p = |\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T|$. The group operations in these groups are *efficiently* computable.
- $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is an *efficiently* computable map, called *pairing*, such that
  1. (Bilinearity) For all $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$, it holds that
  $$e({g_1}^a, {g_2}^b) = e(g_1, g_2)^{ab}.$$
  2. (Non-degeneracy) For $g_1 \in \mathbb{G}_1 \setminus \{1\}$ and $g_2 \in \mathbb{G}_2 \setminus \{1\}$, it holds $e(g_1, g_2) \neq 1$.

Pairings can be categorized into three types. Following Galbraith, Paterson, and Smart [GPS08] these types are:

- *Type 1*: $\mathbb{G}_1 = \mathbb{G}_2$.
- *Type 2*: $\mathbb{G}_1 \neq \mathbb{G}_2$ and there is an efficient isomorphism $\phi\colon \mathbb{G}_2 \to \mathbb{G}_1$, but no efficient one in the other direction.
- *Type 3*: $\mathbb{G}_1 \neq \mathbb{G}_2$ and there is an efficient isomorphism, for neither $\mathbb{G}_1 \to \mathbb{G}_2$ nor $\mathbb{G}_2 \to \mathbb{G}_1$.

We distinguish corresponding types of bilinear groups. For example, a type 3 bilinear group is a bilinear group such that the pairing is a type 3 pairing. The signature scheme we use for most of our constructions is based on type 3 bilinear groups. It is crucial for this construction that the group is of the mentioned kind; if not, security cannot be guaranteed.

To use these groups in our constructions, we need to be able to construct them. For this we define a *bilinear group generators.*

Definition 3.2. We call a ppt algorithm G a *bilinear group generator* if the following holds: On input 1^n, G outputs a prime p with $p \geq 2^n$, and the description of group $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ of prime order p. In addition it outputs a bilinear map $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ such that $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ form a bilinear group.

A group generator is quite important in terms of the security. It ensures that the bilinear group can be set up in polynomial time in our security parameter and thus is representable in polynomial space in the security parameter. Moreover, security is always relative to the group generator.

3.2 Computational Assumptions

The security of most of the building blocks we use are based on the hardness of computational problems. In this section, we formally define these computational assumptions.

$\mathsf{DLog}_{\mathcal{A},\mathsf{G}}(n)$

1 : Run $\mathsf{G}(1^n)$ to obtain $(p, \mathbb{G})$, where $\mathbb{G}$ is a cyclic group of prime order p.
2 : Pick $g \in \mathbb{G} \setminus \{1\}$ and $r \in \mathbb{Z}_p$ uniformly at random.
3 : $\mathcal{A}$ is given $\mathbb{G}, p, g, h := g^r$, and outputs $x \in \mathbb{Z}_p$.
4 : The output of the experiment is defined to be 1 if $g^x = h$, and 0 otherwise.

Figure 3.1: Discrete Logarithm Problem

3.2.1 Discrete Logarithm

A basic assumption we need is the *discrete logarithm problem*. In a cyclic, prime order group $\mathbb{G}$ with generator g, we assume the computation of $\log_g(h)$ for a uniformly chosen element $h \in \mathbb{G}$ to be hard. We define the problem using the experiment given in Figure 3.1. It uses a *group generator*, which we define as a generalization of bilinear group generators given in Definition 3.2:

Definition 3.3. We call a ppt algorithm G a *group generator* if the following holds: On input 1^n, G outputs a prime number p with $p \geq 2^n$ and the description of a cyclic group $\mathbb{G}$ of prime order p.

Based on the experiment defined in Figure 3.1, we define the discrete logarithm problem:

Definition 3.4 (Discrete Logarithm Problem)**.** The discrete logarithm problem is hard relative to the group generator G if for all ppt algorithms $\mathcal{A}$ there is a negligible function μ such that

$$\Pr[\mathsf{DLog}_{\mathcal{A},\mathsf{G}}(n) = 1] \leq \mu(n),$$

where experiment $\mathsf{DLog}_{\mathcal{A},\mathsf{G}}(n)$ is defined in Figure 3.1.

3.2.2 Symmetric Discrete Logarithm Assumption

A variant of the discrete logarithm problem for bilinear groups is the *symmetric discrete logarithm problem*. There, the group generator additionally outputs three groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$, a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and generators for $\mathbb{G}_1$ and $\mathbb{G}_2$. The adversary then has to compute the discrete logarithm of some group element g^x, while also knowing $\tilde{g}^x$ and the other public parameters. Formally, this results in the following definition:

Definition 3.5 (Symmetric Discrete Logarithm Assumption)**.** Let $n \in \mathbb{N}$ be a security parameter. Let G be a type 3 bilinear group generator. The assumption is that for all ppt adversaries there is a negligible function $\mu : \mathbb{N} \to \mathbb{R}^+$ such that

$$\begin{aligned}\Pr[&(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n), x \leftarrow \mathbb{Z}_p, g \leftarrow \mathbb{G}_1 \setminus \{1\}, \tilde{g} \leftarrow \mathbb{G}_2 \setminus \{1\},\\ &y \leftarrow \mathcal{A}(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, \tilde{g}, g^x, \tilde{g}^y) : g^y = g^x] \leq \mu(n).\end{aligned}$$

3.2.3 Decisional Diffie-Hellman

Another assumption we need is the *external decisional Diffie-Hellman* (XDDH) problem. It is a generalization of the decisional Diffie-Hellman (DDH) problem, as defined by Boneh [Bon98], to the context of type 2 and 3 pairings. The DDH problem is about the hardness of distinguishing a tuple $(g, g^\alpha, g^\beta, g^{\alpha\beta})$ from $(g, g^\alpha, g^\beta, g^{\alpha\beta+\gamma})$ for a cyclic, prime order p group, where α, β, γ are chosen uniformly at random from $\mathbb{Z}_p^*$. For XDDH we demand, that DDH is hard within group $\mathbb{G}_1$ according to a bilinear group generator as given in Definition 3.2. We define the XDDH problem using the experiment given in Figure 3.2.

$\mathsf{XDDH}_{\mathcal{A},\mathsf{G}}(n)$

1 : Run $\mathsf{G}(1^n)$ to obtain $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, where the groups are cyclic and of prime order p.
2 : Choose $g \in \mathbb{G}_1 \setminus \{1\}$ and $\tilde{g} \in \mathbb{G}_2 \setminus \{1\}$ uniformly at random.
3 : Choose $\alpha, \beta, \gamma \in \mathbb{Z}_p^*$ and $\delta \in \{0,1\}$ uniformly at random.
4 : $\mathcal{A}$ is given $p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, g^\alpha, g^\beta, g^{\alpha\beta+\gamma\delta}, \tilde{g}$ and outputs $\delta' \in \{0,1\}$.
5 : The output of the experiment is defined to be 1 if $\delta' = \delta$, and 0 otherwise.

Figure 3.2: External Decisional Diffie-Hellman Problem

Definition 3.6 (External Decisional Diffie-Hellman Problem)**.** The external decisional Diffie-Hellman problem is hard relative to the bilinear group generator G if for all ppt algorithms $\mathcal{A}$ there is a negligible function μ such that

$$\Pr[\mathsf{XDDH}_{\mathcal{A},\mathsf{G}}(n) = 1] \leq \frac{1}{2} + \mu(n),$$

where experiment $\mathsf{XDDH}_{\mathcal{A},\mathsf{G}}(n)$ is defined in Figure 3.2.

Note, that the XDDH assumption can only hold in contexts of type 2 and 3 bilinear pairings. In case of a type 1 pairing, an adversary could distinguish, by checking equality of $e(g^\alpha, g^\beta)$ and $e(g, g^{\alpha\beta+\gamma\delta})$. The *symmetric* external decisional Diffie-Hellman (SXDH) assumption demands, that DDH is hard in $\mathbb{G}_2$ as well, which can only hold for type 3 pairings. The homomorphism ϕ from a type 2 pairing, would enable an adversary to compare $e(\phi(\tilde{g}^\alpha), \tilde{g}^\beta) = e(\phi(\tilde{g}), \tilde{g}^{\alpha\beta})$ with $e(\phi(\tilde{g}), \tilde{g}^{\alpha\beta+\gamma\delta})$.

3.2.4 Pointcheval-Sanders Assumptions

In Section 3.4.1, we present the signature scheme that we use for our constructions. The security of this signature scheme, and thus the constructions based on it, rely on assumptions that were stated by the authors of the scheme in [PS16]. Let us start with the first assumption.

Definition 3.7 (Assumption 1)**.** Let G be a type 3 bilinear group generator and let $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n)$. Further, let $(x, y) \in \mathbb{Z}_p^2$. Define the oracle $O_{x,y}(m)$ that on input $m \in \mathbb{Z}_p$ outputs the pair (h, h^{x+ym}) for $h \leftarrow \mathbb{G}_1 \setminus \{1\}$. The assumption is that for all ppt adversaries $\mathcal{A}$, there is a negligible function $\mu \colon \mathbb{N} \to \mathbb{R}^+$ such that

$$\begin{aligned}\Pr\Big[&(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n), x \leftarrow \mathbb{Z}_p, y \leftarrow \mathbb{Z}_p, g \leftarrow \mathbb{G}_1 \setminus \{1\}, \tilde{g} \leftarrow \mathbb{G}_2 \setminus \{1\}, Y := g^y,\\ &\tilde{X} := \tilde{g}^x, \tilde{Y} := \tilde{g}^y, (m, (\sigma_1, \sigma_2)) \leftarrow \mathcal{A}^{O_{x,y}(\cdot)}(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, Y, \tilde{g}, \tilde{X}, \tilde{Y}) :\\ &m \notin Q \wedge m \in \mathbb{Z}_p \wedge \sigma_1 \in \mathbb{G}_1 \setminus \{1\} \wedge \sigma_2 = \sigma_1^{x+y\,m}\Big] \leq \mu(n)\end{aligned}$$

where Q denotes the set of oracle queries of $\mathcal{A}$.

Intuitively this means that no efficient adversary given a description of a type 3 bilinear group, $(g, Y, \tilde{g}, \tilde{X}, \tilde{Y})$, as defined above, and unlimited access to the oracle O is able to generate a pair $\Big(m, (\sigma_1, \sigma_1^{x+y\,m})\Big)$ for m not asked to the oracle.

The second assumption stated by Pointcheval and Sanders is a weaker variant of Definition 3.7. Here, the adversary is only given $(\tilde{g}, \tilde{X}, \tilde{Y})$ instead of $(g, Y, \tilde{g}, \tilde{X}, \tilde{Y})$. Formally, we have the following assumption:

Definition 3.8 (Assumption 2)**.** Let G be a type 3 bilinear group generator and let $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n)$. Define the oracle $O_{x,y}(m)$ that outputs, on input $m \in \mathbb{Z}_p$, the pair (h, h^{x+ym}) for $h \leftarrow \mathbb{G}_1 \setminus \{1\}$. The assumption is that for all ppt adversaries $\mathcal{A}$, there is a negligible function $\mu\colon \mathbb{N} \to \mathbb{R}^+$ such that

$$\Pr\Big[(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n), x \leftarrow \mathbb{Z}_p, y \leftarrow \mathbb{Z}_p, \tilde{g} \leftarrow \mathbb{G}_2 \setminus \{1\}, \tilde{X} := \tilde{g}^x, \tilde{Y} := \tilde{g}^y,$$
$$(m, (\sigma_1, \sigma_2)) \leftarrow \mathcal{A}^{O_{x,y}(\cdot)}(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \tilde{g}, \tilde{X}, \tilde{Y}) : m \notin Q \wedge m \in \mathbb{Z}_p$$
$$\wedge\, \sigma_1 \in \mathbb{G}_1 \setminus \{1\} \wedge \sigma_2 = \sigma_1^{x+y\,m}\Big] \leq \mu(n)$$

where Q denotes the set of oracle queries of $\mathcal{A}$.

As stated in [PS16, Theorem 4], both of these assumptions hold in the generic group model. A proof for this can be found in the full version [PS15, Appendix B].

3.2.5 Modified Strong Diffie-Hellman Assumption

In Section 3.12.3 we introduce a construction of accumulators whose security depends on a modified version of the *q-Strong Diffie Hellman Assumption*. The q-SDH problem was first introduced by D. Boneh and X. Boyen in [BB04]. Intuitively spoken, the assumption says that there is no ppt algorithm that can compute a tuple $(x, g^{(x+s)^{-1}})$ with $x \in \mathbb{Z}_p, s \in \mathbb{Z}_p^*$ from a tuple $t = (g, g^s, g^{s^2}, \dots g^{s^q})$. In contrast to the original assumption, we modify it slightly by adding a tuple $(\tilde{g}, \tilde{g}^s)$ to t, since we use type 3 pairings.

Definition 3.9 (Modified q-Strong Diffie-Hellman (q-SDH) Assumption)**.** Let G be a type 3 bilinear group generator. The assumption is that for all ppt adversaries $\mathcal{A}$, there is a negligible function $\mu\colon \mathbb{N} \to \mathbb{R}^+$ such that

$$\Pr\Big[(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n), s \leftarrow \mathbb{Z}_p, g \leftarrow \mathbb{G}_1 \setminus \{1\}, \tilde{g} \leftarrow \mathbb{G}_2 \setminus \{1\}, t := (\tilde{g}, \tilde{g}^s, g, g^s, g^{s^2}, \dots g^{s^q}),$$
$$(c_1, c_2) \leftarrow \mathcal{A}(p, e, g, \tilde{g}, t) : (c_1 \in \mathbb{Z}_p) \wedge (c_2 \in \mathbb{G}_1) \wedge (c_2 = g^{(s+c_1)^{-1}})\Big] \leq \mu(n)$$

3.3 Hash Functions

A helpful primitive in cryptography is the concept of a cryptographic hash function. Like a normal hash function, a cryptographic hash function maps an input from some big domain into a smaller range. The difference is that cryptographic hash functions may fulfill some additional properties that speak about difficulty in finding relations between pre-images and images. From here on, we only speak about cryptographic hash functions.

Definition 3.10. A cryptographic hash function is a tuple of ppt algorithms $(\mathsf{Setup}, \mathsf{Gen}_h, h)$, where $\mathsf{Setup}(1^n)$ outputs public parameters pp with $|\mathsf{pp}| \geq n$ and $\mathsf{Gen}(\mathsf{pp})$ outputs a key k. Furthermore, $h(k, m)$ takes as input a key k and a message $m \in \{0,1\}^*$ to output a hash value $y \in M(\mathsf{pp})$ if k was generated by $\mathsf{Gen}_h(\mathsf{pp})$ with $|y| \leq \{0,1\}^{l(n)}$ for some polynomial $l(\cdot)$.

We write $h_k(m)$ for $h(k, m)$. We also say that H hashes into $M(\mathsf{pp})$. If h is only defined for $m \in \{0,1\}^{l'(n)}$ for a polynomial $l'(\cdot)$, call the hash function a fixed-length hash function for $l'(\cdot)$.

One important property of a hash function is the so called *collision resistance*. This tells us that it is difficult to find two pre-images that map to the same image. To properly formalize this, we define a game. Let $n \in \mathbb{N}$, let $H = (\mathsf{Setup}, \mathsf{Gen}_h, h)$ be a hash function, and let $\mathcal{A}$ be a ppt adversary.

$\mathsf{HashColl}_{H,\mathcal{A}}(n)$

1 : Generate $\mathsf{pp} \leftarrow \mathsf{Setup}(1^n)$.
2 : Generate a key $k \leftarrow \mathsf{Gen}_h(\mathsf{pp})$.
3 : Run $\mathcal{A}(\mathsf{pp}, k)$.
4 : $\mathcal{A}$ outputs some $m_1, m_2 \in \{0,1\}^*$.
5 : If $m_1 \neq m_2$ and $h_k(m_1) = h_k(m_2)$, output 1. Else, output 0.

If H was a fixed-length hash function for a polynomial $l(\cdot)$, we also require that $m_1, m_2 \in \{0,1\}^{l(n)}$. If the output of the experiment is 1, we say that $\mathcal{A}$ wins.

Definition 3.11. Let H be a hash function. We call H *collision resistant*, if for all ppt adversaries $\mathcal{A}$ there exists a negligible function $\mathrm{negl}(n)$, such that

$$\Pr[\mathsf{HashColl}_{H,\mathcal{A}}(n) = 1] \leq \mathrm{negl}(n).$$

3.4 Digital Signatures

The essential primitive we use to construct our anonymous credential and reputation system are *digital signatures*. Concretely, in our system credentials are digital signatures on a set of attributes and a user secret key. To formally define a digital signature scheme, we slightly adjust the standard definition of a digital signature scheme given by Goldwasser, Micali, and Rivest [GMR88]. In addition, we have a fourth algorithm that deals with the common setup of the signature scheme. This algorithm, for example, generates the group that is used by every entity working with the signature scheme in a higher level system. Consider the following definition:

Definition 3.12 (Digital Signature Scheme)**.** A *(digital) signature scheme* Π with message space $\mathbb{M}$ is a tuple of ppt algorithms $(\mathsf{Setup}, \mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$, where

1. $\mathsf{Setup}(1^n)$: On input security parameter 1^n, it outputs public parameters pp with $|\mathsf{pp}| \geq n$.
2. $\mathsf{Gen}(\mathsf{pp})$: On input public parameters pp, it outputs a key pair $(\mathsf{pk}, \mathsf{sk})$ with $|\mathsf{pk}|, |\mathsf{sk}| \geq n$.
3. $\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}, m)$: On input secret key sk and a message $m \in \mathbb{M}$, it outputs a signature σ.
4. $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{pk}, m, \sigma)$: On input public key pk, a message $m \in \mathbb{M}$ and a signature σ, it outputs $b \in \{0,1\}$. We interpret 1 as *valid* and 0 as *invalid*.

For convenience we often assume that given the secret key sk, the public key pk is (implicitly) known as well. However, Definition 3.12 does not necessarily yield a meaningful signature scheme. Therefore, we additionally impose the *correctness* of a signature scheme.

Correctness of a Signature Scheme For all $n \in \mathbb{N}$, all $\mathsf{pp} \in [\mathsf{Setup}(1^n)]$, all $(\mathsf{pk}, \mathsf{sk}) \in [\mathsf{Gen}(\mathsf{pp})]$ and all $m \in \mathbb{M}$, it holds that

$$\mathsf{Vrfy}(\mathsf{pp}, \mathsf{pk}, m, \mathsf{Sign}(\mathsf{pp}, \mathsf{sk}, m)) = 1.$$

Having defined digital signatures syntactically, we proceed with the security requirements, illustrated in the following example. Bob has a key pair $(\mathsf{pk}_B, \mathsf{sk}_B)$ and publishes his public key pk_B. He wants to send a message to his friend Alice. Therefore he signs the message using his signing key sk_B and sends it alongside the signature to Alice. Upon receiving message and signature, Alice wants to verify whether the message was really sent by Bob and uses his publicly

Experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n)$:

1. $\mathsf{pp} \leftarrow \mathsf{Setup}(1^n)$
2. $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(\mathsf{pp})$
3. Adversary $\mathcal{A}$ is given pp, pk and oracle access to $\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}, \cdot)$, and outputs (m, σ). Let Q be the set of queries made by $\mathcal{A}$ to oracle $\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}, \cdot)$ during its execution.
4. Output is 1 iff $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{pk}, m, \sigma) = 1$, and $m \notin Q$.

Figure 3.3: Existential Unforgeability under a Chosen-Message Attack

known verification key pk_B to do so. Assuming the public key pk_B was not exchanged by a corrupted one, we need to ensure the infeasibility of generating some message and a corresponding new valid signature without possession of secret key sk_B. This should even be infeasible for Mallory, who successfully can convince Bob to sign messages of her choice.

These considerations are reflected in an experiment, where the adversary, given pk and oracle access to a signing oracle, is challenged to output some (not necessarily meaningful) message m and a corresponding valid signature σ under pk, where the oracle has not been queried for a signature on m. If the adversary outputs such a pair this is called an *existential forgery*. We call a digital signature scheme secure if no ppt adversary can generate existential forgeries efficiently. This notion of security is standard for digital signature schemes and goes back to Goldwasser, Micali, and Rivest [GMR88]. We formalize it in Definition 3.13 using the experiment given in Figure 3.3.

Definition 3.13. A signature scheme $\Pi = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ is said to be *existentially unforgeable under an adaptive chosen-message attack*, or *secure*, if for every ppt adversary $\mathcal{A}$ there is a negligible function $\mu\colon \mathbb{N} \to \mathbb{R}^+$ such that

$$\Pr[\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n) = 1] \leq \mu(n),$$

where the experiment $\mathsf{Sig\text{-}forge}_{\mathcal{A},\Pi}(n)$ is defined in Figure 3.3.

3.4.1 Pointcheval-Sanders Signature Scheme

In 2016, Pointcheval and Sanders [PS16] proposed a new signature scheme. It shares many features with the widely used Camenisch-Lysyanskaya (CL) signature scheme [CL04]. The latter is widely used because of its flexibility in many cryptographic protocols. The scheme designed by Pointcheval and Sanders preserves every desirable feature of the scheme by Camenisch and Lysyanskaya, but provides more efficient signing and verification.

This signature scheme (Constructions 3.14 and 3.17) is a good choice for our anoynymous credential system since the features of it together with the algebraic structure of the signatures result in efficient protocols, for example the issuance and showing of a credential. Pointcheval and Sanders [PS16] already provide good base protocols for issuing and showing credentials, which we use as foundation of our anonymous credential system.

The Pointcheval-Sanders (PS) signature scheme offers two variants. On the one hand, there is a variant for signing a single message. On the other, we have a generalization that is able to sign blocks of messages. The scheme presented in the following section is a variation of the signature scheme proposed by Pointcheval and Sanders [PS16], where we adapt the public key to be consistent with the protocols presented in Sections 4.1 and 4.2.3.

3.4.1.1 Single-Message Signatures

Construction 3.14 (Single-Message Pointcheval-Sanders Signature Scheme)**.** Let G be a type 3 bilinear group generator (Definition 3.2). The *single-message Pointcheval-Sanders (PS) signature scheme* is defined as $\Pi_S = (\mathsf{Setup}, \mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$, where

1. $\mathsf{Setup}(1^n)$: On input security parameter 1^n, Setup obtains $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n)$ and returns $\mathsf{pp} := (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$.
2. $\mathsf{Gen}(\mathsf{pp})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, Gen chooses generators $g \leftarrow \mathbb{G}_1 \setminus \{1\}$ and $\tilde{g} \leftarrow \mathbb{G}_2 \setminus \{1\}$. It then chooses $x \leftarrow \mathbb{Z}_p$ and $y \leftarrow \mathbb{Z}_p$, and returns the secret key $\mathsf{sk} := (x, y)$ and the public key $\mathsf{pk} := (g, Y, \tilde{g}, \tilde{X}, \tilde{Y})$ with $Y := g^y$, $\tilde{X} := \tilde{g}^x$ and $\tilde{Y} := \tilde{g}^y$.
3. $\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}, m)$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, secret key $\mathsf{sk} = (x, y)$ and message $m \in \mathbb{Z}_p$, Sign chooses a generator $h \leftarrow \mathbb{G}_1 \setminus \{1\}$ and returns the signature $\sigma := (h, h^{x+ym})$.
4. $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{pk}, m, \sigma)$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, public key $\mathsf{pk} = (g, Y, \tilde{g}, \tilde{X}, \tilde{Y})$, message $m \in \mathbb{Z}_p$ and signature $\sigma = (\sigma_1, \sigma_2)$, Vrfy checks whether the equations $\sigma_1 \neq 1$ and $e(\sigma_1, \tilde{X} \cdot \tilde{Y}^m) = e(\sigma_2, \tilde{g})$ hold. If both equations hold, it returns 1, and 0 otherwise.

One might notice that g and Y contained in the public key are not used. The reason for including these becomes clearer when we consider the protocols in our constructions. For now we stress that the scheme can be defined without g and Y (as it was originally done). As a consequence, the security then would rely on Assumption 2 instead of Assumption 1 as in our case.

Next, we shall check whether scheme Π_S defined above is a correct signature scheme.

Lemma 3.15. *The signature scheme Π_S defined in Construction 3.14 is correct.*

Proof. Let G be a type 3 bilinear group generator and $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \in [\mathsf{Setup}(1^n)]$. Let $(\mathsf{pk}, \mathsf{sk}) \in [\mathsf{Gen}(\mathsf{pp})]$. Further, let $m \in \mathbb{Z}_p$ and $\sigma \in [\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}, m)]$. Then, it holds by definition that $\mathsf{sk} = (x, y) \in \mathbb{Z}_p^2$ and $\sigma = (h, h^{x+ym})$ with $h \in \mathbb{G}_1 \setminus \{1\}$. Therefore, we only need to check the verification equation using the pairing e:

$$e(h, \tilde{X} \cdot \tilde{Y}^m) = e(h, \tilde{g}^x \cdot \tilde{g}^{my}) = e(h, \tilde{g}^{x+my}) = e(h, \tilde{g})^{x+my} = e(h^{x+my}, \tilde{g}).$$

Therefore, the verification outputs 1 for every honestly generated signature. □

Before we go over to the security analysis of Construction 3.14, we point out an interesting property of the scheme that becomes important later. Namely, signatures of the PS signature scheme are *randomizable*:

Remark (Randomizability). Let $\sigma = (\sigma_1, \sigma_2)$ be a PS signature on some message m. We can randomize σ by choosing $\eta \leftarrow \mathbb{Z}_p^*$ and setting $\sigma' := (\sigma_1^\eta, \sigma_2^\eta)$. Note that σ' still is a valid signature on message m. Indeed, it is distributed like a fresh signature on m, which allows to hide the original signature in a certain way.

Let us proceed with the security analysis of the single-message PS signature scheme Construction 3.14. Having a look at Assumption 1 (Definition 3.7), one easily sees that the existential unforgeability of Π_S is given by the hardness of this assumption. However, we show this formally by proving Theorem 3.16.

Theorem 3.16 (Security of Π_S)**.** *Let G be a type 3 bilinear group generator. Under Assumption 1 (Definition 3.7) for G, the signature scheme Π_S is existentially unforgeable under an adaptive chosen message attack.*

Proof. Assume that Assumption 1 (Definition 3.7) holds. We need to show that the success probability of any ppt algorithm that tries to forge signatures of Π_S grows smaller than a negligible function. Therefore, fix an arbitrary ppt forger $\mathcal{F}$ such that the success probability of $\mathcal{F}$ is $\epsilon_{\mathcal{F}}(n) := \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{F},\Pi_S}(n) = 1]$. Based on $\mathcal{F}$, construct adversary $\mathcal{A}$ that tries to break Assumption 1. Consider the following construction:

$\mathcal{A}$ on input $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, Y, \tilde{g}, \tilde{X}, \tilde{Y})$ and oracle access to $O_{x,y}(\cdot)$:

1. Set $\mathsf{pk} := (g, Y, \tilde{g}, \tilde{X}, \tilde{Y})$ and $\mathsf{pp} := (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$
2. Simulate $\mathcal{F}$ on input pp and pk. Whenever, it queries its signing oracle with message $m \in \mathbb{Z}_p$:
   a) Set $(\sigma_1, \sigma_2) \leftarrow O_{x,y}(m)$
   b) Return (σ_1, σ_2) to $\mathcal{F}$
3. If $\mathcal{F}$ outputs a forgery attempt (m^*, σ^*), output (m^*, σ^*).

First of all, note that $\mathcal{A}$ is a ppt algorithm. Adversary $\mathcal{A}$ only simulates forger $\mathcal{F}$; if $t(n)$ upper bounds the running time of $\mathcal{F}$ and $q(n)$ upper bounds the number of queries, for some polynomials t and q, the running time of $\mathcal{A}$ is upper bounded by $t(n) + q(n) \cdot \mathcal{O}(1)$, which is a polynomial.

We relate the success probability of $\mathcal{A}$ in breaking Assumption 1 and the success probability of forger $\mathcal{F}$ in forging a signature of Π_S. First, we show that the answers of $\mathcal{A}$'s oracle queries returned by $\mathcal{F}$ are distributed identically to the outputs of the signing algorithm. $\mathcal{A}$ is an adversary against Assumption 1. Therefore, it is provided with the oracle $O_{x,y}(\cdot)$ for $x, y \in \mathbb{Z}$ defined in Definition 3.7. By definition, oracle $O_{x,y}(\cdot)$ outputs, on input $m \in \mathbb{Z}_p$, the pair $\sigma = (h, h^{x+ym})$ for some uniformly chosen generator h of $\mathbb{G}_1$. Having a look at the definition of the signing algorithm in Construction 3.14, we observe that $\mathcal{A}$'s oracle $O_{x,y}(\cdot)$ within Assumption 1 and $\mathcal{F}$'s signing oracle $\mathsf{Sign}(\mathsf{pp}, \mathsf{sk}, \cdot)$ from experiment $\mathsf{Sig\text{-}forge}_{\mathcal{F},\Pi_S}(n)$, are perfectly equivalent.

To summarize, adversary $\mathcal{A}$ perfectly simulates the $\mathsf{Sig\text{-}forge}_{\mathcal{F},\Pi_S}(n)$ game. It remains to argue that adversary $\mathcal{A}$ outputs a valid pair to break Assumption 1 if and only if forger $\mathcal{F}$ outputs a valid signature under pk defined in the construction of $\mathcal{A}$. If $(m^*, \sigma^*) = (m^*, (\sigma_1^*, \sigma_2^*))$ is valid under pk, it holds that $e(\sigma_1^*, \tilde{X} \cdot \tilde{Y}^{m^*}) = e(\sigma_2^*, \tilde{g})$ and $\sigma_1^* \neq 1$. This is only satisfied by the pair $(\sigma_1^*, {\sigma_1^*}^{x+y\,m^*})$ for $x := \log_{\tilde{g}}(\tilde{X})$ and $y := \log_{\tilde{g}}(\tilde{Y})$, which is a valid pair to break Assumption 1. Additionally, note that the set of oracle queries is the same for $\mathcal{A}$ and $\mathcal{F}$. Concretely, if $\mathcal{F}$ outputs an already queried message and loses, $\mathcal{A}$ would have lost in the real experiment as well. Formally, we have that $\mathcal{A}$ succeeds in breaking Assumption 1 with the same probability as $\mathcal{F}$ in forging signatures of Π_S, i. e.

$$\Pr\Big[(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n), x \leftarrow \mathbb{Z}_p, y \leftarrow \mathbb{Z}_p, g \leftarrow \mathbb{G}_1 \setminus \{1\}, \tilde{g} \leftarrow \mathbb{G}_2 \setminus \{1\}, Y := g^y,$$
$$\tilde{X} := \tilde{g}^x, \tilde{Y} := \tilde{g}^y, (m, (\sigma_1, \sigma_2)) \leftarrow \mathcal{A}^{O_{x,y}(\cdot)}(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, Y, \tilde{g}, \tilde{X}, \tilde{Y}) :$$
$$m \notin Q \wedge m \in \mathbb{Z}_p \wedge \sigma_1 \in \mathbb{G}_1 \setminus \{1\} \wedge \sigma_2 = \sigma_1^{x+y\,m}\Big] = \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{F},\Pi_S}(n) = 1]$$

Since we assume that Assumption 1 holds and adversary $\mathcal{A}$ is a ppt algorithm, we have that

$$\Pr[\mathsf{Sig\text{-}forge}_{\mathcal{F},\Pi_S}(n) = 1] = \epsilon_{\mathcal{A}}(n)$$

where $\epsilon_{\mathcal{A}}(n)$ is negligible. Overall, we have that $\epsilon_{\mathcal{A}}(n) = \epsilon_{\mathcal{F}}(n)$, which in turn means that $\epsilon_{\mathcal{F}}(n)$ is negligible. □

3.4.1.2 Multi-Message Signatures

As mentioned above a credential is a signature. To be more precise it is a signature on the user's secret and her attributes. Therefore, it does not suffice to only sign single messages. The PS signature scheme also offers a multi-message variant to sign blocks of messages.

Construction 3.17 (Multi-Message Pointcheval-Sanders Signature Scheme)**.** Let G be a type 3 bilinear group generator (Definition 3.2). Let $\ell \in \mathbb{N}$. The *multi-message Pointcheval-Sanders (PS) signature scheme* is defined as $\Pi_M = (\mathsf{Setup}_\ell, \mathsf{Gen}_\ell, \mathsf{Sign}_\ell, \mathsf{Vrfy}_\ell)$, where

1. $\mathsf{Setup}_\ell(1^n)$: On input security parameter 1^n, Setup_ℓ obtains $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n)$ and returns $\mathsf{pp} \coloneqq (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$.

2. $\mathsf{Gen}_\ell(\mathsf{pp})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, Gen_ℓ chooses generators $g \leftarrow \mathbb{G}_1 \setminus \{1\}$ and $\tilde{g} \leftarrow \mathbb{G}_2 \setminus \{1\}$. It then chooses $(x, y_1, \ldots, y_\ell) \leftarrow \mathbb{Z}_p^{\ell+1}$, and returns secret key $\mathsf{sk} \coloneqq (x, y_1, \ldots, y_\ell)$ and public key $\mathsf{pk} \coloneqq (g, Y_1, \ldots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell)$ with $\tilde{X} \coloneqq \tilde{g}^x$, $Y_i \coloneqq g^{y_i}$ and $\tilde{Y}_i \coloneqq \tilde{g}^{y_i}$ for $i = 1, \ldots, \ell$.

3. $\mathsf{Sign}_\ell(\mathsf{pp}, \mathsf{sk}, m_1, \ldots, m_\ell)$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, secret key $\mathsf{sk} = (x, y_1, \ldots, y_\ell)$ and messages $(m_1, \ldots, m_\ell) \in \mathbb{Z}_p^\ell$, Sign_ℓ chooses a generator $h \leftarrow \mathbb{G}_1 \setminus \{1\}$ and returns the signature $\sigma \coloneqq (h, h^{x + \sum_{i=1}^{\ell} y_i \cdot m_i})$.

4. $\mathsf{Vrfy}_\ell(\mathsf{pp}, \mathsf{pk}, m_1, \ldots, m_\ell, \sigma)$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, public key $\mathsf{pk} = (g, Y_1, \ldots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell)$, messages $(m_1, \ldots, m_\ell) \in \mathbb{Z}_p^\ell$ and signature $\sigma = (\sigma_1, \sigma_2)$, Vrfy_ℓ checks whether the equations $\sigma_1 \neq 1$ and $e\Big(\sigma_1, \tilde{X} \cdot \prod_{i=1}^{\ell} \tilde{Y}_i^{m_i}\Big) = e(\sigma_2, \tilde{g})$ hold. If both equations hold, it returns 1, and 0 otherwise.

Lemma 3.18. *The signature scheme Π_M defined in Construction 3.17 is correct.*

Proof. Let G be a type 3 bilinear group generator, $n \in \mathbb{N}$ and $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \in [\mathsf{Setup}_\ell(1^n)]$. Let $(\mathsf{pk}, \mathsf{sk}) \in [\mathsf{Gen}(\mathsf{pp})]$. Further, let $(m_1, \ldots, m_\ell) \in \mathbb{Z}_p^\ell$ and $\sigma \in [\mathsf{Sign}_\ell(\mathsf{pp}, \mathsf{sk}, m_1, \ldots, m_\ell)]$. Then, it holds by definition that $\sigma = (h, h^{x + \sum_{i=1}^{\ell} y_i \, m_i})$ with $h \in \mathbb{G}_1 \setminus \{1\}$ and $\mathsf{sk} = (x, y_1, \ldots, y_\ell) \in \mathbb{Z}_p^{\ell+1}$. Since σ is an honestly generated signature on $m_1, \ldots, m_\ell$, it holds by definition that $h \neq 1$. Therefore, we only need to check the verification equation using the pairing e:

$$
\begin{aligned}
e(h, \tilde{X} \cdot \textstyle\prod_{i=1}^{\ell} \tilde{Y}_i^{m_i}) = e(h, \tilde{g}^x \cdot \prod_{i=1}^{\ell} \tilde{g}^{y_i m_i}) &= e(h, \tilde{g}^{x + \sum_{i=1}^{\ell} y_i m_i}) \\
= e(h, \tilde{g})^{x + \sum_{i=1}^{\ell} y_i m_i} &= e(h^{x + \sum_{i=1}^{\ell} y_i m_i}, \tilde{g}).
\end{aligned}
$$

Therefore, the verification outputs 1 for every honestly generated signature. □

The security of this scheme relies on the security of the Single-Message Pointcheval-Sanders Signature Scheme shown in the proof of Theorem 3.16. Therefore, it implicitly relies on Assumption 1 following Definition 3.7.

Theorem 3.19 (Security of Π_M)**.** *Let G be a type 3 bilinear group generator. If Π_S defined in Construction 3.14 is existentially unforgeable under an adaptive chosen-message attack relative to G, then Π_M defined in Construction 3.17 is existentially unforgeable under an adaptive chosen-message attack relative to G.*

Proof. Assume that scheme Π_S (Construction 3.14) is secure. Fix an arbitrary ppt forger $\mathcal{F}_M$ such that $\Pr[\mathsf{Sig\text{-}forge}_{\mathcal{F}_M, \Pi_M}(n) = 1] = \epsilon_M(n)$. Based on $\mathcal{F}_M$, construct the following forger $\mathcal{F}_S$ for the single-message scheme Π_S:

Forger $\mathcal{F}_S$ on input pp and pk and oracle access to $O\colon \mathbb{Z}_p \to \mathbb{G}_1^2$:

1. Parse pp as $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and pk as $(g, Y, \tilde{g}, \tilde{X}, \tilde{Y})$.
2. Pick $\alpha_i, \beta_i \leftarrow \mathbb{Z}_p$ for $i = 1, \ldots, \ell$.
3. Set $\tilde{Y}_i \coloneqq \tilde{Y}^{\alpha_i} \cdot \tilde{g}^{\beta_i}$ and $Y_i \coloneqq Y^{\alpha_i} \cdot g^{\beta_i}$ for $i = 1, \ldots, \ell$.
4. Simulate $\mathcal{F}_M$ on input pp and $\mathsf{pk}_M \coloneqq (g, Y_1, \ldots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_1, \ldots, \tilde{Y}_\ell)$. Whenever $\mathcal{F}_M$ queries its oracle with $(m_1, \ldots, m_\ell) \in \mathbb{Z}_p^\ell$:
   a) Query O on message $\sum_{i=1}^{\ell} \alpha_i\, m_i$ to obtain (σ_1, σ_2).
   b) Return $(\sigma_1, \sigma_2 \cdot \sigma_1^{\sum_{i=1}^{\ell} \beta_i\, m_i})$.
5. When $\mathcal{F}_M$ outputs a forgery $((m_1^*, \ldots, m_\ell^*), (\sigma_1^*, \sigma_2^*))$:
   a) Output $\big(\sum_{i=1}^{\ell} \alpha_i\, m_i^*, (\sigma_1^*, \sigma_2^* / \sigma_1^{*\sum_{i=1}^{\ell} \beta_i\, m_i})\big)$.

First, note that $\mathcal{F}_S$ is a ppt algorithm. It simulates forger $\mathcal{F}_M$, which is a ppt algorithm. Related to this, $\mathcal{F}_S$ needs to answers at most polynomially many oracle queries of $\mathcal{F}_M$. Each of these makes use of the group operations and exponentiation, which are feasible in polynomial time in the security parameter n. Same holds for outputting the forgery.

We analyze the success probability of forger $\mathcal{F}_S$ in winning the game $\mathsf{Sig\text{-}forge}_{\mathcal{F}_S,\Pi_S}(n)$. Therefore we first analyze whether it correctly simulates the game $\mathsf{Sig\text{-}forge}_{\mathcal{F}_M,\Pi_M}(n)$. Note, $\mathcal{F}_S$ needs to derive a public key pk_M for the multi-message scheme based on the public key pk it obtains as input. Important is that the derived key looks like a key generated by Π_M's key generation algorithm Gen_ℓ, i. e. the public key pk_M derived in steps 2–4 is distributed like a public key of Π_M. As $\mathcal{F}_S$ reuses the bilinear group pp, the generators g, $\tilde{g}$ and $\tilde{X}$ from his public key pk, we only need to check whether Y_j and $\tilde{Y}_j$ for $j = 1, \ldots, \ell$ are distributed correctly. So, let $\mathsf{sk} \coloneqq (x, y) \in \mathbb{Z}_p^2$ be the secret key corresponding to pk, and let $Y = g^y$ and $\tilde{Y} = \tilde{g}^y$. Step 3 of forger $\mathcal{F}_S$ implicitly defines the multiple-message secret key $\mathsf{sk}_M \coloneqq (x, (\alpha_i\, y + \beta_i)_{i=1,\ldots,\ell})$. Since α_i and β_i are chosen uniformly and independently from $\mathbb{Z}_p$ for every $i = 1, \ldots, \ell$, this means that $\alpha_i\, y + \beta_i$ is distributed uniformly and independently on $\mathbb{Z}_p$. This implies that $\tilde{Y}_i = \tilde{Y}^{\alpha_i} \cdot \tilde{g}^{\beta_i} = \tilde{g}^{\alpha_i\, y + \beta_i}$ are distributed uniformly and independently on $\mathbb{G}_2$. The same holds for Y_i by substituting $\tilde{g}$ by g and group $\mathbb{G}_2$ by $\mathbb{G}_1$. Therefore, the public key pk_M that $\mathcal{F}_M$ gets as input is distributed correctly, and the discrete logarithms of Y_i and $\tilde{Y}_i$ to base g and $\tilde{g}$, respectively, are the same.

Besides $\mathcal{F}_M$'s inputs, the answers to its oracle queries need to be distributed correctly as well. Let $\mathsf{sk} = (x, y)$ again denote the secret key corresponding to the public key pk which $\mathcal{F}_S$ gets as input. On query $\sum_{i=1}^{\ell} \alpha_i\, m_i$, forger $\mathcal{F}_S$ gets a signature (σ_1, σ_2) in return that is valid for the queried message. If (x, y) is the secret key, we have that σ_1 is distributed uniformly on $\mathbb{G}_1 \setminus \{1\}$ and $\sigma_2 = \sigma_1^{x + y \cdot \sum_{i=1}^{\ell} \alpha_i\, m_i}$ according to the single-message signing algorithm under secret key (x, y). Recall that step 3 of forger $\mathcal{F}_S$ implicitly defines the multiple-message secret key $sk_M = (x, (\alpha_i\, y + \beta_i)_{i=1,\ldots,\ell})$. Therefore, a valid signature on messages $(m_i)_{i=1,\ldots,\ell}$ under that secret key would look like the following: (σ_1, σ_2) such that σ_1 is distributed uniformly on $\mathbb{G}_1 \setminus \{1\}$ and $\sigma_2 = \sigma_1^{x + \sum_{i=1}^{\ell} (\alpha_i\, y + \beta_i)\, m_i}$. Now, Forger $\mathcal{F}_S$ returns $(\sigma_1, \sigma_2 \cdot \sigma_1^{\sum_{i=1}^{\ell} \beta_i\, m_i})$ as oracle answer to $\mathcal{F}_M$ such that (σ_1, σ_2) were obtained by querying $\mathcal{F}_S$'s oracle on message $\sum_{i=1}^{\ell} \alpha_i\, m_i$. Combining the considerations above, we get that σ_1 is distributed uniformly on $\mathbb{G}_1 \setminus \{1\}$ and

$$\sigma_2 \cdot \sigma_1^{\sum_{i=1}^{\ell} \beta_i\, m_i} = \sigma_1^{x + y \cdot \sum_{i=1}^{\ell} \alpha_i\, m_i} \cdot \sigma_1^{\sum_{i=1}^{\ell} \beta_i\, m_i} = \sigma_1^{x + \sum_{i=1}^{\ell} \alpha_i\, y\, m_i} \cdot \sigma_1^{\sum_{i=1}^{\ell} \beta_i\, m_i} = \sigma_1^{x + \sum_{i=1}^{\ell} (\alpha_i\, y + \beta_i)\, m_i}.$$

This means that the oracle simulated by $\mathcal{F}_S$ produces signatures distributed correctly according to the public key given to $\mathcal{F}_M$ as input.

In turn, this implies that $\mathcal{F}_S$ perfectly simulates the $\mathsf{Sig\text{-}forge}_{\mathcal{F}_M,\Pi_M}(n)$ game. Let us have a look at the processing of the forgery $((m_1^*,\dots,m_\ell^*),(\sigma_1^*,\sigma_2^*))$ output by $\mathcal{F}_M$ in step 5. We show, that if the forgery output by $\mathcal{F}_M$ is a valid signature under the public key $\mathcal{F}_M$ got as input, then the forgery output by $\mathcal{F}_S$ is a valid signature under pk. If (σ_1^*,σ_2^*) is a valid signature on $(m_1^*,\dots,m_\ell^*)$, we have that $\sigma^* \neq 1$ and $e(\sigma_1^*, \tilde{X}\cdot\prod_{j=1}^{\ell}\tilde{Y}_j^{m_j^*}) = e(\sigma_2^*,\tilde{g})$. It remains to show that for forgery $(\sum_{i=1}^{\ell}\alpha_i\, m_i^*, (\sigma_1^*, \sigma_2^*/\sigma_1^{*\sum_{i=1}^{\ell}\beta_i\, m_i^*}))$ it holds that $e(\sigma_1^*, \tilde{X}\cdot\tilde{Y}^{\sum_{i=1}^{\ell}\alpha_i\, m_i^*}) = e(\sigma_2^*/\sigma_1^{*\sum_{i=1}^{\ell}\beta_i\, m_i^*}, \tilde{g})$. Therefore we compute

$$
\begin{aligned}
e(\sigma_2^*/\sigma_1^{*\sum_{i=1}^{\ell}\beta_i\, m_i^*}, \tilde{g}) &= e(\sigma_2^*,\tilde{g})\cdot e(\sigma_1^{*-\sum_{i=1}^{\ell}\beta_i\, m_i^*},\tilde{g}) = e(\sigma_1^*, \tilde{X}\cdot\prod_{j=1}^{\ell}\tilde{Y}_j^{m_j^*})\cdot e(\sigma_1^{*-\sum_{i=1}^{\ell}\beta_i\, m_i^*},\tilde{g})\\
&= e(\sigma_1^*, \tilde{X}\cdot\prod_{j=1}^{\ell}\tilde{Y}_j^{m_j^*})\cdot(\sigma_1^*, \tilde{g}^{-\sum_{i=1}^{\ell}\beta_i\, m_i^*})\\
&= e(\sigma_1^*, \tilde{X}\cdot\prod_{j=1}^{\ell}(\tilde{Y}^{\alpha_j}\,\tilde{g}^{\beta_j})^{m_j^*})\cdot(\sigma_1^*, \tilde{g}^{-\sum_{i=1}^{\ell}\beta_i\, m_i^*})\\
&= e(\sigma_1^*, \tilde{X}\cdot\prod_{j=1}^{\ell}\tilde{Y}^{\alpha_j\, m_j^*}\cdot\tilde{g}^{\sum_{i=1}^{\ell}\beta_i\, m_i^*})\cdot(\sigma_1^*, \tilde{g}^{-\sum_{i=1}^{\ell}\beta_i\, m_i^*})\\
&= e(\sigma_1^*, \tilde{X}\cdot\tilde{Y}^{\sum_{j=1}^{\ell}\alpha_j\, m_j^*}).
\end{aligned}
$$

Thus, $\mathcal{F}_S$ outputs a valid forgery, if $\mathcal{F}_M$ does. Note that it may happen that $\sum_{i=1}^{\ell}\alpha_i\, m_i^* = \sum_{i=1}^{\ell}\alpha_i\, m_i$ holds for some messages $(m_1,\dots,m_\ell)$ queried by $\mathcal{F}_M$ during its execution. In this case, $\mathcal{F}_S$ would have already queried its oracle with the message $\sum_{i=1}^{\ell}\alpha_i\, m_i^*$ and thus, the $\mathcal{F}_S$ would lose the game $\mathsf{Sig\text{-}forge}_{\mathcal{F}_S,\Pi_S}(n)$, although the forgery was valid. We analyze the probability that this event occurs more formally. Therefore, fix an arbitrary type 3 bilinear group $(p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e)$, arbitrary generators g and $\tilde{g}$ of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, arbitrary $(x,y)\in\mathbb{Z}_p^2$, $(Y,\tilde{X},\tilde{Y}) := (g^y,\tilde{g}^x,\tilde{g}^y)$. Let $Q_M\subseteq\mathbb{Z}_p^\ell$ be the set of queries made by $\mathcal{F}_M$ during its execution and let $q(\cdot)$ be a polynomial such that $|Q_M|\le q(n)$. Note that the Q_S, the set of queries made by $\mathcal{F}_S$, is defined dependent on Q_M. That is, $Q_S := \{\hat{m}\in\mathbb{Z}_p \mid (m_1,\dots,m_\ell)\in Q_m \wedge \hat{m} = \sum_{i=1}^{\ell}\alpha_i\, m_i\}$ with $|Q_S|\le q(n)$. Further, let $m^{(i)} = (m_1^{(i)},\dots,m_\ell^{(i)})\in\mathbb{Z}_p^\ell$ be arbitrary numbers for $i=1,\dots,q(n)$ and let $\sigma^{(i)} = (\sigma_1^{(i)},\sigma_2^{(i)})\in\mathbb{G}_1^2$ arbitrary. Then, View denotes the event such that:

1. $\alpha_i,\beta_i\leftarrow\mathbb{Z}_p$ for $i=1,\dots,\ell$.
2. The input of $\mathcal{F}_M$ is $(p,\mathbb{G}_1,\mathbb{G}_2,\mathbb{G}_T,e)$ and $(g,(Y^{\alpha_i}\,g^{\beta_i})_{i=1,\dots,\ell},\tilde{g},\tilde{X},(\tilde{Y}^{\alpha_i}\,\tilde{g}^{\beta_i})_{i=1,\dots,\ell})$.
3. $Q_M = \{m^{(1)},\dots,m^{(q(n))}\}$, $Q_S = \{\hat{m}_i \mid 1\le i\le q(n)\wedge m^{(i)}\in Q_M\wedge\hat{m}_i = \sum_{j=1}^{\ell}\alpha_j\, m_j^{(i)}\}$.
4. $\sigma^{(i)}$ is the answer to oracle query $m^{(i)}$ for $i=1,\dots,\ell$.

Let $(m^*,(\sigma_1^*,\sigma_2^*))$ be the signature output by $\mathcal{F}_M$ and let $(\hat{m}^*,(\hat{\sigma}_1^*,\hat{\sigma}_2^*))$ the corresponding signature output by $\mathcal{F}_S$. Let fail denote the event that there exists an $i\in\{1,\dots,q(n)\}$ such that $\hat{m}_i\in Q_S$ and $\hat{m}^* = \hat{m}_i$. We have the following

$$\Pr[\mathsf{fail}] = \Pr\left[\bigvee_{j=1}^{q(n)}\hat{m}^* = \hat{m}_j\right].$$

Let us analyze the probability $\Pr[m^* = \hat{m}_j] = \Pr[\sum_{i=1}^{\ell}\alpha_i\, m_i^* = \sum_{i=1}^{\ell}\alpha_i\, m_i^{(j)}]$ for some $j\in\{1,\dots,q(n)\}$. We claim that the view of forger $\mathcal{F}_M$ is independent of the choices of α_i and thus

reveals no information about them. Therefore, let $a_j \in \mathbb{Z}_p$ be arbitrary but fixed for $j = 1, \ldots, \ell$. We show that for every $j = 1, \ldots, \ell$, it holds that $\Pr[\alpha_j = a_j \mid \mathsf{View}] = \Pr[\alpha_j = a_j]$. Obviously, $\Pr[\alpha_j = a_j] = \frac{1}{p}$ for all j. Let us now have a look at $\Pr[\alpha_j = a_j \mid \mathsf{View}]$. Recall that $\tilde{Y}_j = \tilde{Y}^{\alpha_j}\, \tilde{g}^{\beta_j}$ and $\tilde{Y} = \tilde{g}^y$. For every value that a_j could take, we can find a β'_j such that we get $\tilde{Y}_j$. We set $\beta'_j := \beta_j + y\,(\alpha_j - a_j)$, and get

$$\tilde{Y}^{a_j}\, \tilde{g}^{\beta'_j} = \tilde{g}^{y\, a_j}\, \tilde{g}^{\beta_j}\, \tilde{g}^{y\, \alpha_j}\, \tilde{g}^{-y\, a_j} = \tilde{g}^{\beta_j}\, \tilde{g}^{y\, \alpha_j} = \tilde{Y}_j.$$

This implies that the value of $\tilde{Y}_j$ is independent of α_j for all $j = 1, \ldots, \ell$, i. e.

$$\Pr[\alpha_j = a_j \mid \mathsf{View}] = \Pr[\alpha_j = a_j] = \frac{1}{p}.$$

Assume that forger $\mathcal{F}_M$ did not query its oracle on m^* before. Otherwise, it would loose immediately. As its view is independent from the α_j's, the choice of m^* is independent as well. First, we have for every $j \in 1, \ldots, q(n)$:

$$\Pr[\hat{m}^* = \hat{m}_j] = \Pr\left[\sum_{i=1}^{\ell} \alpha_i\, m_i^* = \sum_{i=1}^{\ell} \alpha_i\, m_i^{(j)}\right] = \Pr\left[\sum_{i=1}^{\ell} \alpha_i (m_i^* - m_i^{(j)}) = 0\right].$$

By the assumption that m^* was not queried, we get that for all $j \in \{1, \ldots, q(n)\}$ there exists a $k \in \{1, \ldots, \ell\}$ such that $m_k^* \neq m_k^{(j)}$. The choice of the corresponding α_k determines whether the event $\sum_{i=1}^{\ell} \alpha_i (m_i^* - m_i^{(j)}) = 0$ occurs or not. Intuitively, suppose that all α_i apart from α_k are already fixed. Then, there is exactly one value for α_k in $\mathbb{Z}_p$ such that $\alpha_k (m_k^* - m_k^{(j)})$ is the additive inverse of the rest of the sum that is already fixed. If m^* and $m^{(j)}$ were equal at this position, this would not be the case. Formally, we have for all $j \in \{1, \ldots, q(n)\}$ there exists a $k \in \{1, \ldots, \ell\}$ such that

$$\Pr\left[\sum_{i=1}^{\ell} \alpha_i (m_i^* - m_i^{(j)}) = 0\right] = \Pr\left[\sum_{i \neq k} \alpha_i (m_i^* - m_i^{(j)}) + \alpha_k \underbrace{(m_k^* - m_k^{(j)})}_{\neq 0} = 0\right] = \frac{1}{p}.$$

This implies that for all $j \in \{1, \ldots, q(n)\}$, it holds $\Pr[\hat{m}^* = \hat{m}_j] = \frac{1}{p}$. By the union bound, we get

$$\Pr[\mathsf{fail}] = \Pr\left[\bigvee_{j=1}^{q(n)} \hat{m}^* = \hat{m}_j\right] \leq \sum_{j=1}^{q(n)} \Pr[\hat{m}^* = \hat{m}_j] = \frac{q(n)}{p}.$$

To conclude, we take all the considerations together. Since Forger $\mathcal{F}_S$ is a ppt algorithm and we assume Π_S to be secure, we have that $\Pr[\mathsf{Sig\text{-}forge}_{\mathcal{F}_S, \Pi_S}(n) = 1] = \epsilon_S(n)$, where $\epsilon_S(n)$ is a negligible function. Moreover, forger $\mathcal{F}_S$ outputs a valid signature, if $\mathcal{F}_M$ outputs a valid signature, since $\mathcal{F}_S$ simulates the game $\mathsf{Sig\text{-}forge}_{\mathcal{F}_M, \Pi_M}(n)$ perfectly. However, if the event fail occurs $\mathcal{F}_S$ will lose the game, because it outputs a signature of a message that it already queried its oracle with. Thus, we have

$$\begin{aligned}
\Pr[\mathsf{Sig\text{-}forge}_{\mathcal{F}_S, \Pi_S}(n) = 1] &= \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{F}_S, \Pi_S}(n) = 1 \wedge \mathsf{fail}] + \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{F}_S, \Pi_S}(n) = 1 \wedge \neg\mathsf{fail}] \\
&= \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{F}_S, \Pi_S}(n) = 1 \neg\mathsf{fail}] \cdot \Pr[\neg\mathsf{fail}] \\
&\qquad + \underbrace{\Pr[\mathsf{Sig\text{-}forge}_{\mathcal{F}_S, \Pi_S}(n) = 1 \mid \mathsf{fail}]}_{=0} \cdot \Pr[\mathsf{fail}] \\
&= \Pr[\mathsf{Sig\text{-}forge}_{\mathcal{F}_S, \Pi_S}(n) = 1 \mid \neg\mathsf{fail}] \cdot \Pr[\neg\mathsf{fail}] \\
&\geq \epsilon_M(n) \cdot \left(1 - \frac{q(n)}{p}\right) = \epsilon_M(n) - \epsilon_M(n) \cdot \frac{q(n)}{p}
\end{aligned}$$

$\mathsf{Binding}_{\mathfrak{C},\mathcal{A}}(n)$

1 : *Run* $\mathsf{Setup}(1^n)$ to obtain public parameter pp
2 : $\mathcal{A}$ is given pp and has to output a tuple (c, m_1, d_1, m_2, d_2)
3 : Output of the experiment is 1 if $\mathsf{Open}(\mathsf{pp}, c, d_1) = m_1$,
$\mathsf{Open}(\mathsf{pp}, c, d_2) = m_2, m_1 \neq m_2$ and $m_1, m_2 \neq \perp$.
Otherwise, the output is 0.

Figure 3.4: Computational Binding Experiment

$$\geq \epsilon_M(n) - \frac{q(n)}{p}.$$

Hence, $\epsilon_S(n) \geq \epsilon_M(n) - \frac{q(n)}{p} \iff \epsilon_S(n) + \frac{q(n)}{p} \geq \epsilon_M(n)$ holds. By definition of group generator G (Definition 3.2) the value of p is lower-bounded by 2^n, which implies that $\frac{1}{p}$ is negligible. Therefore, $\frac{q(n)}{p}$ is negligible as the product of a polynomial and a negligible function. Using that ϵ_S and $\frac{q(n)}{p}$ are negligible, we get that $\epsilon_S(n) + \frac{q(n)}{p}$ is negligible as well. Hence, $\epsilon_M(n)$ is upper-bounded by a negligible function and thus negligible. □

3.5 Commitment Schemes

In this section, we define non-interactive commitment schemes and their possible security properties. Commitment schemes will be used to form pseudonyms in our anonymous credential system (cf. Construction 4.25). By doing that, the user can hide her secret key with the commitment while also allowing the user to create multiple different pseudonyms with different random choices for the commitment.

Definition 3.20. A non-interactive *commitment scheme* $\mathfrak{C}$ is a triple of ppt algorithms $(\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$, where

1. $\mathsf{Setup}(1^n)$: On input security parameter 1^n, it outputs the public parameter pp with $|\mathsf{pp}| \geq n$. It also defines a message space $\mathbb{M}$.
2. $\mathsf{Com}(\mathsf{pp}, m)$: On input public parameter pp and a message $m \in \mathbb{M}$, it outputs a pair (c, d) of commitment c and open value d.
3. $\mathsf{Open}(\mathsf{pp}, c, d)$: On input public parameter pp, a commitment c and an open value d it deterministically outputs m or failure symbol $\perp$.

We will call a commitment scheme correct if for every sufficiently large security parameter n and for all $\mathsf{pp} \in [\mathsf{Setup}(1^n)]$, it holds that for all messages $m \in \mathbb{M}$

$$\Pr[\mathsf{Open}(\mathsf{pp}, \mathsf{Com}(\mathsf{pp}, m)) = m] = 1$$

where the probability is taken over the coin tosses of Com and the inputs' distribution.

Now that we defined non-interactive commitment schemes, we want to look at their security properties.

Definition 3.21 (Computational Binding)**.** Let $\mathfrak{C} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be a commitment scheme. We say the commitment scheme $\mathfrak{C}$ is *computationally binding* if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function *negl* such that

$$\Pr[\mathsf{Binding}_{\mathfrak{C},\mathcal{A}}(n) = 1] \leq negl(n),$$

where experiment $\mathsf{Binding}_{\mathfrak{C},\mathcal{A}}(n)$ is defined in Figure 3.4.

In other words this property states, a commitment cannot efficiently be opened to two different values. One speaks of perfectly binding, if even unrestricted adversaries cannot open commitments to two different values.

Definition 3.22 (Perfect Hiding)**.** Let $\mathfrak{C} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be a commitment scheme. Assume that $\mathsf{Setup}(1^n)$ generated public parameters pp for some $n \in \mathbb{N}$. For messages $m_1, m_2 \in \mathbb{M}$, we define the random variables $C(m_1)$ and $C(m_2)$ describing the output of $\mathsf{Com}(\mathsf{pp}, m_1)$ and $\mathsf{Com}(\mathsf{pp}, m_2)$, respectively, where only the first output is considered. Then, $\mathfrak{C}$ is *perfectly hiding* if for all $\mathsf{pp} \in [\mathsf{Setup}(1^n)]$ and all $m_1, m_2 \in \mathbb{M}$ and for all $c \in [\mathsf{Com}(\mathsf{pp}, m_1)] \cup [\mathsf{Com}(\mathsf{pp}, m_2)]$ it holds

$$\Pr[C(m_1) = c] = \Pr[C(m_2) = c].$$

By the means of this definition, perfectly hiding commitment schemes do not allow the receiver to get any information about the message through the commitment. Even the knowledge of one bit could be harmful, since the receiver might be able to then gain even more information about the message through the setting or earlier commitments. A commitment scheme which is perfectly hiding allows a *user* to publish commitments without any concerns about their contents. The binding property ensures that no action after publishing a commitment affects its contents.

3.5.1 Generalized Pedersen Commitment Scheme

In this section, we define a commitment scheme, generalizing the prominent Pedersen Commitments [Ped92]. This generalization will enable us to commit to a fixed number of k messages. If we do not specify it any further, we implicitly assume the special case of $k = 1$. This creates the original scheme, which is often sufficient within our anonymous credential system. Since we want to use this scheme, we also prove its security properties.

Construction 3.23. Let $k \in \mathbb{N}$ be fixed and G be a group generator (Definition 3.3). The generalized Pedersen commitment scheme for k messages is then defined as a triple of ppt algorithms $(\mathsf{Setup}_k, \mathsf{Com}_k, \mathsf{Open}_k)$, such that:

1. $\mathsf{Setup}_k(1^n)$: On input security parameter 1^n, Setup_k runs the group generator $\mathsf{G}(1^n)$ and obtains $(p, \mathbb{G})$. Then, a *generator* g is chosen uniformly at random from $\mathbb{G} \setminus \{1\}$ and k elements $h_1, \dots, h_k$ are chosen uniformly at random from $\mathbb{G}$. The public parameters pp are then set to $(g, h_1, \dots, h_k, p, \mathbb{G})$. This implicitly sets the message space $\mathbb{M}$ to $\mathbb{Z}_p^k$ and Setup_k outputs pp.

2. $\mathsf{Com}_k(\mathsf{pp}, M)$: On input public parameters $\mathsf{pp} = (g, h_1, \dots, h_k, p, \mathbb{G})$ and message $M = (m_1, \dots, m_k) \in \mathbb{Z}_p^k$, Com_k chooses r uniformly at random from $\mathbb{Z}_p$ and computes the commitment $c \coloneqq g^r \prod_{i=1}^k h_i^{m_i}$. It sets the decommit message $d \coloneqq (M, r)$ and outputs (c, d).

3. $\mathsf{Open}_k(\mathsf{pp}, c, d)$: On input public parameters $\mathsf{pp} = (g, h_1, \dots, h_k, p, \mathbb{G})$, commitment c and decommit message $d = (M, r)$, Open_k computes $g^r \prod_{i=1}^k h_i^{m_i}$. If this equals c, it outputs M. Else it outputs $\perp$.

Theorem 3.24. *If the discrete logarithm problem (Figure 3.1) is hard relative to the group generation algorithm* G *and* $k \in \mathbb{N}$ *is fixed, the following properties hold for the generalized Pedersen Commitment Scheme:*

1. *Correctness as in Definition 3.20*

2. *Perfect hiding as in Definition 3.22*
3. *Computational binding as in Definition 3.21*

Theorem 3.24 follows from the three lemmas (Lemma 3.25, Lemma 3.26, Lemma 3.27) stated and proven below.

Lemma 3.25. *Let $k \in \mathbb{N}$ be fixed. The Pedersen commitment scheme (Construction 3.23) is a correct commitment scheme (Definition 3.20).*

Proof. We want to show that for all security parameters $n \in \mathbb{N}$, for all $k \in \mathbb{N}$, for all public parameters $\mathsf{pp} \in [\mathsf{Setup}_k(1^n)]$ it holds that

$$\forall M = (m_1, \ldots, m_k) \in \mathbb{M} : \Pr[\mathsf{Open}_k(\mathsf{pp}, \mathsf{Com}(\mathsf{pp}, M)) = M] = 1.$$

Let $(c, d) \in [\mathsf{Com}(\mathsf{pp}, M)]$ with $c = g^r \prod_{i=1}^k h_i^{m_i}$, $d = (M, r)$. Open_k then checks if $c = g^r \prod_{i=1}^k h_i^{m_i}$, which leads to the output being M. □

Lemma 3.26. *Let $k \in \mathbb{N}$ be fixed. The Pedersen commitment scheme (Construction 3.23) is perfectly hiding (Definition 3.22).*

Proof. We prove that the scheme is perfectly hiding by showing that for all security parameters $n \in \mathbb{N}$, all $k \in \mathbb{N}$ and for all public parameters $\mathsf{pp} \in [\mathsf{Setup}_k(1^n)]$, for all messages $M, M' \in \mathbb{M}$ with corresponding random variables $C(M)$ and $C(M')$ as defined in Definition 3.22 and for all commitments $c \in [\mathsf{Com}_k(\mathsf{pp}, M)] \cup [\mathsf{Com}_k(\mathsf{pp}, M')]$ it holds that

$$\Pr[C(M) = c] = \Pr[C(M') = c].$$

We achieve this by proving that for all messages $M \in \mathbb{M}$, $C(M)$ is distributed identically to the uniform distribution on $\mathbb{G}$. If this is the case, the distribution is identical for every message $M = (m_1, \ldots, m_k)$ and hence the property holds.
Let $c \in \mathbb{G}$, then

$$\Pr[C(M) = c] = \Pr[r \leftarrow \mathbb{Z}_p : c = g^r \prod_{i=1}^{k} h_i^{m_i}] = \Pr[r \leftarrow \mathbb{Z}_p : g^r = c \cdot \prod_{i=1}^{k} h_i^{-m_i}] = \frac{1}{|\mathbb{G}|}$$

since g is a generator in $\mathbb{G}$ and hence there is exactly one $r \bmod |\mathbb{G}|$ such that $g^r = c \cdot \prod_{i=1}^k h_i^{-m_i}$. □

Lemma 3.27. *If the discrete logarithm problem (Figure 3.1) is hard relative to the group generation algorithm G and $k \in \mathbb{N}$ is fixed, then the Pedersen commitment scheme (Construction 3.23) is computational binding (Definition 3.21).*

Proof. Let $k \in \mathbb{N}$ be fixed and $\mathcal{A}$ be an arbitrary ppt adversary. Define ϵ as $\epsilon(n) \coloneqq \Pr[\mathsf{Binding}_{\mathfrak{C},\mathcal{A}}(n) = 1]$. Using $\mathcal{A}$ we construct an adversary $\mathcal{B}$ with $\Pr[\mathsf{DLog}_{\mathcal{B},\mathcal{G}}(n) = 1] \geq \frac{1}{k} \cdot \epsilon(n)$. In the following $[k]$ denotes the set $\{1, \ldots, k\}$. Define $\mathcal{B}$ such that:

1. $\mathcal{B}$ receives $(\mathbb{G}, p, g, h)$ from the challenger
2. $\mathcal{B}$ chooses $i^* \leftarrow [k]$ and sets $I \coloneqq [k] \setminus \{i^*\}$
3. $\mathcal{B}$ chooses for all $i \in I$ an $r_i \leftarrow \mathbb{Z}_p$ and sets $h_i \coloneqq g^{r_i}$, as well as $h_{i^*} \coloneqq h$
4. $\mathcal{B}$ runs $\mathcal{A}$ on input $\mathsf{pp} = (g, h_1, \ldots, h_k, p, \mathbb{G})$
5. Eventually, $\mathcal{A}$ outputs (c, M, d, M', d') with $M = (m_1, \ldots, m_k) \neq (m'_1, \ldots, m'_k) = M'$. If $\mathcal{A}$ outputs $m_{i^*} = m'_{i^*} \mod p$, $\mathcal{B}$ outputs $\perp$

6. $\mathcal{B}$ parses $d = (M, r)$, $d' = (M', r')$ and outputs $(m'_{i^*} - m_{i^*})^{-1} \cdot (r - r' + \sum_{i \in I} r_i(m_i - m'_i)) = x \mod |\mathbb{G}|$.

Analysis of $\mathcal{B}$:
If $\mathcal{B}$ does not output $\perp$ and $\mathcal{A}$ succeeds in $\mathsf{Binding}_{\mathfrak{C},\mathcal{A}}(n)$, then x is the discrete logarithm such that $g^x = h$ since

$$\begin{aligned}
& & g^r \prod_{i=1}^{k} h_i^{m_i} &= g^{r'} \prod_{i=1}^{k} h_i^{m'_i} \\
\Leftrightarrow & & g^r h_{i^*}^{m_{i^*}} \prod_{i \in I} g^{r_i m_i} &= g^{r'} h_{i^*}^{m'_{i^*}} \prod_{i \in I} g^{r_i m'_i} \\
\Leftrightarrow & & g^r h_{i^*}^{m_{i^*}} g^{\sum_{i \in I} r_i m_i} &= g^{r'} h_{i^*}^{m'_{i^*}} g^{\sum_{i \in I} r_i m'_i} \\
\Leftrightarrow & & g^{r-r'} g^{\sum_{i \in I} r_i(m_i - m'_i)} &= h^{m'_{i^*} - m_{i^*}} \\
\Leftrightarrow & & g^{(m'_{i^*} - m_{i^*})^{-1}(r - r' + \sum_{i \in I} r_i(m_i - m'_i))} &= h \\
\Leftrightarrow & & (m'_{i^*} - m_{i^*})^{-1}(r - r' + \sum_{i \in I} r_i(m_i - m'_i)) &= \log_g(h) = x \mod |\mathbb{G}|.
\end{aligned}$$

Note, if $\mathcal{B}$ does not output $\perp$ then $(m'_{i^*} - m_{i^*})^{-1}$ exists, since then $m'_{i^*} \neq m_{i^*}$ holds (step 5) and $m'_{i^*} - m_{i^*} \neq 0 \mod p$ is invertible in $\mathbb{Z}_p$. Furthermore, for any security parameter n, the public parameters pp of $\mathcal{B}$ are distributed *identical* to $\mathsf{Setup}_k(1^n)$ by the DLog setup (i. e. $(p, \mathbb{G}) \leftarrow \mathsf{G}(1^n)$, $g \leftarrow \mathbb{G} \setminus \{1\}$ and $h_{i^*} := h \leftarrow \mathbb{G}$ by $h = g^r$ with $r \leftarrow \mathbb{Z}_p$) and step 3 (i. e. $h_i \leftarrow \mathbb{G}$ by $h_i := g^{r_i}$ with $r_i \leftarrow \mathbb{Z}_p$ for all $i \in I$).

By the previous calculations $\mathcal{B}$ succeeds in the DLog game, if $\mathcal{A}$ wins $\mathsf{Binding}$ with the two different messages $M = (m_1, \dots, m_k), M' = (m'_1, \dots, m'_k)$, for which additionally $m_{i^*} \neq m'_{i^*}$ holds. Note, that under the condition $\mathsf{Binding}_{\mathfrak{C},\mathcal{A}}(n) = 1$, we know $m_i \neq m'_i$ holds for *some* $i \in [k]$. With μ defined as $\mu(n) := \Pr[\mathsf{DLog}_{\mathcal{B},\mathcal{G}}(n) = 1]$, we can compute

$$\begin{aligned}
\mu(n) =& \Pr[\mathsf{DLog}_{\mathcal{B},\mathcal{G}}(n) = 1] \\
=& \Pr[i^* \leftarrow [k] : \mathsf{Binding}_{\mathfrak{C},\mathcal{A}}(n) = 1 \wedge m_{i^*} \neq m'_{i^*}] \\
=& \Pr[i^* \leftarrow [k] : m_{i^*} \neq m'_{i^*} | \mathsf{Binding}_{\mathfrak{C},\mathcal{A}}(n) = 1] \Pr[\mathsf{Binding}_{\mathfrak{C},\mathcal{A}}(n) = 1] \\
\geq& \frac{1}{k} \cdot \Pr[\mathsf{Binding}_{\mathfrak{C},\mathcal{A}}(n) = 1] \\
=& \frac{1}{k} \cdot \epsilon(n).
\end{aligned}$$

The inequality holds since the view of $\mathcal{A}$ is distributed *independently* of the uniform choice of i^*. Hence, if $\mathcal{A}$ wins, the message vectors differ *at least* on one of the k positions, independent of i^*. By the uniform choice of i^* the probability they differ at position i^* is at least $1/k$. Equivalently $\epsilon(n) \leq k \cdot \mu(n)$ with k polynomial in n holds, since k is fixed. Assuming the discrete logarithm problem is hard relative to the group generator G, $k \cdot \mu$ is a negligible upper bound of ϵ. Hence, as $\mathcal{A}$ was an arbitrary ppt, no ppt algorithm $\mathcal{A}$ wins $\mathsf{Binding}$ with more than negligible probability. Under this assumption the generalized Pedersen commitment scheme (Construction 3.23) is computationally binding. □

3.5.2 Trapdoor Commitment Schemes

There exists an extension of commitment schemes, where one can forgo the binding property (Definition 3.21), if she knows a special secret. This secret is called a trapdoor. With the trapdoor, one is able to output some commitment and open it to any message she wants. Without the trapdoor, the original binding property still holds. We need to introduce the notion of a trapdoor commitment scheme to be able to use Damgård's Technique (Section 3.9).

Definition 3.28 (Trapdoor Commitment Scheme)**.** A *trapdoor commitment scheme (TCS)* $\mathfrak{C}$ is a triple of ppt algorithms $(\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$, where

1. $\mathsf{Setup}(1^n)$: On input security parameter 1^n, it outputs the public parameter pp with $|\mathsf{pp}| \geq n$ and a trapdoor t. It also defines a message space $\mathbb{M}$.
2. $\mathsf{Com}(\mathsf{pp}, m)$: On input public parameter pp, and a message $m \in \mathbb{M}$ it outputs a pair (c, d) of commitment c and open value d.
3. $\mathsf{Open}(\mathsf{pp}, c, d)$: On input public parameter pp, a commitment c and an open value d it outputs m or failure symbol $\perp$.

Furthermore, the TCS must be correct, which is defined as in a standard commitment scheme (Definition 3.20), and it must have the *trapdoor property*:

There exists a tuple of ppt algorithms (T_1, T_2), such that for all m, all $(\mathsf{pp}, t) \in [\mathsf{Setup}(1^n)]$, all $(c, st) \in [T_1(\mathsf{pp}, t)]$ and $d \in [T_2(m, st)]$ we have that $\mathsf{Open}(\mathsf{pp}, c, d) = m$ and the commitments of $\mathsf{Com}(\mathsf{pp}, \cdot)$ and $T_1(\mathsf{pp}, \cdot)$ are distributed the same.

In other words, we can output a commitment with T_1 that is distributed as a normal commitment, but with the trapdoor and the state st we can open the commitment to any message we want.

3.5.3 Hash-Then-Commit

Until now, commitment schemes all had only a predefined message space, which may depend on the public parameters. But in some cases it may be beneficial to be able to commit to any bit string. Thus, we want to create a scheme that is able to do exactly that.

Construction 3.29. Let $\mathfrak{C} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be a commitment scheme with message space $M(\cdot)$. Let $H = (\mathsf{Setup}, \mathsf{Gen}_h, h)$ be a hash function that maps to $M(\cdot)$ (Theorem 3.10). Construct a tuple of ppt algorithms $\mathfrak{C}'_{\mathfrak{C},H} = (\mathsf{Setup}', \mathsf{Com}', \mathsf{Open}')$ as follows:

- $\mathsf{Setup}'(1^n)$: Choose $\mathsf{pp} \leftarrow \mathsf{Setup}(1^n)$ and $k \leftarrow \mathsf{Gen}_h(\mathsf{pp})$. Output $\mathsf{pp}' = (\mathsf{pp}, k)$.
- $\mathsf{Com}'(\mathsf{pp}', m)$: Parse $(\mathsf{pp}, k) = \mathsf{pp}'$. Compute $(com, d) \leftarrow \mathsf{Com}(\mathsf{pp}, h_k(m))$. Output $(com, (m, d))$.
- $\mathsf{Open}'(\mathsf{pp}', com, d')$: Parse $(\mathsf{pp}, k) = \mathsf{pp}'$ and $(m, d) = d'$. Compute $y \leftarrow \mathsf{Open}(\mathsf{pp}, com, d)$. If $h_k(m) = y$, output m.

Theorem 3.30. *Let $\mathfrak{C} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be a commitment scheme with message space $M(\cdot)$. Let $H = (\mathsf{Setup}, \mathsf{Gen}_h, h)$ be a hash function that maps to $M(\cdot)$ (Theorem 3.10). If $\mathfrak{C}$ is computationally binding and perfectly hiding and H is collision-resistant, then the construction $\mathfrak{C}'_{\mathfrak{C},H}$ from Construction 3.29 is a computationally binding (Definition 3.21) and perfectly hiding (Definition 3.22) commitment scheme with message space $\{0,1\}^*$.*

Proof. We prove correctness, computational binding and perfect hiding separately.

Correctness Let $n \in \mathbb{N}$, $pp' = (\mathsf{pp}, k) \in [\mathsf{Setup}'(1^n)]$. Then, for an $m \in \{0,1\}^*$, $\mathsf{Com}'(\mathsf{pp}', m)$ outputs $(com, (m, d))$. Since, (com, d) was generated by $\mathsf{Com}(\mathsf{pp}, h_k(m))$, pp was generated by $\mathsf{Setup}(1^n)$, and the underlying commitment scheme $\mathfrak{C}$ is correct, we have $\mathsf{Open}(\mathsf{pp}, com, d) = h_k(m)$. Thus, $\mathsf{Open}'(\mathsf{pp}', com, (m, d))$ opens to m, therefore the commitment scheme $\mathfrak{C}'_{\mathfrak{C},H}$ is correct.

Computational Binding Let $\mathcal{A}'$ be an adversary against the computationally binding property of $\mathfrak{C}'_{\mathfrak{C},H}$. Construct an adversary $\mathcal{A}$ against the computational binding property of $\mathfrak{C}$ that works as follows:

Adversary $\mathcal{A}(\mathsf{pp})$

1 : Generate $k \leftarrow \mathsf{Gen}_h(\mathsf{pp})$.
2 : Simulate $\mathcal{A}'(\mathsf{pp}, k)$.
3 : Receive $(c, m_1, (y_1, d_1), m_2, (y_2, d_2))$ from $\mathcal{A}'$.
4 : If $h_k(m_1) \neq h_k(m_2)$, return $(c, h_k(m_1), d_1, h_k(m_2), d_2)$, else abort.

Furthermore, construct an adversary $\mathcal{A}_{coll}$ against the collision-resistance of H as follows:

Adversary $\mathcal{A}_{coll}(\mathsf{pp}, k)$

1 : Simulate $\mathcal{A}'(\mathsf{pp}, k)$.
2 : Receive (c, m_1, d_1, m_2, d_2) from $\mathcal{A}'$.
3 : If $h_k(m_1) = h_k(m_2)$, return (m_1, m_2), else abort.

Call EQ the event that $h_k(m_1) = h_k(m_2)$. Then, we have

$$\begin{aligned}\Pr[Binding_{\mathfrak{C}'_{\mathfrak{C},H},\mathcal{A}'}(n) = 1] &= \Pr[Binding_{\mathfrak{C}'_{\mathfrak{C},H},\mathcal{A}'}(n) = 1 \wedge EQ] \\ &+ \Pr[Binding_{\mathfrak{C}'_{\mathfrak{C},H},\mathcal{A}'}(n) = 1 \wedge \neg EQ] \\ &\leq \Pr[\mathsf{HashColl}_{\mathcal{A}_{coll},H}(n) = 1] + \Pr[Binding_{\mathfrak{C},\mathcal{A}}(n) = 1 \mid \neg EQ] \\ &\leq \mathrm{negl}_1(n) + \mathrm{negl}_2(n)\end{aligned}$$

for some negligible functions $\mathrm{negl}_1, \mathrm{negl}_2$, since $\mathcal{A}$ and $\mathcal{A}_{coll}$ perfectly simulate the view of $\mathcal{A}'$, $\mathfrak{C}$ is computationally binding, and H is collision-resistant.

Perfect Hiding Since $\mathfrak{C}$ is perfectly hiding, we have that for $n \in \mathbb{N}$, $\mathsf{pp} \in [\mathsf{Setup}(1^n)]$, $m_1, m_2 \in M(\mathsf{pp})$ the following equation holds:

$$\Pr[\mathsf{Com}(\mathsf{pp}, m_1) = (com, \cdot)] = \Pr[\mathsf{Com}(\mathsf{pp}, m_2) = (com, \cdot)]$$

Since this does not depend on the distribution of the input and $\mathfrak{C}'$ only changes the input, it follows that

$$\Pr[\mathsf{Com}(\mathsf{pp}, h_k(y_1)) = (com, \cdot)] = \Pr[\mathsf{Com}(\mathsf{pp}, h_k(y_2)) = (com, \cdot)]$$

for $y_1, y_2 \in \{0,1\}^*$ and $k \in \mathsf{Gen}_h(1^n)$, since images of h_k are in $M(\mathsf{pp})$. Thus, $\mathfrak{C}'$ is also perfectly hiding.

□

3.6 Secret-Sharing Schemes

We proceed with the definition of secret-sharing schemes. They are used to distribute secrets among participating parties, such that only qualified groups of them can (efficiently) reconstruct the secret with their shares. Therefore, such a scheme is always defined with respect to a context-dependent *access structure*, listing these qualified sets of participants. A secret-sharing schemes is helpful to achieve proofs of partial knowledge [CDS94]. In the following we stick to the definitions from [CK93; BC93; CDS94]. We start defining access structures and their dual counterparts with respect to a given set of parties.

Definition 3.31 ((Dual) Access Structure)**.** Let $M = \{p_1, \ldots, p_n\}$ be a set of parties. A collection $\Gamma \subseteq \mathcal{P}(M)$ is monotone if $A \in \Gamma$ and $A \subseteq B \subseteq M$ imply $B \in \Gamma$. An *access structure* is a monotone collection $\Gamma \subseteq \mathcal{P}(M)$ of non-empty subsets of M. Sets in Γ are called *qualified*, sets not in Γ are called *non-qualified*. The dual access structure of Γ is defined as $\Gamma^* := \{A \subseteq M \mid M \setminus A \notin \Gamma\}$.

Note, that in many cases it is reasonable and no restriction to assume monotonicity of an access structure, as a qualified set extended by other parties usually remains qualified. Access structures and especially their duals will become relevant in the context of proofs of partial knowledge. Such structures may be given implicitly, rather than enumerating all qualified sets.

The next lemma states a useful property of monotone access structure that we later will use to prove the properties of the proof of partial knowledge construction given in Construction 3.59.

Lemma 3.32. *Let Γ be a monotone access structure. A set A is qualified in Γ if and only if for all $B \in \Gamma^*$ it holds $A \cap B \neq \emptyset$.*

Proof. We show the two directions separately by contradiction. In the following, let $M := \{p_1, \ldots, p_n\}$ denote the set of parties of the access structure Γ.

Suppose $A \in \Gamma$ and there exists a $B \in \Gamma^*$ such that $A \cap B = \emptyset$. Therefore, it holds $B \subseteq (M \setminus A)$. Now, we have $B \in \Gamma^*$ and $B \subseteq (M \setminus A) \subseteq M$ giving us by monoticity that $M \setminus A \in \Gamma^*$. By definition of Γ^*, we in turn have $(M \setminus A) \in \Gamma^* \iff M \setminus (M \setminus A) = A \notin \Gamma$. Contradicting the assumption that $A \in \Gamma$.

Suppose $A \notin \Gamma$ and for every $B \in \Gamma^*$ it holds $A \cap B \neq \emptyset$. Since $A \notin \Gamma$, we have by definition $(M \setminus A) \in \Gamma^*$. This contradicts the assumption that every qualified set in Γ^* has a non-empty intersection with A. ☐

Access structures can be given implicitly rather than listing all qualified sets explicitly. One method to do so are monotone predicates, which we define in the following.

Definition 3.33 (Monotone Predicate)**.** A predicate $\phi(X_1, \ldots, X_n)$ on n variables $X_i \in \{0, 1\}$ is called *monotone*, if and only if for a set of parties $\{p_1, \ldots, p_n\}$

$$\Gamma_\phi := \{A \subseteq \{p_1, \ldots, p_n\} \mid \forall i = 1, \ldots, n : X_i = [p_i \overset{?}{\in} A] \wedge \phi(X_1, \ldots, X_n) = 1\},$$

is a (monotone) access structure.

This means any monotone predicate can identify a (monotone) access structure and vice versa. Monotone predicates could be considered as boolean formulas without negated variables. However, the representation via a boolean formula can become exponential in the number of parties. This can happen for example in d-out-of-n access structures, where any set with d and more parties is qualified. Here, an efficiently computable predicate would check whether $\sum_{i=1}^{n} X_i \geq d$ or not, rather than using a boolean formula. With respect to access structures we define secret-sharing schemes.

Definition 3.34 (Secret-Sharing Scheme)**.** Let Γ be an access structure with n parties and K be a finite set of secrets, where $|K| \geq 2$. A *(perfect) secret-sharing scheme* with domain of secrets K realizing an access structure Γ is a tuple $(\mathsf{Share}, \mathsf{Recon})$, where

1. The ppt algorithm Share on input secret $s \in K$ outputs shares $(s_i)_{i=1}^n \in K_1 \times \ldots \times K_n$, where, for $j = 1, \ldots, n$, K_j is called the domain of shares of party p_j.

2. Correctness: The secret $s \in K$ can be reconstructed by any qualified set. The deterministic algorithm Recon outputs, on input set $A = \{p_{i_1}, \ldots, p_{i_{|A|}}\} \in \Gamma$ and shares $(s_{i_1}, \ldots, s_{i_{|A|}}) \in K_{i_1} \times \cdots \times K_{i_{|A|}}$, a secret $s \in K$. We demand that for every $s \in K$,

$$\Pr[\mathsf{Recon}(A, \mathsf{Share}(s)_A) = s] = 1,$$

where the probability is taken over the coin tosses of Share and $\mathsf{Share}(s)_A$ denotes the restriction of the output to its A-entries.

3. Perfect Privacy: Every non-qualified set cannot learn anything about the secret from their shares. Formally, for any set $A \notin \Gamma$, for every two secrets $s, s' \in K$ and every possible vector of shares $(s_j)_{p_j \in A}$

$$\Pr[\mathsf{Share}(s)_A = (s_j)_{p_j \in A}] = \Pr[\mathsf{Share}(s')_A = (s_j)_{p_j \in A}].$$

To share a secret s among the parties Share is run and each share s_j is privately communicated to party p_j. The trivial case $|K| = 1$ is excluded, since the only secret would be "reconstructable" by any party. Given $|K| \geq 2$, the requirements ensure, that a secret can be reconstructed if and only if shares corresponding to a qualified set are known. By perfect privacy we see that, for any non-qualified set A, the $\mathsf{Share}(s)_A$ (treated as random variable) is distributed independently of s, and we can write simply Share_A.

3.6.1 Smooth Secret-Sharing Schemes

In case of proofs of partial knowledge, we demand more properties from secret-sharing schemes [CDS94].

Definition 3.35. Let Γ be an access structure with n parties, K be a finite set of secrets, $|K| \geq 2$, and K_i be the domain of shares of party p_i, $i = 1, \ldots, n$. A *t-smooth secret-sharing scheme* is a tuple $(\mathsf{Share}, \mathsf{Recon}, \mathsf{CheckConsistency}, \mathsf{Complete})$, where

1. $(\mathsf{Share}, \mathsf{Recon})$ forms a secret-sharing scheme such that Share and Recon run in time at most t.

2. The deterministic algorithm CheckConsistency outputs, on input secret s and a full set of n shares, a bit $b \in \{0, 1\}$. Here, we interpret $b = 1$ as the given secret s being consistent with the full set, which means that all qualified sets of shares determine s as the secret. The running time of CheckConsistency is at most t.

3. The probabilistic algorithm Complete outputs, on input any secret s and a set of shares corresponding to an unqualified set A, a full set of n shares completing A that is consistent with s. This algorithms runs in time at most t. If the set of shares is distributed according to Share_A, then the full set output by Complete is distributed according to $\mathsf{Share}(s)$.

4. For any non-qualified set A, the probability distribution Share_A is such that shares for the participants in A are independent and uniformly distributed.

Moreover, a t-semi-smooth secret-sharing scheme is a tuple $(\mathsf{Share}, \mathsf{Recon}, \mathsf{CheckConsistency}, \mathsf{Complete})$ fulfilling properties 1–3.

As Cramer, Damgård, and Schoenmakers [CDS94] note, the scheme of Shamir [Sha79] is smooth and can realize threshold structures. Further, the recursive construction of Benaloh and Leichter [BL90] yields a semi-smooth secret-sharing scheme for any access structure given by a monotone boolean formula [CDS94].

3.7 Zero-Knowledge Arguments of Knowledge

In an anonymous credential system, one is often asked to prove to know something. Since we focus on anonymity, we want to achieve that the user is able to convince another party of her knowledge without revealing the exact knowledge to the other parties. To illustrate, consider the following example: The user wants to use a service that requires her to be over 18 years old, but the user does not want to reveal her exact age. This can be achieved by the technique of

zero-knowledge proofs of knowledge or *zero-knowledge arguments of knowledge*. In this section, we formally define this notion. Let us first give some preliminaries.

In the following, we often refer to *interactive protocols*. An interactive protocol is a computation between two *interactive algorithms*, i. e. the algorithms can exchange messages. We call these two algorithms the *prover* $\mathcal{P}$ and the *verifier* $\mathcal{V}$. The prover $\mathcal{P}$ somehow proves to the verifier that she has a solution to a public problem. The problem and solution are formalized by *(formal) languages* and *binary relations*.

Definition 3.36. Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be a binary relation where for all $(v,w) \in R$ we have $|w| \leq p(|v|)$ for some polynomial $p(\cdot)$. We call this *polynomially bounded*. Furthermore, we want R to be *polynomially verifiable*, meaning one can compute $R(\cdot,\cdot)$ in polynomial time. If R is both polynomially bounded and polynomially verifiable, we call R an *NP-Relation*. For a $v \in \{0,1\}^*$ call $W(v) = \{w \in \{0,1\}^* : (v,w) \in R\}$ the witness set of v and $w \in W(v)$ a witness for v. Define $L_R = \{v \in \{0,1\}^* : W(v) \neq \emptyset\}$.

In other words, $v \in L_R$ denotes our problem known to both $\mathcal{P}$ and $\mathcal{V}$, and w denotes the secret witness solving our problem. The common input of $\mathcal{P}$ and $\mathcal{V}$ is always v. Apart from the common input we allow another *secret* or *auxiliary input* for each algorithms that are only known to them. For $\mathcal{P}$, the secret input is the witness $w \in W(v)$ corresponding to instance v, while the secret input y for a verifier may give her some additional a-priori information. When a prover P on input (v,w) interacts with a verifier V on input v, y, we denote this by $P(v,w) \leftrightarrow V(v,y)$. Moreover, we refer to the messages exchanged by $\mathcal{P}$ and $\mathcal{V}$ as *transcripts*. Denote by $T(\mathcal{P}(v,w) \leftrightarrow V(v,y))$ the random variable of the possible transcripts of the interaction of $P(v,w)$ and $V(v,y)$. Then, we can define what an interactive argument (cf. [Gol01, p.193]) is:

Definition 3.37. A protocol $(\mathcal{P},\mathcal{V})$ is called an interactive argument for a language L, if $\mathcal{P}$ and $\mathcal{V}$ are ppt and

- at the end of their interaction, $\mathcal{V}$ outputs a 1 or 0 indicating she *accepted* the proof or not, which we denote by $(\mathcal{P}(v,w) \leftrightarrow \mathcal{V}(v)) \to b$,
- for every $v \in L$ there exists a $w \in \{0,1\}^{p(|v|)}$ such that

$$\Pr[(\mathcal{P}(v,w) \leftrightarrow \mathcal{V}(v)) \to b : b = 1] \geq \frac{2}{3},$$

- for every $v \notin L$ sufficiently long, every ppt interactive algorithm $\mathcal{P}^*$ and every $w \in \{0,1\}^{p(|v|)}$, we have

$$\Pr[(\mathcal{P}^*(v,w) \leftrightarrow \mathcal{V}(v)) \to b : b = 1] \leq \frac{1}{3},$$

where $p(\cdot)$ is some polynomial.

Note that in this definition $\mathcal{V}$ does not get any additional information y. For such an interactive argument, call a transcript accepting, if the verifier accepted at the end of the interaction described by the transcript.

As a remark, there exists a definition similar to the definition of an interactive argument for an *interactive proof*. While such a proof does not give the prover a witness, it lifts the polynomial time restriction for the prover, meaning she can compute the witness before engaging with a verifier. We want to restrict ourself to only arguments, since we require our provers to have polynomial time. Therefore, all our security definitions only speak about arguments and not proofs.

3.7.1 Zero-Knowledge Arguments

One of the goals of our protocols is to not leak knowledge about the prover's secret during the protocol execution. This means that transcripts of the protocol generally do not help the verifier to learn something secret. For the formal definition, we draw inspiration from [Gol01, p.207].

Definition 3.38 (Honest Verifier Zero-Knowledge)**.** An interactive argument $(\mathcal{P}, \mathcal{V})$ for a language L_R is called *honest verifier zero-knowledge (HVZK)* for an NP-relation R, if there exists a ppt algorithm S, which for all $v \in L_R$ generates transcripts that are identically distributed as transcripts of real protocol runs. Formally defined, this means that for all $(v, w) \in R$ we have

$$\Pr[T(\mathcal{P}(v, w) \leftrightarrow V(v)) = \tau] = \Pr[S(v) = \tau].$$

This means, we are not only able to generate accepting transcripts without any interaction with a prover, but we can do it even with the same probability distribution. Thus there cannot be any gain of knowledge by running the protocol with an *honest* verifier. Note that this definition only covers honest verifiers. The simulator might not have the same guarantees if the verifier is dishonest.

There also exists a definition for (general) zero-knowledgeness, which includes dishonest verifiers. Here, we require similarly to Definition 3.38 that we are able to generate accepting transcripts, only this time we want to do this for every (possibly cheating) verifier. Furthermore, a cheating verifier may have some additional information that she could try to use to learn the prover's witness. We model this by giving the (possibly cheating) verifier an additional input representing her extra information (cf. [Gol01, p.213]).

Definition 3.39 (Zero-Knowledge)**.** An interactive argument $(\mathcal{P}, \mathcal{V})$ for a language L_R is called *zero-knowledge (ZK)* for an NP-relation R, if for every interactive algorithm $\mathcal{V}^*$ there exists a ppt algorithm $S_{\mathcal{V}^*}$, such that for all $(v, w) \in R$, all $y \in \{0,1\}^{p(|v|)}$, we have

- S_{V^*} may output an error symbol $\perp$ with $\Pr[S_{V^*}(v, y) = \perp] \leq \frac{1}{2}$
- $\Pr[T(\mathcal{P}(v, w) \leftrightarrow V^*(v, y)) = \tau] = \Pr[S_{V^*}(v, y) = \tau \mid S_{V^*(v,y)} \neq \perp]$,

where $p(\cdot)$ is some polynomial.

In other words, for every interactive algorithm V^* (i.e. possibly cheating verifier) there exists a simulator S_{V^*} that generates accepting transcripts with the same distribution V^* would expect.

If there exists a single simulator for all interactive algorithms $\mathcal{V}^*$ with oracle access to $\mathcal{V}^*$ that fulfills the requirements above, we speak of *black-box zero-knowledge.*

When using one of the zero-knowledge definitions, one has to be careful of concurrent verifiers that are able to run multiple instances of a protocol in parallel, as they are more powerful in general. For a special subset of protocols there is a way to transform them into general concurrent zero-knowledge protocols with cheating verifiers (cf. Section 3.9).

3.7.2 Arguments of Knowledge

Since we want to use our protocols to prove knowledge of something, we want to make sure that only algorithms with knowledge of a secret can convince a verifier. Thus, we need to define what it means for a machine to have knowledge of something. This is caught by the idea, that if there actually is some knowledge in a machine, one should be able to *extract* this (cf. [Gol01, p.264]).

Definition 3.40 (Argument of Knowledge)**.** An interactive argument $(\mathcal{P}, \mathcal{V})$ for a language L_R is called *argument of knowledge (AoK)* for an NP-relation R with knowledge error $\kappa(\cdot)$, if

- for all $(v, w) \in R$ we have $\Pr[(\mathcal{P}(v, w) \leftrightarrow \mathcal{V}(v, y)) \to b : b = 1] = 1$.

- there exists a ppt algorithm E with black-box access to a prover $\mathcal{P}^*$ that on input $v \in L_R$ outputs a witness w such that $(v, w) \in R$.
  Furthermore, for any ppt interactive algorithm $\mathcal{P}^*$ and any long enough v, if $\epsilon(v) > \kappa(v)$, the extractor must run in expected time

$$\frac{p(|v|)}{\epsilon(v) - \kappa(v)},$$

  where $p(\cdot)$ is some polynomial and $\epsilon(v)$ is the probability that $\mathcal{P}^*$ convinces the verifier on input v.

Here, black-box access means that the extractor has oracle access to a prover, i.e. an interactive algorithm following the protocol in the role of a prover, and may get outputs from it on any input she chooses, including choosing the randomness. Note that since the prover is an interactive algorithm, the extractor sometimes needs to give messages to the oracle to receive answers of a specific step.

3.7.3 Camenisch-Stadler Notation

There is a useful notation introduced by Camenisch and Stadler Camenisch and Stadler [CS97], which describes proofs of knowledge in a very compact manner. For example,

$$\mathsf{PK}\Big\{(\alpha, \beta) : y = g^\alpha \wedge z = g^\beta h^\alpha\Big\}$$

denotes an interactive proof of knowledge of the discrete logarithm for y to the base g and a representation for z to the bases g and h, where additionally h is raised with the discrete logarithm of y to base g. In other words, we prove knowing the discrete logarithm α of y and additionally β such that $z = g^\beta h^\alpha$.

3.7.4 Non-interactive Arguments

In some cases, someone wants to prove knowledge of a secret, but circumstances prohibit the usage of an interactive protocol. For example, a user wants to add to a rating that she is over 18 years old. Since the rating should be verifiable without talking to the rating user, we need a non-interactive way of proving knowledge. Thus, we want to introduce non-interactive arguments, where a prover outputs some form of proof and a verifier either accepts or rejects the proof by outputting 1 or 0. We adapt the definition of Goldreich [Gol01, p.299] to the random oracle model (cf. [BR93]).

Definition 3.41. A tuple of ppt algorithms $(\mathcal{P}, \mathcal{V})$ is called a *non-interactive argument* for a language L, if

- for every $v \in L$ there exists a $w \in \{0,1\}^{p(|v|)}$ such that

$$\Pr[b \leftarrow \mathcal{V}(v, \mathcal{P}(v, w)) : b = 1] \geq \frac{2}{3},$$

  where $p(\cdot)$ is some polynomial.

- for every $v \notin L$ and every ppt algorithm B we have

$$\Pr[b \leftarrow \mathcal{V}(v, B(v), y) : b = 1] \leq \frac{1}{3}.$$

Both $\mathcal{P}$ and $\mathcal{V}$ can make queries to a random oracle.

Similarly to an interactive argument, we want to make sure that a verifier does not learn anything about the secret witness. Since we are in a non-interactive setting, the verifier cannot influence the generation of the proof, thus she does not need to be considered for zero-knowledgeness. As before, we require that there exists a simulator, such that an adversary cannot distinguish between the output of the simulator and an honestly generated proof. For a formal definition, look into Bernhard, Pereira, and Warinschi [BPW12].

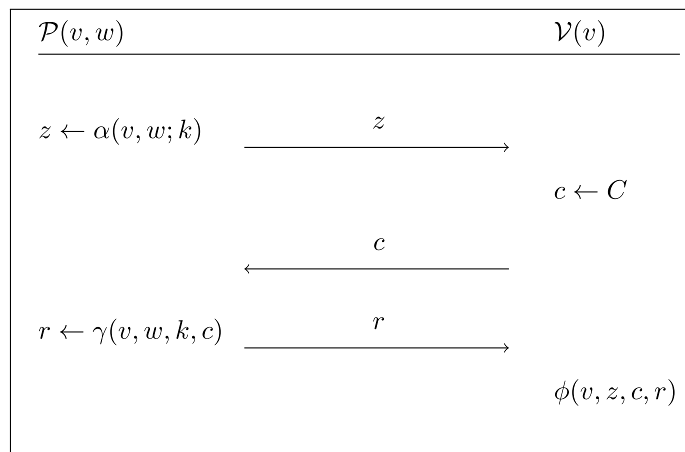
Furthermore, we need some notion regarding soundness to ensure that only provers with some knowledge about a witness can convince a verifier with significant probability. Since the definition of an interactive argument of knowledge does not apply, we look into the notion of *simulation sound extractability* of Bernhard, Pereira, and Warinschi [BPW12]. There, a malicious prover may see simulated proofs and outputs several proofs together with the oracle queries he queried. Then, an extractor tries to extract witnesses from these proofs. During this, the extractor may rewind the prover, but only with the same randomness. If an extractor exists, that for any prover is able to extract witnesses with non-negligible probability, we have simulation sound extractability. For a formal definition, see Bernhard, Pereira, and Warinschi [BPW12].

3.8 Σ-Protocols

Whenever a user wants to convince another party that she knows something, we need to have a protocol which both parties follow to exchange their messages. There exists a pattern that is generally usable for proving knowledge of something, called Σ-protocol [Dam10]. This pattern is useful as a building block, since we can base many different constructions on it. Since a Σ-protocol is an interactive argument (Definition 3.37), we prove knowledge of a witness corresponding to a problem of a NP-Relation.

First, we describe the form of a Σ-protocol.

Construction 3.42. Let R be a NP-Relation. Let α, γ be ppt algorithms. Let C be a finite set. Let ϕ be a predicate that is computable in polynomial time. Then, let $(\mathcal{P}, \mathcal{V})$ be a three-round interactive argument for L_R (Definition 3.37) of the following form:



Here, three-round means that transcripts consist of three messages. We call z the announcement, c the challenge and r the response.

Definition 3.43. Call $\mathfrak{S} = (\mathcal{P}, \mathcal{V})$ of form as in Construction 3.42 a Σ-*protocol* for a NP-Relation R, if it has the following properties:

Correctness: If $(v, w) \in R$, and $\mathcal{P}$ and $\mathcal{V}$ follow the protocol, $\mathcal{V}$ accepts with probability 1.

Special Soundness There exists a ppt algorithm E, called *extractor*, that given $v \in L_R$ and any two accepting transcripts $(z, c, r), (z, c', r')$ with $c \neq c'$ outputs a witness w with $(v, w) \in R$.

Special Honest-Verifier Zero-Knowledge There exists a ppt algorithm S, called *simulator*, that when given $v \in L_R$ and any challenge $c \in C$ outputs an accepting transcript with the same distribution as in the real protocol, i.e. for all $(v, w) \in R$ and all $c \in C$ we have

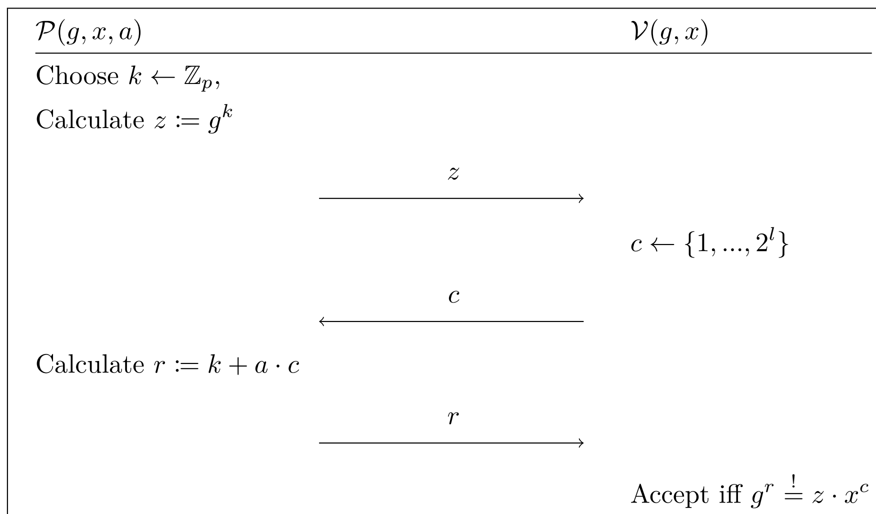
$$\Pr[T(\mathcal{P}(v, w) \leftrightarrow \mathcal{V}(v)) = (z, c, r)] = \Pr[S(v, c) = (z, c, r)].$$

Note that correctness together with special soundness reminds of Definition 3.40, since we fix the probability that the verifier accepts on correct input to 1 and there exists an extractor outputting a witness. This is due to the fact that for Σ-protocols, correctness and special soundness imply that the argument is a proof of knowledge.

3.8.1 Schnorr Protocol

A basic example of a Σ-protocol is the Schnorr protocol [Sch91]. The simplicity and flexibility of it is useful for our anonymous credential system, because we need slightly different Σ-protocols at different points of the system (cf. Section 4.1.3). We need the discrete logarithm problem (Definition 3.4) as a computational assumption, which we use to prove the knowledge of an exponent in the following way:

Construction 3.44. Let $\mathbb{G}$ be a cyclic group of prime order p and let g be a generator in $\mathbb{G}$. Then, let $g^a = x$ for $x \in \mathbb{G}$ and $a \in \mathbb{Z}_p$. The following protocol can be executed between a prover $\mathcal{P}$ with input (g, x, a) and a verifier $\mathcal{V}$ with input (g, x).



Lemma 3.45. *Construction 3.44 is a Σ-protocol for the relation*

$$R_{\mathbb{G}} = \{(g, x, a) \mid g^a = x\}$$

with $g \in \mathbb{G} \setminus \{1\}$.

Proof. We show the three properties stated in Definition 3.43 hold for Construction 3.44. These are correctness, special soundness and special honest verifier zero-knowledgeness.

In the following, let g be a generator of $\mathbb{G}$. Further, let $a \in \mathbb{Z}_p$ and $x = g^a \in \mathbb{G}$.

Correctness We have to show that if both parties follow the protocol, the verifier always accepts. If prover and verifier act honestly, we have $z = g^k$ for some $k \in \mathbb{Z}_p$, and $r = k + a \cdot c$ and challenge $c \in \{1, \ldots, 2^l\}$. Thus, we have

$$g^r = g^{k+a \cdot c} = g^k \cdot (g^a)^c = z \cdot x^c.$$

Hence, the honest verifier will accept.

Special Soundness Let (z, c, r) and (z, c', r') be two accepting transcripts with $c \neq c'$. For special soundness, we have to construct a ppt extractor E which on input $(g, x, (z, c, r), (z, c', r'))$ outputs a witness $\tilde{a}$ such that $g^{\tilde{a}} = x$.

Since both transcripts are accepting, we have

$$g^r = z \cdot x^c \wedge g^{r'} = z \cdot x^{c'} \iff z = g^r\, x^{-c} \wedge z = g^{r'}\, x^{-c'} \iff g^{r-r'} = x^{c-c'} \iff \tilde{a} = \frac{r - r'}{c - c'}.$$

Hence, E outputs $\tilde{a} = \dfrac{r - r'}{c - c'}$.

Special Honest Verifier Zero-Knowledge Fix some challenge $c \in \mathbb{Z}_p$. Special honest verifier zero-knowledge is satisfied if a ppt simulator S on input (g, x, c) outputs an accepting transcript with the same distribution as in the real protocol. Observe that in the real protocol, a transcript (z, c, r) is fixed by fixing r and c. In the real protocol, k is chosen uniformly at random from $\mathbb{Z}_p$. Since challenge c and witness a are fixed, r is distributed uniformly on $\mathbb{Z}_p$ as well. Taking this together with the observation that r and c fix the transcript, we get that every transcript appears with probability $\frac{1}{|\mathbb{Z}_p|} = \frac{1}{p}$.

Now, we construct the simulator S in the following way:

1. Choose $r \leftarrow \mathbb{Z}_p$ uniformly at random
2. Set $z := g^r \cdot x^{-c}$
3. Output (z, c, r)

First, note that the transcripts output by S are always accepting. Moreover, for every transcript output by S, we have that r is distributed uniformly on $\mathbb{Z}_p$ and z is fixed by fixing r. Thus, every transcript again appears with probability $\frac{1}{|\mathbb{Z}_p|} = \frac{1}{p}$. This results in identical distributions, hence special honest verifier zero-knowledgeness is satisfied. $\square$

3.8.2 Generalized Schnorr Protocol

The Schnorr protocol allows the user to prove knowledge of a single exponent. Of course it would be useful to extend this to multiple exponents or even arbitrary group elements. This extension is given by the generalized Schnorr protocol.

Construction 3.46. We consider groups $(\mathbb{G}_1, ..., \mathbb{G}_m) := \mathcal{G}$, where all groups are of prime order p. The prover then proves knowledge of some $(x_1, ..., x_n) \in \mathbb{Z}_p^n$. For that, we consider m equations $A_j = \prod_{i=1}^n g_{j,i}^{x_i}$ for $j \in \{1, ..., m\}$ with $A_j, g_{j,i} \in \mathbb{G}_j$ for all $i \in \{1, ..., n\}, j \in \{1, ..., m\}$. For shorter notation, we write $(A_1, ..., A_m) =: A$ and $((g_{j,i})_{i=1}^n)_{j=1}^m =: g$. As common input, the prover $\mathcal{P}$ and verifier $\mathcal{V}$ get $(\mathcal{G}, A, g)$. The generalized Schnorr protocol is defined in Figure 3.5.

Lemma 3.47. *The generalized Schnorr-Protocol as in Construction 3.46 is a Σ-protocol for the relation $R_\mathcal{G}$ where*

$$((A_1, ..., A_m, g, (x_1, ..., x_n)) \in R_\mathcal{G} \iff \forall i \in \{1, ..., n\} \forall j \in \{1, ..., m\} : A_j := \prod_{i=1}^n g_{j,i}^{x_i}.$$

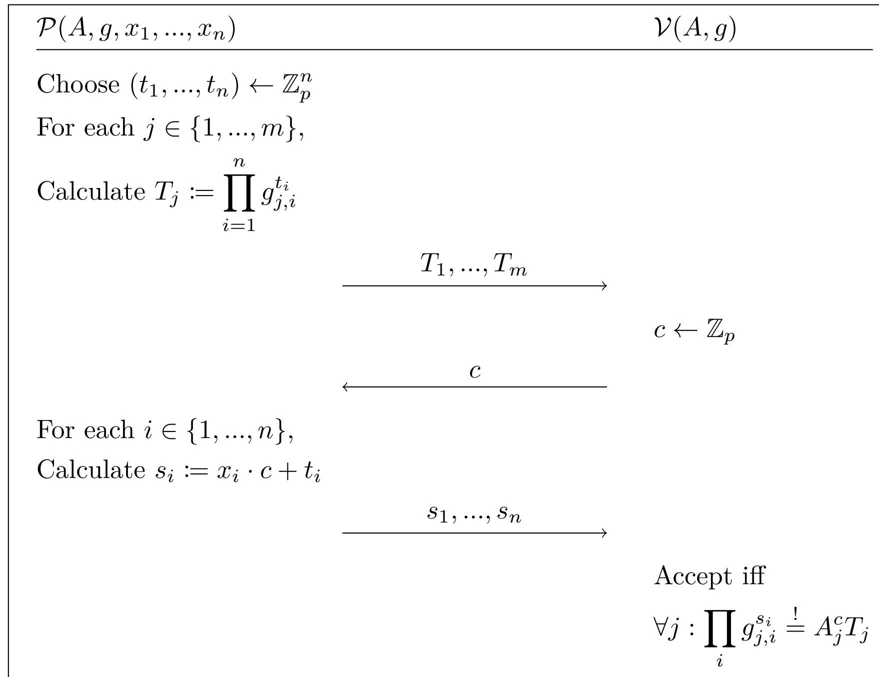


Figure 3.5: Generalized Schnorr Protocol

Proof. To prove that the generalized Schnorr protocol is a Σ-protocol, we have to show three properties (see Definition 3.43). These are completeness, special soundness and special honest verifier zero-knowledgeness.

Completeness Completeness holds if the verifier always accepts while both parties follow the protocol. For every $j \in \{1, ..., m\}$, we get

$$\prod_i g_{j,i}^{s_i} = \prod_i g_{j,i}^{x_i \cdot c + t_i} = \prod_i \left(g_{j,i}^{x_i}\right)^c \cdot \prod_i g_{j,i}^{t_i} = A_j^c \cdot T_j$$

and hence, the verifier accepts.

Special Soundness We define an extractor E that on input $(A_1, ..., A_m, ((g_{j,i})_{i=1}^n)_{j=1}^m)$ and transcripts $((T_1, ..., T_m), c, (s_1, ..., s_n)), ((T_1, ..., T_m), c', (s'_1, ..., s'_n))$ outputs a witness $\tilde{w} = (\tilde{x}_1, ..., \tilde{x}_n)$. Given that input, the extractor can just compute $\tilde{x}_i = \frac{s_i - s'_i}{c - c'}$ for each $i \in \{1, ..., n\}$. The result $(\tilde{x}_1, ..., \tilde{x}_n)$ is a valid witness because

$$\prod_i g_{j,i}^{s_i} = A_j^c \cdot T_j \wedge \prod_i g_{j,i}^{s'_i} = A_j^{c'} T_j$$

for all $j \in \{1, ..., m\}$ and therefore

$$\prod_i g_{j,i}^{s_i - s'_i} = A_j^{c - c'} \iff \prod_i g_{j,i}^{\frac{s_i - s'_i}{c - c'}} = A_j$$

since $c - c' \neq 0$.

Special Honest Verifier Zero-Knowledgeness Fix some challenge $c \in \mathbb{Z}_p$. For *special honest-verifier zero-knowledgeness* we have to simulate interactions between $\mathcal{V}$ and $\mathcal{P}$. On input $(A_1, ..., A_m, ((g_{j,i})_{i=1}^n)_{j=1}^m, c)$, the simulator $\mathcal{S}$ does the following:

1. Choose $(s_1, ..., s_n) \leftarrow \mathbb{Z}_p^n$ uniformly at random
2. Set $T_j = \frac{\prod_i g_{j,i}^{s_i}}{A_j^c}$ for all $j \in \{1, ..., m\}$
3. Output $((T_1, ..., T_m), c, (s_1, ..., s_n))$

In the protocol, the s_i's are calculated using the uniformly and independently distributed t_i's, and hence the s_i's are also distributed uniformly and independently at random in $\mathbb{Z}_p$. By choosing them directly uniformly at random, the simulator therefore fixes the transcript. In the protocol, the T_j's are calculated using the uniformly and independently distributed t_i's This means that the T_j's are also uniformly and independently distributed. The simulator S uses the random s_i's to calculate the T_j's, also resulting in a uniform and independent distribution. Altogether, this means that the transcript produced by the simulator S has the same distribution as transcripts of the protocol for a fixed challenge c where every transcript appears with probability $(\frac{1}{p})^n$ since we choose n s_i's uniformly and independently at random out of $\mathbb{Z}_p$. □

As mentioned before, this protocol allows the prover to prove knowledge of an arbitrary number of elements in $\mathbb{Z}_p$. This can also be extended to group elements by blinding the group element with elements chosen uniformly from $\mathbb{Z}_p$ and then later derandomizing it. Hence we can prove knowledge of any elements using this protocol while preserving the useful properties of Σ-protocols, resulting in a very flexible protocol.

3.9 Damgård's Technique

Through Σ-Protocols we are able to prove knowledge of a secret without revealing the secret, although only against a honest verifier that chooses a challenge with the correct distribution. Since in the real world attackers against the anonymous credential system will not necessarily stay honest, we want to have protocols that have the zero-knowledge property even against dishonest verifiers. Furthermore, we want security against concurrent adversaries. Concurrent here means that an adversary is able to run multiple protocols in parallel against the same prover while trying to learn something about the secret of the prover. Note that the concurrency implies that the adversary may interleave messages of multiple protocols and make them dependent of each other.

To improve the Σ-Protocol, we want the prover and the verifier to have some additional information that they can use for some additional computation steps. We do this by giving them an additional common input called the auxiliary string. Such an addition requires that we define a new model and security in this model.

Definition 3.48. A triple $(G, \mathcal{P}, \mathcal{V})$ is called an interactive argument in the auxiliary string model (IAASM) for a language L, if

- G is a ppt algorithm and $G(1^n)$ outputs an auxiliary string pp,
- $\mathcal{P}, \mathcal{V}$ are ppt interactive algorithms,
- $\mathcal{P}$ gets (pp, v, w) as input, while $\mathcal{V}$ gets only (pp, v) as input,
- At the end of their interaction, $\mathcal{V}$ either accepts or rejects,
- For all $v \in L$, there exists a $w \in \{0,1\}^{p(|v|)}$, such that $\mathcal{V}(\mathsf{pp}, v)$ accepts when interacting with $\mathcal{P}(\mathsf{pp}, v, w)$,

- For every $v \notin L$, every $w \in \{0,1\}^{p(|v|)}$ and every ppt interactive algorithm $\mathcal{P}^*$, we have that

$$\Pr[\mathsf{pp} \leftarrow G(1^n) : (\mathcal{P}^*(\mathsf{pp}, v, w) \leftrightarrow \mathcal{V}(\mathsf{pp}, v)) \rightarrow b : b = 1]$$

is negligible in n,

where $p(\cdot)$ is some polynomial.

Based on this model, we can define when an argument in the new model is secure, i.e. argument of knowledge and zero-knowledge. The latter follows the common approach that there exists a simulator producing accepting transcripts, where the difference is that the simulator gets the auxiliary string and a corresponding trapdoor as additional input.

Definition 3.49. An interactive argument in the auxiliary string model $(G, \mathcal{P}, \mathcal{V})$ for a NP-relation R is called *zero-knowledge* if there exists a ppt algorithm $\mathsf{TrapdoorGen}(1^n)$ outputting a tuple (pp, t) such that pp is distributed as in $G(1^n)$. We call t trapdoor.

Furthermore, for any interactive algorithm $\mathcal{V}^*$ there exists a simulator $M_{\mathcal{V}^*}$, such that for all $(v, w) \in R$, all $y \in \{0,1\}^{p(|v|)}$ and all $(\mathsf{pp}, t) \in [\mathsf{TrapdoorGen}(1^n)]$ it holds that $M_{\mathcal{V}^*}$ has expected polynomial runtime in its input length and

$$\Pr[T(\mathcal{P}(\mathsf{pp}, v, w) \leftrightarrow \mathcal{V}^*(\mathsf{pp}, v, y)) = \tau] = \Pr[M_{\mathcal{V}^*}(1^n, v, \mathsf{pp}, y, t) = \tau],$$

where $p(\cdot)$ is some polynomial.

Definition 3.50. An interactive argument in the auxiliary string model $(G, \mathcal{P}, \mathcal{V})$ for a relation R is an argument of knowledge for R in the auxiliary string model with knowledge error $\kappa(\cdot)$, if there exists an extractor E with black-box access to a prover that on input $v \in L_R$ outputs a witness w such that $(v, w) \in R$.

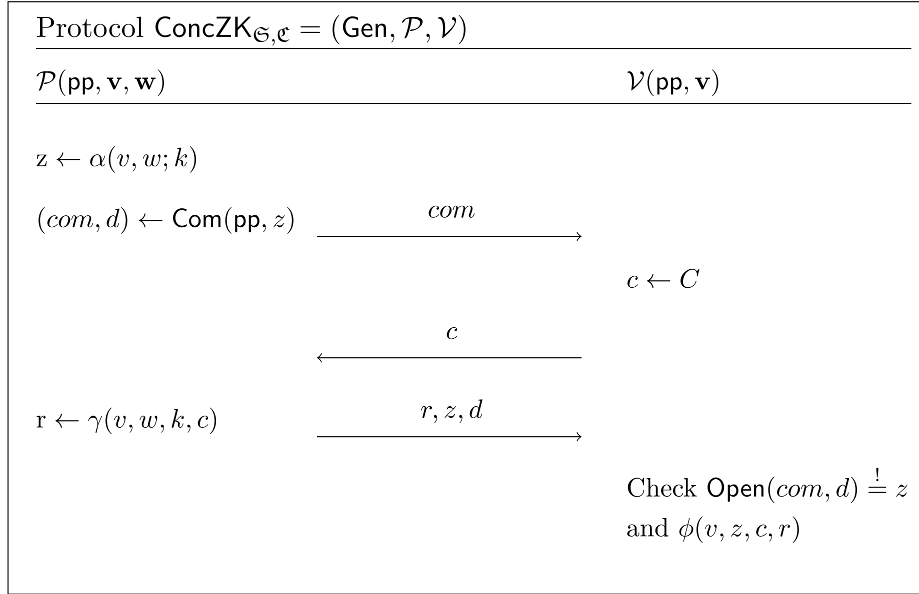
Furthermore, for a security parameter n, any ppt prover $\mathcal{P}^*$ and any sufficiently long v, if $\epsilon(n, v) > \kappa(n)$, the extractor must run in expected time

$$\frac{p(|v|)}{\epsilon(n, v) - \kappa(n)},$$

where $p(\cdot)$ is some polynomial and $\epsilon(n, v) = \Pr[\mathsf{pp} \leftarrow G(1^n) : (\mathcal{P}^*(\mathsf{pp}, v) \leftrightarrow \mathcal{V}(\mathsf{pp}, v)) \rightarrow b : b = 1]$.

With the definitions in place, we can construct a new protocol in the auxiliary string model.

Construction 3.51. Let $\mathfrak{S} = (\mathcal{P}_\Sigma, \mathcal{V}_\Sigma)$ be a Σ-Protocol (Construction 3.42) for a relation R with challenge space C, where α and γ are used to compute the first and third message respectively and ϕ is the predicate $\mathcal{V}_\Sigma$ checks at the end. Let $\mathfrak{C} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be a trapdoor commitment scheme (Definition 3.28). Let Gen be the algorithm that computes $(\mathsf{pp}, t) \leftarrow \mathsf{Setup}$ and outputs pp. Then, construct a IAASM $\mathsf{ConcZK}_{\mathfrak{S},\mathfrak{C}} = (\mathsf{Gen}, \mathcal{P}, \mathcal{V})$ for a relation R as seen below:



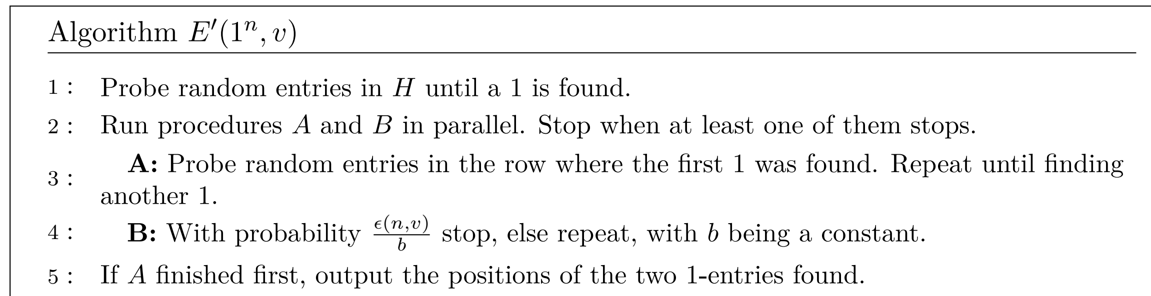
The new protocol simply sends the commitment of the announcement of the original Σ-Protocol instead of the announcement. Then, the last message also contains the original announcement and the open-value of the commitment, so that the verifier can check the validity of the commitment as well as the original check from the Σ-Protocol.

We then have the following results.

Lemma 3.52. *For a security parameter n, Σ-Protocol $\mathfrak{S}$ for a relation R and a computationally binding trapdoor commitment scheme $\mathfrak{C}$, the IAASM $\mathsf{ConcZK}_{\mathfrak{S},\mathfrak{C}}$ from Construction 3.51 is an argument of knowledge for R with knowledge error $\kappa(n) \leq \mu(n) + \frac{1}{|C|}$ in the auxiliary string model, assuming $\epsilon(n, v) > \frac{4}{|C|}$, where $\mu(\cdot)$ is a negligible function, C is the challenge space of $\mathfrak{S}$ and $\epsilon(n, v) = \Pr[\mathsf{pp} \leftarrow G(1^n) : (\mathcal{P}^*(\mathsf{pp}, v) \leftrightarrow \mathcal{V}(\mathsf{pp}, v)) \rightarrow b : b = 1]$.*

Proof. Let n be a security parameter, $v \in L_R$ and $\kappa(n) \leq \mu(n) + \frac{1}{|C|}$. Let $\mathcal{P}^*$ be a prover with $\epsilon(n, v) > \kappa(n)$. Furthermore, let H be a matrix with one row for each possible set of random choices of the prover and one column for every possible challenge in C. Write 1 in the matrix if the verifier accepts with this random choice and challenge, 0 otherwise. We call a row heavy if it contains a fraction of more than $\frac{\epsilon(n,v)}{2}$ 1's.

We construct an algorithm E' that can generate entries of H by using black-box access to $\mathcal{P}^*$:



Algorithm $E'(1^n, v)$

1 : Probe random entries in H until a 1 is found.
2 : Run procedures A and B in parallel. Stop when at least one of them stops.
3 : **A:** Probe random entries in the row where the first 1 was found. Repeat until finding another 1.
4 : **B:** With probability $\frac{\epsilon(n,v)}{b}$ stop, else repeat, with b being a constant.
5 : If A finished first, output the positions of the two 1-entries found.

Call *HeavyRow* the event that E' finds a heavy row in step 1. Then, we have

$$\Pr[A \text{ finds a second 1 in one try} | \mathit{HeavyRow}] \geq \frac{\frac{\epsilon(n,v)}{2} \cdot |C| - 1}{|C|},$$

since a row has $|C|$ entries, thus there are at least $\frac{\epsilon(n,v)}{2} \cdot |C|$ 1's in a heavy row (minus one since we need to find a new one). Let T be the expected number of probes A does until it finds a second 1. Thus, we have

$$
\begin{aligned}
T \leq & \frac{|C|}{\frac{\epsilon(n,v)}{2} \cdot |C| - 1} \\
= & \frac{4 \cdot \frac{1}{4} \cdot |C|}{\frac{\epsilon(n,v)}{2} \cdot |C| - 1} \\
= & \frac{4}{(\frac{\epsilon(n,v)}{2} \cdot |C| - 1) \cdot \frac{4}{|C|}} \\
= & \frac{4}{2 \cdot \epsilon(n,v) - \frac{4}{|C|}} = \frac{4}{\epsilon(n,v) + \underbrace{\epsilon(n,v) - \frac{4}{|C|}}_{>0 \text{ by assumption}}} \\
< & \frac{4}{\epsilon(n,v)}.
\end{aligned}
$$

Therefore, the expected runtime of procedure A is in $\mathcal{O}(\frac{1}{\epsilon(v)})$, if a heavy row was found in the first step. B, and subsequently E', obviously have expected runtime $\mathcal{O}(\frac{1}{\epsilon(n,v)})$ in either case. We now want to ensure that if a heavy row was found in the first step, A outputs a second 1 before B stops. Call the random variable Num_A the number of repetitions of A in an execution of E', Num_B equivalently. By Markov's inequality, we know that

$$\Pr[Num_A < 2 \cdot T | HeavyRow] \geq \frac{T}{2 \cdot T} = \frac{1}{2}.$$

By choosing b large enough we can ensure that

$$\Pr[Num_B \geq 2 \cdot T] \geq \frac{1}{g}$$

for some constant g.

Since we hit a heavy row with probability greater than $\frac{1}{2}$ in step one, we have

$$
\begin{aligned}
\Pr[A \text{ finishes before } B] \geq & \Pr[HeavyRow] \cdot \Pr[Num_A < 2 \cdot T | HeavyRow] \cdot \Pr[Num_B \geq 2 \cdot T] \\
\geq & \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{g} = \frac{1}{4 \cdot g}.
\end{aligned}
$$

Thus, when running E' we find the positions of two 1's with probability $\frac{1}{4 \cdot g}$ in expected time $\mathcal{O}(\frac{1}{\epsilon(n,v)})$.

Now, build an extractor E that repeatedly runs E' until E' returns the positions of two 1's. Then, E calculates the transcripts $(com, c, (r, z, d))$ and $(com, c', (r', z', d'))$ corresponding to the 1's. Note that the commitments are the same, since both 1's are from the same row, thus the prover uses the same randomness. Then, if $z = z'$, the extractor can use the special soundness extractor from the underlying Σ-Protocol to compute a witness and output it. If $z \neq z'$, the extractor simply repeats the whole process until $z = z'$.

To find the positions of two 1's, the extractor takes expected time $4 \cdot g \cdot \mathcal{O}(\frac{1}{\epsilon(n,v)}) = \mathcal{O}(\frac{1}{\epsilon(n,v)})$. By the computationally binding property of the trapdoor commitment scheme we know that the case $z \neq z'$ happens at most with negligible probability $\mu(n)$. Thus, E has to repeat the whole process an expected number of $\frac{1}{1-\mu(n)}$ times, therefore E has an expected runtime of $\frac{1}{1-\mu(n)} \cdot \mathcal{O}(\frac{1}{\epsilon(n,v)}) < \frac{p(|v|)}{\epsilon(n,v)-\kappa(n)}$. □

Lemma 3.53. *For a Σ-Protocol $\mathfrak{S}$ for a relation R and a computationally hiding trapdoor commitment scheme $\mathfrak{C}$, the IAASM $\mathsf{ConcZK}_{\mathfrak{S},\mathfrak{C}}$ from Construction 3.51 is a zero-knowledge protocol in the auxiliary string model.*

Proof. Let R be a NP-relation. Let $\mathfrak{S}$ be a Σ-Protocol for R and $\mathfrak{C} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be a trapdoor commitment scheme. Assume that $\mathfrak{C}$ can commit to the announcement of $\mathfrak{S}$. Define the $\mathsf{TrapdoorGen}$ for the zero-knowledge property to be the Setup from $\mathfrak{C}$. Let (T_1, T_2) be algorithms that fulfill the trapdoor property of $\mathfrak{C}$. Let $\mathcal{V}^*$ be any (adversarial concurrent) verifier. Let $v \in L_R$, $(\mathsf{pp}, t) \in \mathsf{TrapdoorGen}(1^n)$ and $y \in \{0,1\}^{p(|v|)}$ for some polynomial $p(\cdot)$. Construct a simulator $M_{\mathcal{V}^*}$ that on input $(1^n, v, \mathsf{pp}, y, t)$ works the following way:

$M_{\mathcal{V}^*}(1^n, v, \mathsf{pp}, y, t)$

1 : Start simulating V^* with input (pp, v, y).
2 : Give com with $(com, St) \leftarrow T_1(\mathsf{pp}, t)$ to $\mathcal{V}^*$.
3 : Receive c.
4 : Run the honest verifier simulator of the Σ-Protocol on input v, c to get z, r.
5 : Compute $d \leftarrow T_2(z, St)$.
6 : Send r, z, d to $\mathcal{V}^*$

Then, V^* sees com, r, z and d and expects distributions as in a normal protocol for these values. By definition of T_1, the value com is correctly distributed. The same holds for z and r, since they are generated by the honest verifier simulator of the underlying Σ-Protocol. Because d is generated by $T_2(z, St)$, $\mathsf{Open}(\mathsf{pp}, com, d)$ opens to z, therefore d is correct as well. □

Theorem 3.54. *If one-way functions exist, then there exists a concurrent black-box zero-knowledge three-round argument of knowledge with negligible knowledge error in the auxiliary string model for any relation for which there exists a Σ-Protocol.*

This follows directly from Lemmas 3.52 and 3.53 and the result of [FS90b], which is that a computationally binding and computationally hiding trapdoor commitment schemes exists if one-way functions exist.

3.10 Fiat-Shamir Heuristic

In this section, we present the *Fiat-Shamir heuristic* proposed by Fiat and Shamir [FS87]. The Fiat-Shamir heuristic can be used to transform a Σ-protocol (Definition 3.43) into a non-interactive zero-knowledge argument of knowledge (Section 3.7.4) or a signature scheme (Definition 3.12) (called *signatures of knowledge*—for a formal treatment, see [CL06]). Its main idea is instead of having a challenge chosen by a verifier, to let the prover compute the challenge herself by applying a function to the announcement, or in the case of signatures, to the announcement and the message to be signed. The non-interactive proof and the signature, respectively, are a transcript of the protocols which result from the self-computed challenge. The function used in this scheme is formally modeled as a random oracle to emulate the random sampling that would be done by an honest verifier. However, in practice the theoretical construct of a random oracle is replaced by a good cryptographic hash function. For more information about random oracles and their practical use, see Bellare and Rogaway [BR93].

We use the Fiat-Shamir heuristic in several ways. Our main application is our reputation system. In particular, a rating is signed using a signature scheme that is obtained by applying the Fiat-Shamir heuristic. For example, this is used to form a proof that a rater bought some item or purchased a service. Therefore, everyone can publicly check whether the product rated was really bought by the rater.

Next, we present the Fiat-Shamir heuristic for both obtaining a non-interactive argument of knowledge and a signature scheme. We consider the variant that is called *strong Fiat-Shamir transformation* in [BPW12].

3.10.1 Non-Interactive Arguments via the Fiat-Shamir Heuristic

As described above, one application for the Fiat-Shamir heuristic is to transform a Σ-protocol, or any kind of three-round interactive argument, into a non-interactive one (Section 3.7.4):

Construction 3.55. Let R be an NP-relation. Let Σ be a three-round interactive argument for relation R with ppts α, γ, ϕ, announcement space A and challenge space C. Further, let $H\colon A \to C$ be a function. We define the following algorithms $\mathcal{P}$ and $\mathcal{V}$:

1. $\mathcal{P}(x, w)$: On input $(x, w) \in R$, it outputs the tuple $\pi := (a, c, r)$, where $a \leftarrow \alpha(x, w; s)$ for $s \leftarrow \mathsf{Coins}(\alpha)$, $c := H(x, a)$ and $r \leftarrow \gamma(x, w, s, c)$.

2. $\mathcal{V}(x, \pi)$: On input x and $\pi = (a, c, r)$, it outputs $b = 1$ if and only if $\phi(x, a, c, r) = 1$ and $c = H(x, a)$, where $b = 1$ is interpreted as $\mathcal{V}$ accepting π, and $b = 0$ otherwise.

Theorem 3.56. *If Σ is a Σ-Protocol and H is modeled as a random oracle, then Construction 3.55 is a non-interactive zero-knowledge argument of knowledge.*

The proof of this theorem is omitted in this work.

3.10.2 Signature Schemes via the Fiat-Shamir Heuristic

In the previous section we have seen how to turn a three round interactive argument into a non-interactive one. By slightly adapting Construction 3.55, we are also able to instantiate a signature scheme based on the interactive argument.

Construction 3.57. Let R be a NP-relation. Let Σ be a three round interactive argument for relation R with ppts α, γ, ϕ, announcement space A and challenge space C. Further, let $H\colon A \times \{0,1\}^* \to C$ be a function. Then, we define the following algorithms Sign and Vrfy:

1. $\mathsf{Sign}(x, w, m)$: On input $(x, w) \in R$ and $m \in \{0,1\}^*$, it outputs the tuple $\sigma := (a, c, r)$, where $a \leftarrow \alpha(x, w; s)$ for $s \leftarrow \mathsf{Coins}(\alpha)$, $c := H(x, a, m)$ and $r \leftarrow \gamma(x, w, s, c)$.

2. $\mathsf{Vrfy}(x, m, \sigma)$: On input x and $\sigma = (a, c, r)$, it output $b = 1$ if and only if $\phi(x, a, c, r) = 1$ and $c = H(x, a, m)$, where $b = 1$ is interpreted as $\mathcal{V}$ accepting σ and $b = 0$ otherwise.

Note that Construction 3.57 does not yield a syntactically correct signature scheme as given in Definition 3.12. To be precise, the Setup and Gen algorithms are missing. In practice, the setup of the signature scheme looks exactly like the protocol's setup. For the key generation one would implement an *instance generator* that outputs a pair $(x, w) \in R$, where x is the public key and w the secret key. We decided to keep the construction simple and omit these details.

Theorem 3.58. *If Σ is a Σ-Protocol and H is modeled as a random oracle, then Construction 3.57 is existentially unforgeable under an adaptive chosen-message attack.*

The proof of this theorem is omitted. A proof for a concrete Σ-protocol can, for example, be found in [PS96].

3.11 Proofs of Partial Knowledge

In the construction of our anonymous credential system we integrate predicates which, for example, are satisfied if a credential from issuer i contains an attribute a equal to value v_1 *or* v_2. Concretely, these predicates consist of AND-gates, OR-gates and even threshold gates and use a set of predefined relations like equality and inequality (cf. Section 4.1.3). Using the technique of *proofs of partial knowledge* [CDS94] users can prove their credentials satisfy the predicate,

but do not reveal the concrete satisfying configuration. In particular the technique enables us to combine different Σ-protocols into a single one according to a given access structure (Definition 3.31) and a corresponding t-semi-smooth secret-sharing scheme (Definition 3.35). The verifier of such a Σ-protocol is eventually convinced, that the prover's witness corresponds to some qualified set of the given access structure.

We adapt the construction by Cramer, Damgård, and Schoenmakers [CDS94] to distinct relations and show that the construction yields a Σ-protocol instead of a witness indistinguishable proof of knowledge.

Construction 3.59. Let $n \in \mathbb{N}$ and let t be a polynomial. Let $\phi(X_1, \dots, X_n)$ be a polynomial-time computable, monotone predicate on variables $X_i \in \{0, 1\}$ with the corresponding (monotone) access structure Γ_ϕ (Definition 3.33). Let $(\mathcal{P}_i, \mathcal{V}_i)$, for $i = 1, \dots, n$, be a Σ-protocol for some NP-Relation R_i. We denote the components of protocol $(\mathcal{P}_i, \mathcal{V}_i)$ by $(\alpha_i, C_i, \gamma_i, \psi_i, \mathcal{S}_i)$, where α_i is a ppt computing the announcement, C_i is the finite challenge space, γ_i is a ppt computing the responses, ψ_i is a polynomial-time verifiable predicate and $\mathcal{S}_i$ is the special honest-verifier zero-knowledge simulator. Further, let $(\mathsf{Share}, \mathsf{Recon}, \mathsf{CheckConsistency}, \mathsf{Complete})$ be a $t(n)$-semi-smooth secret-sharing scheme (Definition 3.35) for access structure Γ^*_ϕ with n parties, the set of secrets K and the domain of shares $K_j = C_j$ of party p_j for $j = 1, \dots, n$. We define the relation

$$R_\phi := \{((x_1, \dots, x_n), (w_1, \dots, w_n)) \mid \forall i = 1, \dots, n : X_i = R_i(x_i, w_i) \wedge \phi(X_1, \dots, X_n) = 1\}.$$

Consider the following construction of a protocol $(\mathcal{P}, \mathcal{V})$ with challenge space $C = K$:

1. The prover starts by identifying the relations that can be satisfied by its witnesses. We denote the set of all indices i such that $(x_i, w_i) \in R_i$ by $\mathbb{A}$.

   Remark. Assuming $\mathcal{P}$ can satisfy the predicate, $\mathbb{A}$ is a qualified set for Γ_ϕ, i. e. $\mathbb{A} \in \Gamma_\phi$. By definition of access structures (Definition 3.31), we have that its complement in $\{1, \dots, n\}$, $\bar{\mathbb{A}}$, is unqualified in Γ^*_ϕ.

2. Based on set $\mathbb{A}$, the prover computes the announcement:
   a) Choose an arbitrary (not necessarily random) secret s' and share it using Share, i. e. obtain $(s'_1, \dots, s'_n) \leftarrow \mathsf{Share}(s')$.
   b) For all $j \in \bar{\mathbb{A}}$, set $c_j := s'_j$, and then discard $(s'_1, \dots, s'_n)$.
   c) Now, compute the actual announcement:
      i. For all $i \in \mathbb{A}$, compute $a_i := \alpha_i(x_i, w_i; k_i)$ with $k_i \leftarrow \mathsf{Coins}(\alpha_i)$.
      ii. For all $j \in \bar{\mathbb{A}}$, compute $(a_j, c_j, r_j) \leftarrow \mathcal{S}_j(x_j, c_j)$.
      iii. Send $(a_1, \dots, a_n)$ to the verifier.

3. The verifier chooses a challenge $c \leftarrow C$ and sends it to the prover.

4. Upon receiving c, the prover does the following:
   a) Obtain a full set of shares: $(c_1, \dots, c_n) \leftarrow \mathsf{Complete}(c, \{c_j \mid j \in \bar{\mathbb{A}}\})$.
   b) For all $j \in \bar{\mathbb{A}}$, we already have computed responses r_j in step 2c).
   c) For all $i \in \mathbb{A}$, compute $r_i := \gamma_i(x_i, w_i, k_i, c_i)$.
   d) Send $(r_1, c_1), \dots, (r_n, c_n)$ as response to the verifier.

5. Upon receiving $(r_1, c_1), \dots, (r_n, c_n)$, the verifier accepts if and only if
   a) The shares $(c_1, \dots, c_n)$ are all consistent with c, i. e. $\mathsf{CheckConsistency}(c, (c_1, \dots, c_n)) = 1$.

b) $\forall i = 1, \ldots, n : \psi_i(x_i, a_i, c_i, r_i) = 1$.

Theorem 3.60. *Let ϕ be a polynomial-time computable, monotone predicate on n variables $X_i \in \{0, 1\}$. Let R_ϕ be as defined in Construction 3.59. Then, Construction 3.59 is a Σ-protocol for relation R_ϕ.*

Theorem 3.60 follows from the three lemmas stated and proven next.

Lemma 3.61. *Construction 3.59 is complete.*

Proof. We need to show that verifier $\mathcal{V}$ accepts with probability 1, if prover $\mathcal{P}$ and $\mathcal{V}$ follow the protocol. Let $((x_1, \ldots, x_n), (w_1, \ldots, w_n)) \in R_\phi$. We show that the checks 5a) and b) always accept. Let us start with check b). Consider an arbitrary $i \in \{1, \ldots, n\}$. Distinguish the two cases i) $i \in \mathbb{A}$ and ii) $i \in \bar{\mathbb{A}}$. In both cases the verifier will accept, i. e. $\psi_i(x_i, a_i, c_i, r_i) = 1$:

i) By completeness of Σ_i, the check will accept for the chosen challenge c_i.

ii) By the honest-verifier zero-knowledge property of Σ_i, the transcripts output by $\mathcal{S}_i$ are accepting for the given challenge c_i implying that this check also accepts.

It remains to show that the check 5a) will accept, i. e. the challenges $(c_1, \ldots, c_n)$ computed are consistent with the secret c. By definition of $\bar{\mathbb{A}}$, it is unqualified for the secret-sharing's access structure Γ^*_ϕ. Property 4 of semi-smooth secret-sharing (Definition 3.35) yields that $\mathsf{Complete}$, given a secret and an unqualified set of shares, outputs a full set of shares that is consistent with the given secret n step 4a). Hence, check 5a) will accept. □

Lemma 3.62. *Construction 3.59 is special sound.*

Proof. Fix an instance $(x_1, \ldots, x_n)$ of R_ϕ. We show, that given two arbitrary accepting transcripts for instance $(x_1, \ldots, x_n)$, $((a_i)_{i=1}^n, c, (r_i, c_i)_{i=1}^n)$ and $((a_i)_{i=1}^n, c', (r'_i, c'_i)_{i=1}^n)$ with $c \neq c'$, one can efficiently extract a witness for $(x_1, \ldots, x_n)$. Since the transcripts are accepting, we have that $(c_1, \ldots, c_n)$ is consistent with c and $(c'_1, \ldots, c'_n)$ is consistent with c', respectively.

We define the extractor $\mathcal{E}$ for protocol $(\mathcal{P}, \mathcal{V})$ as follows: On input $(x_1, \ldots, x_n)$, $((a_i)_{i=1}^n, c,$ $(r_i, c_i)_{i=1}^n)$ and $((a_i)_{i=1}^n, c', (r'_i, c'_i)_{i=1}^n)$, output $(w_1, \ldots, w_n)$, where $w_i \leftarrow \mathcal{E}_i(x_i, a_i, c_i, c'_i, r_i, r'_i)$ and $\mathcal{E}_i$ is the special soundness extractor of $(\mathcal{P}_i, \mathcal{V}_i)$. It remains to show that $\mathcal{Q} := \{i \mid c_i \neq c'_i\}$ has a non-trivial intersection with all qualified sets contained in Γ^*, and thus by Lemma 3.32 it holds $\mathcal{Q} \in \Gamma$. To show this we consider an arbitrary qualified set $B \in \Gamma^*$. We have $c \neq c'$, which implies that there needs to be an $i \in B$ such that $c_i \neq c'_i$; otherwise, this would contradict the secret-sharing scheme's correctness or $c \neq c'$. This, in turn, gives us two accepting transcripts (a_i, c_i, r_i) and (a_i, c'_i, r'_i) with $c_i \neq c'_i$ of protocol $(\mathcal{P}_i, \mathcal{V}_i)$ for instance x_i, meaning that $\mathcal{E}_i$ will extract a valid witness w_i for x_i. As stated above, it holds $|\{i \in B \mid c_i \neq c'_i\}| \geq 1$ for all $B \in \Gamma^*$. Thus, it holds $\bigcup_{B \in \Gamma^*} \{i \in B \mid c_i \neq c'_i\} \bigcap B' \neq \emptyset$ for all $B' \in \Gamma^*$. Since $\bigcup_{B \in \Gamma^*} \{i \in B \mid c_i \neq c'_i\} \subseteq \mathcal{Q}$, we also have $\mathcal{Q} \cap B' \neq \emptyset$ for all $B' \in \Gamma^*$.

Consequently, the extractor $\mathcal{E}_i$ will output a valid witness w_i for x_i for all $i \in \mathcal{Q} = \{i \mid c_i \neq c'_i\}$, $\mathcal{Q}$ is qualified in Γ. Note that for $i \notin \mathcal{Q}$ it is not necessarily the case that extractor $\mathcal{E}_i$ obtains a valid input and therefore might output an error symbol. However, this suffices since the relation R_ϕ only considers the witnesses corresponding to a qualified set in Γ. □

Lemma 3.63. *Construction 3.59 is special honest-verifier zero-knowledge.*

Proof. We construct a simulator $\mathcal{S}$ that outputs transcripts distributed as in the real protocol given an instance $x_1, \ldots, x_n$ and challenge c. Let $\mathcal{S}_i$ be the simulator given by the special honest-verifier zero-knowledge property of protocol $(\mathcal{P}_i, \mathcal{V}_i)$. The simulator $\mathcal{S}$ outputs, on input $(x_1, \ldots, x_n)$ and c, the transcript $((a_i)_{i=1}^n, c, (r_i, c_i)_{i=1}^n)$ such that $(c_1, \ldots, c_n) \leftarrow \mathsf{Share}(c)$ and $(a_i, c_i, r_i) \leftarrow \mathcal{S}_i(x_i, c_i)$.

Let us argue that the distribution of transcripts output by $\mathcal{S}$ and the real protocol are identical. The sub-challenges $(c_1, \ldots, c_n)$ output by $\mathcal{S}$ are generated by running Share on secret c. In the real protocol, $(c_1, \ldots, c_n)$ are obtained by running Complete on input c and an unqualified set of shares $\{c_j \mid j \in \bar{\mathbb{A}}\}$ (step 2c) i). The shares c_j in turn are obtained in steps 2a) and b) by running Share on an arbitrary secret and discarding every share corresponding to a party in the qualified set $\mathbb{A}$. By perfect privacy of secret-sharing schemes (Definition 3.34), we have that the shares $\{c_j \mid j \in \bar{\mathbb{A}}\}$ are distributed independently of the secret, meaning that the distributions $\mathsf{Share}(c)_{\bar{\mathbb{A}}}$ and $\mathsf{Share}(s')_{\bar{\mathbb{A}}}$, for some arbitrary secret s', are identically. As this distribution is independent of s' we write $\mathsf{Share}_{\bar{\mathbb{A}}}$. By Property 3 of semi-smooth secret-sharing schemes (Definition 3.35) we have that algorithm Complete, on input secret c and unqualified $\{c_j \mid j \in \bar{\mathbb{A}}\}$ distributed according to $\mathsf{Share}_{\bar{\mathbb{A}}}$, outputs shares $(c_1, \ldots, c_n)$ consistent with challenge c and distributed according to $\mathsf{Share}(c)$.

It remains to argue that (a_i, r_i) are distributed correctly. In the real protocol, we simulate for all $j \in \bar{\mathbb{A}}$ before obtaining the challenge and react on the challenge for all $i \in \mathbb{A}$. Given that c_j is distributed as in the real protocol (see above), it is easy to see that the simulated (a_j, r_j) are distributed identically. For (a_i, r_i), we run the algorithms α_i and γ_i corresponding to $(\mathcal{P}_i, \mathcal{V}_i)$. The algorithms are used just as in a run of $(\mathcal{P}_i, \mathcal{V}_i)$ except that the challenge c_i was obtained by secret-sharing instead of given by a verifier. Since a special honest-verifier simulator is given a challenge, simulator $\mathcal{S}_i$ outputs (a_i, r_i) distributed identical to algorithms α_i and γ_i of $(\mathcal{P}_i, \mathcal{V}_i)$. □

3.12 Accumulators

An accumulator can be used to prove that a value x belongs to a set X. The advantage of accumulators is that this set membership can be proven efficiently by using an accumulator value which does not depend on the size of the set. Proving set membership is done by using a witness w_x for x to show that this x was used to create the accumulator value V. Witnesses and accumulator values will only be one group element each in our construction, which is important for the efficiency of this accumulator. On the other side, it is computationally not feasible to prove that a value $x' \notin X$ is part of the accumulator value.

We start to define static accumulators where X is fixed at the beginning and state when such an accumulator is correct and secure. Afterwards, we generalize these definitions to dynamic accumulators where values can be inserted or deleted to X.

3.12.1 Static Accumulators

We first define the syntax of a static accumulator. Note that there are various existing definitions for accumulators. However, we will concentrate on the accumulators where no secret key is needed to generate accumulator value V and witnesses w_i for accumulated values i. This means that everyone can create an accumulator value and witnesses. Moreover, we only talk about deterministic accumulators here, which means that the creation of an accumulator value is done deterministically.

Definition 3.64 (Static Accumulator Scheme)**.** A *static accumulator scheme* Π_{acc} is a tuple of (ppt) algorithms (Gen, AccCreate, WitCreate, Vrfy), where

1. $\mathsf{Setup}(1^\lambda, 1^q)$: On inputs $1^n, 1^q$ it, (probabilistically) outputs the accumulator public parameters pp. These public parameters contain a set of values U. $\lambda \in \mathbb{N}$ denotes the security parameter and $q \in \mathbb{N}$ the upper bound for the number of accumulated values.

2. AccCreate(pp,X): On inputs public parameters pp and set $X \subseteq U$, it (deterministically) outputs an accumulator value V.

3. $\mathsf{WitCreate}(\mathsf{pp},X,\ i)$: On inputs public parameters pp, set $X \subseteq U$ and value $i \in U$, it (deterministically) outputs a witness w_i or a failure symbol $\bot$.

4. $\mathsf{Vrfy}(\mathsf{pp}, V, i, w_i)$: On inputs public parameters pp, accumulator value V, value $i \in U$ and witness w_i, it deterministically outputs 0 or 1. We interpret 1 as *valid* and 0 as *invalid.*

We consider an accumulator scheme *correct* if for all $\lambda, q \in \mathbb{N}$, all $\mathsf{pp} \in [\mathsf{Setup}(1^\lambda, 1^q)], X \subseteq U, |X| \leq q, i \in X$:

$$\mathsf{Vrfy}(\mathsf{pp}, \mathsf{AccCreate}(\mathsf{pp}, X), i, \mathsf{WitCreate}(\mathsf{pp}, X, i) = 1.$$

The definition of (static) accumulators does not include any security yet, which we want to add in the next definition. It should be computationally infeasible to compute a witness w_i for an $i \notin X$.

Definition 3.65 (Security of an Accumulator Scheme). An accumulator scheme Π_{acc} is secure if for all ppt algorithms $\mathcal{A}$ there exists a negligible function μ such that for all $\lambda \in \mathbb{N}$

$$\Pr[Exp^{\text{witforge}}_{\mathcal{A},\Pi_{acc}}(\lambda) = 1] \leq \mu(\lambda)$$

where $Exp^{\text{witforge}}_{\mathcal{A},\Pi_{acc}}(\lambda)$ is defined as follows:

1. $\mathcal{A}$ chooses a value $q \in \mathbb{N}$ and receives public parameters $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, 1^q)$

2. Eventually, $\mathcal{A}$ outputs (X, V, i, w_i)

3. The experiment returns 1 if $V = \mathsf{AccCreate}(\mathsf{pp}, X)$, $\mathsf{Vrfy}(\mathsf{pp}, V, i, w_i) = 1$, and $i \notin X \subseteq U$ with $|X| \leq q$. Otherwise it returns 0.

The definition for static accumulators does not allow for changes to the set of accumulated values X. This would mean that inserting or deleting a single element to X requires a completely new accumulator. Therefore, one would again need time linear in the set of X for such a small change. To overcome this drawback, we define *dynamic accumulators* in the next section.

3.12.2 Dynamic Accumulators

In contrast to static accumulators, we now want to be able to insert and delete values to X. This means that we need two additional algorithms, namely one to insert values and one to delete values. Moreover, existing witnesses have to be updated after a change to the set X. After inserting/deleting a value to X the accumulator value V changes and an old witness w_i would not be valid anymore even if i is still accumulated. This means we need additional algorithms which can be used to update existing witnesses after inserting or deleting a value. Note that we have to ensure the set X is still of a valid size smaller or equal to q after inserting a new value.

Definition 3.66 (Dynamic Accumulator Scheme). A *dynamic accumulator scheme* Π_{acc} is a static accumulator scheme with following additional deterministic, polynomial-time algorithms:

1. $\mathsf{AccInsert}(\mathsf{pp}, V, i, X)$: On input public parameters pp, accumulator value V, value $i \in U$ and set $X \subseteq U$, it outputs the accumulator value V' or a failure symbol $\bot$.

2. $\mathsf{AccDelete}(\mathsf{pp}, V, i, X)$: On input public parameters pp, accumulator value V, value $i \in U$ and set $X \subseteq U$, it outputs the accumulator value V' or a failure symbol $\bot$.

3. $\mathsf{WitUpdateInsert}(\mathsf{pp}, X, X', V, i, i', w_i)$: On input public parameters pp, sets $X, X' \subseteq U$, accumulator value V, values i, i' and witness w_i, it outputs witness w_i'.

4. $\mathsf{WitUpdateDelete}(\mathsf{pp}, X, X', V', i, i', w_i)$: On input public parameters pp, sets $X, X' \subseteq U$, accumulator value V, values i, i' and witness w_i, it outputs witness w_i'.

Additionally to the correctness of a static accumulator, we require that the insertion and deletion of values as well as updating a witness work correctly. More formally, this means:

- For all $i \in U, i \notin X$: $\mathsf{AccInsert}(\mathsf{pp}, \mathsf{AccCreate}(\mathsf{pp}, X), i, X) = \mathsf{AccCreate}(\mathsf{pp}, X \cup \{i\})$
- For all $i \in U, i \in X$: $\mathsf{AccDelete}(\mathsf{pp}, \mathsf{AccCreate}(\mathsf{pp}, X), i, X) = \mathsf{AccCreate}(\mathsf{pp}, X \setminus \{i\})$
- For an inserted value $i \in U, i' \in X' \setminus X$ where $X \subset X'$ and $X' \setminus X = \{i'\}$, and for all $i \in X$:

$$\mathsf{WitUpdateInsert}(\mathsf{pp}, X, X', V, i, i', \mathsf{WitCreate}(\mathsf{pp}, X, i)) = \mathsf{WitCreate}(\mathsf{pp}, X', i)$$

- For a deleted value $i' \in X \setminus X'$ where $X' \subset X$ and $X \setminus X' = \{i'\}$, and for all $i \in X$:

$$\mathsf{WitUpdateDelete}(\mathsf{pp}, X, X', V', i, i', \mathsf{WitCreate}(\mathsf{pp}, X, i)) = \mathsf{WitCreate}(\mathsf{pp}, X', i)$$

Because we can just express these new functions by algorithms that already existed for static accumulators, we do not need to change the security definition here. An adversary $\mathcal{A}$ could just use several different sets during the experiment where one element is inserted/deleted before creating its output.

3.12.3 Nguyen Accumulator

This section introduces a concrete instantiation of a dynamic accumulator, namely the one described by Nguyen in [Ngu05]. We will start with an instantiation of the algorithms introduced in the previous two sections and show that these algorithms work correctly. Afterwards, we prove the security of the Nguyen accumulator if the modified q-SDH assumption holds (Definition 3.9). Note that our construction differs in the type of the pairing. Nguyen uses a type 1 pairing, while we stick to a type 3 pairing to achieve more consistency with our other constructions.

Construction 3.67 (Nguyen Accumulator)**.** Let G be a type 3 bilinear group generator (Definition 3.2). A Nguyen accumulator is a tuple of (ppt) algorithms (Setup, $\mathsf{AccCreate}$, $\mathsf{WitCreate}$, $\mathsf{AccInsert}$, $\mathsf{AccDelete}$, $\mathsf{WitUpdate}$, Vrfy) such that:

- Setup $(1^\lambda, 1^q)$ uses the generator G to get $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^\lambda)$. It outputs the public parameters pp in form of $(p, e, g, \tilde{g}, \tilde{g}^s, t)$ with $t = (g^s, g^{s^2}, \ldots, g^{s^q})$. Here, $g \leftarrow \mathbb{G}_1 \setminus \{1\}$, $\tilde{g} \leftarrow \mathbb{G}_2 \setminus \{1\}$ are generators of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. The upper bound on the number of accumulated values is q and $s \leftarrow \mathbb{Z}_p^*$. While $\mathbb{Z}_p \setminus \{-s\}$ is the domain of values that can be accumulated, i.e. $U = \mathbb{Z}_p \setminus \{-s\}$.
- $\mathsf{AccCreate}$ (pp, X) Takes the set $X \subset \mathbb{Z}_p$ with $|X| \leq q$ and outputs the accumulator value V for this set with $V = g^{\prod_{x \in X}(x+s)}$. This value can be computed without the knowledge of s using the tuple t. To do so, one can compute the coefficients a_i of the polynomial $\sum_{i=0}^{|X|} a_i s^i = \prod_{x \in X}(x+s)$. Using this, we get:

$$V = g^{\prod_{x \in X}(x+s)} = g^{\sum_{i=0}^{|X|} a_i s^i} = \prod_{i=0}^{|X|} (g^{s^i})^{a_i}$$

  The coefficients a_i can be computed from the set X while the g^{s^i} are contained in t.

- $\mathsf{WitCreate}(\mathsf{pp}, X, i)$ calculates and outputs the witness $w_i = g^{\prod_{x \in X \setminus \{x_i\}}(x+s)}$. This can be done using a polynomial like described above.

- $\mathsf{Vrfy}(\mathsf{pp}, V, x_i, w_i)$ checks if $e(w_i, \tilde{g}^{x_i} \cdot \tilde{g}^s) = e(V, \tilde{g})$. If the equation holds, Vrfy outputs 1, otherwise 0.

- $\mathsf{AccInsert}(\mathsf{pp}, V, X, i)$: If the number of accumulated values $|X| = q$ or the value x_i, which is to be inserted, is not in the right domain ($x_i \notin \mathbb{Z}_p \setminus \{-s\}$) output the failure symbol $\perp$. The new accumulator value V' can be computed using AccCreate (pp, $X \cup \{x_i\}$).

- $\mathsf{AccDelete}(\mathsf{pp}, V, X, i)$: If the value x_i, which is to be deleted, is not in the accumulated so far ($x_i \notin X$) output the failure symbol $\perp$. The new accumulator value V' can be computed using AccCreate (pp, $X \setminus \{x_i\}$).

- $\mathsf{WitUpdateInsert}(\mathsf{pp}, X, X', V, i, x', w_i)$: Computes the updated witness $w_i' = V \cdot w_i^{x'-x_i}$ for the new accumulator value V' after the insertion of x'.

- $\mathsf{WitUpdateDelete}(\mathsf{pp}, X, X', V', i, x', w_i)$: Computes the updated witness $w_i' = (\frac{w_i}{V'})^{(x'-x_i)^{-1}}$ for the new accumulator value V' that changed due to the deletion of x'.

Note that because the accumulator value is computed using a product, V is invariant over the order in which elements are accumulated. Nevertheless, the order of executing WitUpdateDelete and WitUpdateInsert is important when WitUpdate is called after multiple values were inserted/deleted to the set X. One could keep track of an archive for that which keeps track of the inserted/deleted values. This archive can then be used when updating a witness. Additionally, it can keep track of the updated accumulator values, such that a user does not have to calculate the different Vs herself. If s is known to a trusted party which computes the accumulator values V, insertion and deletion can be done more efficiently:

- Insertion: The new accumulator value V' over the set $X \cup \{x_i\}$ is computed as $V' = V^{x_i+s}$.

- Deletion: The new accumulator value V' over the set $X \setminus \{x_i\}$ is computed as $V' = V^{(x_i+s)^{-1}}$.

It remains to show that this construction works correctly and is secure. We start by showing the correctness by proving the correctness of the accumulator's algorithms:

Lemma 3.68. *The accumulator presented in Construction 3.67 is correct.*

Proof. 1. $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{AccCreate}(\mathsf{pp}, X), i, \mathsf{WitCreate}(\mathsf{pp}, X, i) = 1$

Let $V = \mathsf{AccCreate}(\mathsf{pp}, X)$ and $w_i = \mathsf{WitCreate}(\mathsf{pp}, X, i)$, then
$\mathsf{Vrfy}(\mathsf{pp}, V, i, w_i) = 1 \Leftrightarrow e(w_i, \tilde{g}^{x_i} \cdot \tilde{g}^s) = e(V, \tilde{g})$.
With $V = g^{\prod_{x \in X}(x+s)}$ and $w_i = g^{\prod_{x \in X \setminus \{x_i\}}(x+s)}$, we get:

$$\begin{aligned} e(w_i, \tilde{g}^{x_i} \cdot \tilde{g}^s) &= e(g^{\prod_{x \in X \setminus \{x_i\}}(x+s)}, \tilde{g}^{x_i+s}) \\ &= e(g, \tilde{g})^{(\prod_{x \in X \setminus \{x_i\}}(x+s)) \cdot (x_i+s)} \\ &= e(g, \tilde{g})^{\prod_{x \in X}(x+s)} \\ &= e(g^{\prod_{x \in X}(x+s)}, \tilde{g}) \\ &= e(V, \tilde{g}) \end{aligned}$$

2. For all $i \in U, i \notin X$: AccInsert (pp, AccCreate (pp, X), i, X) = AccCreate (pp, $X \cup \{i\}$)

This is true by definition, thus we only show that the more efficient way with knowledge of s is correct:

With $\mathsf{AccCreate}(\mathsf{pp}, X) = g^{\prod_{x\in X}(x+s)}$, we get:

$$\begin{aligned}\mathsf{AccInsert}(\mathsf{pp}, \mathsf{AccCreate}(\mathsf{pp}, X), i, X) &= (g^{\prod_{x\in X}(x+s)})^{(i+s)}\\ &= g^{\prod_{x\in X\cup\{i\}}(x+s)}\\ &= \mathsf{AccCreate}(\mathsf{pp}, X \cup \{i\})\end{aligned}$$

3. For all $i \in U, i \in X$: $\mathsf{AccDelete}$ (pp, $\mathsf{AccCreate}$ (pp, X), i, X) $=$ $\mathsf{AccCreate}$ (pp, $X \setminus \{i\}$)

   This is true by definition, thus we only show that the more efficient way with knowledge of s is correct:
   With $\mathsf{AccCreate}(\mathsf{pp}, X) = g^{\prod_{x\in X}(x+s)}$, we get:

$$\begin{aligned}\mathsf{AccDelete}(\mathsf{pp}, \mathsf{AccCreate}(\mathsf{pp}, X), i, X) &= (g^{\prod_{x\in X}(x+s)})^{(i+s)^{-1}}\\ &= g^{\prod_{x\in X\setminus\{i\}}(x+s)}\\ &= \mathsf{AccCreate}(\mathsf{pp}, X \setminus \{i\})\end{aligned}$$

4. For an inserted value $x' \in X' \setminus X$ and for all $x_i \in X \cap X'$:
   $\mathsf{WitUpdateInsert}$ $(\mathsf{pp}, X, X', V, x_i, x', \mathsf{WitCreate}(\mathsf{pp}, X)) = \mathsf{WitCreate}(\mathsf{pp}, X', x_i)$
   With $V = g^{\prod_{x\in X}(x+s)}$ and $w_i = g^{\prod_{x\in X\setminus\{x_i\}}(x+s)}$, we get:

$$\begin{aligned}V \cdot w_i^{x'-x_i} &= g^{\prod_{x\in X}(x+s)} \cdot g^{(x'-x_i)\prod_{x\in X\setminus\{x_i\}}(x+s)}\\ &= g^{(\prod_{x\in X\setminus\{x_i\}}(x+s))((x_i+s)+(x'-x_i))}\\ &= g^{(\prod_{x\in X\setminus\{x_i\}}(x+s))(x'+s)}\\ &= g^{\prod_{x\in(X\setminus\{x_i\})\cup\{x'\}}(x+s)}\\ &= \mathsf{WitCreate}(\mathsf{pp}, X', x_i)\end{aligned}$$

5. For a deleted value $x' \in X \setminus X'$ and for all $x_i \in X \cap X'$:
   $\mathsf{WitUpdateDelete}$ $(\mathsf{pp}, X, X', V', x_i, x', \mathsf{WitCreate}(\mathsf{pp}, X)) = \mathsf{WitCreate}(\mathsf{pp}, X', x_i)$
   With $V' = g^{\prod_{x\in X\setminus\{x'\}}(x+s)}$ and $w_i = g^{\prod_{x\in X\setminus\{x_i\}}(x+s)}$, we get:

$$\begin{aligned}(w_i \cdot V'^{-1})^{(x'-x_i)^{-1}} &= (\frac{g^{\prod_{x\in X\setminus\{x_i\}}(x+s)}}{g^{\prod_{x\in X\setminus\{x'\}}(x+s)}})^{(x'-x_i)^{-1}}\\ &= g^{(\prod_{x\in X\setminus\{x_i,x'\}}(x+s))((x'+s)-(x_i+s))(\frac{1}{x'-x_i})}\\ &= g^{(\prod_{x\in X\setminus\{x_i,x'\}}(x+s))(\frac{x'-x_i}{x'-x_i})}\\ &= g^{\prod_{x\in X\setminus\{x_i,x'\}}(x+s)}\\ &= \mathsf{WitCreate}(\mathsf{pp}, X', x_i)\end{aligned}$$

□

The last remaining step is showing that this construction fulfills the security requirement defined in Definition 3.65. We will show that breaking the security of the Nguyen Accumulator would also break the (modified) q-SDH assumption. As described in Section 3.12.2, the algorithms for deletion, insertion and updating a witness do not help an adversary performing in the security experiment.

Theorem 3.69 (Security of the Nguyen accumulator). *The Nguyen accumulator Π_{acc} introduced in Construction 3.67 is secure with respect to Definition 3.65 if the modified q-SDH assumption holds (Definition 3.9). Here, q denotes the upper bound of the number of accumulated values.*

Proof. Assume that there is a ppt adversary $\mathcal{A}$ that is able to break the security of Π_{acc}. This means $\mathcal{A}$ wins the game $Exp^{\text{witforge}}_{\mathcal{A},\Pi_{acc}}(\lambda)$ with non-negligible probability (Definition 3.65). Using $\mathcal{A}$, we construct an adversary $\mathcal{B}$ that can break the modified q-SDH assumption. Let G be a type 3 bilinear generator and let $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n)$ with $g, \tilde{g}$ as generators of $\mathbb{G}_1, \mathbb{G}_2$. The additional input $\mathcal{B}$ gets from the q-SDH challenge is the tuple $t = (\tilde{g}, \tilde{g}^s, g, g^s, g^{s^2}, \ldots, g^{s^q})$ with $s \leftarrow \mathbb{Z}_p^*$. To use the adversary $\mathcal{A}$ we include those inputs into the public parameters pp of Π_{acc}. We further denote the output of $\mathcal{A}$ with (X, V, c, W) with $X = \{x_1, \ldots, x_k\} \subseteq \mathbb{Z}_p \setminus \{-s\}$ for a $k \leq q$, $c \in \mathbb{Z}_p \setminus (\{-s\} \cup X)$ and $V, W \in \mathbb{G}_1$.

Because $\mathcal{A}$ wins the game with non-negligible probability, we know $V = \mathsf{AccCreate}(\mathsf{pp}, X) = g^{\prod_{x \in X}(x+s)}$, which can be computed from t and X without knowledge of s using $\mathsf{AccCreate}$ as stated in the construction. For the same reason, it holds with non-negligible probability that $\mathsf{Vrfy}(\mathsf{pp}, V, c, W) = 1$ where c and W is the rest of $\mathcal{A}$'s output.

We now have X, c and W breaking the security of Π_{acc} where the groups, the pairing and the public parameter correspond to the input for the q-SDH challenge. However, $\mathcal{B}$ does yet have to compute the tuple $(c, g^{(c+s)^{-1}})$ breaking this challenge.

Because the verification of Π_{acc} succeeds and g and $\tilde{g}$ are generators, we have

$$\begin{aligned} e(W, \tilde{g}^c \cdot \tilde{g}^s) &= e(V, \tilde{g}) \\ \Leftrightarrow e(W, \tilde{g})^{(c+s)} &= e(V, \tilde{g}) \\ \Leftrightarrow e(W^{c+s}, \tilde{g}) &= e(V, \tilde{g}) \\ \Rightarrow W^{c+s} &= V. \end{aligned}$$

$\mathcal{B}$ can combine V and W to compute $g^{(c+s)^{-1}}$ with c being part of the output of $\mathcal{A}$. This can be done by first applying polynomial division and using this result to compute the tuple which breaks the q-SDH assumption. We describe the necessary steps in more detail now. As stated above, we know that $V = g^{\prod_{x \in X}(x+s)} = g^{P(s)}$ with $P(s) = \prod_{x \in X}(x+s)$ being a polynomial with s as the variable. Using polynomial division, we can rewrite $P(s)$:

$$P(s) = (c+s) \cdot P'(s) + r$$

Since for c, X output by $\mathcal{A}$ it holds $c \notin X$, the remainder has to be unequal to zero, i.e. $r \neq 0$. Additionally, the degree of r has to be smaller than the degree of (c+s) which has a degree of one. Combining these two facts, we know that $r \in \mathbb{Z}_p \setminus \{0\}$ which $\mathcal{B}$ can compute using polynomial division. This yields

$$W^{c+s} = V = g^{\prod_{x \in X}(x+s)} = g^{P(s)} = g^{(c+s) \cdot P'(s) + r}.$$

Taking a look at the following equations, one can see that $\mathcal{B}$ now has every necessary information to compute $g^{(c+s)^{-1}}$.

$$\begin{aligned} W^{c+s} &= g^{(c+s) \cdot P'(s) + r} \\ \Leftrightarrow W &= g^{P'(s)} \cdot g^{\frac{r}{c+s}} \\ \Leftrightarrow \frac{W}{g^{P'(s)}} &= g^{\frac{r}{c+s}} \\ \Leftrightarrow \left(\frac{W}{g^{P'(s)}}\right)^{r^{-1}} &= g^{(c+s)^{-1}} \end{aligned}$$

Therefore, $\mathcal{B}$ can compute $(\frac{W}{g^{P'(s)}})^{r^{-1}}$ to get $g^{(c+s)^{-1}}$. Thus, the adversary $\mathcal{B}$ can output the tuple $(c, g^{(c+s)^{-1}})$ using the output created by the adversary $\mathcal{A}$ against Π_{acc} and hereby break the q-SDH assumption. □

4 Anonymous Credential and Reputation System

Contents

4.1 Preliminaries . . . **54**
- 4.1.1 Signing Partially Committed Values . . . 54
  - 4.1.1.1 Construction of a Scheme for Signing Partially Committed Values . . . 56
- 4.1.2 Proving Knowledge of a Signature . . . 59
  - 4.1.2.1 Construction of a Scheme for Proving Knowledge of a Signature . . . 59
- 4.1.3 Predicates . . . 60
  - 4.1.3.1 Equality Proofs for Attributes . . . 61
  - 4.1.3.2 Inequality Proofs for Attributes . . . 62
  - 4.1.3.3 Membership Proofs for Attributes . . . 64

4.2 Basic Anonymous Credential System . . . **68**
- 4.2.1 Definition . . . 68
- 4.2.2 Security Notions . . . 70
  - 4.2.2.1 Anonymity . . . 71
  - 4.2.2.2 Soundness . . . 72
- 4.2.3 Construction of an ACS . . . 74
- 4.2.4 Security Proofs . . . 76
  - 4.2.4.1 Anonymity . . . 77
  - 4.2.4.2 Soundness . . . 79

4.3 Extended Anonymous Credential System . . . **83**
- 4.3.1 Definition . . . 83
- 4.3.2 Security Notions . . . 85
  - 4.3.2.1 Anonymity . . . 85
  - 4.3.2.2 Soundness . . . 87
- 4.3.3 Construction . . . 90
- 4.3.4 Security Proofs . . . 93
  - 4.3.4.1 Anonymity . . . 93
  - 4.3.4.2 Soundness . . . 97

4.4 Attribute-Based Anonymous Credential and Reputation System . . . **100**
- 4.4.1 Definition . . . 100
- 4.4.2 Construction . . . 103

4.5 Further Extensions . . . **107**
- 4.5.1 Revocation . . . 107
  - 4.5.1.1 Revocation of Users . . . 107
  - 4.5.1.2 Revocation of Credentials . . . 108
  - 4.5.1.3 Expiration of Credentials . . . 108
- 4.5.2 Non-Frameability of Users via Judge Algorithm . . . 108
- 4.5.3 Removing the Random Oracle . . . 108

4.5.4 Credentials for k-time Use . . . 109
4.5.5 Disable Rating Own Products . . . 109
4.5.6 Invalidation of Ratings . . . 110
4.5.7 Editability of Ratings . . . 110

In this chapter, we develop the construction of our attribute-based anonymous credential and reputation system. We start with important basics for our constructions. These are special protocols for signing partially committed values used in the credential issuance, for proving knowledge of a signature used when showing a credential and for allowing certain predicates as an access policy. Subsequently, we present three stages of our system. Firstly, we give a construction for a basic anonymous credential system including our own security model and a corresponding security proof. Secondly, we extend the basic anonymous credential system by the feature of traceability of users. This extension includes extending our security model and our construction given before. For this construction, we also give a security proof. Lastly, we further extend the system by including features of an anonymous reputation system in our system. Here, we only give extentions to our construction other than extending the security model and security proof. Instead, we outline the modifications and additionally describe future adjustments to the final stage of our system. These adjustments are for example revocation of users and credentials.

4.1 Preliminaries

According to Camenisch and Lysyanskaya [CL03; CL04], for the construction of an anonymous credential system it suffices to use a hiding and binding commitment scheme (Definitions 3.20 to 3.22) and a signature scheme (Definition 3.12) with corresponding efficient zero-knowledge protocols for (1) receiving a signature on partially committed values (without revealing the values) and (2) proving knowledge of a signature on a message (without revealing either). In this section, we provide general definitions and concrete constructions for these protocols using generalized Pedersen commitments (Construction 3.23) and Pointcheval-Sanders signatures (Construction 3.17), where the latter is used as it allows for efficient protocols [PS16].

4.1.1 Signing Partially Committed Values

A scheme for signing committed values allows users to obtain signatures on values hidden in a commitment from some signing instance. If some values are publicly known, we speak of a scheme for signing partially committed values. First, we define the syntax of such a scheme. In this definition, we assume that the public parameters as well as the public/secret key pair of the signature scheme and the commitment scheme are already generated and given to the parties involved.

Definition 4.1. Let $\ell \in \mathbb{N}$. Let $\mathfrak{C} = \{\mathfrak{C}_k = (\mathsf{Setup}_k, \mathsf{Com}_k, \mathsf{Open}_k)\}_{1 \leq k \leq \ell}$ be a family of commitment schemes with message space $\mathbb{M}^k$ and let $\Pi_\ell = (\mathsf{Setup}_\ell, \mathsf{Gen}_\ell, \mathsf{Sign}_\ell, \mathsf{Vrfy}_\ell)$ be a signature scheme with message space $\mathbb{M}^\ell$. A scheme for signing partially committed values using Π_ℓ and $\mathfrak{C}$ is defined by the following algorithms:

- $\mathsf{BlindInit}(\mathsf{pp}, \mathsf{pk}, S)$: On input public parameters pp, public key pk and a set $S \subseteq \{0, \dots, \ell-1\}$, it fixes the commitment scheme $\mathfrak{C}_{|S|}$ and outputs commitment parameters $\mathsf{pp}_{\mathfrak{C}}$.
- $\left(\mathsf{BlindRcv}(\mathsf{pp}, \mathsf{pk}, C, S, (m_i)_{i=0}^{\ell-1}, d), \mathsf{BlindIssue}(\mathsf{pp}, \mathsf{pk}, C, S, (m_i)_{i \in \bar{S}}, \mathsf{sk})\right)$ is an interactive protocol, where the user runs $\mathsf{BlindRcv}$ and the signer runs $\mathsf{BlindIssue}$. The common inputs are public parameters pp, the signer's public key pk, a commitment C, a set $S \subseteq \{0, \dots, \ell-1\}$

and messages $(m_i)_{i\in\bar{S}}$ with $\bar{S} \coloneqq \{0,\dots,\ell-1\} \setminus S$. The user's private inputs are messages $(m_i)_{i\in S}$ and an opening value d, and the signer's private input is her secret key sk. After the interaction, $\mathsf{BlindRcv}$ outputs a signature σ.

Correctness of a Scheme for Signing Partially Committed Values We say that a scheme for signing partially committed values is *correct* if for all $n, \ell \in \mathbb{N}$, all $\mathsf{pp} \in [\Pi_\ell.\mathsf{Setup}_\ell(1^n)]$, all $(\mathsf{pk}, \mathsf{sk}) \in [\Pi_\ell.\mathsf{Gen}_\ell(\mathsf{pp})]$, all $(m_i)_{i=0}^{\ell-1} \in \mathbb{M}^\ell$ and all $S \subseteq \{0,\dots,\ell-1\}$, we have

$$\Pr\Big[\mathsf{pp}_{\mathfrak{C}} \leftarrow \mathsf{BlindInit}(\mathsf{pp},\mathsf{pk},S), (C,d) \leftarrow \mathsf{Com}(\mathsf{pp}_{\mathfrak{C}}, (m_i)_{i\in S}), \\ \sigma \leftarrow \Big(\mathsf{BlindRcv}(\mathsf{pp},\mathsf{pk},C,S,(m_i)_{i=0}^{\ell-1},d), \mathsf{BlindIssue}(\mathsf{pp},\mathsf{pk},C,S,(m_i)_{i\in\bar{S}},\mathsf{sk})\Big) : \\ \mathsf{Vrfy}_\ell\,(\mathsf{pk}, ((m_0,\dots,m_{\ell-1}),\sigma)) = 1\Big] = 1.$$

Intuitively, a scheme for signing partially committed values is correct if the user has a valid signature of $(m_0,\dots,m_{\ell-1})$ under the signer's public key after interaction with the signer.

We assume the following usage of the scheme: The public parameters of the system are generated and known to every party using the scheme. The signer has generated her signing key pair and published it in the existing public key infrastructure. A user having messages $(m_i)_{i\in S}$ for some set $S \subseteq \{0,\dots,\ell-1\}$, that she possibly does not want to reveal to the signer, then generates commitment parameters using $\mathsf{BlindInit}(\mathsf{pp},\mathsf{pk},S)$ and computes a commitment on the messages $(m_i)_{i\in S}$. The commitment along with set S is sent to the signer. The signer then blindly signs the messages hidden in the commitment and (possibly) adds messages $(m_i)_{i\in\bar{S}}$. This blinded signature is sent to the user, who unblinds it and outputs the resulting signature.

The main goal of such a scheme is obtaining a signature on messages that are only partially known to the signer. Both, the user and the signer, have separate security requirements when executing the protocol. On the one hand, the user wants to protect her secret messages $(m_i)_{i\in S}$, and thus the signing protocol must not reveal any information about it. The signer, on the other hand, wants to make sure that a user cannot output a valid signature under the signer's public key, unless it was derived by executing the protocol in interaction with her. This essentially pours down to the signer wanting to protect her secret key.

Formally, we split these requirements into two definitions. Definition 4.2 formalizes the security for the user, whereas Definition 4.3 formalizes the security for the signer.

Definition 4.2. A scheme for signing partially committed values using signature scheme Π_ℓ and a family of commitment schemes $\mathfrak{C}$ is *secure for the user* if for all $n \in \mathbb{N}$, all $\mathsf{pp} \in [\Pi_\ell.\mathsf{Setup}_\ell(1^n)]$, all $(\mathsf{pk},\mathsf{sk}) \in [\Pi_\ell.\mathsf{Gen}_\ell(\mathsf{pp})]$, all $S \subseteq \{0,\dots,\ell-1\}$, all $\mathsf{pp}_{\mathfrak{C}} \in [\mathsf{BlindInit}(\mathsf{pp},\mathsf{pk},S)]$, all messages $(m_0,\dots,m_{\ell-1}), (m'_0,\dots,m'_{\ell-1}) \in \mathbb{M}^\ell$ with $(m_i)_{i\in\bar{S}} = (m'_i)_{i\in\bar{S}}$, and all (unrestricted) adversaries $\mathcal{A}$, the following distributions are identical:

- $\mathrm{output}_{\mathcal{A}}[(C,d) \leftarrow \mathsf{Com}(\mathsf{pp}_{\mathfrak{C}}, (m_i)_{i\in S}) : \mathcal{A}(C,S,(m_i)_{i\in\bar{S}},\mathsf{sk}) \leftrightarrow \mathsf{BlindRcv}(\mathsf{pp},\mathsf{pk},C,S,(m_i)_{i=1}^{\ell},d)]$
- $\mathrm{output}_{\mathcal{A}}[(C',d') \leftarrow \mathsf{Com}(\mathsf{pp}_{\mathfrak{C}}, (m'_i)_{i\in S}) : \mathcal{A}(C',S,(m_i)_{i\in\bar{S}},\mathsf{sk}) \leftrightarrow \mathsf{BlindRcv}(\mathsf{pp},\mathsf{pk},C',S,(m'_i)_{i=1}^{\ell},d')]$

In this formalization, the adversary takes the role of the signer. If our scheme fulfills the requirement of Definition 4.2 the adversary cannot distinguish whether the user was started with message $(m_0,\dots,m_{\ell-1})$ or $(m'_0,\dots,m'_{\ell-1})$, which gives us that the secret is protected. Note that this definition can only be fulfilled if the commitment scheme $\mathfrak{C}$ is perfectly hiding. Otherwise, the distributions are not necessarily identical.

Let us continue with security for the signer.

Definition 4.3. A scheme for signing partially committed values using signature scheme Π_ℓ and a family of commitment schemes $\mathfrak{C}$ is *secure for the signer*, if there exists a ppt simulator $\mathcal{S}$, such that for all $n \in \mathbb{N}$, all $\mathsf{pp} \in [\Pi_\ell.\mathsf{Setup}_\ell(1^n)]$, all $(\mathsf{pk}, \mathsf{sk}) \in [\Pi_\ell.\mathsf{Gen}_\ell(\mathsf{pp})]$, all messages $(m_i)_{i=0}^{\ell-1} \in \mathbb{M}^\ell$, all sets $S \subseteq \{0, \dots, \ell-1\}$, all $\mathsf{pp}_\mathfrak{C} \in [\mathsf{BlindInit}(\mathsf{pp}, \mathsf{pk}, S)]$, all (C, d) with $\mathsf{Open}_{|S|}(\mathsf{pp}_\mathfrak{C}, C, d) = (m_i)_{i \in S}$ and all (unrestricted) adversaries $\mathcal{A}$ the two distributions

- $\text{output}_\mathcal{A}[\mathcal{A} \leftrightarrow \mathsf{BlindIssue}(\mathsf{pp}, \mathsf{pk}, C, S, (m_i)_{i \in \bar{S}}, \mathsf{sk})]$
- $\text{output}_\mathcal{A}[\mathcal{A} \leftrightarrow \mathcal{S}^{\mathsf{1Sign}(\mathsf{sk}, \cdot)}(\mathsf{pp}, \mathsf{pk}, C, S, (m_i)_{i=0}^{\ell-1}, d)]$

are identical. Here $\mathsf{1Sign}(\mathsf{sk}, \cdot)$ denotes a one-time signing oracle that on input $(m_i)_{i=0}^{\ell-1} \in \mathbb{M}^\ell$ outputs a signature σ such that $\sigma \leftarrow \mathsf{Sign}_\ell(\mathsf{pp}, \mathsf{sk}, (m_i)_{i=0}^{\ell-1})$.

Informally this means, executing BlindIssue reveals as much of the secret key as outputting a single signature on a chosen message does. This is due to the fact, that the simulator needs no information on sk beyond a single oracle query to perfectly imitate the signer. Here, the simulator is provided a valid pair of open value and message(s) for the commitment C, corresponding to the commitment scheme with message space $\mathbb{M}^{|S|}$ instantiated by BlindInit. If such values are known to an interacting party, she could perfectly simulate the interaction with BlindIssue, despite a single signature query. This naturally leads to another notion for the signer, where we demand these values to be extractable from an interacting party after successfully receiving a blinded signature in a significant number of cases. A zero-knowledge argument of knowledge over opening the commitment would be a way to provide such a property. However, this is left to the scope of the credential system, since an argument of knowledge over more statements than just opening the commitment can be used there.

4.1.1.1 Construction of a Scheme for Signing Partially Committed Values

We provide a construction for signing partially committed values. The idea is to first commit to the private messages using a generalized Pedersen environment created via BlindInit from the given public Pointcheval-Sanders key pk. Given this commitment, the signer returns a "blinded" (invalid) signature. Afterwards, using the commitment's open value, the user can compute a valid signature.

Construction 4.4. Let $n, \ell \in \mathbb{N}$, let $\mathfrak{C} = \{\mathfrak{C}_k = (\mathfrak{C}.\mathsf{Setup}_k, \mathsf{Com}_k, \mathsf{Open}_k)\}_{1 \le k \le \ell}$ be a family of generalized Pedersen commitment schemes (Construction 3.23) with message space $\mathbb{M}^k$ and let $\Pi_\ell = (\Pi_\ell.\mathsf{Setup}_\ell, \mathsf{Gen}_\ell, \mathsf{Sign}_\ell, \mathsf{Vrfy}_\ell)$ be the Pointcheval-Sanders signature scheme (Construction 3.17) with message space $\mathbb{M}^\ell$.

- $\mathsf{BlindInit}(\mathsf{pp}, \mathsf{pk}, S)$ on input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, $\mathsf{pk} = (g, Y_0, \dots, Y_{\ell-1}, \tilde{g}, \tilde{X}, \tilde{Y}_0, \dots, \tilde{Y}_{\ell-1})$ and set $S \subseteq \{0, \dots, \ell-1\}$ it outputs $\mathsf{pp}_\mathfrak{C} = (g, (Y_i)_{i \in S}, p, \mathbb{G}_1)$.
- $\Big(\mathsf{BlindRcv}(\mathsf{pp}, \mathsf{pk}, C, S, (m_i)_{i=0}^{\ell-1}, d), \mathsf{BlindIssue}(\mathsf{pp}, \mathsf{pk}, C, S, (m_i)_{i \in \bar{S}}, \mathsf{sk})\Big)$ is an interactive protocol on common inputs $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, $\mathsf{pk} = (g, Y_0, \dots, Y_{\ell-1}, \tilde{g}, \tilde{X}, \tilde{Y}_0, \dots, \tilde{Y}_{\ell-1})$ and set $S \subseteq \{0, \dots, \ell-1\}$. The user's private input are messages $(m_i)_{i \in S}$ and $d = ((m_i)_{i \in S}, r)$, while the issuer gets messages $(m_i)_{i \in \bar{S}}$ and secret key $\mathsf{sk} = (x, y_0, \dots, y_k)$. The issuer chooses $u \leftarrow \mathbb{Z}_p^*$ and computes $(\sigma_1', \sigma_2') \coloneqq (g^u, (g^x \cdot C \cdot \prod_{i \in \bar{S}} Y_i^{m_i})^u)$ and sends (σ_1', σ_2') to the user. Finally, the user unblinds the signature by computing $\sigma = (\sigma_1', \sigma_2' \cdot (\sigma_1')^{-r})$ and outputs it.

Since the protocol for signing partially committed value is the foundation for the credential issuance protocol of our construction given in Section 4.2.3, we now show that Construction 4.4 is correct and secure for both the user (Definition 4.2) and the signer (Definition 4.3). In favor of readability, we split up these three properties into three lemmas.

Lemma 4.5 (Correctness). *Let $n, \ell \in \mathbb{N}$, let $\mathfrak{C} = \{\mathfrak{C}_k = (\mathfrak{C}.\mathsf{Setup}_k, \mathsf{Com}_k, \mathsf{Open}_k)\}_{1 \leq k \leq \ell}$ be a family of generalized Pedersen commitment schemes (Construction 3.23) with message space $\mathbb{M}^k$ and let $\Pi_\ell = (\Pi_\ell.\mathsf{Setup}_\ell, \mathsf{Gen}_\ell, \mathsf{Sign}_\ell, \mathsf{Vrfy}_\ell)$ be the Pointcheval-Sanders signature scheme (Construction 3.17) with message space $\mathbb{M}^\ell$. Construction 4.4 is a correct scheme for signing partially committed values.*

To prove that Construction 4.4 is correct, we need to show that the execution of ($\mathsf{BlindRcv}$, $\mathsf{BlindIssue}$) always yields a valid Pointcheval-Sanders signature under the public key of the issuer.

Proof. Fix $\mathsf{pp} \in [\Pi_\ell.\mathsf{Setup}_\ell(1^n)]$, $(\mathsf{pk}, \mathsf{sk}) \in [\Pi_\ell.\mathsf{Gen}_\ell(\mathsf{pp})]$, $(m_i)_{i=0}^{\ell-1} \in \mathbb{M}^\ell$, $S \subseteq \{0, \ldots, \ell - 1\}$ and $\bar{S} := \{0, \ldots, \ell - 1\} \setminus S$. Further let $\mathsf{pp}_\mathfrak{C} \leftarrow \mathsf{BlindInit}(\mathsf{pp}, \mathsf{pk}, S)$, $(C, d) \leftarrow \mathsf{Com}(\mathsf{pp}_\mathfrak{C}, (m_i)_{i \in S})$ and $\sigma \leftarrow (\mathsf{BlindRcv}(\mathsf{pp}, \mathsf{pk}, C, S, (m_i)_{i=0}^{\ell-1}, d), \mathsf{BlindIssue}(\mathsf{pp}, \mathsf{pk}, C, S, (m_i)_{i \in \bar{S}}, \mathsf{sk}))$.

By definition of $\mathsf{BlindRcv}$, it holds that $\sigma = (\sigma_1', \sigma_2' \cdot (\sigma_1')^{-r})$ for opening value $d = ((m_i)_{i \in S}, r)$ and (σ_1', σ_2') obtained from $\mathsf{BlindIssue}$. The signer obtains $(\sigma_1', \sigma_2') := (g^u, (g^x \cdot C \cdot \prod_{i \in \bar{S}} Y_i^{m_i})^u)$ for $u \leftarrow \mathbb{Z}_p^*$. By the correctness of the Pedersen commitment (Lemma 3.25), we have $C = g^r \prod_{i \in S} Y_i^{m_i}$ for commitment parameters $\mathsf{pp}_\mathfrak{C}$ obtained by running $\mathsf{BlindInit}$. Therefore, we have

$$\begin{aligned}
\sigma &= (\sigma_1', \sigma_2' \cdot (\sigma_1')^{-r}) = \left(g^u, \left(g^x \cdot C \cdot \prod_{i \in \bar{S}} Y_i^{m_i}\right)^u \cdot (g^u)^{-r}\right) \\
&= \left(g^u, \left(g^x \cdot g^r \prod_{i \in S} Y_i^{m_i} \cdot \prod_{i \in \bar{S}} Y_i^{m_i}\right)^u \cdot (g^u)^{-r}\right) = \left(g^u, \left(g^x \cdot g^r \prod_{i \in S} Y_i^{m_i} \cdot \prod_{i \in \bar{S}} Y_i^{m_i} \cdot g^{-r}\right)^u\right) \\
&= \left(g^u, \left(g^x \prod_{i=0}^{\ell-1} Y_i^{m_i}\right)^u\right) = \left(g^u, \left(g^x \prod_{i=0}^{\ell-1} g^{m_i y_i}\right)^u\right) = \left(g^u, (g^u)^x \prod_{i=0}^{\ell-1} (g^u)^{m_i y_i}\right) \\
&= \left(g^u, (g^u)^{x + \sum_{i=0}^{\ell-1} m_i y_i}\right).
\end{aligned}$$

By definition and correctness of the Pointcheval-Sanders signature scheme (Lemma 3.18), σ is a valid signature on the messages $(m_i)_{i=0}^{\ell-1}$ under a public key $\mathsf{pk} = (g, Y_0, \ldots, Y_{\ell-1}, \tilde{g}, \tilde{X}, \tilde{Y}_0, \ldots, \tilde{Y}_{\ell-1})$ and the corresponding secret key $sk = (x, y_0, \ldots, y_{\ell-1})$. □

As described above we consider security for the user (Definition 4.2) and signer (Definition 4.3) separately. We start with the security for the user followed by the security of the signer.

Lemma 4.6 (Security for the User). *Let $n, \ell \in \mathbb{N}$, let $\mathfrak{C} = \{\mathfrak{C}_k = (\mathfrak{C}.\mathsf{Setup}_k, \mathsf{Com}_k, \mathsf{Open}_k)\}_{1 \leq k \leq \ell}$ be a family of generalized Pedersen commitment schemes (Construction 3.23) with message space $\mathbb{M}^k$ and let $\Pi_\ell = (\Pi_\ell.\mathsf{Setup}_\ell, \mathsf{Gen}_\ell, \mathsf{Sign}_\ell, \mathsf{Vrfy}_\ell)$ be the Pointcheval-Sanders signature scheme (Construction 3.17) with message space $\mathbb{M}^\ell$. Then, Construction 4.4 is a secure scheme for signing partially committed values for the user according to Definition 4.2.*

Recall the intuition we have given about the user's security requirement. It said that an adversary should not be able to distinguish whether the user has secret message $(m_i)_{i \in S}$, for some set S, or secret message $(m_i')_{i \in S}$. This is the essence of what we need to show. More formally, the distribution on the outputs of an adversary interacting with a user having secret $(m_i)_{i \in S}$ and with a user having secret $(m_i')_{i \in S}$ need to be identical.

Proof. Fix any public parameters $\mathsf{pp} \in [\Pi_\ell.\mathsf{Setup}_\ell(1^n)]$, key pair $(\mathsf{pk}, \mathsf{sk}) \in [\mathsf{Gen}_\ell(\mathsf{pp})]$, a set of indices of private message $S \subseteq \{0, \ldots, \ell - 1\}$, messages $(m_i)_{i=0}^{\ell-1}$, $\mathsf{pp}_\mathfrak{C} := \mathsf{BlindInit}(\mathsf{pp}, \mathsf{pk}, S)$ and an adversary $\mathcal{A}$ interacting with $\mathsf{BlindRcv}$. Let $(C, d) \leftarrow \mathsf{Com}(\mathsf{pp}_\mathfrak{C}, (m_i)_{i \in S})$.

First of all, we show that $\mathsf{pp}_\mathfrak{C}$ output by $\mathsf{BlindInit}$ are valid parameters for the Pedersen commitment. These consist of a description of a cyclic group of prime order p, a generator of this group and a group element for every message to commit to. The signature public parameters

pp only contain a description of a bilinear group. By definition of a bilinear group, the group $\mathbb{G}_1$ is cyclic and of prime order p. Moreover, by definition of the signature scheme's public key, g is a generator of $\mathbb{G}_1$ and $(Y_i)_{i\in S}$ are elements of $\mathbb{G}_1$. Taking this together, we conclude that $\mathsf{BlindInit}$ generates valid parameters for the Pedersen commitment.

Now, consider the two distributions given in Definition 4.2. Every input of the adversary $\mathcal{A}$ and the user are identical except for the commitment given to them. Therefore, $\mathcal{A}$ can only attack the commitment given to her. We have seen above that $\mathsf{BlindInit}$ generates valid commitment parameters for the Pedersen commitment. Since the Pedersen commitment scheme we use fulfills the property of perfect hiding (Lemma 3.26), it holds that the commitment is distributed independently of the committed messages and thus reveals no information about them. Hence, the two distributions in Definition 4.2 are identical. □

Having shown that Construction 4.4 is secure for the user, it remains to show that it also is secure for the signer. Consider the next lemma.

Lemma 4.7 (Security for the Signer). *Let $n, \ell \in \mathbb{N}$, let $\mathfrak{C} = \{\mathfrak{C}_k = (\mathfrak{C}.\mathsf{Setup}_k, \mathsf{Com}_k, \mathsf{Open}_k)\}_{1\le k\le \ell}$ be a family of generalized Pedersen commitment schemes (Construction 3.23) with message space $\mathbb{M}^k$ and let $\Pi_\ell = (\Pi_\ell.\mathsf{Setup}_\ell, \mathsf{Gen}_\ell, \mathsf{Sign}_\ell, \mathsf{Vrfy}_\ell)$ be the Pointcheval-Sanders multi-message signature scheme (Construction 3.17) with message space $\mathbb{M}^\ell$. Then, Construction 4.4 is a secure scheme for signing partially committed values for the signer according to Definition 4.3.*

Remember that security for the signer demands existence of a ppt simulator outputting messages distributed exactly as messages from $\mathsf{BlindIssue}$. Besides valid open value and messages corresponding to the commitment for the scheme instantiated by $\mathsf{BlindInit}$ using pk, the simulator obtains access to a one-time signing oracle on sk. The only message sent in the construction is by $\mathsf{BlindIssue}$, which needs to be simulated. It suffices to show the outputted message by simulator and $\mathsf{BlindIssue}$ on any allowed input is distributed identically.

Proof. Fix *any* public parameters $\mathsf{pp} \in [\Pi_\ell.\mathsf{Setup}_\ell(1^n)]$, key pair $(\mathsf{sk}, \mathsf{pk}) \in [\mathsf{Gen}_\ell(\mathsf{pp})]$, set of indices of private messages $S \subseteq \{0, \dots, \ell-1\}$, messages $(m_i)_{i=0}^{\ell-1} \in \mathbb{M}^\ell$, $\mathsf{pp}_\mathfrak{C} := \mathsf{BlindInit}(\mathsf{pp}, \mathsf{pk}, S)$, tuple (C, d) with $\mathsf{Open}_{|S|}(\mathsf{pp}_\mathfrak{C}, C, d) = (m_i)_{i\in S}$ and adversary $\mathcal{A}$.

On inputs set S and Pointcheval-Sanders public key and parameters, $\mathsf{BlindInit}$ outputs an environment for generalized Pedersen commitments on $|S|$ messages with group $\mathbb{G}_1$ of order p and the elements $(g, (Y_i)_{i\in S})$. By $\mathsf{Open}_{|S|}(\mathsf{pp}_\mathfrak{C}, C, d) = (m_i)_{i\in S}$, we know that d is of form $(r, (m_i)_{i\in S})$ and $C = g^r \prod_{i\in S} Y_i^{m_i}$ holds.

Now the simulator queries $(\sigma_1, \sigma_2) \leftarrow \mathsf{1Sign}(\mathsf{sk}, (m_i)_{i=0}^{\ell-1})$. The received signature (σ_1, σ_2) has form $(h, h^{x+\sum_{i=0}^{\ell-1} y_i m_i})$ for a randomly chosen $h \leftarrow \mathbb{G}_1 \setminus \{1\}$. The simulator outputs $(\sigma_1, \sigma_2 \cdot \sigma_1^r)$. Note, that for any generators h and g there exists a unique $u \in \mathbb{Z}_p^*$ with $h = g^u$. By substituting h with g^u we get

$$\begin{aligned}
(\sigma_1, \sigma_2 \cdot \sigma_1^r) &= (h, h^{x+\sum_{i=0}^{\ell-1} y_i m_i} \cdot h^r) = (h, h^{x+\sum_{i\in \bar{S}} y_i m_i} \cdot h^{\sum_{i\in S} y_i m_i} \cdot h^r) \\
&= \left(g^u, \left(g^{x+\sum_{i\in \bar{S}} y_i m_i} \cdot g^{\sum_{i\in S} y_i m_i} \cdot g^r\right)^u\right) = \left(g^u, \left(g^{x+\sum_{i\in \bar{S}} y_i m_i} \cdot C\right)^u\right) \\
&= (g^u, (g^x \cdot C \cdot \prod_{i\in \bar{S}} Y_i^{m_i})^u),
\end{aligned}$$

where the latter has exactly the form of $\mathsf{BlindIssue}$'s output. Since its only random choice is $u \leftarrow \mathbb{Z}_p^*$, and g^u is distributed the same as choosing $h \leftarrow \mathbb{G}_1 \setminus \{1\}$ the outputs of the simulator and $\mathsf{BlindIssue}$ are distributed the same, independent of any (interacting) adversary. □

4.1.2 Proving Knowledge of a Signature

Now we need a protocol for proving knowledge of a valid signature on a certain message with respect to public key pk. This is necessary in the ACS (see Section 4.2), since verifiers need to be sure a user really possesses a valid credential from some issuer, i. e. a signature over her user secret and certain attributes. Assuming unforgeability, such a proof assures a signature was issued by the party corresponding to pk. A simple proof would be to send signature σ and message m to the verifier, who could check validity. However, in the ACS the signed message contains private data, and a user would become linkable by always using the same signature. Hence, the user should only reveal she knows that *some* valid signature σ on *some* message m under pk. Formally, for given $\mathsf{pp} \in [\mathsf{Setup}(\cdot)]$ of a signature scheme (Definition 3.12), we demand a zero-knowledge proof of knowledge (Definitions 3.39 and 3.40) for the relation (Definition 3.36)

$$R_{\mathsf{pp}} = \{((\sigma, m), \mathsf{pk}) \mid (\mathsf{sk}, \mathsf{pk}) \in [\mathsf{Gen}(\mathsf{pp})], \mathsf{Vrfy}(\mathsf{pk}, \sigma, m) = 1\}.$$

In short, we write $\mathsf{PK}\{(\sigma, m) : \mathsf{Vrfy}(\mathsf{pk}, \sigma, m) = 1\}$.

4.1.2.1 Construction of a Scheme for Proving Knowledge of a Signature

Now we provide a protocol for proving knowledge of a valid Pointcheval-Sanders multi-message signature (Construction 3.17) on $\ell + 1$ messages. The idea is to randomize the existing signature (see remark on p. 16), such that $\sigma' = (\sigma'_1, \sigma'_2)$ is a valid signature on $(m_0, \ldots, m_\ell, r)$ for $r \leftarrow \mathbb{Z}_p$, where the public key is extended with $(Y_{\ell+1} = g, \tilde{Y}_{\ell+1} = \tilde{g})$. This statement can then simply be proven via the generalized Schnorr approach (Construction 3.44). Of course, the randomization needs to ensure σ' itself does not reveal any knowledge on the original message or signature. This protocol was already given by Pointcheval and Sanders [PS16], with the missing but necessary $\sigma'_1 \neq 1$ check.

Construction 4.8. Let $\Pi_{\ell+1}$ be the Pointcheval-Sanders multi-message signature scheme (Construction 3.17) for $(\ell{+}1)$ messages on given public parameters $\mathsf{pp} \in [\mathsf{Setup}(\cdot)]$. Let $\mathsf{pk} = (g, Y_0, \ldots Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_0, \ldots, \tilde{Y}_\ell) \in [\mathsf{Gen}_{\ell+1}(\mathsf{pp})]$ be a valid public key for the scheme. Let $\sigma = (\sigma_1, \sigma_2)$ and messages $(m_0, \ldots, m_\ell)$ be the user's private input, such that $\mathsf{Vrfy}_{\ell+1}(\mathsf{pk}, \sigma, (m_i)_{i=0}^{\ell}) = 1$ holds. First, the user chooses $(u, r) \leftarrow \mathbb{Z}_p^* \times \mathbb{Z}_p$, sets $\sigma' := (\sigma'_1, \sigma'_2) := (\sigma_1^u, (\sigma_2 \cdot \sigma_1^r)^u)$ and sends σ' to the verifier. Next, the user initiates and runs a proof of knowledge of the form

$$\mathsf{PK}\left\{(m_0, \ldots, m_\ell, r) : e(\sigma'_1, \tilde{g})^r \prod_{i=0}^{\ell} e(\sigma'_1, \tilde{Y}_i)^{m_i} = \frac{e(\sigma'_2, \tilde{g})}{e(\sigma'_1, \tilde{X})}\right\}$$

with the verifier. This proof can be instantiated as Σ-protocol (Definition 3.43), for example via the generalized Schnorr protocol (Construction 3.44). The verifier accepts, if and only if $\sigma'_1 \neq 1$ *and* it accepts within the proof.

Note, that we can simply send σ' alongside the first message of the proof of knowledge. With this consideration, we can prove Construction 4.8 is a Σ-protocol (Definition 3.43) for the desired relation, which implies it is an interactive zero-knowledge proof of knowledge.

Theorem 4.9. *Let $\ell \in \mathbb{N}_0$. Construction 4.8 is a Σ-protocol (Definition 3.43) for the relation $R_{\mathsf{pp}} = \{((\sigma, m), \mathsf{pk}) \mid (\mathsf{sk}, \mathsf{pk}) \in [\mathsf{Gen}_{\ell+1}(\mathsf{pp})], \mathsf{Vrfy}_{\ell+1}(\mathsf{pk}, \sigma, m) = 1\}$ corresponding to the Pointcheval-Sanders multi-message signature scheme $\Pi_{\ell+1} = (\mathsf{Setup}, \mathsf{Gen}_{\ell+1}, \mathsf{Sign}_{\ell+1}, \mathsf{Vrfy}_{\ell+1})$ (Construction 3.17) and public parameters $\mathsf{pp} \in [\mathsf{Setup}(\cdot)]$.*

Proof. For *correctness* note, that any valid PS-signature $\sigma = (\sigma_1, \sigma_2)$ fulfills $\sigma_1 \neq 1$. Thus $\sigma_1^u \neq 1$ holds for $u \in \mathbb{Z}_p^*$. By this fact and correctness of the generalized Schnorr protocol (Lemma 3.45), the verifier accepts after an honest execution with $((\sigma, (m_0, \ldots, m_\ell)), \mathsf{pk}) \in R_{\mathsf{pp}}$.

For *special soundness* we extract a witness, i. e. some valid signature on some message $(m_0, \dots, m_\ell)$, from two accepting transcripts. Using the generalized Schnorr protocol's extractor (cf. Lemma 3.45), we can get $(m_0, \dots, m_\ell, r)$, such that

$$\begin{aligned} e(\sigma_1', \tilde{g})^r \prod_{i=0}^{\ell} e(\sigma_1', \tilde{Y}_i)^{m_i} &= \frac{e(\sigma_2', \tilde{g})}{e(\sigma_1', \tilde{X})} \\ \iff e(\sigma_1', \tilde{g})^r e(\sigma_1', \tilde{X}) \prod_{i=0}^{\ell} e(\sigma_1', \tilde{Y}_i)^{m_i} &= e(\sigma_2', \tilde{g}) \\ \iff \qquad e(\sigma_1', \tilde{g})^r e(\sigma_1', \tilde{X} \prod_{i=0}^{\ell} \tilde{Y}_i^{m_i}) &= e(\sigma_2', \tilde{g}) \\ \iff \qquad e(\sigma_1', \tilde{X} \prod_{i=0}^{\ell} \tilde{Y}_i^{m_i}) &= e(\sigma_2', \tilde{g}) e(\sigma_1', \tilde{g})^{-r} \\ \iff \qquad e(\sigma_1', \tilde{X} \prod_{i=0}^{\ell} \tilde{Y}_i^{m_i}) &= e(\sigma_2' \cdot \sigma_1'^{-r}, \tilde{g}). \end{aligned}$$

The last equality simply states $\mathsf{Vrfy}(\mathsf{pk}, (\sigma_1', \sigma_2' \cdot \sigma_1'^{-r}), (m_0, \dots, m_\ell)) = 1$, because for an accepting transcript $\sigma_1' \neq 1$ holds. The extractor outputs the valid witness $((\sigma_1', \sigma_2' \cdot \sigma_1'^{-r}), (m_0, \dots, m_\ell))$.

It remains to show it is a *special honest verifier zero-knowledge* protocol by constructing a simulator S. Given any challenge c, the simulator S picks $\sigma_1' \leftarrow \mathbb{G}_1 \setminus \{1\}$ and $\sigma_2 \leftarrow \mathbb{G}_1$. In an honest protocol execution $\sigma_1' := \sigma_1^u$ is distributed uniformly in $\mathbb{G}_1 \setminus \{1\}$ by $\sigma_1 \neq 1$ and $u \leftarrow \mathbb{Z}_p^*$. Independent of that, $\sigma_2' := \sigma_2^u \cdot (\sigma_1^u)^r$ is distributed uniformly in $\mathbb{G}_1$ by the choice of $r \leftarrow \mathbb{Z}_p$. Since (σ_1', σ_2') is distributed as in the real protocol, S now can run the simulator of the generalized Schnorr protocol (Lemma 3.45) with the correctly distributed $\mathbb{G}_T$ elements (e. g. $e(\sigma_1', \tilde{g}), e(\sigma_2', \tilde{g})$) and challenge c. By that it obtains the remaining transcript elements with the same distribution as in a real protocol execution. □

4.1.3 Predicates

As mentioned before, our system should provide the possibility of proving statements over issued attribute values. In particular users can prove in a zero-knowledge-fashion (Definition 3.39), that their credentials fulfill a published predicate. These predicates are evaluated by using threshold operators where $(X_1, X_2, \dots, X_n, d)$ expresses d of the n boolean variables need to be fulfilled. Formulas with these operators can replace boolean formulas by exchanging $\wedge$ with $(2,2)$-thresholds and $\vee$ with $(1,2)$-ones. For example, $((X_{11}, X_{12}, 2), (X_{21}, X_{22}, 2), 1)$ is equivalent to $(X_{11} \wedge X_{12}) \vee (X_{21} \wedge X_{22})$. Such predicates define monotone access structures (cf. Section 3.6).

We want to give zero-knowledge arguments/proofs for the following cases:

- *Equality*, respectively *inequality*, of an attribute a_i to
  1. an (unknown) discrete logarithm of public value y, expressed by $a_i = \log_g(y)$, respectively $a_i \neq \log_g(y)$.
  2. a public value s, expressed by $a_i = s$, respectively $a_i \neq s$.
  3. another attribute a_j expressed by $a_i = a_j$, respectively $a_i \neq a_j$.
- *Membership* of an attribute a_i in a public
  1. *set* $\Omega = \{\omega_1, \dots, \omega_m\}$ denoted by $a_i \in \Omega$.
  2. *range* of integers $[A, B] := \{A, A+1, \dots, B\}$ expressed by $a_i \in [A, B]$.

Set and range membership proofs could be achieved by OR-concatenations of equality-proofs, which becomes inefficient for large sets or ranges. As an example, for two attributes $a_1 =$ 'USA' and $a_2 = 20$, a predicate could be described by

$$\phi(a_1, a_2) = 1 :\iff (a_1 = \text{'Germany'} \wedge a_2 \in [18, 150]) \vee (a_1 = \text{'USA'} \wedge a_2 \in [21, 150]).$$

This would be evaluated to 0, as the expressions are evaluated to $(0 \wedge 1) \vee (1 \wedge 0)$.

To prove such formulas in zero-knowledge, we use the technique of proofs of partial knowledge from Construction 3.59. In order to apply the technique, we define zero-knowledge arguments corresponding to each of the previously stated expressions. In the following, we assume publicly known Pedersen commitments (Construction 3.23) on the involved attributes, where only the prover knows how to open them. A proof like "the credential consists of attributes $(a_1, \dots, a_\ell)$ *and* C_i is a Pedersen commitment to a_i for all $i = 1, \dots, \ell$ " will ensure, that the commitments are bound to the issued attributes. We assume that all involved groups are cyclic, multiplicative and of prime order p.

4.1.3.1 Equality Proofs for Attributes

First of all, we provide protocols for proving equality of attributes in different scenarios. We focus on equality of an attribute to a discrete logarithm, to a public value and another attribute. Providing the open value(s) for the involved commitment(s) allows a verifier to check equality herself. In case of selective disclosure, where an attribute value should be revealed, this can be meaningful. But, if several equality proofs are OR-concatenated, we want to reveal that one but not which of the equalities holds, e. g. an attribute either equals "Germany" or "USA". Therefore, we employ zero-knowledge arguments of knowledge in the following.

Equality to (Unknown) Discrete Logarithm

We want to provide a Σ-protocol (Definition 3.43) to prove α, committed to in $C = g_1^r h^\alpha$, is equal to the discrete logarithm of a (publicly known) value y to base g_2. More formally, we provide a proof of form $\mathsf{PK}\Big\{(\alpha, r) : C = \mathsf{Com}(\alpha; r) \wedge \alpha = \log_{g_2}(y)\Big\}$. This is valuable, for example, if public keys of form $y = g_2^x$ are known and a user wants to disclose her identity without leaking the secret exponent unknown to the verifier. Based on such a protocol, she could show her public key is contained in a set via OR-concatenations. This proof is simply achieved using the generalized Schnorr protocol.

Construction 4.10. Let $g_1, h, C \in \mathbb{G}_1,\ g_2, y \in \mathbb{G}_2$ be commonly known, where g_1, g_2 are generators. The prover's private input is r and α, such that $C = g_1^r h^\alpha$. Prover and verifier run the generalized Schnorr proof (Construction 3.44)

$$\mathsf{PK}\{(r, \alpha) : C = g_1^r h^\alpha \wedge y = g_2^\alpha\}.$$

The verifier accepts, if and only if she accepts within this proof.

Equality to Public Value

Next, we want a Σ-protocol, that proves equality of an attribute to a public value s. More formally, this is a proof of form $\mathsf{PK}\{(\alpha, r) : C = \mathsf{Com}(\alpha; r) \wedge \alpha = s\}$. Doing this in a Σ-protocol only makes sense if several such proofs are OR-combined. This comes by the fact, that revealing the commitment's open value would have the same effect, but is more efficient. For example, an attribute value could be proven to be Boolean, i. e. either equal to 1 or 0.

Construction 4.11. Let $g, h, C \in \mathbb{G}$ and $s \in \mathbb{Z}_p$ be commonly known, where g is a generator. The prover's private input is r, such that $C = g^r h^\alpha$. Prover and verifier run the generalized Schnorr proof (Construction 3.44) of form

$$\mathsf{PK}\{(r) : Ch^{-s} = g^r\}.$$

The verifier accepts, if and only if she accepts within this proof.

The extractor of this Schnorr protocol can extract a representation $C = g^r h^s$, i.e. an open value for C to value s.

Equality of Two Attributes

Finally, we want a protocol, showing two committed values are equal. More formally, this is a proof of form $\mathsf{PK}\{(\alpha_1, \alpha_2, r_1, r_2) : C_1 = \mathsf{Com}(\alpha_1; r_1) \wedge C_2 = \mathsf{Com}(\alpha_2; r_2) \wedge \alpha_1 = \alpha_2\}$. It is hard to figure out meaningful examples for such a proof for attribute values within a single credential. But it could be reasonable to prove, whether two attributes in different credentials are equal, for example, contain the same age. Furthermore, as pseudonyms in our ACS are commitments to the user secret, such a protocol allows proving two pseudonyms belong to the same user.

A simple Schnorr proof $\mathsf{PK}\{(\alpha, r_1, r_2) : C_1 = g^{r_1} h^\alpha \wedge C_2 = g^{r_2} h^\alpha\}$ could achieve such a protocol. However, in the ACS we generically perform proofs of form $\mathsf{PK}\{(\alpha_i, r_i) : C_i = g^{r_i} h^{\alpha_i}\}$. Therefore it is slightly more efficient to use Construction 4.11 alongside these proofs for opening C_1 and C_2. To instantiate Construction 4.11, prover and verifier compute a "new" commitment $C := C_1 \cdot C_2^{-1}$ and use $s = 0$. The prover's private input then is $r_1 - r_2$. The reasoning is that the prover can open the new commitment to 0, only if both committed values are equal, i.e. $C = g^{r_1 - r_2} h^{\alpha_1 - \alpha_2} = g^{r_1 - r_2}$. Else, she would again be able to compute $\log_g(h)$.

4.1.3.2 Inequality Proofs for Attributes

We proceed with proofs showing inequality of an attribute to a discrete logarithm, a public value and another attribute. Such proofs constitute, that the committed attribute value could be any (committable) value but the one inequality has been shown to. Indeed this is equivalent to an OR-concatenation of equality proofs, which would be less efficient than the following protocols or even be infeasible.

Inequality to (Unknown) Discrete Logarithm

First we want to provide a protocol assuring a value α, committed to in C, is not equal to a discrete logarithm s of $y := g^s$ for some generator g. This is an argument of form $\mathsf{PK}\Big\{(\alpha, r) : C = \mathsf{Com}(\alpha; r) \wedge \alpha \neq \log_g(y)\Big\}$. Such a proof becomes necessary, if for example a user with secret key α has to deny to be the party with public key of form $y = g^s$. In a reputation system, for example, a proof of the above form could disable vendors to rate their own products.

For the construction we adapt the protocol from Camenisch and Shoup [CS03, Section 6]. There, the common input is a value g^α instead of the hiding commitment C on α. This input contradicts the anonymity demands of an ACS, as it allows for linking several protocol runs, if the value α was uniquely chosen within the system.

Construction 4.12. Let $g_1, h, C \in \mathbb{G}_1$, $g_2, y \in \mathbb{G}_2$ be commonly known, such that g_1, g_2 are generators. The prover's private input is r and α, such that $C = g_1^r h^\alpha$.

The prover chooses $z \leftarrow \mathbb{Z}_p^*$ and computes $W := (g_2^\alpha \cdot y^{-1})^z$. Only if $W \neq 1$ (i.e. the exponents are not equal), she continues and sets $x_1 := rz, x_2 := \alpha z, x_3 := z$, sends W to the verifier and runs the generalized Schnorr proof (Construction 3.44)

$$\mathsf{PK}\Big\{(x_1, x_2, x_3) : 1 = g_1^{x_1} h^{x_2} (C^{-1})^{x_3} \wedge W = (y^{-1})^{x_3} g_2^{x_2}\Big\}.$$

The verifier accepts if and only if $W \neq 1$ and she accepts in the proof.

Construction 4.12 relies on the fact, that for $s = \log_{g_2}(y)$, in case of inequality, $\alpha - s \neq 0$ holds. Therefore the value $\alpha - s$ can be distributed uniformly at random in $\mathbb{Z}_p^*$ by multiplication with an element $z \leftarrow \mathbb{Z}_p^*$. Hence $(g_2^{\alpha z} \cdot y^{-z})$ is distributed uniformly in $\mathbb{G}_2 \setminus \{1\}$, and can be sent as an auxiliary, hiding commitment. Then, essentially, knowledge of z and αz is proven as well as (implicitly) knowledge of opening the original commitment on α.

Theorem 4.13. *Let $\mathfrak{C} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be the Pedersen commitment scheme for a single value (Construction 3.23) and $(g_1, h, p, \mathbb{G}_1) \leftarrow \mathsf{Setup}(\cdot)$. If the discrete logarithm problem (Definition 3.4) is hard relative to the group generation algorithm G (used in Setup), Construction 4.12 is a special honest-verifier zero-knowledge argument of knowledge for the relation $R \coloneqq \{((C, g_1, g_2, h, y), (r, \alpha) \mid C = g_1^r h^\alpha \wedge \alpha \neq \log_{g_2}(y) \wedge g_1 \neq 1 \neq g_2\}$, with negligible knowledge error.*

Proof. Completeness follows by inspection of the protocol and the fact that $(\alpha - s) \cdot z \neq 0$ for $(\alpha - s) \neq 0 \neq z$, and therefore $W \neq 1$.

To handle the computational *soundness* of the protocol, we provide a *relaxed* special soundness extractor. This extractor obtains two accepting transcripts with the same announcement but different challenges and is allowed to fail with a negligible probability. The input transcripts are distributed as obtained from the standard rewinding experiment. For such transcripts we can use the special soundness extractor of the Schnorr proof to obtain (x_1, x_2, x_3) fulfilling

$$1 = g_1^{x_1} h^{x_2} (C^{-1})^{x_3} \text{ and } W = (y^{-1})^{x_3} g_2^{x_2}.$$

We make a case distinction over x_3 being zero or non-zero:

Case $\mathbf{x_3 = 0}$: Due to $1 \neq W = g^0 \cdot g_2^{x_2} = g_2^{x_2}$ and g_2 being a generator this implies $x_2 \neq 0$. But then we can compute $\log_{g_1}(h) = \frac{x_1}{-x_2}$ due to $1 = g_1^{x_1} h^{x_2} \cdot 1 \iff h^{-x_2} = g_1^{x_1}$. In this case, we break the binding property of the Pedersen commitment scheme (and therefore the discrete logarithm problem). For a ppt prover, this happens with negligible probability (as the Setup algorithm was run honestly by assumption) and the extractor outputs a failure symbol.

Case $\mathbf{x_3 \neq 0}$: In this case we always obtain a valid witness. We set $r \coloneqq \frac{x_1}{x_3}, \alpha \coloneqq \frac{x_2}{x_3}$ and see

$$1 = g_1^{x_1} h^{x_2} (C^{-}1)^{x_3} \iff C^{x_3} = g_1^{x_1} h^{x_2} \iff C = g_1^r h^\alpha.$$

We only need to show $g_2^\alpha \neq y = g_2^s$ or equivalently $(\alpha - s) \neq 0$, which holds since

$$1 \neq W = (y^{-1})^{x_3} g_2^{x_2} = g_2^{-s x_3} g_2^{x_2} = g_2^{-s x_3} g_2^{\alpha x_3} = g_2^{(\alpha - s) x_3} \overset{\substack{x_3 \neq 0 \\ g_2 \neq 1}}{\iff} (\alpha - s) \neq 0.$$

Finally, the *special honest-verifier simulator* chooses W uniformly from $\mathbb{G}_2 \setminus \{1\}$, which distributes the auxiliary commitment as in the protocol (since $g_2^\alpha \cdot y^{-1} \neq 1$ and the exponent z is chosen from $\mathbb{Z}_p^*$). The remaining part of the protocol is simulated with the simulator corresponding to the particular Schnorr instantiation (using W). □

Inequality to Public Value

The previous protocol can be employed to show a value α, committed to in C, is not equal to a public value $s \in \mathbb{Z}_p$ (instead of s hidden in y), i. e. $\mathsf{PK}\{(\alpha, r) : C = \mathsf{Com}(\alpha; r) \wedge s \neq \alpha\}$. Such a feature can be useful, if parties with certain attributes are excluded from using a service. For example citizens from certain states might be forbidden to access a service due to licensing.

Such a proof can be instantiated based on the inequality proof from Construction 4.12. To instantiate the construction both parties compute $y \coloneqq g^s$ and then execute the protocol.

Inequality of Two Attributes

We provide a protocol on two publicly known Pedersen commitments $C_i = g^{r_i} h^{\alpha_i}$ for $i \in \{1, 2\}$, where $\alpha_1 \neq \alpha_2$ is proven additionally to the knowledge of exponents. That is, an argument of knowledge of the form $\mathsf{PK}\{(\alpha_1, \alpha_2, r_1, r_2) : C_1 = \mathsf{Com}(\alpha_1; r_1) \wedge C_2 = \mathsf{Com}(\alpha_2; r_2) \wedge \alpha_1 \neq \alpha_2\}$.

Again such a proof can be achieved by using Construction 4.12, AND-composed with proofs for opening C_1 and C_2. To instantiate it, $C := C_1 \cdot C_2^{-1}$ is computed at verifier and prover site. The prover can open this new commitment with the private values $r_1 - r_2$ and $\alpha_1 - \alpha_2$. If $\alpha_1 \neq \alpha_2$, the committed value is not equal to 0. Therefore we need to prove inequality to 0, hence run Construction 4.12 with $y = g_2^0 = 1$ and C.

4.1.3.3 Membership Proofs for Attributes

We focus on proofs showing an attribute value lies within a given set, or, as a special case, in a given range. This could be achieved by OR-concatenations of equality proofs, which is inefficient or even infeasible for too large sets and ranges. Here we will rely on the work of Camenisch, Chaabouni, and shelat [CCs08].

Set Membership of Attribute

First we focus on the problem where we want to prove an attribute α, committed to in C, is contained in a public set $\Omega := \{\omega_1, \ldots, \omega_m\}$. More formally, $\mathsf{PK}\{(\alpha, r) : C = \mathsf{Com}(\alpha; r) \wedge \alpha \in \Omega\}$ denotes such a *set membership proof*. An OR-concatenation of m equality proofs would result in a proof of size $\mathcal{O}(m)$. One could argue, the communication complexity always needs to be $\Omega(m)$, as the set has to be announced somehow. However, publishing the set can be performed once, and afterwards more efficient, i. e. constant-time, set membership proofs could amortize this communication cost.

Camenisch, Chaabouni, and shelat [CCs08] mainly focus on the idea to provide signatures on the elements in Ω with a signature scheme allowing for zero-knowledge proofs of a signature. They name cryptographic accumulators (cf. Section 3.12) as an alternative, on which we concentrate. In the following construction we use the accumulator of Nguyen [Ngu05] following Construction 3.67 in Section 3.12.3.

Construction 4.14. Let Π_{acc} be the Nguyen accumulator from Construction 3.67, $\mathsf{pp} = (p, e, g_1, \tilde{g}_1, \tilde{g}_1^s, t) \in [\mathsf{Setup}(\cdot, 1^q)]$ with $t = (g_1^s, g_1^{s^2}, \ldots, g_1^{s^q})$, $\Omega = \{\omega_1, \ldots, \omega_m\} \in \mathbb{Z}_p^m$ with $m \leq q$, $C = g_2^r h^\alpha$, where $g_2, h, C \in \mathbb{G}'$, pp, Ω are common, and $\alpha, r \in \mathbb{Z}_p$ the prover's private input, such that $\alpha \in \Omega$.

In a first step, the prover and verifier (locally) compute $V := \mathsf{AccCreate}(\mathsf{pp}, \Omega)$. For $\alpha = \omega_i$ the prover computes $W_\alpha := \mathsf{WitCreate}(\mathsf{pp}, \Omega, i)$. Next, she chooses $z \leftarrow \mathbb{Z}_p^*$, computes $W := W_\alpha^z$, sends W to the verifier and runs the Schnorr proof (Construction 3.44)

$$\mathsf{PK}\Big\{(r, \alpha, z) : C = g_2^r h^\alpha \wedge e(W, \tilde{g}_1)^\alpha \cdot e(V, \tilde{g}_1^{-1})^z = e(W, \tilde{g}_1^s)^{-1}\Big\}.$$

The verifier accepts, if and only if $W \neq 1$ and she accepts in this proof of knowledge.

Here prover and verifier can (locally) accumulate all elements from Ω to the same value V. The prover then computes a witness for her element α lying in V, which is only feasible if $\alpha \in \Omega$. What follows is a zero-knowledge argument of knowing a witness, such that the element committed to in C lies in V as well. Therefore, the protocol's soundness relies on the security of the accumulator.

Theorem 4.15. *Let Π_{acc} be the Nguyen accumulator from Construction 3.67, and $\mathsf{pp} = (p, e, g_1, \tilde{g}_1, \tilde{g}_1^s, t) \leftarrow [\mathsf{Setup}(\cdot, 1^q)]$ with $t = (g_1^s, g_1^{s^2}, \ldots, g_1^{s^q})$. Then Construction 4.14 is a special honest-verifier zero-knowledge argument of knowledge for the relation $R := \{((\Omega, C, g_2, h), (r, \alpha)) \mid C = g_2^r h^\alpha \wedge \alpha \in \Omega\}$ with negligible knowledge error.*

Proof. The *completeness* of Construction 4.14 follows by the properties of the Nguyen accumulator: For $V := \mathsf{AccCreate}(\mathsf{pp}, \Omega)$, and $W_\alpha := \mathsf{WitCreate}(\mathsf{pp}, \Omega, i)$ (assuming $\alpha = \omega_i$) we know $V = g_1^{\prod_{i=1}^{|\Omega|}(\omega_i+s)}$ and $W_\alpha = V^{\frac{1}{(\alpha+s)}}$, respectively. Therefore, we see

$$e(W, \tilde{g}_1)^\alpha \cdot e(V, \tilde{g}_1^{-1})^z = e(W, \tilde{g}_1^s)^{-1} \iff e(W_\alpha, \tilde{g}_1^\alpha \tilde{g}_1^s)^z = e(V, \tilde{g}_1)^z \iff e(V, \tilde{g}_1)^z = e(V, \tilde{g}_1)^z.$$

The rest follows from correctness of the generalized Schnorr proof.

We proceed to the *relaxed special soundness* property as in the proof of Theorem 4.13. Again, for two accepting transcripts, the special soundness extractor of the Schnorr proof outputs a witness (r, α, z), fulfilling

$$C = g_2^r h^\alpha \text{ and } e(W, \tilde{g}_1)^\alpha \cdot e(V, \tilde{g}_1^{-1})^z = e(W, \tilde{g}_1^s)^{-1}.$$

Whereas the first yields a valid witness for opening C to α, the second yields a witness for α being accumulated in V. To show this, we make a case distinction over z being zero or not.

Case $\mathbf{z = 0}$: In this case, we can deduce that $e(W, \tilde{g}_1)^\alpha = e(W, \tilde{g}_1)^{-s}$. By $W \neq 1 \neq g_1$ it follows, that $e(W, \tilde{g}_1) \neq 1$ and therefore $\alpha = -s$ holds. This information would enable us, to break the security of the accumulator scheme, according to Definition 3.65. Hence this case happens with negligible probability and the extractor fails.

Case $\mathbf{z \neq 0}$: In this case we know z^{-1} exists and we can compute

$$\begin{aligned} e(W, \tilde{g}_1)^\alpha \cdot e(V, \tilde{g}_1^{-1})^z = e(W, \tilde{g}_1^s)^{-1} &\iff e(W, \tilde{g}_1^\alpha \tilde{g}_1^s) = e(V, \tilde{g}_1)^z \\ &\iff e(W^{z^{-1}}, \tilde{g}_1^\alpha \tilde{g}_1^s) = e(V, \tilde{g}_1). \end{aligned}$$

The latter states that $W^{z^{-1}}$ is a valid witness for V within the Nguyen accumulator. Formally this means, $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{AccCreate}(\mathsf{pp}, \Omega), \alpha, W^{z^{-1}}) = 1$ holds. If $\alpha \notin \Omega$ we broke the accumulator's security, and fail (again with negligible probability). Else we can output the *valid* witness (r, α) for the relation.

Therefore, the relaxed special soundness extractor succeeds in all, but a negligible fraction of cases.

The *special honest-verifier simulator* on input challenge c simply chooses $W \leftarrow \mathbb{G} \setminus \{1\}$ (group of g_1), which is distributed as in the real protocol, assuming W_α is a generator. This is the case, as only values unequal to $-s$ are accumulated by the Nguyen accumulator (else the secret would have been found). Then the simulator for the Schnorr proof (instantiated with W) is run. ☐

The advantage of using the accumulator instead of signatures is, that additional public parameters need to be published only *once*, allowing set membership proofs for *all* sets with q or less elements (q depends on the parameters). The signature-based variant would need a distinct signing key pair and corresponding signatures for each set Ω. The drawback to be bound to q elements can be overcome by splitting bigger sets up, and then performing several OR-concatenated set membership proofs. Another small drawback is the time complexity $\mathcal{O}(n \log n)$ for computing accumulator value and witness without auxiliary information [Zho+16].

Range Proof for Attribute in Range $[0, u^\ell)$

The problem to prove that a secret attribute α lies within a range of integers $[A, B] := \{A, A+1, \ldots, B\}$ is a special case of a set membership proof. Here, we first focus on ranges of form $[0, u^\ell)$, i. e. an argument for $\mathsf{PK}\Big\{(\alpha, r) : C = \mathsf{Com}(\alpha; r) \wedge \alpha \in [0, u^\ell)\Big\}$. One could simply employ the protocol from Construction 4.14. If the range includes several thousands of elements, we would either need an accumulator for many elements meaning big public parameters or several OR-concatenated set membership proofs. If the range spans over millions of integers, then this

approach becomes impractical. However, the special case of a range gives allows more efficient protocols.

Camenisch, Chaabouni, and shelat [CCs08] provide a method, which makes use of the set membership proof as above, resulting in a communication complexity of $\mathcal{O}(\frac{\log L}{\log\log L})$, where L denotes the size of the range [CLs10]. We first consider a range of form $[0, u^l)$ for $u, \ell \in \mathbb{N}$ with $u^\ell < p$. This will be generalized to an arbitrary interval $[A, B]$ in Section 4.1.3.3.

Construction 4.16. Let Π_{acc} be the Nguyen accumulator from Construction 3.67, $\mathsf{pp} = (p, e, g_1, \tilde{g}_1, \tilde{g}_1^s, t) \in [\mathsf{Setup}(\cdot, 1^q)]$ with $t = (g_1^s, g_1^{s^2}, \ldots, g_1^{s^q})$, $u, \ell \in \mathbb{N}$ with $u^\ell - 1 < p$ and $u \leq q$, $\Omega = \{0, \ldots, u-1\}$, $C = g_2^r h^\alpha$. Here $C, \mathsf{pp}, \Omega, u, \ell$ are common, and $\alpha \in [0, u^\ell)$ and r the prover's private input.

In a first step, the prover and verifier (locally) compute $V := \mathsf{AccCreate}(\mathsf{pp}, \Omega)$. Then, the prover computes a representation for α to base u, such that $\alpha = \sum_{j=0}^{\ell-1} \alpha_j \cdot u^j$. For $j = 0, \ldots, \ell-1$ she computes a witness $W_j := \mathsf{WitCreate}(\mathsf{pp}, \Omega, \alpha_j)$, chooses $z_j \leftarrow \mathbb{Z}_p^*$ and computes $\hat{W}_j := W_j^{z_j}$. Then, she sends the $(\hat{W}_j)_{j=0}^{\ell-1}$ to the verifier and runs the Schnorr proof (Construction 3.44)

$$\mathsf{PK}\left\{\left(r, (\alpha_j, z_j)_{j=0}^{\ell-1}\right) : C = g_2^r \prod_{j=0}^{\ell-1} (h^{u^j})^{\alpha_j} \wedge \bigwedge_{j=0}^{\ell-1} e(\hat{W}_j, \tilde{g}_1)^{\alpha_j} \cdot e(V, \tilde{g}_1^{-1})^{z_j} = e(\hat{W}_j, \tilde{g}_1^s)^{-1}\right\}.$$

The verifier accepts if and only if $\hat{W}_j \neq 1$ for all $j = 0, \ldots, \ell-1$ and she accepts in this proof of knowledge.

This protocol relies on the fact, that any $\alpha \in [0, u^\ell)$ can (uniquely) be rewritten in u-ary notation as the sum $\alpha = \sum_{j=0}^{\ell-1} \alpha_j \cdot u^j$ with $\alpha_j \in \Omega := \{0, \ldots, u-1\}$. The main idea is to prove the latter, by executing ℓ-many set membership proofs for $\alpha_j \in \Omega$, while ensuring correspondence of the α_j to α. To prove this correspondence, it suffices to prove knowledge of $C = g^r \prod_{j=0}^{\ell-1} (h^{u^j})^{\alpha_j}$, which implies knowledge of opening C to α. The protocol's soundness again relies on the security of the accumulator scheme.

Theorem 4.17. *Let Π_{acc} be the Nguyen accumulator from Construction 3.67, and $\mathsf{pp} = (p, e, g_1, \tilde{g}_1, \tilde{g}_1^s, t) \leftarrow [\mathsf{Setup}(\cdot, 1^q)]$ with $t = (g_1^s, g_1^{s^2}, \ldots, g_1^{s^q})$ and $u \leq q$. Then Construction 4.16 is a special honest-verifier zero-knowledge argument of knowledge for the relation*

$$R := \left\{((\Omega, C, g_2, h, u, \ell), (r, \alpha)) \;\middle|\; C = g_2^r h^\alpha \wedge \alpha \in [0, u^\ell)]\right\}$$

with negligible knowledge error.

Proof. *Completeness* of the protocol follows from completeness of Construction 4.14 and the computable representation $\alpha = \sum_{j=0}^{\ell-1} \alpha_j u^j$ for $\alpha \in [0, u^\ell)$ and $\alpha_j \in \{0, 1, \ldots, u-1\}$.

Regarding *soundness* and extraction, we first consider the ℓ-many proofs for knowing a witness for the accumulated value. Assuming ℓ is related polynomially to the security parameter, the chance at least one of them breaks the accumulator's security, and making the extractor fail, is again negligible. Therefore we can assume α_j to lie in $\{0, 1, \ldots, u-1\}$. By defining $\alpha := \sum_{j=0}^{\ell-1} \alpha_j u^j$, we know $\alpha \in [0, u^\ell)$. Further,

$$C = g_2^r h^\alpha = g_2^r h^{\sum_{j=0}^{\ell-1} \alpha_j u^j} = g_2^r \prod_{j=0}^{\ell-1} (h^{u^j})^{\alpha_j},$$

implies (r, α) is a valid witness for the relation.

The *special honest-verifier simulator* chooses $W_j \leftarrow \mathbb{G} \setminus \{1\}$ for $i = 0, \ldots, \ell-1$, instantiates and runs the Schnorr protocol's simulator with these values. □

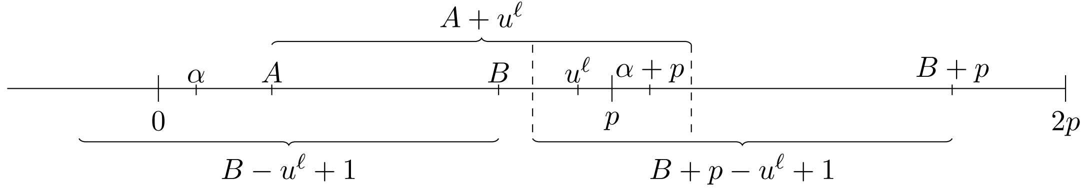


Figure 4.1: Illustration of intersection problem without limiting the difference $B - A$

Range Proof for Attribute in Arbitrary Range $[A, B]$

Now, we construct a range proof for (nearly) arbitrary integer ranges $[A, B]$, with $0 \leq A < B < p$, based on the previous proof for a $[0, u^\ell)$ range. Here, we will employ a method presented by Schoenmakers [Sch05; Sch01]. The idea is to represent the $[A, B]$-interval by the intersection of two intervals of size u^ℓ. Then shift each interval, such that it starts in 0, and shift the committed attribute value by the same amount. What follows is an AND-composition of two "standard" range proofs.

In particular, for integers α and $u^\ell > B - A$, we can rewrite an interval $[A, B]$ by

$$\begin{aligned} &\alpha \in [A, B] \\ \iff &\alpha \in [A, A + u^\ell) \cap [B - u^\ell + 1, B + 1) \\ \iff &\alpha - A \in [0, u^\ell) \wedge \alpha - B + u^\ell - 1 \in [0, u^\ell). \end{aligned}$$

This observation then leads to following Construction 4.18.

Construction 4.18. Let Π_{acc} be the Nguyen accumulator from Construction 3.67 and $\mathsf{pp} \leftarrow \mathsf{Setup}(\cdot, 1^q)$. The common input is pp, $A, B, u, \ell \in \mathbb{N}$, $\Omega = \{0, \dots, u - 1\}$ and $C = g_2^r h^\alpha$, such that $0 < B - A < \min\{\frac{p+1}{2u-1}, u^\ell - 1\}$, $u \leq q$ and $u^\ell - 1 < p$. The prover's private input is $\alpha \in [A, B]$ and r.

First prover and verifier (locally) compute $C_A = Ch^{-A}$ and $C_B = Ch^{-B+u^\ell-1}$. The prover's private open value for C_A is $(r, \alpha - A)$ and $(r, \alpha - B + u^\ell - 1)$ for C_B. Next, Construction 4.16 is instantiated twice, once with C_A and once with C_B and the corresponding private open values. The verifier accepts, if and only if she accepts in both proofs.

Note, that a ppt prover can only open C_A to $\alpha - A$ and C_B to $\alpha - B + u^\ell - 1$, if she has proven knowledge of opening C to α. Therefore this protocol matches the observation above perfectly for integers. However, there is an additional constraint, such that $B - A < \frac{p+1}{2u-1}$ needs to hold. This is necessary, as the committed value α represents a $\mathbb{Z}_p$ value, this shifting technique can cause overflows, such that $\alpha \notin [A, B]$ would be accepted. This problem is depicted in Figure 4.1.

For the chosen parameters $A < B$ and $u^\ell < p$ any $\mathbb{Z}_p$ element between the dashed lines would be accepted in a range proof. These values, like α in the sketch, are clearly not contained in $[A, B]$. We need to ensure there is no such intersection as between the dashed lines. We achieve this by the restriction of the maximal range size in dependence of the base u.

We first observe that the problematic intersection is empty, if

$$A + u^\ell < p + B - u^\ell + 1 \iff A - B + 2u^\ell < p + 1.$$

We assume $u^{\ell-1} < B - A < u^\ell$ (i. e. ℓ is minimal to represent the interval size to base u). We demand the range size $B - A$ to be less than $\frac{p+1}{2u-1}$. This ensures the intersection is empty, since we can compute

$$A - B + 2u^\ell = A - B + 2uu^{\ell-1} < A - B + 2u(B - A) = (B - A)(2u - 1) < \frac{p+1}{2u-1}(2u - 1) = p + 1.$$

The constraint on the size of $[A, B]$ does not need to concern us, as p needs to be exponential in the security parameter. However, u and ℓ should be chosen in dependence of the interval size. To get close to an optimal solution with respect to communication complexity one would first choose a numerical approach like in [Bla97]. Then, it would be checked whether $B - A < \frac{p+1}{2u-1}$ holds for the computed u and ℓ. By the exponential size of p, this should be the case for usual ranges, but if not, one could refine the optimization constraints. For simplicity it is possible to start with $u = 2$ and $\ell = \lceil \log_2(B - A) \rceil$.

4.2 Basic Anonymous Credential System

In this section, we present the first stage of our attribute-based credential system. In the previous sections, we presented all necessary tools to implement such a system. In an (attribute-based) credential system, we have *users* and *organizations* (in our setting we divide organizations into two roles: *issuer* and *verifier*, the latter is sometimes called *service provider*). The main goal of a user is to stay *anonymous*, i. e. nobody should learn about the user's secret value called the *user secret* when interacting with the user. The main tool to achieve this is the *zero knowledge argument of knowledge* presented in Section 3.7. A user interacts with issuers to obtain *credentials* attesting certain attributes. Basically, a credential is a digital signature (Section 3.4) on the user secret and the attributes. Since the user is interested in staying anonymous, and the issuer needs to issue a digital signature on the user secret, we implement the credential issuance with the aid of the scheme for signing partially committed value presented in Section 4.1.1, where the private message is the user secret. After having obtained credentials, the user uses these to gain access to a service. The access policy of a service provider is formalized by a predicate. A user then needs to verify that the attributes she was issued in form of her credentials match the given predicate. To realize the showing of a credential, the main tool is the protocol for proving knowledge of a signature presented in Section 4.1.2. Further, we want to reduce the information transferred by the proof of the predicate satisfiability to a minimum. Here, the technique of proofs of partial knowledge presented in Section 3.11 is a crucial tool. Our system supports predicates over the relation introduced in Section 4.1.3. This means a service provider can demand attributes to be (un)equal to some possibly public values, or even to be an element of a set or a range. We emphasize at this point that proofs of partial knowledge allow predicates of arbitrary combinations of relations as well as an arbitrary combination of conjunctions and disjunctions. Not that we implicitly also support negation, by using the inequality relation in place of the equality relation and using considering of membership in the complement of a set/range.

4.2.1 Definition

We now define algorithms and interactive protocols, their inputs and outputs, which reflect these considerations and build a *basic credential system*. This definition is similar to the framework proposed by Camenisch and Lysyanskaya [CL01], but adds predicates and splits up the $\mathsf{FormNym}$ protocol.

Definition 4.19. A *basic credential system* consists of the following (ppt) algorithms and interactive protocols:

- $\mathsf{Setup}(1^n)$: On input security parameter 1^n, it outputs public parameters pp with $|\mathsf{pp}| \geq n$, an attribute universe $\mathcal{U}_A$ and a predicate universe $\mathcal{U}_\Phi \subseteq \{\phi \mid \phi : \mathcal{U}_A^k \to \{0,1\}, k \in \mathbb{N}\}$. Additionally, it (implicitly) outputs application-dependent auxiliary parameters $\mathsf{pp}_{\mathrm{aux}}$.
- $\mathsf{U.Init}(\mathsf{pp})$: On input public parameters pp, it outputs a user secret key usk.

- $\mathsf{I.Init}(\mathsf{pp}, 1^\ell)$: On input public parameters pp and 1^ℓ with $\ell \in \mathbb{N}$, it outputs an issuer public key ipk and secret key isk. The number ℓ denotes the number of attributes supported by the issuer.

- $\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})$: On input public parameters pp and user secret usk, it outputs a pseudonym nym and corresponding pseudonym secret psk.

- $(\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}), \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym}))$ is an interactive protocol on common inputs public parameters pp and pseudonym nym, where the user has user secret usk and pseudonym secret psk as private input. After execution the verifier outputs a bit $b \in \{0, 1\}$ and we interpret 1 as accepting, 0 as denying.

- $(\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}), \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{isk}))$ is an interactive protocol on common inputs public parameters pp, pseudonym nym and attributes $(a_1, \ldots, a_\ell) \in \mathcal{U}_A^\ell$. The user has user secret usk, pseudonym secret psk and issuer public key ipk as additional input, whereas the issuer has secret key isk. After the interaction, the user outputs (locally) either the credential cred corresponding to ipk over her user secret usk and the attributes $(a_1, \ldots, a_\ell)$ or a failure symbol represented by $\perp$.

- $(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}), \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi))$ is an interactive protocol on common inputs public parameters pp, issuer public key ipk, pseudonym nym, and a predicate $\phi \in \mathcal{U}_\Phi$, where the user has user secret usk, pseudonym secret psk and credential cred as private input. After execution the verifier outputs a bit $b \in \{0, 1\}$ and we interpret 1 such that the credential cred satisfies the predicate ϕ and the verifier accepts, 0 as denying.

Usage of the System We now give a *concrete* example of how these algorithms are supposed to be used. At the very beginning, some *(trusted) system manager* runs Setup, in dependency of a security parameter n, to generate the public parameters of the system. Note that all algorithms have $|\mathsf{pp}| \geq n$ as input, and thus are ppt algorithms in n. A user runs $\mathsf{U.Init}$ to obtain a user secret usk. An issuer runs $\mathsf{I.Init}$ with additional input $\ell = 3$, to receive secret key isk for issuing three attributes, and a corresponding public key ipk. The number of attributes is given in unary notation, to allow the algorithm to be ppt in ℓ *and* n. The public key ipk needs to be made available to all participants, e. g. by a public-key-infrastructure (cf. Chapter 5). Then, the user runs $\mathsf{CreateNym}$ to get a new pseudonym nym unknown to any other participant, and the corresponding secret psk. Before requesting a credential, user and issuer run some *application-dependent* protocol to agree on the attributes to be issued. For example, the user knows $(a_1 = 25, a_2 = \text{Germany}, a_3 = 2894)$, where the first represents the user's age and the second the user's living place. Given all these inputs, they run the $(\mathsf{RcvCred}, \mathsf{IssCred})$ protocol, resulting in a credential cred corresponding to ipk over usk and the attributes. The issuer possibly stores an association of nym with this particular issuance protocol. However, the user does not want any connection to this protocol run and creates another pseudonym nym'. Using nym' she executes $(\mathsf{ProveCred}, \mathsf{VrfyCred})$ with some verifier, who demands to match the predicate

$$\phi(a_1, a_2) = 1 :\Longleftrightarrow (a_1 \in [18, 150] \wedge a_2 = \text{Germany}) \vee (a_1 \in [21, 150] \wedge a_2 = \text{USA}).$$

After accepting, the verifier could associate nym' with fulfilling the particular ϕ. This allows them to run $(\mathsf{ProveNym}, \mathsf{VrfyNym})$ with nym', instead of the larger, and therefore less efficient, protocol $(\mathsf{ProveCred}, \mathsf{VrfyCred})$.

In the next paragraph, we formally describe when a credential system is functional, i.e. when it is *correct*. Before we do this, we state the idea of what is meant by a correct credential system. Note that this correctness does not include any security requirements, which are defined in

Section 4.2.2. Intuitively, we want that in a system, that was set up using Setup, users can create pseudonyms and receive credentials as well as prove the possession of some attributes via these credentials. For this, it is important that the users and issuers generate their keys using U.Init, respectively I.Init. Using their secret keys, users can create a pseudonym nym via CreateNym which can be used in the (ProveNym, VrfyNym) protocol to prove that they actually know the secret key belonging to this pseudonym. We call a pseudonym that is used in this protocol with the corresponding parameters valid if the protocol outputs 1 with a significantly high probability. Using these valid pseudonyms, one can receive a credential cred via the (RcvCred, IssCred) protocol. This cred can then successfully be used in the (ProveCred, VrfyCred) protocol to prove the possession of the attributes, which the credential was issued over. Here, we call a credential cred valid regarding the parameters in the (ProveCred, VrfyCred) protocol, if using cred in this protocol results in an output of 1 with significantly high probability.

Correctness We now want to formally define when an anonymous credential system works correctly.

We say that an anonymous credential system is *correct* if

- **Honestly generated pseudonyms are valid**: For all $n \in \mathbb{N}$, $(\mathsf{pp}, \cdot, \cdot) \in [\mathsf{Setup}(1^n)]$, $\mathsf{usk} \in [\mathsf{U.Init}(\mathsf{pp})]$, $(\mathsf{nym}, \mathsf{psk}) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})]$ we have that

  $$\Pr[(\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym})) \to 1] = 1 - \mu(|pp|)$$

  for some negligible function μ. We call such a nym *vaild*.

- **Honestly issued credentials are valid**:

  For all $n, \ell \in \mathbb{N}$,
  $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \in [\mathsf{Setup}(1^n)]$,
  $\mathsf{usk} \in [\mathsf{U.Init}(\mathsf{pp})]$,
  $(\mathsf{ipk}, \mathsf{isk}) \in [\mathsf{I.Init}(\mathsf{pp}, 1^\ell)]$,
  $(\mathsf{nym}, \mathsf{psk}) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})]$,
  $(\mathsf{nym}', \mathsf{psk}') \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})]$,
  $(a_1, \ldots, a_\ell) \in \mathcal{U}_A^\ell$,
  all $\phi \in \mathcal{U}_\Phi$ with $\phi(a_1, \ldots, a_\ell) = 1$,
  for all credentials cred output by $\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^\ell, \mathsf{usk}, \mathsf{psk}) \leftrightarrow$ $\mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^\ell, \mathsf{isk})$

  we have that

  $$\Pr[(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}', \phi, \mathsf{usk}, \mathsf{psk}', \mathsf{cred}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}', \phi)) \to 1] = 1 - \mu(|pp|)$$

  for some negligible function μ. We call such a cred *valid*.

Note that in *Honestly issued credentials are valid* the pseudonyms are valid since they are honestly generated.

4.2.2 Security Notions

The defined credential system still lacks clear security notions. As briefly mentioned before, we want to achieve *anonymity* and *soundness* of the system. To prevent linking of user actions, anonymity essentially pours down to hide the user identity, respectively the user secret, and the used credential in any protocol execution from (cheating) issuers and verifiers. Next, soundness basically assures, that *only* honestly issued credentials, over attributes matching a given predicate, may be used to convince verifiers. Additionally, only the user, who created a pseudonym or received a credential, should be able to use either.

4.2.2.1 Anonymity

This section states what we require of a credential system to be anonymous, where anonymity means that actions of users cannot be linked. To achieve this property, the algorithms and interactive protocols defined in 4.19 have to hide the user secret. We will first give an intuitive description what that means for the individual algorithms/protocols and conclude with a formal definition of anonymity.

1. (CreateNym **hides the user secret**) Essentially, users want to use pseudonyms to stay anonymous. Therefore, it is important that a created pseudonym does not tell anything about the user secret. This means that an adversary cannot distinguish whether a pseudonym nym was created using a secret key usk or another key usk′.

2. (ProveNym **hides the user secret**) When a user proves that she knows the key corresponding to a pseudonym, she wants to be sure that taking part in this protocol does not tell the other participant anything about her key. This means that an adversary cannot tell which user secret was incorporated in a pseudonym, even if this adversary participates in the protocol.

3. (**Anonymity when executing** RcvCred) A user that likes to receive a credential wants to do this without the risk of being linked to other actions, which she already performed. Therefore, it is important that performing the RcvCred part of the protocol executed when issuing a credential does not tell anything about the user secret. Additionally, this has to hold if the adversary can take the role of the issuer in this protocol execution.

4. (**Anonymity when executing** ProveCred) After a user obtained a (valid) credential over certain attributes via RcvCred, she does not want her actions to be linked, when she proves possession of it satisfying some predicate. This means, executing ProveCred must not reveal anything about the user secret and anything beyond possession of *some* valid credential for the demanded predicate. This even holds, if the adversary interacted with RcvCred and then ProveCred.

We will now give a formal definition of the properties which we just described informally. These formal definitions are inspired by the definitions given by Blömer and Bobolz [BB17].

Definition 4.20. We call a credential system $\Pi = (\mathsf{Setup}, \mathsf{U.Init}, \mathsf{I.Init}, \mathsf{CreateNym}, (\mathsf{ProveNym}, \mathsf{VrfyNym}), (\mathsf{RcvCred}, \mathsf{IssCred}), (\mathsf{ProveCred}, \mathsf{VrfyCred}))$ (Definition 4.19) *anonymous* if the following properties are fulfilled:

1. (CreateNym **hides the user secret**) For all $n \in \mathbb{N}$, all $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \in [\mathsf{Setup}(1^n)]$, all $\mathsf{usk}, \mathsf{usk}' \in [\mathsf{U.Init}(\mathsf{pp})]$ and all (unrestricted) adversaries $\mathcal{A}$ the following distributions are identical:
   - $\text{output}_\mathcal{A}[\mathsf{nym} \leftarrow \mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}) : \mathcal{A}(\mathsf{pp}, \mathsf{nym})]$
   - $\text{output}_\mathcal{A}[\mathsf{nym} \leftarrow \mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}') : \mathcal{A}(\mathsf{pp}, \mathsf{nym})]$

2. (ProveNym **hides the user secret**) For all $n \in \mathbb{N}$, $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \in [\mathsf{Setup}(1^n)]$, all $\mathsf{usk}, \mathsf{usk}' \in [\mathsf{U.Init}(\mathsf{pp})]$, all $((\mathsf{nym}, \mathsf{psk}), (\mathsf{nym}, \mathsf{psk}')) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})] \times [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}')]$ and all (unrestricted) adversaries $\mathcal{A}$ the following distributions are identical:
   - $\text{output}_\mathcal{A}[\mathcal{A}(\mathsf{pp}, \mathsf{nym}) \leftrightarrow \mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk})]$
   - $\text{output}_\mathcal{A}[\mathcal{A}(\mathsf{pp}, \mathsf{nym}) \leftrightarrow \mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}', \mathsf{psk}')]$

3. (**Anonymity when executing** RcvCred) For all $n, \ell \in \mathbb{N}$, $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \in [\mathsf{Setup}(1^n)]$, all $\mathsf{usk}, \mathsf{usk}' \in [\mathsf{U.Init}(\mathsf{pp})]$, all $((\mathsf{nym}, \mathsf{psk}), (\mathsf{nym}, \mathsf{psk}')) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})] \times [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}')]$, all ipk, all $(a_i)_{i=1}^{\ell} \in \mathcal{U}_A^{\ell}$ and all (unrestricted) adversaries $\mathcal{A}$ the following distributions are identical:

- $\text{output}_{\mathcal{A}}[\mathcal{A}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}) \leftrightarrow \mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk})]$
- $\text{output}_{\mathcal{A}}[\mathcal{A}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}) \leftrightarrow \mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}', \mathsf{psk}')]$

4. (**Anonymity when executing** $\mathsf{ProveCred}$) For all $n, \ell \in \mathbb{N}$, $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \in [\mathsf{Setup}(1^n)]$, all $\mathsf{usk}, \mathsf{usk}' \in [\mathsf{U.Init}(\mathsf{pp})]$, all $((\mathsf{nym}, \mathsf{psk}), (\mathsf{nym}, \mathsf{psk}')) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})] \times [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}')]$, all ipk, all $\phi \in \mathcal{U}_\Phi$, all $(a_i)_{i=1}^{\ell}, (a_i')_{i=1}^{\ell} \in \mathcal{U}_A^{\ell}$ with $\phi((a_i)_{i=1}^{\ell}) = \phi((a_i')_{i=1}^{\ell})$, all (unrestricted) adversaries $\mathcal{A}$', all $\mathsf{cred} \neq \perp$ output by $\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathcal{A}'((a_i)_{i=1}^{\ell})$, all $\mathsf{cred}' \neq \perp$ output by $\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i')_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathcal{A}'((a_i')_{i=1}^{\ell})$ and all (unrestricted) adversaries $\mathcal{A}$ the following distributions are identical:
   - $\text{output}_{\mathcal{A}}[\mathcal{A}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi) \leftrightarrow \mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred})]$
   - $\text{output}_{\mathcal{A}}[\mathcal{A}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi) \leftrightarrow \mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}', \mathsf{psk}', \mathsf{cred}')]$

4.2.2.2 Soundness

In an anonymous credential system no user or group of users should be able to use or create a credential for verification that was not issued to one of them. A user also should not be able to successfully identify herself using a pseudonym which was created by another user.

Before we start with the formal definition of soundness, we will give the intuition of what a possible adversary $\mathcal{A}$ is able to do in an anonymous credential system and when we consider this system sound. We assume that the adversary wants to achieve its goals in a system that was correctly set up using Setup. Respectively, all users and issuers used $\mathsf{U.Init}$ and $\mathsf{I.Init}$ to create their keys. Afterwards, the adversary can make users and issuers execute the algorithms and interactive protocols of the system. When $\mathcal{A}$ lets a user create a pseudonym, it receives the created nym. For the (interactive) protocols, the adversary gets to know if these protocol runs were successful, i.e. output 1. However, $\mathcal{A}$ does not get the credential created by the corresponding issuing protocol. Moreover, the adversary can corrupt users and issuers from which it learns all, previously secret, information. In the protocol runs, the adversary takes the role of the corrupted user/issuer in the execution. By giving $\mathcal{A}$ the ability to corrupt, we ensure that our system is secure for honest users and issuers even if some dishonest users and issuers collaborate. We denote two goals of possible adversaries: Non-Impersonation and Credential-Unforgeability. Non-Impersonation means that no one can successfully use a pseudonym nym for identification without knowing the corresponding secret keys usk and psk. Credential-Unforgeability means that no adversary is able to forge a valid credential that was not issued to her. We say that a credential system is sound if no adversary, given the described abilities, can achieve any of the mentioned goals. The experiment and definitions used in this chapter are roughly based on the definitions proposed by Pashalidis and Mitchell [PM04]

Definition 4.21 (Soundness experiment)**.** Define the following Experiment $\mathsf{Exp}_{\Pi,\mathcal{A}}^{Soundness}(n)$ for an ACS Π between a challenger and a ppt adversary $\mathcal{A}$. Note here that we assume secure communication for honest entities in the experiment (cf. Chapter 5). The adversary only sees outputs which are explicitly mentioned in the definition below.

- The challenger runs $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \leftarrow \mathsf{Setup}(1^n)$.
- The adversary $\mathcal{A}$ chooses $1^{n_U}, 1^{n_I}, 1^{\ell_1}, \ldots, 1^{\ell_{n_I}}$ with $n_U, n_I, \ell_1, \ldots, \ell_{n_I} \in \mathbb{N}$.
- The challenger creates sets U and I for users and issuers, with $|U| = n_U$ and $|I| = n_I$, and initially empty subsets $\hat{U}$ and $\hat{I}$ for corrupted users and issuers respectively. The challenger also creates an initially empty pseudonym-user-pair set S_{nym} and a credential-ID-credential-pair set S_{cred}. The user and issuer sets contain unique IDs for each entity, the mapping from the IDs to the actual entities is done by the challenger. The IDs are chosen independently from any secret keys.

- For each user $u \in U$ the challenger makes u run $\mathsf{U.Init}(\mathsf{pp})$ to obtain a user secret key usk.
- For each issuer $i \in I$ the challenger makes i run $\mathsf{I.Init}(\mathsf{pp}, 1^{\ell_j})$ to obtain an issuer public key ipk and secret key isk. Using 1^{ℓ_j} for the jth issuer. The ipk is made publicly available.
- After this initial setup the adversary gets as input the sets U, I and the public parameters of the ACS Π. $\mathcal{A}$ may now issue the following oracle type queries to the challenger.
  - *corruptUser*(u): $\mathcal{A}$ may arbitrarily select a user $u \in U$ to gain control over her. The challenger hands all private information of u to $\mathcal{A}$. This includes the user's usk, all of her pseudonym/key pairs $(\mathsf{nym}, \mathsf{psk})$, her credentials and all her past protocol views. From that point on, $\mathcal{A}$ has full control over u. Note here that, since $\mathcal{A}$ has full control over u, it can arbitrarily choose a new usk for u and use it in following protocols. The challenger adds u to $\hat{U}$.
  - *corruptIssuer*(i, ipk'): $\mathcal{A}$ may arbitrarily select an issuer $i \in I$ to gain control over her. The challenger hands all private information of i to $\mathcal{A}$. This includes the issuer's secret key isk, all pseudonyms users have used in past protocols with her and all her past protocol views. From that point on $\mathcal{A}$ has full control over i. If ipk' is not equal to the ipk of i, ipk' becomes i's new public key. The challenger adds i to $\hat{I}$.
  - *runCreateNym*(u): $\mathcal{A}$ may select a user, which then runs $\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})$. $\mathcal{A}$ learns the pseudonym nym but not the corresponding psk. The challenger adds (nym, u) to S_{nym}.
  - *runProveVrfyNym*(u, v, nym): Where $u \in U$ and $v \in U \cup I$. The challenger makes u and v execute $\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym})$, where usk is u's secret key and psk is the pseudonym secret of nym. If one or both of those entities are corrupted, $\mathcal{A}$ takes the corresponding role in the protocol execution. The challenger returns 1 to $\mathcal{A}$ if v outputs 1, 0 otherwise.
  - *runRcvIssCred*$(u, i, \mathsf{nym}, (a_i)_{i=1}^{\ell})$: Where $u \in U$, $i \in I$, and where ℓ is the ℓ_j that $\mathcal{A}$ has chosen for i in the setup. The challenger makes u and i execute $\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{isk})$, where usk is u's secret key, psk is the pseudonym secret of nym and ipk and isk are i's public and secret key. If one or both of those entities are corrupted, $\mathcal{A}$ takes the corresponding role in the protocol execution. If the protocol is successful, the challenger returns 1 and a unique *credID* to $\mathcal{A}$ and adds $(credID, \mathsf{cred})$ to S_{cred}, else 0.
  - *runProveVrfyCred*$(u, v, \mathsf{nym}, credID, \phi)$: Where $u \in U$ and $v \in U \cup I$. If u is honest and *credID* is not contained in a pair in S_{cred} the challenger returns 0. Otherwise, the challenger makes u and v execute $\mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi)$, where cred is the credential in the $(credID, \mathsf{cred})$ pair in S_{cred}, ipk is the public key of the issuer who issued cred, usk is u's user secret and psk the pseudonym secret corresponding to nym. If one or both of those entities are corrupted, $\mathcal{A}$ takes the corresponding role in the protocol execution. The challenger returns 1 to $\mathcal{A}$ if v outputs 1, 0 otherwise.

Definition 4.22 (Non-Impersonation)**.** In $\mathsf{Game}_{\Pi,\mathcal{A}}^{NonImp}(n)$ an adversary $\mathcal{A}$ first participates in $\mathsf{Exp}_{\Pi,\mathcal{A}}^{Soundness}(n)$. After that it outputs nym, chooses a corrupted user $u \in \hat{U}$ and an honest verifier $v \in (I \cup U) \setminus (\hat{I} \cup \hat{U})$. Then, the challenger makes u and v execute $\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym})$, where pp are the public parameters of the prior experiment. Since $u \in \hat{U}$, $\mathcal{A}$ fully controls the user during protocol execution. $\mathcal{A}$ wins if and only if v accepts and the following requirement is true:

- $\exists\ (\mathsf{nym}, u') \in S_{nym}$ with $u' \notin \hat{U}$. Which means that nym does belong to a non-corrupted user.

If for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function μ such that $\Pr[\mathsf{Game}_{\Pi,\mathcal{A}}^{NonImp}(1) = 1] \leq \mu(n)$, we say that Π offers non-impersonation.

Definition 4.23 (Credential Unforgeability)**.** In $\mathsf{Game}_{\Pi,\mathcal{A}}^{CredForge}(n)$ an adversary $\mathcal{A}$ first participates in $\mathsf{Exp}_{\Pi,\mathcal{A}}^{Soundness}(n)$. After that it outputs $(\mathsf{ipk}, \mathsf{nym}, \phi)$, chooses a user $u \in \hat{U}$ and an honest verifier $v \in (I \cup U) \setminus (\hat{I} \cup \hat{U})$. Then, the challenger makes u and v execute $\mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi)$, where pp are the public parameters of the prior experiment. Since $u \in \hat{U}$, $\mathcal{A}$ fully controls the user during protocol execution. $\mathcal{A}$ wins if and only if v accepts and the following requirements are true:

- *ipk* is the public key of a non-corrupted issuer.
- $\mathcal{A}$ has never queried $runRcvIssCred(u', i, \mathsf{nym}', (a_i)_{i=1}^{\ell})$, where $\phi((a_i)_{i=1}^{\ell}) = 1$, i is the issuer corresponding to ipk and $u' \in \hat{U}$. The latter includes instances where $corruptUser(u')$ got queried after the credential got granted.

If for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function μ such that $\Pr[\mathsf{Game}_{\Pi,\mathcal{A}}^{CredForge}(n) = 1] \leq \mu(n)$, we say that Π offers credential unforgeability.

Note here that we, at this point, do not look at the case where two fully colluding users share a credential. If the first user gives a credential and her user secret to another user, in regards to the credential system the second user is now the first one.

Definition 4.24 (Soundness)**.** We call Π *sound* if it offers both non-impersonation and credential unforgeability.

4.2.3 Construction of an ACS

Now we can construct a basic credential system according to Definition 4.19. In short, pseudonyms are Pedersen commitments on a generated user secret, and credentials are (partially) blindly signed Pointcheval-Sanders signatures on the user secret and certain attributes. The protocols from Section 4.1 help to achieve *soundness* and *anonymity* of the system. Note that each following $\mathsf{PK}\{(\ldots) : \ldots\}$-representation can be instantiated as Σ-protocol, and will be transformed into a concurrent zero-knowledge argument of knowledge by Damgårds technique.

Construction 4.25. Let G be a type 3 bilinear group generator (Definition 3.2), let $\Pi_{\mathrm{com}} = (\Pi_{\mathrm{com}}.\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be the Pedersen commitment scheme (Construction 3.23) and let $\Pi_{\mathrm{sign}} = (\Pi_{\mathrm{sign}}.\mathsf{Setup}, \mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be the Pointcheval-Sanders signature scheme (Construction 3.17).

- $\mathsf{Setup}(1^n)$: On input security parameter 1^n, Setup generates a type 3 bilinear group $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n)$ and pseudonym public parameters $\mathsf{pp}_{\mathrm{nym}} := (g_{\mathrm{nym}}, h_{\mathrm{nym}}, p, \mathbb{G}_1)$ with $g_{\mathrm{nym}} \leftarrow \mathbb{G}_1 \setminus \{1\}$ and $h_{\mathrm{nym}} \leftarrow \mathbb{G}_1$. Additionally, it creates $\mathsf{pp}_{\mathrm{aux}}$ by generating public parameters of Nguyen accumulator (Section 3.12.3) based on the bilinear group, to enable membership proofs in predicates (Section 4.1.3.3). It returns $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$ and implicitly $\mathsf{pp}_{\mathrm{aux}}$.
- $\mathsf{U.Init}(\mathsf{pp})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$, $\mathsf{U.Init}$ returns user secret key usk with $\mathsf{usk} \leftarrow \mathbb{Z}_p$.
- $\mathsf{I.Init}(\mathsf{pp}, 1^{\ell})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$ and number $\ell \in \mathbb{N}$, $\mathsf{I.Init}$ generates and returns issuer key pair $(\mathsf{ipk}, \mathsf{isk}) \leftarrow \Pi_{\mathrm{sign}}.\mathsf{Gen}_{\ell+1}(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ with $\mathsf{ipk} = (g, Y_0, Y_1, \ldots, Y_{\ell}, \tilde{g}, \tilde{X}, \tilde{Y}_0, \tilde{Y}_1, \ldots \tilde{Y}_{\ell})$.
- $\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$ and user secret usk, $\mathsf{CreateNym}$ generates and returns $(\mathsf{nym}, \mathsf{psk}) \leftarrow \mathsf{Com}_1(\mathsf{pp}_{\mathrm{nym}}, \mathsf{usk})$ with $\mathsf{psk} = (\mathsf{usk}, d)$.

- $(\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{psk}), \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym}))$ is an interactive protocol with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$ and $\mathsf{pp}_{\text{nym}} = (g_{\text{nym}}, h_{\text{nym}}, p, \mathbb{G}_1)$. The prover parses the pseudonym secret psk as (usk, d) and performs an interactive zero-knowledge argument of knowledge of form $\mathsf{PK}\Big\{(\mathsf{usk}, d) : \mathsf{nym} = g_{\text{nym}}^d h_{\text{nym}}^{\mathsf{usk}}\Big\}$ with the verifier.

- $(\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}), \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{isk}))$ is an interactive protocol with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$, $\mathsf{pp}_{\text{nym}} = (g_{\text{nym}}, h_{\text{nym}}, p, \mathbb{G}_1)$ and issuer public key $\mathsf{ipk} = (g, Y_0, \ldots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_0, \ldots, \tilde{Y}_\ell)$. The receiver parses the pseudonym secret psk as (usk, d) and sets $pp_{\text{iss}} = (g, Y_0, p, \mathbb{G}_1) := \mathsf{BlindInit}(\mathsf{pp}, \mathsf{ipk}, \{0\})$ (ignoring pp_{nym} in pp) and computes $(C, (\mathsf{usk}, r)) \leftarrow \mathsf{Com}_1(\mathsf{pp}_{\text{iss}}, \mathsf{usk})$. Then, she sends C to the issuer and runs a Σ-protocol of form

$$\mathsf{PK}\Big\{(\mathsf{usk}, d, r) : \mathsf{nym} = g_{\text{nym}}^d h_{\text{nym}}^{\mathsf{usk}} \wedge C = g^r Y_0^{\mathsf{usk}}\Big\}$$

  with the issuer. As in $(\mathsf{BlindRcv}, \mathsf{BlindIssue})$, the issuer chooses $u \leftarrow \mathbb{Z}_p^*$, computes $(\sigma_1', \sigma_2') := (g^u, (g^x \cdot C \cdot \prod_{i=1}^{\ell} Y_i^{a_i})^u)$ and sends (σ_1', σ_2') to the receiver. The receiver computes $\sigma = (\sigma_1', \sigma_2' \cdot (\sigma_1')^{-r})$, and checks the signature's validity via $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{ipk}, (\mathsf{usk}, a_1, \ldots, a_\ell), \sigma)$. If the check fails, she (locally) outputs $\perp$ and else a credential $\mathsf{cred} = (\sigma, (a_1, \ldots, a_\ell), \mathsf{ipk})$.

- $(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}), \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi))$ is an interactive protocol with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$, $\mathsf{pp}_{\text{nym}} = (g_{\text{nym}}, h_{\text{nym}}, p, \mathbb{G}_1)$ and issuer public key $\mathsf{ipk} = (g, Y_0, \ldots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_0, \ldots, \tilde{Y}_\ell)$.

  The prover parses her private input to $\mathsf{psk} = (\mathsf{usk}, d)$, $\mathsf{cred} = ((\sigma_1, \sigma_2), (a_1, \ldots, a_\ell), \mathsf{ipk})$. If $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{ipk}, (\mathsf{usk}, a_1, \ldots, a_\ell), \sigma) = 0$ or $\phi(a_1, \ldots, a_\ell) = 0$ holds, she outputs $\perp$ and stops. Else she chooses $(u, r) \leftarrow \mathbb{Z}_p^* \times \mathbb{Z}_p$, sets $\sigma' := (\sigma_1', \sigma_2') := (\sigma_1^u, (\sigma_2 \cdot \sigma_1^r)^u)$ and sends σ' to the verifier. For $i = 1, \ldots, \ell$ she generates $(C_i, (a_i, d_i)) \leftarrow \mathsf{Com}(\mathsf{pp}_{\text{nym}}, a_i)$ and sends C_i to the verifier. Then, she runs a zero-knowledge argument of knowledge of form

$$\mathsf{PK}\left\{(\mathsf{usk}, d, (a_i, d_i)_{i=1}^{\ell}, r) : \begin{array}{r} e(\sigma_1', \tilde{g})^r e(\sigma_1', \tilde{Y}_0)^{\mathsf{usk}} \prod_{i=1}^{\ell} e(\sigma_1', \tilde{Y}_i)^{a_i} = \dfrac{e(\sigma_2', \tilde{g})}{e(\sigma_1', \tilde{X})} \\ \wedge\, \mathsf{nym} = g_{\text{nym}}^d h_{\text{nym}}^{\mathsf{usk}} \bigwedge_{i=1}^{\ell} C_i = g_{\text{nym}}^{d_i} h_{\text{nym}}^{a_i} \wedge \phi(a_1, \ldots, a_\ell) = 1 \end{array}\right\}$$

  with the verifier, where the proof of $\phi(a_1, \ldots, a_\ell) = 1$ is instantiated via the technique for proofs of partial knowledge (Construction 3.59) and the protocols from Section 4.1.3. The verifier accepts, if and only if $\sigma_1' \neq 1$ *and* it accepts within the proof.

Before we give further information about the usage of the concrete construction of the ACS, we show that it is a correct credential system according to Definition 4.19.

Lemma 4.26. *Construction 4.25 is a correct anonymous credential system (Definition 4.19).*

Proof. An anonymous credential system is correct if 1) honestly generated pseudonym are valid and 2) honestly issued credentials are valid. We show these two properties separately.

Let us start by showing that honestly generated pseudonyms are valid. More formally, we fix arbitrary $n \in \mathbb{N}$, $(\mathsf{pp}, \cdot, \cdot) \in [\mathsf{Setup}(1^n)]$, $\mathsf{usk} \in [\mathsf{U.Init}(\mathsf{pp})]$ and $(\mathsf{nym}, \mathsf{psk}) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})]$. Assume the user and the verifier act honestly in $(\mathsf{ProveCred}, \mathsf{VrfyCred})$. We show that in this case it holds

$$\Pr[(\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym})) \rightarrow 1]] = 1 - \mu(|\mathsf{pp}|)$$

for some negligible function $\mu(\cdot)$. Since $(\mathsf{nym}, \mathsf{psk}) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})]$, it holds $\mathsf{nym} = g_{\text{nym}}^r h_{\text{nym}}^{\mathsf{usk}}$ by definition of the Pedersen commitment scheme. This together with the assumption

that the user and verifier act honestly, we get that the zero-knowledge argument of knowledge carried out in $(\mathsf{ProveCred}, \mathsf{VrfyCred})$ will be accepted by the verifier with probability 1 implying

$$\Pr[(\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym})) \to 1]] = 1.$$

It remains to show that honestly issued credentials are valid. Fix arbitrary $n, \ell \in \mathbb{N}$, $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \in [\mathsf{Setup}(1^n)]$, $\mathsf{usk} \in [\mathsf{U.Init}(\mathsf{pp})]$, $(\mathsf{ipk}, \mathsf{isk}) \in [\mathsf{I.Init}(\mathsf{pp}, 1^\ell)]$, $(\mathsf{nym}, \mathsf{psk}) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})]$, $(\mathsf{nym}', \mathsf{psk}') \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})]$, $(a_1, \ldots, a_\ell) \in \mathcal{U}_A^\ell$, $\phi \in \mathcal{U}_\Phi$ with $\phi(a_1, \ldots, a_\ell) = 1$ and a credential $\mathsf{cred} = (\sigma, (a_1, \ldots, a_\ell), \mathsf{ipk})$ output by $\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^\ell, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^\ell, \mathsf{isk})$. We need to show that

$$\Pr[(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}', \phi, \mathsf{usk}, \mathsf{psk}', \mathsf{cred}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}', \phi)) \to 1] = 1 - \mu(|\mathsf{pp}|)$$

for some negligible function $\mu(\cdot)$. Since cred was output by running $(\mathsf{RcvCred}, \mathsf{IssCred})$ and Construction 4.4 is a correct scheme for signing partially committed values, we have that σ contained in cred is a valid PS-signature of $(\mathsf{usk}, a_1, \ldots, a_\ell)$ under ipk. Moreover, by assumption nym and nym' are pseudonyms for the same usk, and $\phi(a_1, \ldots, a_\ell) = 1$ holds. The proof used in the protocol

$$\mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}', \phi, \mathsf{usk}, \mathsf{psk}', \mathsf{cred}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}', \phi)$$

is an AND-composition of three Σ-protocols, which yields a Σ-protocol as well. A Σ-protocol always fulfills the property of completeness. As mentioned above cred contains a valid signature, nym' is a pseudonym for usk and ϕ is satisfied by the given attributes. By the completeness of the used Σ-protocol, it is implied that

$$\Pr[(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}', \phi, \mathsf{usk}, \mathsf{psk}', \mathsf{cred}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}', \phi)) \to 1] = 1.$$

□

Thus, we have the following usage of the system. The Setup algorithm should be run by a trusted party and generates public parameters for both, Pointcheval-Sanders signatures used for credentials, and Pedersen commitments used for pseudonyms. Users run $\mathsf{U.Init}$ to obtain a user secret, chosen uniformly at random from $\mathbb{Z}_p$. They can create pseudonyms with $\mathsf{CreateNym}$, which simply outputs a Pedersen commitment on usk and the corresponding open value called pseudonym secret psk. To prove some pseudonym belongs to a user, $(\mathsf{ProveNym}, \mathsf{VrfyNym})$ is an interactive zero-knowledge proof of knowledge of opening the commitment. Issuers, which want to issue credentials on ℓ attributes, receive a Pointcheval-Sanders key pair for $\ell+1$ messages from $\mathsf{I.Init}$. They need to be able to sign $\ell+1$ messages, since the user secret is part of the credential, alongside the ℓ attributes. Credentials are obtained in $(\mathsf{RcvCred}, \mathsf{IssCred})$ via the protocol for signing partially committed values, such that in any case the user secret is perfectly hidden in the sent commitment C (by running $\mathsf{BlindInit}$ with $\{0\}$). The protocol $(\mathsf{ProveCred}, \mathsf{VrfyCred})$ is a zero-knowledge proof of knowledge of a signature, such that the user secret matches the pseudonym and the signed attributes fulfill the given predicate ϕ.

4.2.4 Security Proofs

In the following, let G be a type 3 bilinear group generator (Definition 3.2), let $\Pi_{\text{com}} = (\Pi_{\text{com}}.\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be the Pedersen commitment scheme (Construction 3.23) and let $\Pi_{\text{sign}} = (\Pi_{\text{sign}}.\mathsf{Setup}, \mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be the Pointcheval-Sanders signature scheme (Construction 3.17). Let $\Pi = (\mathsf{Setup}, \mathsf{U.Init}, \mathsf{I.Init}, \mathsf{CreateNym}, (\mathsf{ProveNym}, \mathsf{VrfyNym}), (\mathsf{RcvCred}, \mathsf{IssCred}), (\mathsf{ProveCred}, \mathsf{VrfyCred}))$ be the credential system from Construction 4.25 using Π_{com} and Π_{sign}.

4.2.4.1 Anonymity

In the following proofs, we use an implication of the zero-knowledge property; namely that the transcripts generated by such proofs are distributed independently of the witness. In the literature, this property is called *witness indistinguishability* and was introduced by Feige and Shamir [FS90a]. For convenience, we state this implication in the next lemma and prove it before going over to show the anonymity of the credential system.

Lemma 4.27. *Let $(\mathcal{P}, \mathcal{V})$ be a zero-knowledge proof (Definition 3.39) for some language L_R of an NP-Relation R. Then, it holds for every verifier $\mathcal{V}^*$ that for all $(x, w) \in R$ and all $y \in \{0,1\}^{p(|v|)}$, for some polynomial $p(\cdot)$, the transcripts, $T(\mathcal{P}(v, w) \leftrightarrow \mathcal{V}^*(v, y))$, are distributed independently of w.*

Proof. Fix an arbitrary interactive algorithm $\mathcal{V}^*$ and arbitrary v, w, w', y such that $(v, w), (v, w') \in R$ and $y \in \{0,1\}^{p(|v|)}$, for some $p(\cdot)$. Since $(\mathcal{P}, \mathcal{V})$ is a zero-knowledge proof, there exists a simulator $\mathcal{S}$ for verifier $\mathcal{V}^*$ that on input (v, y) outputs accepting transcripts with the same distribution as $T(\mathcal{P}(v, w) \leftrightarrow \mathcal{V}^*(v, y))$ regardless of the behavior of $\mathcal{V}^*$. Moreover, $\mathcal{S}$ outputs, on input (v, y), accepting transcripts with the same distribution as $T(\mathcal{P}(v, w') \leftrightarrow \mathcal{V}^*(v, y))$. This implies that the random variables $T(\mathcal{P}(v, w) \leftrightarrow \mathcal{V}^*(v, y))$ and $T(\mathcal{P}(v, w') \leftrightarrow \mathcal{V}^*(v, y))$ are distributed identically. Hence, the transcripts $T(\mathcal{P}(v, w) \leftrightarrow \mathcal{V}^*(v, y))$ are distributed independently of the witness. □

Let us proceed by showing that our basic credential system given in Construction 4.25 is anonymous according to Definition 4.20. To this end, we show that each of the properties stated in Definition 4.20 holds for Construction 4.25 in separate lemmas. Subsequently, we conclude the anonymity in Theorem 4.32 based on these lemmas.

Lemma 4.28. $\Pi.\mathsf{CreateNym}$ *hides the user secret (Definition 4.20.1).*

Proof. Let n be a security parameter. Let $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{nym}) \in [\mathsf{Setup}(1^n)]$. Let $\mathsf{usk} \in [\mathsf{U.Init}(\mathsf{pp})]$. Since $\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})$ is essentially just a Pedersen commitment with public parameters pp_{nym} on message usk, we only need to look at the properties of the commitment scheme. We have that pp_{nym} is generated the same way as in $\Pi_{\text{com}}.\mathsf{Setup}$, and usk and usk' are in the message space of Π_{com}, thus, by the perfect hiding property of the Pedersen commitment we have that the distributions of commitments on two different messages are the same. In particular, we have that the distributions of

$$\text{output}_{\mathcal{A}}[\mathsf{nym} \leftarrow \mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}) : \mathcal{A}(\mathsf{pp}, \mathsf{nym})] \qquad \text{and}$$
$$\text{output}_{\mathcal{A}}[\mathsf{nym} \leftarrow \mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}') : \mathcal{A}(\mathsf{pp}, \mathsf{nym})]$$

are exactly the same for all unrestricted adversaries $\mathcal{A}$, all $n \in \mathbb{N}$ and all $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \in [\mathsf{Setup}(1^n)]$. □

Lemma 4.29. $\Pi.\mathsf{ProveNym}$ *hides the user secret (Definition 4.20.2).*

Proof. The only messages that are exchanged between an adversary and $\mathsf{ProveNym}$ are that of an interactive zero-knowledge argument of knowledge, thus it only remains to have look at this argument. By Lemma 4.27 we get that these exchanged messages are independent of the witness, meaning user secret usk and pseudonym secret psk. Hence, we have that the distributions

$$\text{output}_{\mathcal{A}}[\mathcal{A}(\mathsf{pp}, \mathsf{nym}) \leftrightarrow \mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk})] \qquad \text{and}$$
$$\text{output}_{\mathcal{A}}[\mathcal{A}(\mathsf{pp}, \mathsf{nym}) \leftrightarrow \mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}', \mathsf{psk}')]$$

are the same for all unrestricted $\mathcal{A}$, for all $n \in \mathbb{N}$, $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \in [\mathsf{Setup}(1^n)]$, all $\mathsf{usk}, \mathsf{usk}' \in [\mathsf{U.Init}(\mathsf{pp})]$ and all $((\mathsf{nym}, \mathsf{psk}), (\mathsf{nym}, \mathsf{psk}')) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})] \times [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}')]$. □

Lemma 4.30. *A user stays anonymous when executing* $\Pi.\mathsf{RcvCred}$ *(Definition 4.20.3).*

Proof. $\mathsf{RcvCred}$ consists of two stages, where first a commitment C on usk is announced to the interacting party, and second a zero-knowledge argument of knowledge over opening C and the commonly known pseudonym nym to the same usk is performed. We show, that all messages sent by $\mathsf{RcvCred}$ are independent of user secret usk and pseudonym secret $\mathsf{psk} = (r_1, \mathsf{usk})$.

By Lemma 4.6, which states that the constructed scheme for signing partially committed values is secure for the user, we know that C reveals no information on the private usk and its open value r_2 chosen in $\mathsf{RcvCred}$. This holds, since on a valid ipk from Pointcheval-Sanders signature scheme, $\mathsf{BlindInit}$ outputs an environment for *hiding* generalized Pedersen commitments on $|S|$ values. In the particular case, the set S is statically set to be $\{0\}$, ensuring that usk is always hidden in C.

For the second part, Lemma 4.27 gives us that messages sent within the zero-knowledge argument of knowledge $\mathsf{PK}\Big\{(\mathsf{usk}, r_1, r_2) : \mathsf{nym} = g_{\mathrm{nym}}^{r_1} h_{\mathrm{nym}}^{\mathsf{usk}} \wedge C = g^{r_2} Y_0^{\mathsf{usk}}\Big\}$ are independent of the witnesses (usk, r_1, r_2). Hence all messages exchanged are independent of psk and usk, thus the distributions

$$\mathrm{output}_{\mathcal{A}}[\mathcal{A}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}) \leftrightarrow \mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk})]$$
$$\mathrm{output}_{\mathcal{A}}[\mathcal{A}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}) \leftrightarrow \mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}', \mathsf{psk}')]$$

are identical for all unrestricted adversaries $\mathcal{A}$, all $n, \ell \in \mathbb{N}$, all $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \in [\mathsf{Setup}(1^n)]$, all $\mathsf{usk}, \mathsf{usk}' \in [\mathsf{U.Init}(\mathsf{pp})]$, all $((\mathsf{nym}, \mathsf{psk}), (\mathsf{nym}, \mathsf{psk}')) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})] \times [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}')]$, all ipk and all $(a_i)_{i=1}^{\ell} \in \mathcal{U}_A^{\ell}$. ☐

Lemma 4.31. *A user stays anonymous when executing* $\Pi.\mathsf{ProveCred}$ *(Definition 4.20.4).*

Proof. To show users executing $\mathsf{ProveCred}$ stay anonymous, we essentially prove that messages sent by the algorithm, are independent of its private inputs cred, usk and psk. For that, we fix an adversary.

As demanded in Definition 4.20.4, $\mathsf{cred} \neq \perp$ is any output of $\mathsf{RcvCred}$ run on attributes $(a_1, \ldots, a_\ell)$, user secret usk from $\mathsf{U.Init}$ and some ipk in interaction with the adversary. The last check of $\mathsf{RcvCred}$ in Construction 4.25 ensures, that for the stated inputs $\mathsf{cred} \neq \perp$ only holds if cred is of form $(\sigma, (a_i)_{i=1}^{\ell}, \mathsf{ipk})$ with $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{ipk}, (\mathsf{usk}, a_1, \ldots, a_\ell), \sigma) = 1$. The latter is independent of any interaction with $\mathcal{A}$.

Further, if $\phi((a_i)_{i=1}^{\ell}) = 0$ holds, $\mathsf{RcvCred}$ stops without any output. In this case, nothing, but the attributes not satisfying the predicate, is revealed. In the other case, the attributes fulfill ϕ, and $\mathsf{RcvCred}$ has an opening value for the pseudonym and a valid Pointcheval-Sanders signature on the attributes. The Pedersen commitments on the attribute values are perfectly hiding and thus independent of the private inputs. Then $(\mathsf{ProveCred}, \mathsf{VrfyCred})$ is the Σ-protocol from Construction 4.8 for proving knowledge of a signature, AND-composed with zero-knowledge arguments for satisfying ϕ with the signed attributes and opening nym and the Pedersen commitments. This composition is a Σ-protocol again and is instantiated as zero-knowledge argument of knowledge in the ACS using the technique of proofs of partial knowledge. Lemma 4.27 yields, that the messages exchanged are independent of the witnesses $((\sigma_1, \sigma_2), (a_1, \ldots, a_\ell), (d, \mathsf{usk}))$. The exchanged messages do still depend on whether ϕ is satisfied or not. However any two attribute tuples from $\mathcal{U}_A^{\ell}$, with the same outcome evaluated by ϕ, cannot be distinguished.

Concluding, the distributions

$$\mathrm{output}_{\mathcal{A}}[\mathcal{A}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi) \leftrightarrow \mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred})]$$
$$\mathrm{output}_{\mathcal{A}}[\mathcal{A}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi) \leftrightarrow \mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}', \mathsf{psk}', \mathsf{cred}')]$$

are identical for all unrestricted $\mathcal{A}$, all $n, \ell \in \mathbb{N}$, $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \in [\mathsf{Setup}(1^n)]$, all $\mathsf{usk}, \mathsf{usk}' \in [\mathsf{U.Init}(\mathsf{pp})]$, all $((\mathsf{nym}, \mathsf{psk}), (\mathsf{nym}, \mathsf{psk}')) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})] \times [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}')]$, all ipk,

all $\phi \in \mathcal{U}_\Phi$, all $(a_i)_{i=1}^{\ell}, (a'_i)_{i=1}^{\ell} \in \mathcal{U}_A^{\ell}$ with $\phi((a_i)_{i=1}^{\ell}) = \phi((a'_i)_{i=1}^{\ell})$, all (unrestricted) adversaries $\mathcal{A}$', all $\mathsf{cred} \neq \perp$ output by $\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathcal{A}'((a_i)_{i=1}^{\ell})$, and all $\mathsf{cred}' \neq \perp$ output by $\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a'_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathcal{A}'((a'_i)_{i=1}^{\ell})$.

□

Consequently, Lemmas 4.28 to 4.31 imply the following statement:

Theorem 4.32. *The credential system given in Construction 4.25 is anonymous regarding Definition 4.20.*

4.2.4.2 Soundness

In the following section, we will prove the soundness of our Construction 4.25 according to Definition 4.24. We will do that by giving individual proofs for non-impersonation (Definition 4.22) and credential unforgeability (Definition 4.23). Subsequently, we conclude the overall soundness in Theorem 4.36.

For the soundness proofs, we often make use of an extractor of an argument of knowledge when constructing an adversary. Since the extractor only has expected polynomial runtime in the security parameter, an adversary using it would also have expected polynomial runtime. Due to our security models requiring ppt adversaries, this does not suffice. Thus we introduce the following lemma to transform an adversary with expected polynomial runtime into an adversary with probabilistic polynomial time but marginally worse success probability.

Lemma 4.33. *For a probabilistic algorithm $\mathcal{A}$ with expected polynomial runtime, there exists a ppt algorithm $\mathcal{A}'$ with* $\Pr[x \leftarrow \mathcal{A}' : x = X] \geq \frac{1}{2}\Pr[x \leftarrow \mathcal{A} : x = X]$ *for all $X \neq \perp$, where $\perp$ is some error symbol.*

Proof. Let RT be the random variable describing the runtime of $\mathcal{A}$. Then, construct $\mathcal{A}'$ to simulate $\mathcal{A}$ on the same input up to a maximum number of $2 \cdot \mathbb{E}[\mathsf{RT}]$ steps. $\mathcal{A}'$ outputs whatever $\mathcal{A}$ outputs. If the runtime bound is reached $\mathcal{A}'$ instead outputs an error symbol $\perp$. Furthermore, for every output $X \neq \perp$ of $\mathcal{A}$, we have

$$\begin{aligned}
\Pr[x \leftarrow \mathcal{A}' : x = X] &= \Pr[x \leftarrow \mathcal{A} : x = X \wedge \mathsf{RT} < 2 \cdot \mathbb{E}[\mathsf{RT}]] \\
&= \Pr[x \leftarrow \mathcal{A} : \mathsf{RT} < 2 \cdot \mathbb{E}[\mathsf{RT}] \mid x = X] \cdot \Pr[x \leftarrow \mathcal{A} : x = X] \\
&= (1 - \Pr[x \leftarrow \mathcal{A} : \mathsf{RT} \geq 2 \cdot \mathbb{E}[\mathsf{RT}] \mid x = X]) \cdot \Pr[x \leftarrow \mathcal{A} : x = X] \\
&\overset{\text{Markov}}{\geq} \frac{1}{2} \Pr[x \leftarrow \mathcal{A} : x = X]
\end{aligned}$$

□

Lemma 4.34. *If the generalized Pedersen commitment scheme is computationally binding (Lemma 3.27), then the credential system Π from Construction 4.25 offers non-impersonation (Definition 4.22).*

Proof. Let $\mathcal{A}_{cred}$ be an adversary in the non-impersonation game against Π. Let E be an extractor for the argument of knowledge in $(\mathsf{ProveNym}, \mathsf{VrfyNym})$ of Π. Construct an extractor E' that works the same as E, but, in parallel, chooses some usk' and exhaustively searches for some psk' such that $\mathsf{Open}(\mathsf{pp}_C, \mathsf{nym}, \mathsf{psk}') = \mathsf{usk}'$ holds. E' outputs the witness of whichever process finds one first. Construct an adversary $\mathcal{A}_C$ against the computational binding property of Π_{com}:

Adversary $\mathcal{A}_C(\mathsf{pp}_C)$

1 : Perform $\mathsf{Exp}^{Soundness}_{\Pi,\mathcal{A}_{cred}}(n)$ by taking the role of the challenger and simulating $\mathcal{A}_{cred}$ with the following exceptions:

- After $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{nym})$ is generated, replace it with $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_C)$.
- When $\mathcal{A}_{cred}$ queries *runCreateNym*, also store the psk in S_{nym}.

Receive $(\mathsf{pp}, \mathsf{nym}, \mathsf{psk})$ and $(u, v) \in \hat{U} \times (I \cup U) \setminus (\hat{I} \cup \hat{U})$ from $\mathcal{A}_{cred}$.

2 : Simulate $(\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym})) \to b$ between u and v. If $u \in \bar{U}$, $\mathcal{A}_{cred}$ still controls u.

3 : If $b = 0$ or any of the conditions in the real experiment do not hold, abort.

4 : Run $E'(\mathsf{pp}, \mathsf{nym})$ with black-box access to $\mathcal{A}_{cred}$ in the argument of knowledge of $\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym})$ between u and v.

5 : The extractor outputs some (usk_2, d_2).

6 : Search for a $(\mathsf{nym}, u', \mathsf{psk}_1)$ with $u' \notin \hat{U}$ in S_{nym} and let usk_1 be the secret key corresponding to u'.

7 : Output $(\mathsf{nym}, (\mathsf{usk}_1, \mathsf{psk}_1), (\mathsf{usk}_2, d_2))$.

Then, we have

$$\Pr[Binding_{\Pi_{com},\mathcal{A}_C}(n) = 1] = \Pr[\mathsf{Game}^{CredForge}_{\Pi,\mathcal{A}_{cred}}(n) = 1 \land (\mathsf{usk}_1, \mathsf{psk}_1) \neq (\mathsf{usk}_2, d_2)],$$

due to the following reasons:

- The view of $\mathcal{A}_{cred}$ is perfectly simulated.
- If $\mathcal{A}_C$ passes step 3, then there exists an honest user with the pseudonym output by $\mathcal{A}_{cred}$. Thus, $\mathcal{A}_C$ can find a $(\mathsf{nym}, u', \mathsf{psk}_1)$ with $u' \notin \hat{U}$ in S_{nym}.
- $\mathsf{Open}(\mathsf{pp}, \mathsf{nym}, \mathsf{psk}_1) = \mathsf{usk}_1 \neq \perp$, since it was honestly generated.
- $\mathsf{Open}(\mathsf{pp}, \mathsf{nym}, \mathsf{psk}_2) = \mathsf{usk}_2 \neq \perp$, since it was output by the extractor.

Since Π_{com} is perfectly hiding, nym is independent from usk_1. Furthermore, there exist exactly p different combinations of (a, b) such that $\mathsf{Open}(\mathsf{nym}, b) = a$. Thus, we have that $\Pr[(\mathsf{usk}_1, \mathsf{psk}_1) = (\mathsf{usk}_2, \mathsf{psk}_2)] = \frac{1}{p}$ and that this event is independent from $\mathcal{A}_{cred}$'s action. Therefore, we have

$$\Pr[Binding_{\Pi_{com},\mathcal{A}_C}(n) = 1] = \frac{p-1}{p} \cdot \Pr[\mathsf{Game}^{CredForge}_{\Pi,\mathcal{A}_{cred}}(n) = 1].$$

What is left to prove is that $\mathcal{A}_C$ has expected polynomial runtime. Then we can use Lemma 4.33 to receive a ppt adversary $\mathcal{A}'$ which is marginally worse. Let RT be the random variable that describes the runtime of $\mathcal{A}_C$. Let $run(\cdot)$ be a polynomial bounding the runtime of $\mathcal{A}_C$ except for the extracting step. Let $\epsilon(n, v)$ be the probability of u convincing v in step 2. Assume that the randomness r for the proof is fixed. Thus, the announcement the adversary outputs is fixed, and, after the challenge is chosen by the honest verifier, the response is also fixed. Therefore, $\epsilon(n, v)$ corresponds to the number of challenges the adversary can answer divided by the size of the challenge space, i.e. we have $\epsilon(n, v) = i \cdot \kappa(n)$ for some $i \in \mathbb{N}_0$, where $\kappa(\cdot)$ is the knowledge error of the argument of knowledge. If $\epsilon(n, v) > \kappa(n)$ and thus $i \geq 2$, we have the following:

$$\begin{aligned}\mathbb{E}[\mathsf{RT}|i \geq 2] =& run(n) + \epsilon(n, v) \cdot \mathbb{E}[\mathsf{RT}_E|r] + (1 - \epsilon(n, v)) \cdot 0 \\ =& r(n) + \epsilon(n, v) \cdot \frac{p(|v|)}{\epsilon(n, v) - \kappa(n)}\end{aligned}$$

$$=r(n) + \frac{i \cdot \kappa(n) \cdot p(|v|)}{(i-1) \cdot \kappa(n)}$$
$$=r(n) + \frac{i \cdot p(|v|)}{i-1} \leq 2 \cdot p(|v|)$$

If $\epsilon(n, v) = \kappa(n)$, then E never finds two pairs of the same randomness and same challenge for which the verifier accepts. Instead, E' outputs a witness due to the parallel process after a running time of $\mathcal{O}(p)$. But, since we have this extractor runtime only with probability $\kappa(n) = \frac{1}{p}$ and else an extractor runtime of 0, the expected runtime of the extractor is $\mathbb{E}[\mathsf{RT}_E] = \frac{1}{p} \cdot \mathcal{O}(p) = \mathcal{O}(1)$. If $\epsilon(n, v) = 0$, then the extractor has an expected runtime of 0, since it will never be used. Therefore, $\mathbb{E}[\mathsf{RT}]$ is bounded by a polynomial in expectation.

Then, by Lemma 4.33, there exists an adversary $\mathcal{A}'_C$ with $\Pr[x \leftarrow \mathcal{A}'_C : X = x] \geq \frac{1}{2} \cdot \Pr[x \leftarrow \mathcal{A}_C : X = x]$, for an $X \neq \perp$ output by $\mathcal{A}_C$. Therefore, we have

$$\Pr[Binding_{\Pi_{com}, \mathcal{A}'_C}(n) = 1] \geq \frac{p-1}{2p} \cdot \Pr[\mathsf{Game}^{CredForge}_{\Pi, \mathcal{A}_{cred}}(n) = 1].$$

Thus, if $\Pr[Binding_{\Pi_{com}, \mathcal{A}'_C}(n) = 1]$ is negligible, $\Pr[\mathsf{Game}^{NonImp}_{\Pi, \mathcal{A}_{cred}}(n) = 1]$ is also negligible. □

Lemma 4.35. *If the multi-message Pointcheval-Sanders signature scheme is existentially unforgeable (Theorem 3.19),* Π *offers credential unforgeability (Definition 4.23).*

To proof the credential unforgeability, we show that given an existing adversary $\mathcal{A}_{cred}$ that can forge a credential with high probability, we can construct an adversary $\mathcal{A}_{sign}$ who can forge a signature with high probability.

Proof. If the Pointcheval-Sanders signature scheme is existentially unforgeable, $(\mathsf{BlindInit}, (\mathsf{BlindRcv}, \mathsf{BlindIssue}))$ from Construction 4.4 is secure for the signer by Lemma 4.7. Thus, there exists a simulator $\mathcal{S}'$ for $\mathsf{BlindIssue}$. From that we want to construct a simulator $\mathcal{S}$ for $\mathsf{IssCred}$: $\mathcal{S}$ on input $(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell})$ receives a commitment C from the receiver and then partakes in the argument of knowledge as $\mathsf{IssCred}$ does. If the argument of knowledge succeeds, $\mathcal{S}$ extracts $(\mathsf{usk}, \mathsf{psk})$ from it by using the associated extractor with oracle access to $\mathcal{A}_{cred}$ in the argument of knowledge. In parallel, the extractor chooses some usk' and searches exhaustively for some psk' and r' that form a witness for the proof. Whichever process finds a witness first, returns it. After that, $\mathcal{S}$ parses pp to $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{nym})$ and simulates $\mathcal{S}'((p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e), \mathsf{ipk}, C, \{0\}, (a_i)_{i=1}^{\ell}, \mathsf{psk})$ to receive a blind signature σ. After that, $\mathcal{S}$ sends σ to u. This simulates $\mathsf{BlindIssue}$ perfectly, since the argument of knowledge is done the same way and the blinded signature is generated by the simulator for blind issue on correct input.

Let E be an extractor for the argument of knowledge in $(\mathsf{ProveCred}, \mathsf{VrfyCred})$ of Π. Construct an extractor E' that does the same as E, but does the following in parallel: It computes some $a_1, .., a_\ell$ such that $\phi(a_1, .., a_\ell) = 1$. For each C_i it exhaustively searches for some d_i such that $C_i = g_{nym}^{d_i} h_{nym}^{a_i}$. Then, it uniformly chooses some usk and exhaustively searches for some d such that $\mathsf{nym} = g_{nym}^{d} h_{nym}^{\mathsf{usk}}$. After that, it exhaustively searches for some r, such that the last equation of the argument of knowledge is fulfilled. E' returns the witness of the process that finished first. Let $\mathcal{A}_{cred}$ be an adversary in the credential-unforgeability game against Π. Construct an adversary $\mathcal{A}_{sign}$ against the multi-message Pointcheval-Sanders signature scheme as follows:

$\mathcal{A}_{sign}^{\mathcal{O}}(\mathsf{pp}_{ps}, \mathsf{pk}_{ps})$

1 : Create an initially empty multi-set R.

2 : Perform $\mathsf{Exp}_{\Pi,\mathcal{A}_{cred}}^{Soundness}(n)$ by taking the role of the challenger and simulating $\mathcal{A}_{cred}$ with the following exceptions:

- After $\mathcal{A}_{cred}$ chooses n_I, choose $j \leftarrow \{1, ..., n_I\}$.
- After $\mathsf{pp} = (\cdot, \mathsf{pp}_{nym})$ is generated, replace it with $\mathsf{pp} = (\mathsf{pp}_{ps}, \mathsf{pp}_{nym})$.
- After running I.Init for each issuer, replace the ipk of the j-th issuer by pk_{ps}.

Answer all oracles as specified in the experiment except for the following:

- $corruptIssuer(i)$: if $i = j$, abort the experiment. Else do the same as in the real experiment.
- $runRcvIssCred(u, i, \mathsf{nym}, (a_i)_{i=1}^{\ell})$:
  - if $i = j$ and $u \in \hat{U}$, execute $\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathcal{S}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell})$, where the parameters are of the corresponding u and i. If the simulator queries for the signing of an m, answer only the first query with $\mathcal{O}(m)$. If u is corrupted, $\mathcal{A}_{cred}$ still has control over it. If the protocol is successful, return 1 to $\mathcal{A}_{cred}$, else 0.
  - if $i = j$ and $u \notin \hat{U}$, do not generate any protocol messages. Instead, add $(u, \mathsf{nym}, (a_i)_{i=1}^{\ell})$ to R.
  - else, answer the query as in the real experiment.
- $corruptUser(u)$: if $u \notin \bar{U}$, do the following: for each $(u, \mathsf{nym}, (a_i)_{i=1}^{\ell}) \in R$ execute $\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathcal{S}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell})$ as in *runRcvIssCred*. After that, do the same things $corruptUser(u)$ does in the normal experiment.

3 : $\mathcal{A}_{cred}$ outputs $(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred})$, $u \in \hat{U}$ and $v \in I \cup U \setminus (\hat{I} \cup \hat{U})$.

4 : Run $(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi)) \rightarrow b$ between u and v, while $\mathcal{A}_{cred}$ controls u. If $b = 0$ or any of the conditions in the real experiment do not hold, abort.

5 : Run $E'(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi)$ with black-box access to $\mathcal{A}_{cred}$ in the argument of knowledge of $\mathsf{ProveCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi)$ between u and v (the adversary controls u).

6 : The extractor outputs some $w = (\mathsf{usk}', \mathsf{psk}', \mathsf{cred}')$.

7 : Parse $cred'$ to $(\sigma, (a_1, ..., a_\ell), \mathsf{ipk}')$.

8 : Let j' be the issuer corresponding to ipk' in cred'.

9 : If $j' = j$, output $((a_1, ..., a_\ell), \sigma)$.

Then, we have

$$\Pr[\mathsf{Sig\text{-}forge}_{\mathsf{PS},\mathcal{A}_{sign}}(n) = 1] = \Pr[\mathsf{Game}_{\Pi,\mathcal{A}_{cred}}^{CredForge}(n) = 1 \wedge j' = j]$$

due to the following reasons:

- $\mathcal{A}_{sign}$ only outputs a signature if $\mathcal{A}_{cred}$ outputs parameters so that u proves knowledge of the credential and $j' = j$.
- The view of $\mathcal{A}_{cred}$ is perfectly simulated, unless $\mathcal{A}_{cred}$ asks for $corruptIssuer(i)$ with $i = j$, but then $\mathcal{A}_{cred}$ would lose anyways.
- The extractor always outputs some valid credential, unless it was aborted due to a too high runtime.
- The signing oracle is only used iff $\mathcal{A}_{cred}$ queries *runRcvIssCred* or he corrupts a user that received a credential from j. Then, if $\mathcal{A}_{sign}$ loses due to having queried for a signature he outputs, then $\mathcal{A}_{cred}$ loses due to having queried for a *cred* that he later outputs.

Since $\mathcal{A}_{cred}$ is a ppt, n_I is $p_I(n)$ for some polynomial $p_I(\cdot)$. Furthermore, the choice of j is independent from the rest. Thus, we have that

$$\Pr[\mathsf{Sig\text{-}forge}_{\mathsf{PS},\mathcal{A}_{sign}}(n) = 1] = \frac{1}{p_I(n)} \cdot \Pr[\mathsf{Game}^{CredForge}_{\Pi,\mathcal{A}_{cred}}(n) = 1].$$

What is left to prove is that $\mathcal{A}_{sign}$ has expected polynomial runtime. Then, we can use *Lemma* 4.33 to obtain a ppt adversary $\mathcal{A}'_{sign}$ with marginally worse success probability. Since $\mathcal{A}_{sign}$ obviously has polynomial runtime when ignoring the extractors, we only look at the runtimes of the latter. For both, it holds that their parallel process has a runtime of $\mathcal{O}(p)$, since they exhaustively search for some witness, else they have polynomial runtime. But, similar to the proof of Lemma 4.34, the extractors are used only with probability $\frac{1}{p}$, thus their expected runtime is polynomial. Thus, by Lemma 4.33 there exists an adversary $\mathcal{A}'_{sign}$ with $\Pr[x \leftarrow \mathcal{A}'_{sign} : X = x] \geq \frac{1}{2} \cdot \Pr[x \leftarrow \mathcal{A}_{sign} : X = x]$ for some output $X \neq \perp$ of $\mathcal{A}_{sign}$. Thus, we have that

$$\Pr[\mathsf{Sig\text{-}forge}_{\mathsf{PS},\mathcal{A}'_{sign}}(n) = 1] \geq \frac{1}{2 \cdot p_I(n)} \cdot \Pr[\mathsf{Game}^{CredForge}_{\Pi,\mathcal{A}_{cred}}(n) = 1]$$

and furthermore that if $\Pr[Game^{CredForge}_{\Pi,\mathcal{A}_{cred}}(n) = 1]$ is non-negligible, then $\Pr[\mathsf{Sig\text{-}forge}_{\mathsf{PS},\mathcal{A}'_{sign}}(n) = 1]$ is also non-negligible. □

From the two lemmas above we get the following theorem:

Theorem 4.36. *The ACS from Construction 4.25 is* sound *in regards to Definition 4.24.*

4.3 Extended Anonymous Credential System

We extend the basic ACS from Construction 4.25 by features like traceability of users by a trusted system manager. Therefore we first provide an adapted definition for an extended ACS, modeling these features. Next, we adjust the demanded security notions from Section 4.2.2 accordingly. We provide a new construction, which builds upon of the basic ACS. Finally we, prove the extended ACS matches the updated security notions.

4.3.1 Definition

To achieve the extended feature traceability, we need to adapt the definition of the basic ACS (Definition 4.19) accordingly. Users have to be able to join a system in which they receive registration information. This information can later be opened to the corresponding user with help of an *open secret key*. We therefore generate a key pair for opening and a registry using MGen, which are used when a user joins the system by executing the interactive protocol $(\mathsf{Join}, \mathsf{MJoin})$. After a successful execution of this protocol, the registry gets updated to include the newly joined user and the user receives the registration information $\mathsf{reginfo}$ which is included when proving the possession of a credential over some attributes. Moreover, the algorithm Open is introduced to actually open a transcript to a public key of a user. In total, this leads to the following definition:

Definition 4.37. An *extended credential system* consists of the following (ppt) algorithms and interactive protocols:

- $\mathsf{Setup}(1^n)$: On input security parameter 1^n, it outputs public parameters pp with $|\mathsf{pp}| \geq n$, an attribute universe $\mathcal{U}_A$ and a predicate universe $\mathcal{U}_\Phi \subseteq \{\phi \mid \phi : \mathcal{U}_A^k \to \{0,1\}, k \in \mathbb{N}\}$. Additionally, it (implicitly) outputs application-dependent auxiliary parameters $\mathsf{pp}_{\mathrm{aux}}$.
- $\mathsf{MGen}(\mathsf{pp})$: On input public parameters pp, it outputs a tuple $(\mathsf{osk}, \mathsf{opk}, \mathsf{reg})$ of open secret key, open public key and a user registry.

- $\mathsf{U.Init}(\mathsf{pp}, \mathsf{opk})$: On input public parameters pp and open public key, it outputs a user secret key usk and a user public key upk.
- $\mathsf{I.Init}(\mathsf{pp}, 1^\ell)$: On input public parameters pp and 1^ℓ with $\ell \in \mathbb{N}$, it outputs an issuer public key ipk and secret key isk. The number ℓ denotes the number of attributes supported by the issuer.
- $\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})$: On input public parameters pp and user secret usk, it outputs a pseudonym nym and corresponding pseudonym secret psk.
- $(\mathsf{Join}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{usk}), \mathsf{MJoin}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{osk}, \mathsf{reg}))$ is an interactive protocol on common inputs public parameters pp, open public key opk, user public key upk. Secret inputs are user secret key usk for the user and open secret key osk and user registry reg for the trusted party. At the end of the interaction, MJoin either outputs an error symbol $\perp$ or the updated reg'. Join outputs a $\mathsf{reginfo}$ containing registration information.
- $(\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}), \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym}))$ is an interactive protocol on common inputs public parameters pp and pseudonym nym, where the user has user secret usk and pseudonym secret psk as private input. After execution the verifier outputs a bit $b \in \{0, 1\}$ and we interpret 1 as accepting, 0 as denying.
- $(\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^\ell, \mathsf{usk}, \mathsf{psk}), \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^\ell, \mathsf{isk}))$ is an interactive protocol on common inputs public parameters pp, pseudonym nym and attributes $(a_1, \ldots, a_\ell) \in \mathcal{U}_A^\ell$. The user has user secret usk, pseudonym secret psk and issuer public key ipk as additional input, whereas the issuer has secret key isk. After the interaction, the user outputs (locally) either the credential cred corresponding to ipk over her user secret usk and the attributes $(a_1, \ldots, a_\ell)$ or a failure symbol represented by $\perp$.
- $(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}, \mathsf{reginfo}), \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi))$ is an interactive protocol on common inputs public parameters pp, open public key opk, issuer public key ipk, pseudonym nym, and a predicate $\phi \in \mathcal{U}_\Phi$, where the user has user secret usk, pseudonym secret psk, credential cred on attributes $(a_1, .., a_\ell) \in \mathcal{U}_A^\ell$ and registration information $\mathsf{reginfo}$ as private input. After execution the verifier outputs a bit $b \in \{0, 1\}$ and we interpret 1 such that the credential cred satisfies the predicate ϕ and the verifier accepts, 0 as denying.
- $\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}, t)$: On input open public key opk, open secret key osk, registry reg and a transcript t, it outputs a user public key upk or an error symbol $\perp$.

It remains to state when an extended anonymous credential system is correct. The main difference to the basic ACS is that we now have to be able to successfully open a transcript to a user public key, which is mentioned in point 3 in the following.

We say that an extended anonymous credential system is *correct* if for

all $n, \ell, c \in \mathbb{N}$,

$(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \leftarrow \mathsf{Setup}(1^n)$,

$(\mathsf{osk}, \mathsf{opk}, \mathsf{reg}) \leftarrow \mathsf{MGen}(\mathsf{pp})$,

all $(a_1, \ldots, a_\ell) \in \mathcal{U}_A^\ell$,

all $\phi \in \mathcal{U}_\Phi$ with $\phi(a_1, \ldots, a_\ell) = 1$,

$(\mathsf{ipk}, \mathsf{isk}) \leftarrow \mathsf{I.Init}(\mathsf{pp}, 1^\ell)$,

$(\mathsf{usk}_i, \mathsf{upk}_i) \leftarrow \mathsf{U.Init}(\mathsf{pp})$ for $1 \leq i \leq c$,

all $j \in \{1, \ldots, c\}$,

$(\mathsf{nym}, \mathsf{psk}), (\mathsf{nym}', \mathsf{psk}') \leftarrow \mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}_j)$,

$\mathsf{reginfo}_i \leftarrow (\mathsf{Join}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}_i, \mathsf{usk}_i) \leftrightarrow \mathsf{MJoin}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}_i, \mathsf{osk}, \mathsf{reg}_i)) \rightarrow \mathsf{reg}_{i+1}$ for $1 \leq i \leq c$,

$\mathsf{cred} \leftarrow (\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}_j, \mathsf{psk}) \leftrightarrow \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{isk}))$,

all transcripts t from $(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}', \phi, \mathsf{usk}_j, \mathsf{psk}', \mathsf{cred}, \mathsf{reginfo}_j) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}', \phi))$

1. **Honestly generated pseudonyms are valid**:

$$1 - \mu(|pp|) = \Pr[(\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}_j, \mathsf{psk}) \leftrightarrow \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym})) \rightarrow 1]$$

   holds for some negligible function μ, where the probability is taken over the coin tosses of the algorithms and their inputs' distribution. We call such a nym *valid*.

2. **Honestly issued credentials are valid**:

$$1 - \mu(|pp|) = \Pr[(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}', \phi, \mathsf{usk}_j, \mathsf{psk}', \mathsf{cred}, \mathsf{reginfo}_j) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}', \phi)) \rightarrow 1]$$

   holds for some negligible function μ, where the probability is taken over the coin tosses of the algorithms and their inputs' distribution. We call such a cred *valid*.

3. ProveCred **is traceable by the open manager**:

$$1 - \mu(|pp|) = \Pr[\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}_{c+1}, t) = \mathsf{upk}_j]$$

   holds for some negligible function μ, where the probability is taken over the coin tosses of the algorithms and their inputs' distribution.

Note that we do not necessarily have to integrate the open mechanism into the (ProveCred, VrfyCred) protocol. It is also possible that the registration information is included in the (ProveNym, VrfyNym) protocol. This would mean that this protocol has to be executed when accessing some service in order to provide a traceable transcript. However, we see the proving of a credential as the main aspect that should include the registration information and want to keep proving of a pseudonym efficient.

4.3.2 Security Notions

Since we added the traceability feature to the extended ACS, we have to adapt the security notions we introduced in Section 4.2.2. Regarding anonymity the (ProveCred, VrfyCred) protocol is the only protocol a user runs with a non-trusted party which changed compared to the basic ACS (cf. Definition 4.37). Therefore we need to redefine the corresponding part of the anonymity using an experiment. The structure of the soundness definition stays the same and will only be extended by an additional oracle and a further winning condition.

4.3.2.1 Anonymity

In addition to the already defined anonymity in the basic system, we have to take a look at the case that transcripts of user interactions can be opened to the corresponding user public key. It should be infeasible for an adversary to distinguish between two users of her choice within a (ProveCred, VrfyCred) execution. To define this property formally, we introduce a game in the following definition. An adversary can make users run the different algorithms and protocols of our extended ACS and has access to an open oracle. Since we model a security property for the user we assume that all issuers are corrupted and the adversary has full control over them.

Moreover, she can corrupt users which results in the total control over these users as well as the knowledge of the users' secret keys. Eventually, the adversary chooses two non-corrupted users and an issuer from which the challenger selects one user. The challenger then makes the selected user form a new pseudonym and lets her run ($\mathsf{ProveCred}, \mathsf{VrfyCred}$) with the chosen issuer. Given that information, the adversary wins if she is able to state which user the challenger selected. If no adversary is able to win this game with high probability, we say that our extended ACS fulfills anonymity. The formal definition looks as follows:

Definition 4.38. Define the game $\mathsf{Exp}^{anon}_{\Pi,\mathcal{A}}(n)$ of an extended ACS Π (Definition 4.37) against an adversary $\mathcal{A}$ to be the following:

- The challenger runs $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \leftarrow \mathsf{Setup}(1^n)$.
- The challenger runs $(\mathsf{osk}, \mathsf{opk}, \mathsf{reg}) \leftarrow \mathsf{MGen}(\mathsf{pp})$.
- The challenger creates $S_{\mathsf{nym}} := \emptyset$ to store pseudonym-user-pairs, $S_{\mathsf{cred}} := \emptyset$ to store credential-user-pairs and $t^* := \epsilon$ to store the challenge transcript.
- The challenger gives $\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi$ and opk to the adversary.
- The adversary $\mathcal{A}$ chooses $1^{n_U}, 1^{n_I}$ with $n_U, n_I \in \mathbb{N}$.
- The challenger creates sets U and I for users and issuers, with $|U| = n_U$ and $|I| = n_I$, and an initially empty subset $\hat{U}$ for corrupted users. User and issuer sets contain a unique ID for each entity, chosen independently from any secret keys. The mapping from IDs to actual entities is done by the challenger.
- For each user $u \in U$ the challenger makes u run $\mathsf{U.Init}(\mathsf{pp}, \mathsf{opk})$ to obtain a user secret key usk and a user public key upk. The challenger gives the upk of all users to the adversary.
- For each issuer $i \in \{1, ..., n_I\}$ the adversary gives an ipk_i to the challenger together with an 1^{ℓ_i} describing the number of supported attributes by i.
- The adversary obtains the sets U and I from the challenger. Furthermore, she gets full knowledge of and full control over all issuers. $\mathcal{A}$ may query following oracles, where $\perp$ is returned if the oracle input is invalid.
  - *corruptUser*(u, upk'), where $u \in U$: $\mathcal{A}$ may arbitrarily select a user $u \in U$ to gain control over her. The challenger hands all private information of u to $\mathcal{A}$. This includes the user's usk, her registration information, all of her pseudonym/key pairs $(\mathsf{nym}, \mathsf{psk})$, credentials and past protocol views. From that point on, $\mathcal{A}$ has full control over u. Note here that, since $\mathcal{A}$ has full control over u, it can arbitrarily choose a new usk for u and use it in following protocols. If upk' is not equal to the upk of u, upk' becomes the new public key of u. The challenger adds u to $\hat{U}$.
  - *runCreateNym*(u), where $u \in U \setminus \hat{U}$: The challenger makes u run $\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})$ to obtain $(\mathsf{nym}, \mathsf{psk})$. The challenger adds (nym, u) to S_{nym}, stores the corresponding psk and returns nym to $\mathcal{A}$.
  - *runJoin*(u), where $u \in U$: The challenger makes u execute $\mathsf{Join}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{usk}) \leftrightarrow \mathsf{MJoin}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{osk}, \mathsf{reg})$ with herself. If u is honest, i. e. $u \in U \setminus \hat{U}$, upk and usk correspond to u and the challenger obtains and stores $\mathsf{reginfo}$ in relation to u (and overwrites old versions, if she has any for the same user). The challenger overwrites the old reg with output reg' On success the challenger returns 1 and else 0.
  - *runProveVrfyNym*(u, v, nym), where $u \in U$, $v \in U \cup I$ and $(\mathsf{nym}, u) \in S_{\mathsf{nym}}$: The challenger makes u and v execute $\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym})$, where psk is the pseudonym secret related to nym, and usk corresponds to u. The challenger returns 1 to $\mathcal{A}$ if v outputs 1, 0 otherwise.

- *runRcvIssCred*$(u, i, \mathsf{nym}, (a_j)_{j=1}^{\ell_i})$, where $u \in U$, $i \in I$ and $(\mathsf{nym}, u) \in S_{\mathsf{nym}}$: The challenger makes u and i execute $\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_j)_{j=1}^{\ell_i}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_j)_{j=1}^{\ell_i}, \mathsf{isk})$, where ipk and isk are i's public and secret key. If RcvCred outputs cred and not $\perp$, the challenger chooses a unique index j, adds (cred, j, u) to S_{cred} and returns j to $\mathcal{A}$. Else the challenger returns $\perp$.
- *runProveVrfyCred*$(u, v, \mathsf{nym}, j, \phi)$, where $u \in U$, $v \in U \cup I$, $(\mathsf{nym}, u) \in S_{\mathsf{nym}}$ and $\exists \mathsf{cred}$ with $(\mathsf{cred}, j, u) \in S_{\mathsf{cred}}$: The challenger makes u and v execute $\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}, \mathsf{reginfo}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi)$, where ipk is the public key of the issuer who issued cred, usk is u's user secret and psk the pseudonym secret corresponding to nym. The challenger returns 1 to $\mathcal{A}$ if v outputs 1, 0 otherwise.
- *Open*(t), where $t \neq t^*$: The challenger responds with $\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}, t)$.

- For $k \in \{0, 1\}$ the adversary outputs $(u_k, j_k), \mathsf{ipk}, \phi \in \mathcal{U}_\Phi$ and $i \in I$ such that u_k joined the system, $\exists \mathsf{cred}_k = ((a_0, a_1, \dots, a_{\ell_i}), \cdot, \mathsf{ipk})$, such that $(\mathsf{cred}_k, j_k, u_k) \in S_{\mathsf{cred}}$ and $\phi(a_1, \dots, a_{\ell_i}) = 1$. If a requirement does not hold, $\mathcal{A}$ loses and the experiment outputs 0.
- The challenger chooses a bit $b \leftarrow \{0, 1\}$.
- The challenger runs $\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})$ to obtain $(\mathsf{nym}^*, \mathsf{psk}^*)$ (*not* added to S_{nym}), where usk corresponds to u_b.
- The challenger makes u_b and i execute $\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}^*, \phi, \mathsf{usk}_b, \mathsf{psk}^*, \mathsf{reginfo}_b, \mathsf{cred}^*) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}^*, \phi)$ and stores the transcript in t^*, where $\mathsf{cred}^* = (a_0, ..., a_{\ell_i}, \cdot, \mathsf{ipk})$ is some credential with $(\mathsf{cred}^*, \cdot, u_b) \in S_{cred}$ and $\phi(a_1, ..., a_{\ell_i}) = 1$.
- The adversary may query her oracles as before until she outputs a bit b'. If $b = b'$ and $u_0, u_1 \notin \hat{U}$, the experiment's output is 1 and we say that $\mathcal{A}$ wins. Else, the output is 0 and $\mathcal{A}$ loses.

Note that the adversary may use the *corruptUser* oracle more than once on the same user. Then, she does not learn anything new, since she controls the user anyway, but may change the user's upk multiple times.

We move on to the security definitions, which, in general mirror the definitions of Definition 4.20.

Definition 4.39. We call an extended credential system $\Pi = (\mathsf{Setup}, \mathsf{MGen}, \mathsf{U.Init}, \mathsf{I.Init}, \mathsf{CreateNym}, (\mathsf{Join}, \mathsf{MJoin}), (\mathsf{ProveNym}, \mathsf{VrfyNym}), (\mathsf{RcvCred}, \mathsf{IssCred}), (\mathsf{ProveCred}, \mathsf{VrfyCred}), \mathsf{Open})$ *anonymous* if the following properties are fulfilled:

1. CreateNym **hides the user secret** (Definition 4.20.1), ProveNym **hides the user secret** (Definition 4.20.2), **Anonymity when executing** RcvCred (Definition 4.20.3)
2. (**Anonymity when executing** ProveCred) For all ppt adversaries $\mathcal{A}$ we have that $\Pr[\mathsf{Exp}_{\Pi,\mathcal{A}}^{anon}(n) = 1] - \frac{1}{2}$ is negligible.

Note that the reused security definitions of Definition 4.20 need to be altered slightly to accommodate for the changes of the extended ACS. Since they would be altered in a non-meaningful way, we do not specify them.

4.3.2.2 Soundness

With the addition of the new opening feature and the corresponding changes to the definition we also have to incorporate those into the soundness definition. To achieve that, we adapt the oracle queries in $\mathsf{Exp}_{\Pi,\mathcal{A}}^{Soundness}(n)$ (cf. Section 4.2.2.2) and add an opening oracle. We also

slightly adjust the winning conditions for the adversary and add a new one. In the new game an adversary has to show a credential where the protocol transcript is not traceable to a corrupt user.

Definition 4.40 (Soundness experiment). Define the following Experiment $\mathsf{Exp}_{\Pi,\mathcal{A}}^{Ext-Soundness}(n)$ for an extended ACS Π (Definition 4.37) between a challenger and a ppt adversary $\mathcal{A}$. Note here that we assume secure communication for honest entities in the experiment. The adversary only sees outputs which are explicitly mentioned in the definition below.

- The challenger runs $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \leftarrow \mathsf{Setup}(1^n)$.
- The challenger runs $(\mathsf{osk}, \mathsf{opk}, \mathsf{reg}) \leftarrow \mathsf{MGen}(\mathsf{pp})$.
- The adversary $\mathcal{A}$ chooses $1^{n_U}, 1^{n_I}, 1^{\ell_1}, \dots, 1^{\ell_{n_I}}$ with $n_U, n_I, \ell_1, \dots, \ell_{n_I} \in \mathbb{N}$.
- The challenger creates sets U and I for users and issuers, with $|U| = n_U$ and $|I| = n_I$, and initially empty subsets $\hat{U}$ and $\hat{I}$ for corrupted users and issuers respectively. The challenger also creates an initially empty pseudonym-user-pair set S_{nym}, an registered-upk-user-pair set S_{upk} and a credential-ID-credential-pair set S_{cred}. The user and issuer sets contain unique IDs for each entity, the mapping from the IDs to the actual entities is done by the challenger. The IDs are chosen independently from any secret keys.
- For each user $u \in U$ the challenger makes u run $\mathsf{U.Init}(\mathsf{pp}, \mathsf{opk})$ to obtain a user secret key usk and a user public key upk.
- For each issuer $i \in I$ the challenger makes i run $\mathsf{I.Init}(\mathsf{pp}, 1^{\ell_j})$ to obtain an issuer public key ipk and secret key isk. Using 1^{ℓ_j} for the jth issuer. The ipk is made publicly available.
- After this initial setup the adversary gets as input the sets U, I and all public parameters, including the upks and ipks of the ACS Π. $\mathcal{A}$ may query following oracles, where $\perp$ is returned if the oracle input is invalid.
  - *corruptUser*(u, upk'): $\mathcal{A}$ may arbitrarily select a user $u \in U$ to gain control over her. The challenger hands all private information of u to $\mathcal{A}$. This includes the user's usk, all of her pseudonym/key pairs $(\mathsf{nym}, \mathsf{psk})$, her credentials, her registration information and all her past protocol views. From that point on, $\mathcal{A}$ has full control over u. Note here that, since $\mathcal{A}$ has full control over u, it can arbitrarily choose a new usk for u and use it in following protocols. If upk' is not equal to the upk of u, upk' becomes u's new public key. The challenger adds u to $\hat{U}$.
  - *corruptIssuer*(i, ipk'): $\mathcal{A}$ may arbitrarily select an issuer $i \in I$ to gain control over her. The challenger hands all private information of i to $\mathcal{A}$. This includes the issuer's secret key isk, all pseudonyms users have used in past protocols with her and all her past protocol views. From that point on $\mathcal{A}$ has full control over i. If ipk' is not equal to the ipk of i, ipk' becomes i's new public key. The challenger adds i to $\hat{I}$.
  - *runCreateNym*(u): $\mathcal{A}$ may select a user, which then runs $\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})$. $\mathcal{A}$ learns the pseudonym nym but not the corresponding psk. The challenger adds (nym, u) to S_{nym}.
  - *runJoin*(u): $\mathcal{A}$ may select a user, which then executes $\mathsf{Join}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{usk}) \leftrightarrow \mathsf{MJoin}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{osk}, \mathsf{reg})$ with the challenger, where usk is u's secret key and upk is u's public key. If the protocol is successful, the user saves $\mathsf{reginfo}$ (and overwrites old versions, if she has any), the challenger overwrites the old reg with output reg', adds (upk, u) to S_{upk} and returns 1 to $\mathcal{A}$, else 0.

- *runProveVrfyNym*(u, v, nym): Where $u \in U$ and $v \in U \cup I$. The challenger makes u and v execute $\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym})$, where usk is u's secret key and psk is the pseudonym secret of nym. If one or both of those entities are corrupted, $\mathcal{A}$ takes the corresponding role in the protocol execution. The challenger returns 1 to $\mathcal{A}$ if v outputs 1, 0 otherwise.
- *runRcvIssCred*$(u, i, \mathsf{nym}, (a_i)_{i=1}^{\ell})$: Where $u \in U$, $i \in I$, and where ℓ is the ℓ_j that $\mathcal{A}$ has chosen for i in the setup. The challenger makes u and i execute $\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{isk})$, where usk is u's secret key, psk is the pseudonym secret of nym and ipk and isk are i's public and secret key. If one or both of those entities are corrupted, $\mathcal{A}$ takes the corresponding role in the protocol execution. If the protocol is successful, the challenger returns 1 and a unique *credId* to $\mathcal{A}$ and adds $(credId, \mathsf{cred})$ to S_{cred}, else 0.
- *runProveVrfyCred*$(u, v, \mathsf{nym}, credId, \phi)$: Where $u \in U$ and $v \in U \cup I$. If u is honest and either has not successfully joined the system or *credId* is not contained in a pair in S_{cred} the challenger returns 0. Otherwise the challenger makes u and v execute $\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}, \mathsf{reginfo}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi)$, where cred is the credential in the $(credId, \mathsf{cred})$ pair in S_{cred}, ipk is the public key of the issuer who issued cred, usk is u's user secret, psk the pseudonym secret corresponding to nym and $\mathsf{reginfo}$ is u's registration information which u got through Join. If one or both of those entities are corrupted, $\mathcal{A}$ takes the corresponding role in the protocol execution. The challenger returns 1 to $\mathcal{A}$ if v outputs 1, 0 otherwise.
- *Open*(t): The challenger responds with $\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}, t)$.

Note that as in Definition 4.38 the *corruptUser* and *corruptIssuer* oracles may be used on the same entity multiple times to change the entities public key.

Definition 4.41 (Non-Impersonation)**.** In $\mathsf{Game}_{\Pi,\mathcal{A}}^{NonImp}(n)$ an adversary $\mathcal{A}$ first participates in $\mathsf{Exp}_{\Pi,\mathcal{A}}^{Ext-Soundness}(n)$. After that it outputs nym, chooses a corrupted user $u \in \hat{U}$ and an honest verifier $v \in (I \cup U) \setminus (\hat{U} \cup \hat{I})$. Then, the challenger makes u and v execute $\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym})$, where pp are the public parameters of the prior experiment. Since $u \in \hat{U}$, $\mathcal{A}$ fully controls the user during protocol execution. $\mathcal{A}$ wins if and only if v accepts and the following requirement is true:

- $\exists\ (\mathsf{nym}, u') \in S_{nym}$ with $u' \notin \hat{U}$. Which means that nym does belong to a non-corrupted user.

If for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function μ such that $\Pr[\mathsf{Game}_{\Pi,\mathcal{A}}^{NonImp}(1) = 1] \leq \mu(n)$, we say that Π offers non-impersonation.

Definition 4.42 (Credential Unforgeability)**.** In $\mathsf{Game}_{\Pi,\mathcal{A}}^{CredForge}(n)$ an adversary $\mathcal{A}$ first participates in $\mathsf{Exp}_{\Pi,\mathcal{A}}^{Ext-Soundness}(n)$. After that it outputs $(\mathsf{ipk}, \mathsf{nym}, \phi)$, chooses a corrupted user $u \in \hat{U}$ and an honest verifier $v \in (I \cup U) \setminus (\hat{U} \cup \hat{I})$. Then, the challenger makes u and v execute $\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}, \mathsf{reginfo}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi)$, where $\mathsf{pp}, \mathsf{opk}$ are the public parameters of the prior experiment. Since $u \in \hat{U}$, $\mathcal{A}$ fully controls the user during protocol execution. $\mathcal{A}$ wins if and only if v accepts and the following requirements are true:

- ipk is the public key of a non-corrupted issuer.
- $\mathcal{A}$ has never queried *runRcvIssCred*$(u', i, \mathsf{nym}', (a_j)_{j=1}^{\ell_i})$, where $\phi((a_j)_{j=1}^{\ell_i}) = 1$, i is the issuer corresponding to ipk and $u' \in \hat{U}$. The latter includes instances where *corruptUser*(u') got queried after the credential got granted.

If for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function μ such that $\Pr[\mathsf{Game}^{CredForge}_{\Pi,\mathcal{A}}(n) = 1] \leq \mu(n)$, we say that Π offers credential unforgeability.

Definition 4.43 (Traceability). In $\mathsf{Game}^{Trace}_{\Pi,\mathcal{A}}(n)$ an adversary $\mathcal{A}$ first participates in $\mathsf{Exp}^{Ext-Soundness}_{\Pi,\mathcal{A}}(n)$. After that it outputs $(\mathsf{ipk}, \mathsf{nym}, \phi)$, chooses a user $u \in \hat{U}$ and an honest verifier $v \in (I \cup U) \setminus (\hat{U} \cup \hat{I})$. Then, the challenger makes u and v execute $\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}, \mathsf{reginfo}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi)$, where $\mathsf{pp}, \mathsf{opk}$ are the public parameters of the prior experiment. Since $u \in \hat{U}$, $\mathcal{A}$ fully controls the user during protocol execution. Denote the transcript of the protocol execution by t. $\mathcal{A}$ wins if and only if v accepts and the following requirement is true:

- $\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}, t)$ outputs a upk with $\forall u' \in \hat{U}$: $(\mathsf{upk}, u') \notin S_{upk}$.

If for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible function μ such that $\Pr[\mathsf{Game}^{Trace}_{\Pi,\mathcal{A}}(n) = 1] \leq \mu(n)$, we say that Π offers Traceability.

Note here that we, at this point, do not look at the case where two fully colluding users share a credential. If the first user gives a credential and her user secret to another user, in regards to the credential system the second user is now the first one.

Definition 4.44 (Soundness). We call Π *sound* if it offers Non-Impersonation (Definition 4.41), Credential Unforgeability (Definition 4.42) and Traceability (Definition 4.43).

4.3.3 Construction

Now we proceed with the concrete construction of our extended ACS. We use techniques of the group signature scheme proposed by Pointcheval and Sanders [PS16, Appendix B], which yields the demanded security properties. Essentially a system master creates a group, which users need to join to successfully execute $\mathsf{ProveCred}$. This execution then is traceable via the group signature's Open mechanism. Therefore we apply the Fiat-Shamir heuristic from Section 3.10 on the Schnorr proofs (Construction 3.44) mentioned in the following.

Construction 4.45. Let G be a type 3 bilinear group generator (Definition 3.2), let $\Pi_{\text{com}} = (\Pi_{\text{com}}.\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be the Pedersen commitment scheme (Construction 3.23) and let $\Pi_{\text{sign}} = (\Pi_{\text{sign}}.\mathsf{Setup}, \mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be the Pointcheval-Sanders signature scheme (Construction 3.17).

- $\mathsf{Setup}(1^n)$: On input security parameter 1^n, Setup generates a type 3 bilinear group $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n)$ and pseudonym public parameters $\mathsf{pp}_{\text{nym}} := (g_{\text{nym}}, h_{\text{nym}}, p, \mathbb{G}_1)$ with $g_{\text{nym}} \leftarrow \mathbb{G}_1 \setminus \{1\}$ and $h_{\text{nym}} \leftarrow \mathbb{G}_1$. Additionally, it creates pp_{aux} by generating public parameters of Nguyen accumulator (Section 3.12.3) based on the bilinear group, to enable membership proofs in predicates (Section 4.1.3.3). It returns $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$.

- $\mathsf{MGen}(\mathsf{pp})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$, MGen returns $(\mathsf{osk}, \mathsf{opk}, \mathsf{reg})$ with opening key pair $(\mathsf{osk}, \mathsf{opk}) \leftarrow \Pi_{\text{sign}}.\mathsf{Gen}_1(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and registry $\mathsf{reg} := \emptyset$.

- $\mathsf{U.Init}(\mathsf{pp}, \mathsf{opk})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$ and opener public key $\mathsf{opk} = (g_{\text{opk}}, Y_{\text{opk}}, \tilde{g}_{\text{opk}}, \tilde{X}_{\text{opk}}, \tilde{Y}_{\text{opk}})$, $\mathsf{U.Init}$ chooses user secret key $\mathsf{usk} \leftarrow \mathbb{Z}_p$, sets public key $\mathsf{upk} := g_{\text{opk}}^{\mathsf{usk}}$ and returns $(\mathsf{usk}, \mathsf{upk})$.

- $\mathsf{I.Init}(\mathsf{pp}, 1^\ell)$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$ and number $\ell \in \mathbb{N}$, $\mathsf{I.Init}$ generates and returns issuer key pair $(\mathsf{ipk}, \mathsf{isk}) \leftarrow \Pi_{\text{sign}}.\mathsf{Gen}_{\ell+1}(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ with $\mathsf{ipk} = (g, Y_0, Y_1, \ldots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_0, \tilde{Y}_1, \ldots \tilde{Y}_\ell)$ for issuing credentials over ℓ attributes and a usk.

- $\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$ and user secret usk, $\mathsf{CreateNym}$ generates and returns $(\mathsf{nym}, \mathsf{psk}) \leftarrow \mathsf{Com}_1(\mathsf{pp}_{\mathrm{nym}}, \mathsf{usk})$ with $\mathsf{psk} = (\mathsf{usk}, d)$.

- $(\mathsf{Join}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{usk}), \mathsf{MJoin}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{osk}, \mathsf{reg}))$ is an interactive protocol with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$, opener public key $\mathsf{opk} = (g_{\mathrm{opk}}, Y_{\mathrm{opk}}, \tilde{g}_{\mathrm{opk}}, \tilde{X}_{\mathrm{opk}}, \tilde{Y}_{\mathrm{opk}})$ and $\mathsf{pp}_{\mathrm{nym}} = (g_{\mathrm{nym}}, h_{\mathrm{nym}}, p, \mathbb{G}_1)$. The system manager parses the opener secret key osk as (x, y). The user computes $\tilde{\tau} := \tilde{Y}_{\mathrm{opk}}^{\mathsf{usk}}$, sends $\tilde{\tau}$ to the manager and runs a Σ-protocol of form

$$\mathsf{PK}\Big\{(\mathsf{usk}) : \mathsf{upk} = g_{\mathrm{opk}}^{\mathsf{usk}}\Big\}.$$

  The manager proceeds, if and only if she accepts in this proof *and* $e(\mathsf{upk}, \tilde{Y}_{\mathrm{opk}}) = e(g_{\mathrm{opk}}, \tilde{\tau})$ holds. If there is an entry $(\mathsf{upk}, \sigma, \cdot)$ in reg, the manager returns σ and stops. Else the manager chooses $u \leftarrow \mathbb{Z}_p^*$, computes $\sigma := (\sigma_1, \sigma_2) := (g_{\mathrm{opk}}^u, (g_{\mathrm{opk}}^x \cdot \mathsf{upk}^y)^u)$, sends σ to the user, adds the entry $(\mathsf{upk}, \sigma, \tilde{\tau})$ to reg and (locally) outputs the updated reg. Then, the user checks the signature's validity via $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{opk}, \mathsf{usk}, \sigma)$. If the check fails, she (locally) outputs $\perp$, and else registration information $\mathsf{reginfo} = (\sigma, \mathsf{opk})$.

- $(\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}), \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym}))$ is an interactive protocol with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$ and $\mathsf{pp}_{\mathrm{nym}} = (g_{\mathrm{nym}}, h_{\mathrm{nym}}, p, \mathbb{G}_1)$. The prover parses the pseudonym secret psk as (usk, d) and performs an interactive zero-knowledge argument of knowledge of form $\mathsf{PK}\Big\{(\mathsf{usk}, d) : \mathsf{nym} = g_{\mathrm{nym}}^d h_{\mathrm{nym}}^{\mathsf{usk}}\Big\}$ with the verifier.

- $(\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}), \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{isk}))$ is an interactive protocol with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$, $\mathsf{pp}_{\mathrm{nym}} = (g_{\mathrm{nym}}, h_{\mathrm{nym}}, p, \mathbb{G}_1)$ and issuer public key $\mathsf{ipk} = (g, Y_0, \dots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_0, \dots, \tilde{Y}_\ell)$. The receiver parses the pseudonym secret psk as (usk, d) and generates $pp_{\mathrm{iss}} = (g, Y_0, p, \mathbb{G}_1) := \mathsf{BlindInit}(\mathsf{pp}, \mathsf{ipk}, \{0\})$ (ignoring $\mathsf{pp}_{\mathrm{nym}}$ in pp) and computes $(C, (\mathsf{usk}, r)) \leftarrow \mathsf{Com}_1(\mathsf{pp}_{\mathrm{iss}}, \mathsf{usk})$. Then, she sends C to the issuer and runs a Σ-protocol of form

$$\mathsf{PK}\Big\{(\mathsf{usk}, d, r) : \mathsf{nym} = g_{\mathrm{nym}}^d h_{\mathrm{nym}}^{\mathsf{usk}} \wedge C = g^r Y_0^{\mathsf{usk}}\Big\}$$

  with the issuer. As in $(\mathsf{BlindRcv}, \mathsf{BlindIssue})$, the issuer now chooses $u \leftarrow \mathbb{Z}_p^*$, computes $(\sigma_1', \sigma_2') := (g^u, (g^x \cdot C \cdot \prod_{i=1}^{\ell} Y_i^{a_i})^u)$ and sends (σ_1', σ_2') to the receiver. The receiver computes $\sigma = (\sigma_1', \sigma_2' \cdot (\sigma_1')^{-r})$, and checks the signature's validity via $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{ipk}, (\mathsf{usk}, a_1, \dots, a_\ell), \sigma)$. If the check fails, she (locally) outputs $\perp$ and else a credential $\mathsf{cred} = (\sigma, (a_1, \dots, a_\ell), \mathsf{ipk})$.

- $(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{reginfo}, \mathsf{cred}), \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi))$ is an interactive protocol with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$, $\mathsf{pp}_{\mathrm{nym}} = (g_{\mathrm{nym}}, h_{\mathrm{nym}}, p, \mathbb{G}_1)$, opener public key $\mathsf{opk} = (g_{\mathrm{opk}}, Y_{\mathrm{opk}}, \tilde{g}_{\mathrm{opk}}, \tilde{X}_{\mathrm{opk}}, \tilde{Y}_{\mathrm{opk}})$ and issuer public key $\mathsf{ipk} = (g, Y_0, \dots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_0, \dots, \tilde{Y}_\ell)$. The prover parses her private input to $\mathsf{psk} = (\mathsf{usk}, d)$, $\mathsf{reginfo} = ((\hat{\sigma}_1, \hat{\sigma}_2), \mathsf{opk})$ and $\mathsf{cred} = ((\sigma_1, \sigma_2), (a_1, \dots, a_\ell), \mathsf{ipk})$.

  If $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{ipk}, (\mathsf{usk}, a_1, \dots, a_\ell), \sigma) = 0$, $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{opk}, \mathsf{usk}, \hat{\sigma}) = 0$ or $\phi(a_1, \dots, a_\ell) = 0$ holds, she outputs $\perp$ and stops. Else she chooses $s \leftarrow \mathbb{Z}_p^*$, sets $\hat{\sigma}' := (\hat{\sigma}_1^s, \hat{\sigma}_2^s)$ and sends $\hat{\sigma}'$ to the verifier. Next, she chooses $(u, r) \leftarrow \mathbb{Z}_p^* \times \mathbb{Z}_p$, sets $\sigma' := (\sigma_1', \sigma_2') := (\sigma_1^u, (\sigma_2 \cdot \sigma_1^r)^u)$ and sends σ' to the verifier. For $i = 1, \dots, \ell$ she generates $(C_i, (a_i, d_i)) \leftarrow \mathsf{Com}(\mathsf{pp}_{\mathrm{nym}}, a_i)$ and sends C_i to the verifier. The user computes a non-interactive argument π using the Fiat-Shamir

heuristic (Construction 3.55) on the Σ-protocol of form

$$\mathsf{PK}\left\{(\mathsf{usk}, d, (a_i, d_i)_{i=1}^{\ell}, r) : \begin{array}{l} e(\sigma_1', \tilde{g})^r e(\sigma_1', \tilde{Y}_0)^{\mathsf{usk}} \prod_{i=1}^{\ell} e(\sigma_1', \tilde{Y}_i)^{a_i} = \dfrac{e(\sigma_2', \tilde{g})}{e(\sigma_1', \tilde{X})} \\ \wedge\, \mathsf{nym} = g_{\mathrm{nym}}^{d} h_{\mathrm{nym}}^{\mathsf{usk}} \bigwedge_{i=1}^{\ell} C_i = g_{\mathrm{nym}}^{d_i} h_{\mathrm{nym}}^{a_i} \wedge \phi(a_1, \ldots, a_\ell) = 1 \\ \wedge\, e(\hat{\sigma}_1', \tilde{Y}_{\mathrm{opk}})^{\mathsf{usk}} = \dfrac{e(\hat{\sigma}_2', \tilde{g}_{\mathrm{opk}})}{e(\hat{\sigma}_1', \tilde{X}_{\mathrm{opk}})} \end{array}\right\},$$

where the proof of $\phi(a_1, \ldots, a_\ell) = 1$ is instantiated via the technique for proofs of partial knowledge (Construction 3.59) and the protocols from Section 4.1.3. The verifier (deterministically) accepts, if and only if $\sigma_1' \neq 1 \neq \hat{\sigma}_1'$ holds and π is valid according to Construction 3.55.

- $\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}, t)$: On input open public key $\mathsf{opk} = (g_{\mathrm{opk}}, Y_{\mathrm{opk}}, \tilde{g}_{\mathrm{opk}}, \tilde{X}_{\mathrm{opk}}, \tilde{Y}_{\mathrm{opk}})$, open secret key osk, registry information obtained from MJoin and a transcript $t = ((\hat{\sigma}_1', \hat{\sigma}_2'), \cdot)$, Open first uses $\mathsf{VrfyCred}$ to check, whether the transcript is valid. If it is invalid, it outputs $\perp$. Else it iterates through the entries $(\mathsf{upk}, \cdot, \tilde{\tau})$ from reg, until $e(\hat{\sigma}_2', \tilde{g}_{\mathrm{opk}}) e(\hat{\sigma}_1', \tilde{X}_{\mathrm{opk}})^{-1} = e(\hat{\sigma}_1', \tilde{\tau})$ holds. If it finds such an entry, it outputs upk, and else outputs $\perp$.

Lemma 4.46. *Construction 4.45 is a correct extended anonymous credential system (Definition 4.37).*

Proof. According to Definition 4.37 an extended anonymous credential system is correct if 1) honestly generated pseudonyms are valid, 2) honestly issued credentials are valid and 3) $\mathsf{ProveCred}$ is traceable by the open manager. We prove correctness by first fixing arbitrary outputs of the algorithms from Construction 4.45 and then arguing why the three properties hold.

Concretely, we first fix arbitrary $n, c, \ell \in \mathbb{N}$, $j \in \{1, \ldots, c\}$, $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi) \in [\mathsf{Setup}(1^n)]$, $(\mathsf{osk}, \mathsf{opk}, \mathsf{reg}) \in [\mathsf{MGen}(\mathsf{pp})]$, $(a_1, \ldots, a_\ell) \in \mathcal{U}_A^{\ell}$, $\phi \in \mathcal{U}_\Phi$ with $\phi(a_1, \ldots, a_\ell) = 1$, $(\mathsf{ipk}, \mathsf{isk}) \in [\mathsf{I.Init}(\mathsf{pp}, 1^{\ell})]$, $((\mathsf{usk}_i, \mathsf{upk}_i))_{1 \leq i \leq c} \in [\mathsf{U.Init}(\mathsf{pp})]^c$ and $(\mathsf{nym}, \mathsf{psk}), (\mathsf{nym}', \mathsf{psk}') \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}_j)]$. For $i = 1, \ldots, c$ fix arbitrary $\mathsf{reginfo}_i, \mathsf{reg}_{i+1}$ of $(\mathsf{Join}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}_i, \mathsf{usk}_i) \leftrightarrow \mathsf{MJoin}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}_i, \mathsf{osk}, \mathsf{reg}_i))$, cred of $(\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}_j, \mathsf{psk}) \leftrightarrow \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{isk}))$ and transcript t of $(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}', \phi, \mathsf{usk}_j, \mathsf{psk}', \mathsf{cred}, \mathsf{reginfo}_j) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}', \phi))$.

1) Since $(\mathsf{ProveNym}, \mathsf{VrfyNym})$ and the choice of its inputs remained exactly the same in Construction 4.45 compared to our basic ACS (Construction 4.25), we have that *honestly generated pseudonyms are valid* by Lemma 4.26. Hence we know that nym and nym' are *valid pseudonyms.*

2) To prove *honestly issued credentials are valid* we need to show that

$$1 - \mu(|pp|) = \Pr[(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}', \phi, \mathsf{usk}_j, \mathsf{psk}', \mathsf{cred}, \mathsf{reginfo}_j) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}', \phi)) \rightarrow 1]$$

holds for some negligible function μ. We mainly rely on Lemma 4.26, as the only difference between Construction 4.45 and Construction 4.25 is related to $\mathsf{reginfo}_j$. It is left to show that $\mathsf{reginfo}_j$ from $(\mathsf{Join}, \mathsf{MJoin})$ as input of $\mathsf{ProveCred}$ suffices to make $\mathsf{VrfyCred}$ always accept. By construction of MGen and Gen_1 from the Pointcheval-Sanders signature scheme (Construction 3.17) we know $\mathsf{osk} = (x, y) \in \mathbb{Z}_p^2$ and $\mathsf{opk} = (g, Y, \tilde{g}, \tilde{X}, \tilde{Y}) = (g, g^y, \tilde{g}, \tilde{g}^x, \tilde{g}^y)$ for generators $g, \tilde{g}$ (we omit opk as index for the sake of readability). With usk_j as input, Join computes $\tilde{\tau} := \tilde{Y}^{\mathsf{usk}_j} = \tilde{g}^{y \cdot \mathsf{usk}_j}$ and by correctness of the generalized Schnorr protocol succeeds in the argument of knowledge, since $\mathsf{upk}_j = g^{\mathsf{usk}_j}$ is output of $\mathsf{U.Init}$. The check $e(\mathsf{upk_j}, \tilde{Y}) = e(g, \tilde{\tau})$ in MJoin is fulfilled too, shown by

$$e(\mathsf{upk_j}, \tilde{Y}) = e(g^{\mathsf{usk}_j}, \tilde{g}^y) = e(g, \tilde{g})^{y \cdot \mathsf{usk}_j} = e(g, \tilde{g}^{y \cdot \mathsf{usk}_j}) = e(g, \tilde{\tau}).$$

Hence $\mathsf{reginfo}_j = (\hat{\sigma}_1, \hat{\sigma}_2) = (g^u, g^{(x+y\cdot\mathsf{usk}_j)u})$ is output of Join for some $u \in \mathbb{Z}_p^*$. ProveCred computes $(\hat{\sigma}_1', \hat{\sigma}_2') := (\hat{\sigma}_1^s, \hat{\sigma}_2^s) = (g^{us}, g^{(x+y\cdot\mathsf{usk}_j)us})$ for some $s \in \mathbb{Z}_p^*$. Since $\mathbb{Z}_p^*$ is closed under multiplication $us = r$ holds for some $r \in \mathbb{Z}_p^*$. Thus, for generator g, $\hat{\sigma}_1' = g^r \neq 1$ holds, and the corresponding check at the end of VrfyCred is fulfilled. It remains to show the non-interactive argument of knowledge is accepted. We can compute

$$\frac{e(\hat{\sigma}_2', \tilde{g})}{e(\hat{\sigma}_1', \tilde{X}')} = \frac{e(g^{(x+y\cdot\mathsf{usk}_j)r}, \tilde{g})}{e(g^r, \tilde{g}^x)} = \frac{e(g, \tilde{g})^{rx+ry\cdot\mathsf{usk}_j}}{e(g, \tilde{g})^{rx}} = e(g, \tilde{g})^{ry\cdot\mathsf{usk}_j} = e(g^r, \tilde{g}^{y\cdot\mathsf{usk}_j}) = e(\hat{\sigma}_1', \tilde{Y})^{\mathsf{usk}_j}$$

and hence see that usk_j is a valid witness for the argument of knowledge. By correctness of the Fiat-Shamir heuristic and the remaining parts of ProveCred (Lemma 4.26) we conclude that VrfyCred always accepts. Formally we have

$$\begin{aligned}\Pr[(&\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}', \phi, \mathsf{usk}_j, \mathsf{psk}', \mathsf{cred}, \mathsf{reginfo}_j) \\ &\leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}', \phi)) \to 1] \qquad = 1,\end{aligned}$$

meaning *honestly issued credentials are valid* in Construction 4.45.

3) The last point to prove is ProveCred *is traceable by the open manager* or formally that

$$\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}_{c+1}, t) = \mathsf{upk}_j$$

holds. Open first uses VrfyCred to check whether the fixed transcript t is valid, i. e. checks if the deterministic VrfyCred outputs 1 for the non-interactive proof. As shown before, honestly issued credentials are valid, hence the fixed transcript t is accepted and Open iterates through its reg entries. By construction of the algorithms, reg_{c+1} contains an entry $(\mathsf{upk}_i, \cdot, \tilde{\tau}) = (g^{\mathsf{usk}_i}, \cdot, \tilde{Y}^{\mathsf{usk}_i})$ for each $i = 1, \ldots, c$. We know that t contains $(\hat{\sigma}_1', \hat{\sigma}_2') = (g^r, g^{(x+y\cdot\mathsf{usk}_j)r})$ for some $r \in \mathbb{Z}_p^*$ (cf. part 2). For $i = 1, \ldots, c$ Open checks whether $e(\hat{\sigma}_2', \tilde{g}')e(\hat{\sigma}_1', \tilde{X}')^{-1} = e(\hat{\sigma}_1', \tilde{\tau}_i)$ holds. By

$$e(\hat{\sigma}_2', \tilde{g}')e(\hat{\sigma}_1', \tilde{X}')^{-1} = e(g, \tilde{g})^{(x+y\cdot\mathsf{usk}_j)r} e(g, \tilde{g})^{-rx} = e(g, \tilde{g})^{ry\cdot\mathsf{usk}_j} \stackrel{?}{=} e(\hat{\sigma}_1', \tilde{\tau}_i) = e(g, \tilde{g})^{ry\cdot\mathsf{usk}_i}$$

we see, this happens if and only if $ry \cdot \mathsf{usk}_i = ry \cdot \mathsf{usk}_j \mod p$. This holds only for $usk_i = usk_j$, when $ry \in \mathbb{Z}_p^*$. By $r \in \mathbb{Z}_p^*$, the only case where $ry \notin \mathbb{Z}_p^*$ occurs is for $y = 0$. Since y is chosen uniformly at random from $\mathbb{Z}_p$, this happens with probability $\frac{1}{p}$. If we have $y = 0$, the (possibly correct) first upk entry of reg is output by Open. Therefore we have

$$\Pr[\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}_{c+1}, t) = \mathsf{upk}_j] \geq 1 - \frac{1}{p},$$

where $\frac{1}{p}$ is negligible in $|\mathsf{pp}|$. This proves that ProveCred *is traceable by the open manager.* □

4.3.4 Security Proofs

In the following section, we are going to prove the anonymity and soundness of our extended ACS Π (Construction 4.45) in regards to the definitions from Section 4.3.2.

4.3.4.1 Anonymity

We show our extended ACS Π (Construction 4.45) is anonymous according to both requirements of Definition 4.39. The first (Definition 4.39.1) demands anonymity when executing RcvCred and that CreateNym, as well as ProveNym, hides the user secret. These notions essentially remained the same as for anonymity of a basic ACS (Definition 4.20). Since the corresponding protocols did not change, i. e. still are perfect zero-knowledge (in the random oracle model), the proofs are analogue to Lemmas 4.28, 4.29 and 4.30.

To prove anonymity when executing ProveCred (Definition 4.39.2), we essentially need to show that the partially randomized signatures $(\hat{\sigma}_1^s, \hat{\sigma}_2^s)$ in ProveCred are computationally indistinguishable from fully randomized ones. We proceed similar to the selfless anonymity proof by Bichsel et al. [Bic+10]. In particular, we also show a successful adversary $\mathcal{A}$ against the anonymity game (Definition 4.38) breaks the XDDH assumption (Definition 3.6). Different to their model, we allow adversaries to corrupt users.

Lemma 4.47. *Let* Π *be the extended ACS from Construction 4.45. If the XDDH problem (Definition 3.6) is hard relative to the bilinear group generator* G *and* Π *offers traceability, a user stays anonymous when executing* $\Pi.\mathsf{ProveCred}$ *(Definition 4.39.2) in the random oracle model.*

First we give an intuition to the proof of Lemma 4.47. The adversary $\mathcal{A}'$ (Figure 4.2) chooses two of the n_U users uniformly at random and uses g^β from its XDDH challenge $(g, g^\alpha, g^\beta, g^{\alpha\beta+\gamma\delta}, \tilde{g})$ to create their reginfo. Afterwards $\mathcal{A}'$ simulates the view of $\mathcal{A}$ computationally indistinguishable to the original experiment Exp^{anon} until $\mathcal{A}$ selects the users for its challenge. If $\mathcal{A}$ does not select the previously chosen users $\mathcal{A}'$ stops and outputs a bit δ' chosen uniformly at random. Else, happening with non-negligible probability $\Omega(n^{-2})$, $\mathcal{A}'$ chooses one of these users uniformly at random and uses g^α and $g^{\alpha\beta+\gamma\delta}$ to create the challenge transcript. If $\delta = 0$, $\mathcal{A}$ wins with probability $\frac{1}{2} + \epsilon(n) + \mu(n)$, where ϵ corresponds to the advantage of $\mathcal{A}$ and μ is negligible, due to the computational indistinguishability from the real experiment. If $\delta = 1$, $\mathcal{A}$ obtains a transcript independent of the user secrets of either user and wins exactly with probability $\frac{1}{2}$. Since simulation succeeds with non-negligible probability, a non-negligible $\epsilon(n)$ would imply that $\mathcal{A}'$ distinguishes between both cases of δ and wins the XDDH game with non-negligible probability. This would break the XDDH assumption and finishes the proof by contradiction.

Proof. We prove anonymity when executing ProveCred under the XDDH assumption (Definition 3.6) in the random oracle model. Therefore we construct adversary $\mathcal{A}'$ (Figure 4.2) against the XDDH game (Figure 3.2) from an arbitrary ppt adversary $\mathcal{A}$ against the anonymity game (Definition 4.38) and show their winning probabilities are polynomially related. Let $\mathcal{A}$ be an arbitrary but fixed ppt adversary with $\Pr[\mathsf{Exp}^{anon}_{\Pi,\mathcal{A}}(n) = 1] = \frac{1}{2} + \epsilon(n)$, which, without loss of generality, always outputs a bit $b' \in \{0,1\}$ after a polynomial time. Further remember, adversaries in the XDDH game obtain a tuple $(g, g^\alpha, g^\beta, g^{\alpha\beta+\gamma\delta}, \tilde{g})$, where $\alpha, \beta, \gamma \in \mathbb{Z}_p^*$, $\delta \in \{0,1\}$ are chosen uniformly at random, and output a guess δ' for δ.

Let us first argue that the view of $\mathcal{A}$ is simulated computationally indistinguishable compared to the anonymity game (Definition 4.38), assuming the event E that $\{u_0, u_1\} = U^*$ holds in line 10 and $\delta = 0$ for line 12. For all non-corrupted users $u \in U \setminus U^*$ the view is simulated perfectly by looking up their usk and running the real protocols honestly. For $i \in \{0,1\}$ and $u_i^* \in U^*$ the usk is (implicitly) set to βr_i with $r_i \leftarrow \mathbb{Z}_p$, hence distributed identically to an output of U.Init. The corresponding upk is sufficient to create reginfo, as done in *runJoin*$(\cdot)$. Using reginfo, *runProveVrfyCred* can be simulated like the oracles *runProveVrfyNym* and *runRcvIssCred* by programming the random oracle $\mathcal{H}$. It is possible, however, that for challenge c, instance x and simulated announcement z an entry $((x,z),c') \in \mathcal{H}$ with $c \neq c'$ already exists. Since z is chosen uniformly at random from at least p elements (cf. simulator of generalized Schnorr protocols from Lemma 3.47) and the number of elements in $\mathcal{H}$ is polynomial, such a collision occurs with negligible probability. Consequently this difference to the original game can only be detected with negligible probability by $\mathcal{A}$. Furthermore we store transcripts output by *runProveVrfyCred* for a queried user $u \in U^*$ in $\mathcal{T}$ to simulate the *open* oracle. Since *show is traceable by the open manager* (Lemma 4.46) the real Open would output upk of u for these stored transcripts exactly as $\mathcal{A}'$ does. When $\mathcal{A}$ provides a valid transcript for which the simulated *open* outputs 0, a real system manager either could not have opened it herself (obviously contradicting traceability) or would have opened it to a user in U^* (since tracing information of other users $u \in U \setminus U^*$ is

Adversary $\mathcal{A}'(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, g^\alpha, g^\beta, g^{\alpha\beta+\gamma\delta}, \tilde{g})$

1 : Run $\Pi.\mathsf{Setup}(1^n)$ with $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ fixed and output $(\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi)$.

2 : Run $\mathsf{MGen}(\mathsf{pp})$ to obtain $\mathsf{osk} = (x, y)$, $\mathsf{opk} = (g, g^y, \tilde{g}, \tilde{g}^x, \tilde{g}^y)$ and $\mathsf{reg} = \epsilon$.

3 : Simulate $\mathcal{A}$ on input $\mathsf{pp}, \mathcal{U}_A, \mathcal{U}_\Phi, \mathsf{opk}$ until it outputs $1^{n_U}, 1^{n_I}$.

4 : Set $U := \{1, \ldots, n_U\}, \hat{U} := \emptyset, \mathcal{H} := \emptyset, \mathcal{T} := \emptyset, S_{\mathsf{nym}} := \emptyset, S_{\mathsf{cred}} := \emptyset, t^* := \epsilon$.

5 : Choose $u_0^* \leftarrow U, u_1^* \leftarrow U \setminus \{u_0^*\}$. Set $U^* := \{u_0^*, u_1^*\}$.

6 : For $u \in U \setminus U^*$ run $\mathsf{U.Init}(\mathsf{pp}, \mathsf{opk})$ to obtain $(\mathsf{usk}, \mathsf{upk})$.

7 : For $i \in \{0, 1\}$ choose and store $r_i \leftarrow \mathbb{Z}_p$ and set $\mathsf{upk} := (g^\beta)^{r_i}$ for user u_i^*.

8 : Simulate $\mathcal{A}$ on input $\mathsf{pp}, \mathsf{opk}$ and all upk until it outputs for each issuer i an ipk and corresponding 1^{ℓ_i} representing the number of attributes. Set $I := \{1, \ldots, n_I\}$.

9 : Answer oracle queries of $\mathcal{A}$ for $u \notin U^*$ exactly as in the real experiment (Definition 4.38). Answer oracle queries for $u \in U^*$ as in the following.

- $H(x)$: If $(x, y) \in \mathcal{H}$ exists return y. Else choose $y \leftarrow \mathbb{Z}_p$, add (x, y) to $\mathcal{H}$ and return y.
- *corruptUser*(u, upk'), where $u \in U^*$: Return $\perp$.
- *runCreateNym*(u), where $u \in U^*$: Choose $\mathsf{nym} \leftarrow \mathbb{G}_1$, add (nym, u) to S_{nym} and return nym.
- *runJoin*(u), where $u \in U^*$: Look up upk of u, choose $t \leftarrow \mathbb{Z}_p^*$, set $\mathsf{reginfo} := (g^t, (g^x(\mathsf{upk}^y)^t)$ for u and return 1.
- *runProveVrfyNym*(u, v, nym), where $u \in U^*$, $v \in U \cup I$ and $(\mathsf{nym}, u) \in S_{\mathsf{nym}}$: Choose $c \leftarrow \mathbb{Z}_p$, run $(\mathsf{ProveNym}, \mathsf{VrfyNym})$'s simulator (cf. Lemma 4.29) for instance x to obtain transcript (z, c, r), add $((x, z), c)$ to $\mathcal{H}$ and send (z, c, r) to v. If v outputs 1 return 1, and else return 0.
- *runRcvIssCred*$(u, i, \mathsf{nym}, (a_j)_{j=1}^{\ell})$, where $u \in U^*$, $i \in I$, $\ell = \ell_i$ and $(\mathsf{nym}, u) \in S_{\mathsf{nym}}$: Look up ipk related to i and create commitment parameters $pp_{\mathsf{iss}} := \mathsf{BlindInit}(\mathsf{pp}, \mathsf{ipk}, \{0\})$. Generate a commitment on 0 by $(C, (0, r)) \leftarrow \mathsf{Com}_1(\mathsf{pp}_{iss}, 0)$ and send C to i. Choose $c \leftarrow \mathbb{Z}_p$ and run $(\mathsf{RcvCred}, \mathsf{IssCred})$'s simulator (cf. Lemma 4.30) for instance x to obtain transcript (z, c, r). Add $((x, z), c)$ to $\mathcal{H}$ and send $((x, z), c, r)$ to i. Unblind the signature returned by $\mathsf{IssCred}$ using r as in $\mathsf{RcvCred}$ and check its validity under ipk on message $(0, a_1, \ldots, a_{\ell_i})$. If it is valid, choose a unique index j, set $\mathsf{cred} = ((a_1, \ldots, a_{\ell_i}), \cdot, \mathsf{ipk})$, add (cred, j, u) to S_{cred} and return j. Else return $\perp$.
- *runProveVrfyCred*$(u, v, \mathsf{nym}, j, \phi)$, where $u \in U^*$, $v \in U \cup I$, $(\mathsf{nym}, u) \in S_{\mathsf{nym}}$ and $\exists\mathsf{cred}$ with $(\mathsf{cred}, j, u) \in S_{\mathsf{cred}}$: Look up $\mathsf{reginfo} = (\hat{\sigma}_1, \hat{\sigma}_2)$ of u, choose $t \leftarrow \mathbb{Z}_p^*$ and set $(\hat{\sigma}') = (\hat{\sigma}_1^t, \hat{\sigma}_2^t)$. Choose $c \leftarrow \mathbb{Z}_p$, run $(\mathsf{ProveCred}, \mathsf{VrfyCred})$'s simulator (cf. Lemma 4.31) for instance x to obtain transcript (z, c, r). Add $((x, (\hat{\sigma}', z)), c)$ to $\mathcal{H}$, send $t = ((\hat{\sigma}', z), c, r)$ to v and add (t, u) to $\mathcal{T}$. If v outputs 1 output 1 and else 0.
- *Open*(t), where $t \neq t^*$: If there is an entry $(t, u) \in \mathcal{T}$ return upk of user u. Check the validity of t using $\mathsf{VrfyCred}$. If t is invalid return $\perp$. Else run $\mathsf{Open}(t)$. If it outputs upk of user in U^* or $\perp$ output 0. Else return output of $\mathsf{Open}(t)$.

10 : Simulate $\mathcal{A}$ on input $\mathsf{pp}, \mathsf{opk}$, all upk and ipk until it outputs for $k \in \{0, 1\}$ $(u_k, j_k), \mathsf{ipk}, \phi \in \mathcal{U}_\Phi$ and $i \in I$, such that u_k joined the system and $\exists\mathsf{cred}_k = ((a_0, a_1, \ldots, a_{\ell_i}), \cdot, \mathsf{ipk})$ with $(\mathsf{cred}_k, j_k, u_k) \in S_{\mathsf{cred}}$ and $\phi(a_1, \ldots, a_{\ell_i}) = 1$.

11 : If $\{u_0, u_1\} \neq U^*$ choose $\delta' \leftarrow \{0, 1\}$ uniformly at random, output δ' and stop the simulation. Else run *corruptUser*(u, upk) for all $u \in U \setminus U^*$ (w.l.o.g. assume any adversary would do so).

12 : Choose new pseudonym $\mathsf{nym} \leftarrow \mathbb{G}_1$ (*not* added to S_{nym}), $b \leftarrow \{0, 1\}$ and set $(\hat{\sigma}_1', \hat{\sigma}_2') := (g^\alpha, ((g^\alpha)^x (g^{\alpha\beta+\gamma\delta})^{r_b y}))$. Simulate the remaining part of $(\mathsf{ProveCred}, \mathsf{VrfyCred})$ protocol as in the corresponding oracle to obtain transcript t, store it in t^* and send it to $\mathcal{A}$.

13 : Simulate $\mathcal{A}$ until it outputs bit b'. If $b = b'$ holds output 0, else output 1.

Figure 4.2: Adversary $\mathcal{A}'$ for XDDH experiment from $\mathcal{A}$ for anonymity when executing $\mathsf{ProveCred}$

known by $\mathcal{A}$'). The latter breaks the assumed traceability as well, because it requires a valid transcript not in $\mathcal{T}$ but opened to a user in U^*. Therefore $\mathcal{A}$ detects this difference to the original experiment with negligible probability only. Next, assuming $\delta = 0$ and $\{u_0, u_1\} = U^*$ in line 10, the challenge transcript t from line 12 is distributed exactly as in the real experiment. This holds by $\hat{\sigma}' = (g^\alpha, (g^\alpha)^x((g^{\alpha\beta})^{r_b y})) = (g^\alpha, (g^x g^{\beta r_b y})^\alpha)$, where α is distributed uniformly at random in $\mathbb{Z}_p^*$ by the XDDH setup and only used once to create this particular transcript. Finally, assuming event E (i.e. $\{u_0, u_1\} = U^*$ in line 10), without loss of generality an adversary $\mathcal{A}$ against the anonymity game does not corrupt a *challenged* user from U^* and detects the wrong behavior of the *corruptUser* oracle, unless it distinguished simulated and real experiment before. As stated previously the latter happens with negligible probability. Concluding, under event E and $\delta = 0$ (for line 12) $\mathcal{A}$ can distinguish simulated and real experiment (Definition 4.38) with negligible probability only. Therefore the winning probability of $\mathcal{A}$ in this simulated experiment is

$$\Pr[b = b' | E \wedge \delta = 0] = \Pr[\mathsf{Exp}^{anon}_{\Pi,\mathcal{A}}(n) = 1] + \mu(n) = \frac{1}{2} + \epsilon(n) + \mu(n), \tag{4.1}$$

where $|\mu|$ is a negligible function due to the negligible probability of detecting the stated differences.

Remember, that $\mathcal{A}'$ wins in the XDDH game, if $\delta' = \delta$ holds. We analyze

$$\begin{aligned}\Pr[\mathsf{XDDH}_{\mathcal{A}',\mathsf{G}}(n)] &= \sum_{i \in \{0,1\}} (\Pr[\delta = i \wedge \delta' = i \wedge E] + \Pr[\delta = i \wedge \delta' = i \wedge \neg E]) \\ &= \frac{1}{2} \sum_{i \in \{0,1\}} (\Pr[\delta' = i | \delta = i \wedge E] \Pr[E | \delta = i] + \Pr[\delta' = i | \delta = i \wedge \neg E] \Pr[\neg E | \delta = i]).\end{aligned} \tag{4.2}$$

We first prove that event E (i.e. $\mathcal{A}$ chooses $\{u_0, u_1\} = U^*$ in line 10) happens with non-negligible probability independently of δ. Note, that U^* is independent of $\mathcal{A}$'s choice of $\{u_0, u_1\}$ in line 10, up to the detection of one of the previously described differences in the oracles. This holds, since the users from U^* are chosen uniformly at random and their information obtained by $\mathcal{A}$ is distributed identically to any user from U. Furthermore, δ is used the first time in line 12 and therefore cannot affect event E in line 10. We have

$$\Pr[E] = \Pr[E | \delta = 0] = \Pr[E | \delta = 1] = \frac{2}{n_U(n_U - 1)} + \mu'(n) =: p(n), \tag{4.3}$$

where $p(n)$ is non-negligible, because $|\mu'|$ is negligible and n_U polynomial in n.

Under event $\neg E$, $\mathcal{A}'$ outputs a bit $\delta' \in \{0, 1\}$ chosen uniformly at random, i.e. for $i \in \{0, 1\}$

$$\Pr[\delta' = i | \delta = i \wedge \neg E] = \Pr[\delta' \leftarrow \{0, 1\} : \delta' = i] = \frac{1}{2}. \tag{4.4}$$

Conditioned on E and $\delta = 0$ $\mathcal{A}'$ outputs $\delta' = 0$, if and only if $\mathcal{A}$ correctly guesses b. Hence (4.1) yields

$$\Pr[\delta' = 0 | \delta = 0 \wedge E] = \Pr[b = b' | \delta = 0 \wedge E] = \frac{1}{2} + \epsilon(n) + \mu(n). \tag{4.5}$$

The last missing part of (4.2) is outputting $\delta' = 1$ under condition $\delta = 1$ and E. $\mathcal{A}'$ outputs 1 in line 13, if and only if $b \neq b'$ holds. However, under condition $\delta = 1$, the challenge transcript is distributed independently of the chosen user u_b and b by the randomization with $\delta\gamma$ for $\gamma \leftarrow \mathbb{Z}_p^*$ in the XDDH setup. Since we assume that $\mathcal{A}'$ always outputs a bit b' we have

$$\Pr[\delta' = 1 | \delta = 1 \wedge E] = \Pr[b \neq b' | \delta = 1 \wedge E] = \Pr[b \leftarrow \{0, 1\} : b \neq b'] = \frac{1}{2}. \tag{4.6}$$

We plug (4.3), (4.4), (4.5) and (4.6) into (4.2) to obtain

$$\begin{aligned}\Pr[\mathsf{XDDH}_{\mathcal{A}',\mathsf{G}}(n)] &= \frac{1}{2}\Big(\Big(\frac{1}{2}+\epsilon(n)+\mu(n)\Big)p(n)+\Big(1-p(n)\Big)\frac{1}{2}+\frac{1}{2}p(n)+\Big(1-p(n)\Big)\frac{1}{2}\Big)\\ &= \frac{1}{2}\Big(1+p(n)\epsilon(n)+p(n)\mu(n)\Big)\\ &= \frac{1}{2}+\frac{p(n)}{2}\epsilon(n)+\frac{p(n)}{2}\mu(n),\end{aligned}$$

where p is non-negligible and $|\mu|$ is negligible in n as previously shown. Hence, if ϵ were non-negligible $\mathcal{A}'$ would have a non-negligible advantage in the XDDH game. This contradicts the XDDH assumption (Definition 3.6) and implies that ϵ is negligible. Therefore a user stays anonymous when executing $\Pi.\mathsf{ProveCred}$, assuming traceability of the system and that the XDDH problem is hard relative to the bilinear group generator G. □

Thus, by Lemmas 4.28, 4.29, 4.30 and 4.47 we have the following theorem:

Theorem 4.48. *The credential system Π from Construction 4.45 is anonymous, if the XDDH problem (Definition 3.6) is hard relative to the bilinear group generator G and Π offers traceability.*

4.3.4.2 Soundness

To prove soundness of the extended ACS, we have to prove that it offers non-impersonation (Definition 4.41), credential unforgeability (Definition 4.42) and traceability (Definition 4.43). Note that the definitions of non-impersonation and credential unforgeability are the same for the basic ACS and extended ACS, except for the open oracle and the setup of related parameters. Thus, we can reuse the proofs of the basic ACS by generating the open parameters along the setup and simulating the oracle as in the soundness experiment. Therefore, we only have to prove traceability.

Lemma 4.49. *Assuming that the SDLP assumption (Definition 3.5) holds, if the Pointcheval-Sanders multi-message signature scheme is existentially unforgeable (Theorem 3.19), then Construction 4.45 offers traceability.*

Proof. Let Π be the extended ACS from Construction 4.45. Let $\mathcal{A}_{cred}$ be an adversary against Π. To bound the success probability of $\mathcal{A}_{cred}$, we want to construct two adversaries, $\mathcal{A}_{sdlp}$ and $\mathcal{A}_{sign}$ against the SDLP assumption and the unforgeability of the Pointcheval-Sanders signatures.

Before we construct those adversaries, we want to talk about repeating patterns in the constructions: In the extended ACS, we use non-interactive zero-knowledge proofs of knowledge by using the Fiat-Shamir heuristic on the Σ-Protocols described in the protocols. In the constructed adversaries, we want to simulate the environment for an adversary against the ACS. There, we have to simulate those protocols and the random oracle among other things. Assume that the random oracle maps into a space C. To simulate it, we use the following general approach: We define an initially empty set $\mathcal{H}$ and define the random oracle $H(x)$ to output a $y \leftarrow C$ and save (x, y) to C, if there is no $(x, \cdot) \in \mathcal{H}$. Else, i.e. for $H(x)$ there is a $(x, y) \in \mathcal{H}$, H outputs y. Then, suppose we want to generate a valid transcript of a non-interactive version of a Σ-Protocol with $(v, \cdot) \in R$ and challenge space C. We choose a $c \leftarrow C$ and use the honest-verifier simulator of the Σ-Protocol to generate a transcript (a, c, r). Then, we set $H(v||a) = c$, i.e. save $(v||a, c)$ to $\mathcal{H}$. Thus, we have a simulated transcript (a, c, r) and H stays consistent by remembering old values. Note that H and $\mathcal{H}$ stay the same for different protocols.

Furthermore, if we want to extract a witness from a Fiat-Shamir heuristic based on a Σ-Protocol $(\mathcal{P}, \mathcal{V})$ for an NP-relation R and a $v \in L_R$, we use the idea of the Forking Lemma

[BN06]: We use the forking algorithm to extract two transcripts on which we use the special soundness extractor from the Σ-Protocol to extract a witness. Since the forking algorithm only has a success probability of $acc \cdot (\frac{acc}{q} - \frac{1}{p})$, where acc is the success probability of the prover and q is an upper bound for the hash queries of the prover, we repeatedly try to use the forking algorithm until we succeed. Then, for some polynomial runtime $poly(n)$ of the prover, we have an expected runtime of $\mathsf{RT} = (acc \cdot (\frac{acc}{q} - \frac{1}{p}))^{-1} \cdot poly(n)$ of this process, since we repeat it that often in expectation. Since RT is boundable by a polynomial, the extraction takes expected polynomial time. As in the proof of Lemma 4.35, we exhaustively search for a witness to the proof with a runtime of $\mathcal{O}(p)$. Since this case is used only with probability $\frac{1}{p}$ (as in the proof of Lemma 4.35), the parallel process also has expected polynomial time, thus the extractor as a whole has expected polynomial time. Assume that the prover always queries the random oracle to compute the challenge for the non-interactive proof, because else she would only have negligible chance to guess it correctly. Then, if the success probability of the prover is greater than the knowledge error, the extractor always outputs two transcripts with the same announcement except for negligible cases, since the announcement is part of the input for the random oracle. If the success probability is equal to the knowledge error, the extractor outputs an error symbol.

With those two sub-constructions, we can construct the new adversaries. The idea is that $\mathcal{A}_{cred}$ wins in two cases: Either she outputs a proof which opens to $\perp$, which means that the trusted party has no corresponding tracing information, or Open outputs the upk of an honest user. We use the first case to construct an adversary against the SDLP assumption, while we use the latter to construct an adversary against the unforgeability of Pointcheval-Sanders signatures.

$\mathcal{A}_{sdlp}(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, \tilde{g}, a, \tilde{a})$

1 : Perform $\mathsf{Exp}^{Soundness}_{\Pi,\mathcal{A}_{cred}}(n)$ by taking the role of the challenger and simulating $\mathcal{A}_{cred}$ with the following exceptions:

- Instead of running $\mathsf{G}(1^n)$ in Setup, take the public parameters from the input.
- After $\mathcal{A}_{cred}$ returns n_U, choose a $j \leftarrow \{1, .., n_U\}$. Set $\mathsf{upk}_j = a$.

Answer all oracles as specified in the experiment except for the following:

- $runJoin(u)$: If $u = j$, then u sends $\tilde{\tau} = \tilde{a}^y$ to the system manager. Then, the trusted party computes σ as in the original experiment, sends it to u and adds $(a, \sigma, \tilde{\tau})$ to reg. If $u \neq j$, do the same as in the original experiment.
- $corruptUser(u)$: If $u = j$, output $\perp$. If $u \neq j$, do the same as in the original experiment.
- $runProveVrfyNym(u, v, \mathsf{nym})$: If $u = j$, simulate the non-interactive proof as described above. If $u \neq j$, do the same as in the original experiment.
- $runRcvIssCred(u, i, \mathsf{nym}, (a_i)_{i=1}^{\ell})$: If $u = j$, u chooses $C \leftarrow \mathbb{G}_1$ and sends it to v. Then, u engages in the non-interactive proof by simulating as described above. If $u \neq j$, do the same as in the original experiment.
- $runProveVrfyCred(u, v, \mathsf{nym}, credId, \phi)$: If $u = j$, compute and send $\hat{\sigma}'$ and σ' as in the original experiment. Then, u engages in the non-interactive proof by simulating as described above. If $u \neq j$, do the same as in the original experiment.

2 : $\mathcal{A}_{cred}$ outputs $(\mathsf{ipk}, \mathsf{nym}, \phi)$, $u \in \hat{U}$ and an honest verifier $v \in (U \cup I) \setminus (\hat{U} \cup \hat{I})$.

3 : Execute $(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}, \mathsf{reginfo}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi)) \rightarrow b$ between u and v, while $\mathcal{A}_{cred}$ controls u. Let t be the transcript of this interaction. If $b = 0$ or any of the winning conditions in the real experiment do not hold, output $\perp$.

4 : If $\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}, t) \neq \mathsf{upk}_j$, output $\perp$.

5 : Extract a $w = (\mathsf{usk}', \mathsf{psk}', \mathsf{cred}', \mathsf{reginfo}')$ from the argument of knowledge as described above.

6 : Output usk'.

Let PS be the Pointcheval-Sanders signature scheme. Construct an adversary $\mathcal{A}_{sign}$ against the unforgeability of PS.

$\mathcal{A}^{\mathcal{O}}_{sign}(\mathsf{pp}_{sign}, \mathsf{pk}_{sign})$

Perform $\mathsf{Exp}^{Soundness}_{\Pi,\mathcal{A}_{cred}}(n)$ by taking the role of the challenger and simulating $\mathcal{A}_{cred}$ with the following exceptions:

- Instead of running $\mathsf{G}(1^n)$ in Setup, take the public parameters from the input.
- Instead of generating opk, set $\mathsf{opk} = \mathsf{pk}_{sign}$.

1 : Answer all oracles as specified in the experiment except for $runJoin(u)$: If $u \notin \hat{U}$, do the interaction as normal, but instead of computing σ, query $\mathcal{O}(\mathsf{usk}_u)$. If $u \in \hat{U}$, u sends a $\tilde{\tau}$ to the trusted party as before, then both participate in the argument of knowledge and the system manager checks whether $e(\mathsf{upk}, \tilde{Y}) = e(g, \tilde{\tau})$. If the trusted party accepts, it extracts a usk' from the PoK as described above, and then sends $\sigma \leftarrow \mathcal{O}(\mathsf{usk}')$ to u. At last, it saves $(\mathsf{upk}, \sigma, \tilde{\tau})$ to reg.

2 : $\mathcal{A}_{cred}$ outputs $(\mathsf{ipk}, \mathsf{nym}, \phi)$, $u \in \hat{U}$ and an honest verifier $v \in (U \cup I) \backslash (\hat{U} \cup \hat{I})$.

3 : Execute $(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}, \mathsf{reginfo}) \leftrightarrow \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, \phi)) \rightarrow b$ between u and v, while $\mathcal{A}_{cred}$ controls u. Let t be the transcript of this interaction. If $b = 0$ or any of the winning conditions in the real experiment do not hold, output $\perp$.

4 : If $\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}, t) = \perp$, extract a $(\mathsf{usk}', \mathsf{psk}', \mathsf{cred}', \mathsf{reginfo}')$ from the argument of knowledge as described above.

5 : Output $(\mathsf{usk}', \mathsf{reginfo}')$.

Let $Frame$ be the event that Open outputs the upk of an honest user when given the transcript of the challenge proof. Let $Trace$ be the event that Open outputs $\perp$ with the same input. Then, we have that

$$\Pr[\mathsf{Game}^{Trace}_{\Pi,\mathcal{A}_{cred}}(n) = 1] = \Pr[\mathsf{Game}^{Trace}_{\Pi,\mathcal{A}_{cred}}(n) = 1 \wedge Frame] + \Pr[\mathsf{Game}^{Trace}_{\Pi,\mathcal{A}_{cred}}(n) = 1 \wedge Trace],$$

as in the case of $\neg Frame \wedge \neg Trace$ the output of Open would be the upk of a dishonest user, which would mean that the adversary loses.

Let $Guess$ be the event that the j that $\mathcal{A}_{sdlp}$ guesses is the u that Open outputs at the end of $\mathcal{A}_{sdlp}$. Now, consider when $\mathcal{A}_{sdlp}$ wins. This happens exactly in the event of $\mathsf{Game}^{Trace}_{\Pi,\mathcal{A}_{cred}}(n) = 1 \wedge Frame \wedge Guess$. This is true due to the following reasons: $\mathcal{A}_{cred}$ is perfectly simulated by definition of $\mathcal{A}_{sdlp}$, except for the $corruptUser$ oracle when given j, but then $\mathcal{A}_{sdlp}$ would lose anyway, since the output of $\mathcal{A}_{cred}$ would not correspond to a and $\tilde{a}$. Furthermore, the extracted usk' is the discrete logarithm of a: since Open outputs upk_j, there is an entry $(\mathsf{upk}_j, \cdot, \tilde{\tau}) = (a, \cdot, \tilde{a}^y)$ in reg with $e(\hat{\sigma}'_1, \tilde{a}^y) = e(\hat{\sigma}'_2, \tilde{g})e(\hat{\sigma}'_1, \tilde{X}')^{-1}$. Suppose that $a = g^b$. Then, we have that $e(\hat{\sigma}'_1, \tilde{Y})^b = e(\hat{\sigma}'_2, \tilde{g})e(\hat{\sigma}'_1, \tilde{X}')^{-1}$. Thus, the adversary convinces the verifier that it knows a d, such that $g^d = a$, meaning that the usk' that we extract has the same property. Therefore, if $\mathcal{A}_{sdlp}$ is able to extract a witness, it is able to compute the discrete logarithm of a.

Note that $\mathcal{A}_{sdlp}$ makes use of an extractor with only expected polynomial time. By a similar argument to Lemma 4.34, we conclude that $\mathcal{A}_{sdlp}$ has expected polynomial time as well. Then, we can use Lemma 4.33 to get a ppt adversary $\mathcal{A}'_{sdlp}$ with success probability greater than half of the success probability of $\mathcal{A}_{sdlp}$.

We conduct a similar analysis for $\mathcal{A}_{sign}$. Let $Guess$ be defined analogously. Then, we have

$$\Pr[\mathsf{Sig\text{-}forge}_{\mathsf{PS},\mathcal{A}_{sign}}(1^n) = 1] = \Pr[\mathsf{Game}^{Trace}_{\Pi,\mathcal{A}_{cred}}(n) = 1 \wedge Trace \wedge Guess],$$

since in the event of $\mathsf{Game}^{Trace}_{\Pi,\mathcal{A}_{cred}}(n) = 1 \wedge Trace \wedge Guess$ the adversary $\mathcal{A}_{cred}$ is able to convince the verifier in $(\mathsf{ProveCred}, \mathsf{VrfyCred})$ of possession of some witness, such that the transcript opens to $\perp$. This means we can extract some $(\mathsf{usk}', \cdot, \cdot, \mathsf{reginfo}')$, such that $\mathsf{reginfo}'$ is a signature on the message usk, that when verified with $\mathsf{opk} = \mathsf{pk}_{sign}$ is valid. Since $Guess$ is independent from the other events, we have that

$$\Pr[\mathsf{Sig\text{-}forge}_{\mathsf{PS},\mathcal{A}_{sign}}(1^n) = 1] = \frac{1}{n_U} \cdot \Pr[\mathsf{Game}^{Trace}_{\Pi,\mathcal{A}_{cred}}(n) = 1].$$

As before, we have to prove that $\mathcal{A}_{sign}$ has expected polynomial time, since she makes use of extractors with expected polynomial time. But since every extractor in $\mathcal{A}_{sign}$ has expected polynomial runtime, the sum of all runtimes of the extractors is polynomial as well. Thus, $\mathcal{A}_{sign}$ has expected polynomial time as well and we can use Lemma 4.33 to get a ppt adversary $\mathcal{A}'_{sign}$ with $\Pr[\textsf{Sig-forge}_{\textsf{PS},\mathcal{A}'_{sign}}(n) = 1] \geq \frac{1}{2} \cdot \Pr[\textsf{Sig-forge}_{\textsf{PS},\mathcal{A}_{sign}}(n) = 1]$.

Putting everything together, we have

$$\begin{aligned}\Pr[\textsf{Game}^{Trace}_{\Pi,\mathcal{A}_{cred}}(n) = 1] \leq & 2n_U \cdot \big(\Pr[(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n), x \leftarrow \mathbb{Z}_p, g \leftarrow \mathbb{G}_1 \setminus \{1\},\\ & \tilde{g} \leftarrow \mathbb{G}_2 \setminus \{1\}, y \leftarrow \mathcal{A}(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, \tilde{g}, g^x, \tilde{g}^y) : g^y = g^x]\\ & + \Pr[\textsf{Sig-forge}_{\textsf{PS},\mathcal{A}_{sign}}(n) = 1]\big),\end{aligned}$$

where n_U is boundable by a polynomial.

Therefore, if the SDLP assumption (Definition 3.5) holds and the Pointcheval-Sanders multi-message signature scheme is existentially unforgeable (Theorem 3.19), then $\Pr[\textsf{Game}^{Trace}_{\Pi,\mathcal{A}}(n) = 1]$ is negligible for all adversaries $\mathcal{A}$. □

Thus, we have that the extended ACS is sound.

Theorem 4.50. *Assuming that the SDLP assumption holds (Definition 3.5), if the generalized Pederson commitment scheme is computationally binding (Lemma 3.27) and if the multi-message Pointcheval-Sanders signature scheme is existentially unforgeable (Theorem 3.19), then the extended credential system from Construction 4.45 is sound.*

4.4 Attribute-Based Anonymous Credential and Reputation System

In Sections 4.2.3 and 4.3, we developed an anonymous credential system for arbitrary policies that supports identity escrow in case of misuse. The next step is to extend the system presented in Section 4.3 such that it also includes the features of a *reputation system*. To this end, we integrate the functionalities of creating ratings and public linkability. We call this system an *attribute-based anonymous credential and reputation system* (ACRS).

4.4.1 Definition

To formally define what an attribute-based anonymous credential and reputation system is, we further extend our extended ACS given in Definition 4.37 by algorithms and protocols to support the features an reputation system provides.

The extensions to Definition 4.37 given next introduce a new role in the system, that is a *review token issuer*. This entity often is a service provider that verifies that a user bought a product by issuing a *review token*. To support this feature, we add an issuance protocol for review tokens. Moreover, we add an algorithm enabling users to create a rating and one to verify a given rating. Finally, we add an algorithm to link two ratings created by the same user. This algorithm prevents that a user is able to rate some item or service more than once.

Definition 4.51. An *attribute-based credential and reputation system (CRS)* consists of the following (ppt) algorithms and interactive protocols:

- $\mathsf{Setup}(1^n)$: On input security parameter 1^n, it outputs public parameters pp with $|\mathsf{pp}| \geq n$, an attribute universe $\mathcal{U}_A$ and a predicate universe $\mathcal{U}_\Phi \subseteq \{\phi \mid \phi : \mathcal{U}_A^k \to \{0,1\}, k \in \mathbb{N}\}$. Additionally, it (implicitly) outputs application-dependent auxiliary parameters $\mathsf{pp}_{\mathrm{aux}}$.
- $\mathsf{MGen}(\mathsf{pp})$: On input public parameters pp, it outputs a tuple $(\mathsf{msk}, \mathsf{mpk}, \mathsf{reg})$ of master secret key, master public key and a user registry. The key pair $(\mathsf{msk}, \mathsf{mpk})$ also contains the opening secret key osk and the opening public key opk.

- $\mathsf{U.Init}(\mathsf{pp}, \mathsf{opk})$: On input public parameters pp, and open public key, it outputs a user secret key usk and a user public key upk.
- $\mathsf{I.Init}(\mathsf{pp}, 1^\ell)$: On input public parameters pp and 1^ℓ with $\ell \in \mathbb{N}$, it outputs an issuer public key ipk and secret key isk. The number ℓ denotes the number of attributes supported by the issuer.
- $\mathsf{R.Init}(\mathsf{pp})$: On input public parameters pp it outputs a review secret key rsk and a review public key rpk.
- $\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})$: On input public parameters pp and user secret usk, it outputs a pseudonym nym and corresponding pseudonym secret psk.
- $(\mathsf{Join}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{usk}), \mathsf{MJoin}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{osk}, \mathsf{reg}))$ is an interactive protocol on common inputs public parameters pp, open public key opk, user public key upk. Secret inputs are user secret key usk for the user and open secret key osk and user registry reg for the (trusted) system manager. At the end of the interaction, MJoin either outputs an error symbol $\perp$ or the updated reg'. Join outputs a $\mathsf{reginfo}$ containing registration information.
- $(\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{usk}, \mathsf{psk}), \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym}))$ is an interactive protocol on common inputs public parameters pp and pseudonym nym, where the user has user secret usk and pseudonym secret psk as private input. After execution the verifier outputs a bit $b \in \{0, 1\}$ and we interpret 1 as accepting, 0 as denying.
- $(\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^\ell, \mathsf{usk}, \mathsf{psk}), \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^\ell, \mathsf{isk}))$ is an interactive protocol on common inputs public parameters pp, pseudonym nym and attributes $(a_1, \ldots, a_\ell) \in \mathcal{U}_A^\ell$. The user has user secret usk, pseudonym secret psk and issuer public key ipk as additional input, whereas the issuer has secret key isk. After the interaction, the user outputs (locally) either the credential cred corresponding to ipk over her user secret usk and the attributes $(a_1, \ldots, a_\ell)$ or a failure symbol represented by $\perp$.
- $(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{cred}, \mathsf{reginfo}), \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi))$ is an interactive protocol on common inputs public parameters pp, open public key opk, issuer public key ipk, pseudonym nym, and a predicate $\phi \in \mathcal{U}_\Phi$, where the user has user secret usk, pseudonym secret psk, credential cred on attributes $(a_1, .., a_\ell) \in \mathcal{U}_A^\ell$ and register information $\mathsf{reginfo}$ as private input. After execution the verifier outputs a bit $b \in \{0, 1\}$ and we interpret 1 such that the credential cred satisfies the predicate ϕ and the verifier accepts, 0 as denying.
- $\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}, t)$: On input open public key opk, open secret key osk, registry reg and a transcript t, it outputs a user public key upk or an error symbol $\perp$.
- $(\mathsf{RcvToken}(\mathsf{pp}, \mathsf{rpk}, \mathsf{nym}, \mathsf{item}, \mathsf{usk}, \mathsf{psk}), \mathsf{IssToken}(\mathsf{pp}, \mathsf{nym}, \mathsf{item}, \mathsf{rsk}))$ is an interactive protocol with common inputs public parameter pp, pseudonym nym and an item identifier item. The user has rating public key rpk, user secret key usk and pseudonym secret key psk as additional input, whereas the issuer has the rating secret key rsk as additional input. After the interaction, the user outputs a review token token corresponding to the item identified by item or a failure symbol $\perp$.
- $\mathsf{Rate}(\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{item}, \mathsf{reginfo}, \mathsf{token}, \mathsf{usk}, m)$: On inputs public parameters pp, master public key mpk, review public key rpk, item identifier item, registration information $\mathsf{reginfo}$, review token token and a message m. It outputs a rating rating.
- $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{item}, \mathsf{rating}, m)$: On inputs public parameters pp, master public key mpk, review public key rpk, item identifier item, rating rating and message m it checks whether

rating is a valid rating for the item identified by item and contains the message m. It outputs 1 if this check holds and 0 otherwise.

- $\mathsf{Link}(\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{rating}, \mathsf{rating}^*)$: On inputs public parameter pp, master public key mpk, review public key rpk and two ratings rating and rating^* it checks whether these two ratings can be linked. It outputs 1 if this check holds and 0 otherwise.

For completeness most of the algorithms are repeated. The usage of these additional algorithms in combination with the algortihms of an extended ACS is described in detail in Section 4.4.2, where we introduce the construction of our system.

Correctness We continue to state the correctness of the system defined in Definition 4.51. We say an attribute-based credential and reputation system is *correct* if ($\mathsf{Setup}, \mathsf{MGen}, \mathsf{U.Init}, \mathsf{I.Init}, \mathsf{CreateNym}, (\mathsf{Join}, \mathsf{MJoin}), (\mathsf{ProveNym}, \mathsf{VrfyNym}), (\mathsf{RcvCred}, \mathsf{VrfyNym}), (\mathsf{RcvCred}, \mathsf{IssCred}), (\mathsf{ProveCred}, \mathsf{VrfyCred}), \mathsf{Open}$) is a correct extended credential system as defined in Section 4.3.1 and additionally the following holds:

For all $n, c \in \mathbb{N}$
all $(\mathsf{pp}, \cdot, \cdot) \in [\mathsf{Setup}(1^n)]$
all $(\mathsf{msk}, \mathsf{mpk}) \in [\mathsf{MGen}(\mathsf{pp})]$
all $(\mathsf{usk}_i, \mathsf{upk}_i) \in \mathsf{U.Init}$ with $1 \leq i \leq c$
all $j \in \{1, \ldots, c\}$
all $(\mathsf{rsk}, \mathsf{rpk}) \in \mathsf{R.Init}$
all $(\mathsf{nym}, \mathsf{psk}) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})]$
all $(\mathsf{nym}^*, \mathsf{psk}^*) \in [\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk}_j)]$
all $\mathsf{item} \in \{0,1\}^*$
all $m, m^* \in \{0,1\}^*$
all token output by $\mathsf{RcvToken}(\mathsf{pp}, \mathsf{rpk}, \mathsf{nym}, \mathsf{item}, \mathsf{usk}, \mathsf{psk}) \leftrightarrow \mathsf{IssToken}(\mathsf{pp}, \mathsf{nym}, \mathsf{item}, \mathsf{rsk})$
all token^* output by $\mathsf{RcvToken}(\mathsf{pp}, \mathsf{rpk}, \mathsf{nym}^*, \mathsf{item}, \mathsf{usk}_j, \mathsf{psk}^*) \leftrightarrow \mathsf{IssToken}(\mathsf{pp}, \mathsf{nym}^*, \mathsf{item}, \mathsf{rsk})$
all $\mathsf{reginfo}_i$ output by $(\mathsf{Join}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{usk}) \leftrightarrow \mathsf{MJoin}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{osk}, \mathsf{reg}_i) \to \mathsf{reg}_{i+1})$ for $1 \leq i \leq c$
all rating output by $\mathsf{Rate}(\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{item}, \mathsf{reginfo}_j, \mathsf{token}, \mathsf{usk}_j, m)$
all rating^* output by $\mathsf{Rate}(\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{item}, \mathsf{reginfo}_j, \mathsf{token}, \mathsf{usk}_j, m^*)$

1. **Honestly generated ratings are valid:**

$$\Pr[\mathsf{Vrfy}(\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{item}, \mathsf{rating}, m) = 1] = 1 - \mu(|\mathsf{pp}|)$$

holds for some negligible function μ, where the probability is taken over the coin tosses of the algorithms and their inputs' distribution.

2. **Two ratings for the same item by one user can be linked:**

$$\Pr[\mathsf{Link}(\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{rating}, \mathsf{rating}^*) = 1] = 1 - \mu(|\mathsf{pp}|)$$

holds for some negligible function μ, where the probability is taken over the coin tosses of the algorithms and their inputs' distribution.

3. **Ratings are traceable by the open manager:**

$$1 - \mu(|pp|) = \Pr[\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}_{c+1}, \mathsf{rating}) = \mathsf{upk}_j]$$

holds for some negligible function μ, where the probability is taken over the coin tosses of the algorithms and their inputs' distribution.

4.4.2 Construction

In this section, we present the construction of our attribute-based anonymous credential and reputation system. The construction given next adapts the concepts used by Blömer, Juhnke, and Kolb [BJK15] in their reputation system to introduce rating and linking of ratings to our extended ACS (Construction 4.45). In contrast to the construction given in [BJK15], we use credentials to show membership of the system as well as to verify the purchase of an item, and transferred the linking of two ratings into our type 3 bilinear group setting. The use of credentials is a logical consequence of the infrastructure we build up with our credential system. This allows us to add the reputation system features with little effort.

We start with the construction and continue with the use of the thus created system.

Construction 4.52. Let G be a type 3 bilinear group generator (Definition 3.2), let $\Pi_{\text{com}} = (\Pi_{\text{com}}.\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be the Pedersen commitment scheme (Construction 3.23) and let $\Pi_{\text{sign}} = (\Pi_{\text{sign}}.\mathsf{Setup}, \mathsf{Gen}, \Pi_{\text{sign}}.\mathsf{Sign}, \Pi_{\text{sign}}.\mathsf{Vrfy})$ be the Pointcheval-Sanders signature scheme (Construction 3.17). Further, let $H_{\mathbb{G}} \colon \{0,1\}^* \to \mathbb{G}$ be a hash function hashing into a group $\mathbb{G}$ (Theorem 3.10).

- $\mathsf{Setup}(1^n)$: On input security parameter 1^n, Setup generates a type 3 bilinear group $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e) \leftarrow \mathsf{G}(1^n)$ and pseudonym public parameters $\mathsf{pp}_{\text{nym}} \coloneqq (g_{\text{nym}}, h_{\text{nym}}, p, \mathbb{G}_1)$ with $g_{\text{nym}} \leftarrow \mathbb{G}_1 \setminus \{1\}$ and $h_{\text{nym}} \leftarrow \mathbb{G}_1$. Additionally, it creates pp_{aux} by generating public parameters of Nguyen accumulator (Section 3.12.3) based on the bilinear group, to enable membership proofs in predicates (Section 4.1.3.3). It returns $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$.
- $\mathsf{MGen}(\mathsf{pp})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$, MGen returns $(\mathsf{mpk}, \mathsf{msk}) \coloneqq ((\mathsf{opk}, b), \mathsf{osk}, \mathsf{reg})$ with opening key pair $(\mathsf{opk}, \mathsf{osk}) \leftarrow \Pi_{\text{sign}}.\mathsf{Gen}_1(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, public linkability basis $b \leftarrow \mathbb{G}_2$ and registry $\mathsf{reg} \coloneqq \epsilon$.
- $\mathsf{U.Init}(\mathsf{pp}, \mathsf{opk})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$ and opener public key $\mathsf{opk} = (g_{\text{opk}}, Y_{\text{opk}}, \tilde{g}_{\text{opk}}, \tilde{X}_{\text{opk}}, \tilde{Y}_{\text{opk}})$, $\mathsf{U.Init}$ chooses user secret key $\mathsf{usk} \leftarrow \mathbb{Z}_p$, sets public key $\mathsf{upk} \coloneqq g_{\text{opk}}^{\mathsf{usk}}$ and returns $(\mathsf{usk}, \mathsf{upk})$.
- $\mathsf{I.Init}(\mathsf{pp}, 1^\ell)$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$ and number $\ell \in \mathbb{N}$, $\mathsf{I.Init}$ generates and returns issuer key pair $(\mathsf{ipk}, \mathsf{isk}) \leftarrow \Pi_{\text{sign}}.\mathsf{Gen}_{\ell+1}(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ with $\mathsf{ipk} = (g, Y_0, Y_1, \ldots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_0, \tilde{Y}_1, \ldots \tilde{Y}_\ell)$.
- $\mathsf{R.Init}(\mathsf{pp})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$, $\mathsf{R.Init}$ generates and returns a review token key pair $(\mathsf{rpk}, \mathsf{rsk}) \leftarrow \mathsf{I.Init}(\mathsf{pp}, 1^1)$.
- $\mathsf{CreateNym}(\mathsf{pp}, \mathsf{usk})$: On input public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$ and user secret usk, $\mathsf{CreateNym}$ generates and returns $(\mathsf{nym}, \mathsf{psk}) \leftarrow \mathsf{Com}_1(\mathsf{pp}_{\text{nym}}, \mathsf{usk})$ with $\mathsf{psk} = (\mathsf{usk}, d)$.
- $(\mathsf{Join}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{usk}), \mathsf{MJoin}(\mathsf{pp}, \mathsf{opk}, \mathsf{upk}, \mathsf{osk}, \mathsf{reg}))$ is an interactive protocol with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$, opener public key $\mathsf{opk} = (g_{\text{opk}}, Y_{\text{opk}}, \tilde{g}_{\text{opk}}, \tilde{X}_{\text{opk}}, \tilde{Y}_{\text{opk}})$ and $\mathsf{pp}_{\text{nym}} = (g_{\text{nym}}, h_{\text{nym}}, p, \mathbb{G}_1)$. The system manager parses the opener secret key osk as (x, y). The user computes $\tilde{\tau} \coloneqq \tilde{Y}_{\text{opk}}^{\mathsf{usk}}$, sends $\tilde{\tau}$ to the system manager and runs a Σ-protocol of form
$$\mathsf{PK}\Big\{(\mathsf{usk}) : \mathsf{upk} = g_{\text{opk}}^{\mathsf{usk}}\Big\}.$$
The system manager proceeds, if and only if she accepts in this proof *and* $e(\mathsf{upk}, \tilde{Y}_{\text{opk}}) = e(g_{\text{opk}}, \tilde{\tau})$ holds. If there is an entry $(\mathsf{upk}, \sigma, \cdot)$ in reg, the system manager returns σ and stops. Else she chooses $u \leftarrow \mathbb{Z}_p^*$, computes $\sigma \coloneqq (\sigma_1, \sigma_2) \coloneqq (g_{\text{opk}}^u, (g_{\text{opk}}^x \cdot \mathsf{upk}^y)^u)$, sends σ to the user, adds the entry $(\mathsf{upk}, \sigma, \tilde{\tau})$ to reg and (locally) outputs the updated reg. Then,

the user checks the signature's validity via $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{opk}, \sigma)$. If the check fails, she (locally) outputs $\bot$, and else registration information $\mathsf{reginfo} = (\sigma, \mathsf{opk})$.

- $(\mathsf{ProveNym}(\mathsf{pp}, \mathsf{nym}, \mathsf{psk}), \mathsf{VrfyNym}(\mathsf{pp}, \mathsf{nym}))$ is an interactive protocol with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$ and $\mathsf{pp}_{\mathrm{nym}} = (g_{\mathrm{nym}}, h_{\mathrm{nym}}, p, \mathbb{G}_1)$. The prover parses the pseudonym secret psk as (usk, d) and performs an interactive zero-knowledge argument of knowledge of form $\mathsf{PK}\Big\{(\mathsf{usk}, d) : \mathsf{nym} = g_{\mathrm{nym}}^d h_{\mathrm{nym}}^{\mathsf{usk}}\Big\}$ with the verifier.

- $(\mathsf{RcvCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{usk}, \mathsf{psk}), \mathsf{IssCred}(\mathsf{pp}, \mathsf{nym}, (a_i)_{i=1}^{\ell}, \mathsf{isk}))$ is an interactive protocol with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$, $\mathsf{pp}_{\mathrm{nym}} = (g_{\mathrm{nym}}, h_{\mathrm{nym}}, p, \mathbb{G}_1)$ and issuer public key $\mathsf{ipk} = (g, Y_0, \ldots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_0, \ldots, \tilde{Y}_\ell)$. The receiver parses the pseudonym secret psk as (usk, d) and generates $pp_{\mathrm{iss}} = (g, Y_0, p, \mathbb{G}_1) := \mathsf{BlindInit}(\mathsf{pp}, \mathsf{ipk}, \{0\})$ (ignoring $\mathsf{pp}_{\mathrm{nym}}$ in pp) and computes $(C, (\mathsf{usk}, r)) \leftarrow \mathsf{Com}_1(\mathsf{pp}_{\mathrm{iss}}, \mathsf{usk})$. Then, she sends C to the issuer and runs a Σ-protocol of form

$$\mathsf{PK}\Big\{(\mathsf{usk}, d, r) : \mathsf{nym} = g_{\mathrm{nym}}^d h_{\mathrm{nym}}^{\mathsf{usk}} \wedge C = g^r Y_0^{\mathsf{usk}}\Big\}$$

with the issuer. As in $(\mathsf{BlindRcv}, \mathsf{BlindIssue})$, the issuer now chooses $u \leftarrow \mathbb{Z}_p^*$, computes $(\sigma_1', \sigma_2') := (g^u, (g^x \cdot C \cdot \prod_{i=1}^{\ell} Y_i^{a_i})^u)$ and sends (σ_1', σ_2') to the receiver. The receiver computes $\sigma = (\sigma_1', \sigma_2' \cdot (\sigma_1')^{-r})$, and checks the signature's validity via $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{ipk}, (\mathsf{usk}, a_1, \ldots, a_\ell), \sigma)$. If the check fails, she (locally) outputs $\bot$ and else a credential $\mathsf{cred} = (\sigma, (a_1, \ldots, a_\ell), \mathsf{ipk})$.

- $(\mathsf{RcvToken}(\mathsf{pp}, \mathsf{rpk}, \mathsf{nym}, \mathsf{item}, \mathsf{usk}, \mathsf{psk}), \mathsf{IssToken}(\mathsf{pp}, \mathsf{nym}, \mathsf{item}, \mathsf{rsk}))$ is an interactive protocol, where $\mathsf{RcvToken}$ performs the same steps as $\mathsf{RcvCred}$ except that it uses $\mathsf{rpk} = (g, Y_0, Y_1, \tilde{g}, \tilde{Y}_0, \tilde{Y}_1)$ instead of ipk and $H_{\mathbb{Z}_p}(\mathsf{item})$ instead of $(a_i)_{i=1}^{\ell}$. Similarly, $\mathsf{IssToken}$ performs the same steps as $\mathsf{IssCred}$ except that it uses rsk instead of isk, and again, $H_{\mathbb{Z}_p}(\mathsf{item})$ instead of $(a_i)_{i=1}^{\ell}$. After unblinding the computed signature σ, the user checks the signature's validity via $\Pi_{\mathrm{sign}}.\mathsf{Vrfy}(\mathsf{rpk}, (\mathsf{usk}, H_{\mathbb{Z}_p}(\mathsf{item})), \sigma)$. If the check fails, she (locally) outputs $\bot$ and else a review token $\mathsf{token} = (\sigma, \mathsf{item}, \mathsf{rpk})$.

- $(\mathsf{ProveCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi, \mathsf{usk}, \mathsf{psk}, \mathsf{reginfo}, \mathsf{cred}), \mathsf{VrfyCred}(\mathsf{pp}, \mathsf{opk}, \mathsf{ipk}, \mathsf{nym}, \phi))$ is an interactive protocol with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\mathrm{nym}})$, $\mathsf{pp}_{\mathrm{nym}} = (g_{\mathrm{nym}}, h_{\mathrm{nym}}, p, \mathbb{G}_1)$, opener public key $\mathsf{opk} = (g_{\mathrm{opk}}, Y_{\mathrm{opk}}, \tilde{g}_{\mathrm{opk}}, \tilde{X}_{\mathrm{opk}}, \tilde{Y}_{\mathrm{opk}})$ and issuer public key $\mathsf{ipk} = (g, Y_0, \ldots, Y_\ell, \tilde{g}, \tilde{X}, \tilde{Y}_0, \ldots, \tilde{Y}_\ell)$. The prover parses her private input to $\mathsf{psk} = (\mathsf{usk}, d)$, $\mathsf{reginfo} = ((\hat{\sigma}_1, \hat{\sigma}_2), \mathsf{opk})$, $\mathsf{cred} = ((\sigma_1, \sigma_2), (a_1, \ldots, a_\ell), \mathsf{ipk})$.

  If $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{ipk}, (\mathsf{usk}, a_1, \ldots, a_\ell), \sigma) = 0$, $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{opk}, \mathsf{usk}, \hat{\sigma}) = 0$ or $\phi(a_1, \ldots, a_\ell) = 0$ holds, she outputs $\bot$ and stops. Else she chooses $s \leftarrow \mathbb{Z}_p^*$, sets $\hat{\sigma}' := (\hat{\sigma}_1^s, \hat{\sigma}_2^s)$ and sends $\hat{\sigma}'$ to the verifier. Next, she chooses $(u, r) \leftarrow \mathbb{Z}_p^* \times \mathbb{Z}_p$, sets $\sigma' := (\sigma_1', \sigma_2') := (\sigma_1^u, (\sigma_2 \cdot \sigma_1^r)^u)$ and sends σ' to the verifier. For $i = 1, \ldots, \ell$ she generates $(C_i, (a_i, d_i)) \leftarrow \mathsf{Com}(\mathsf{pp}_{\mathrm{nym}}, a_i)$ and sends C_i to the verifier. The user computes a non-interactive argument π using the Fiat-Shamir heuristic (Construction 3.55) on the Σ-protocol of form

$$\mathsf{PK}\left\{(\mathsf{usk}, d, (a_i, d_i)_{i=1}^{\ell}, r) : \begin{array}{l} e(\sigma_1', \tilde{g})^r e(\sigma_1', \tilde{Y}_0)^{\mathsf{usk}} \prod_{i=1}^{\ell} e(\sigma_1', \tilde{Y}_i)^{a_i} = \dfrac{e(\sigma_2', \tilde{g})}{e(\sigma_1', \tilde{X})} \\ \wedge\, \mathsf{nym} = g_{\mathrm{nym}}^d h_{\mathrm{nym}}^{\mathsf{usk}} \bigwedge_{i=1}^{\ell} C_i = g_{\mathrm{nym}}^{d_i} h_{\mathrm{nym}}^{a_i} \wedge \phi(a_1, \ldots, a_\ell) = 1 \\ \wedge\, e(\hat{\sigma}_1', \tilde{Y}_{\mathrm{opk}})^{\mathsf{usk}} = \dfrac{e(\hat{\sigma}_2', \tilde{g}_{\mathrm{opk}})}{e(\hat{\sigma}_1', \tilde{X}_{\mathrm{opk}})} \end{array}\right\},$$

  where the proof of $\phi(a_1, \ldots, a_\ell) = 1$ is instantiated via the technique for proofs of partial knowledge (Construction 3.59) and the protocols from Section 4.1.3. The verifier (deterministically) accepts, if and only if $\sigma_1' \neq 1 \neq \hat{\sigma}_1'$ holds and π is valid according to Construction 3.55.

- $\mathsf{Open}(\mathsf{opk}, \mathsf{osk}, \mathsf{reg}, t)$: On input open public key $\mathsf{opk} = (g_{\text{opk}}, Y_{\text{opk}}, \tilde{g}_{\text{opk}}, \tilde{X}_{\text{opk}}, \tilde{Y}_{\text{opk}})$, open secret key osk, registry information obtained from MJoin and a transcript $t = ((\hat{\sigma}'_1, \hat{\sigma}'_2), \cdot)$, Open first uses $\mathsf{VrfyCred}$ to check, whether the transcript is valid. Instead of a transcript, it is also possible to use a rating as input since it also includes $(\hat{\sigma}'_1, \hat{\sigma}'_2)$. In this case, Open uses Vrfy instead of $\mathsf{VrfyCred}$. If one of the verification steps outputs 0, Open outputs $\perp$. Else it iterates through the entries $(\mathsf{upk}, \cdot, \tilde{\tau})$ from reg, until $e(\hat{\sigma}'_2, \tilde{g}_{\text{opk}}) e(\hat{\sigma}'_1, \tilde{X})^{-1} = e(\hat{\sigma}'_1, \tilde{\tau})$ holds. If it finds such an entry, it outputs upk, and else outputs $\perp$.

- $\mathsf{Rate}(\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{item}, \mathsf{reginfo}, \mathsf{token}, \mathsf{usk}, m)$ is a non-interactive algorithm with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$, $\mathsf{pp}_{\text{nym}} = (g_{\text{nym}}, h_{\text{nym}}, p, \mathbb{G}_1)$, master public key $\mathsf{mpk} = (\mathsf{opk}, b)$ with opener public key $\mathsf{opk} = (g_{\text{opk}}, Y_{\text{opk}}, \tilde{g}_{\text{opk}}, \tilde{X}_{\text{opk}}, \tilde{Y}_{\text{opk}})$ and review token issuer public key $\mathsf{rpk} = (g, Y_0, Y_1, \tilde{g}, \tilde{X}, \tilde{Y}_0, \tilde{Y}_1)$. Further, Rate parses $\mathsf{reginfo} = ((\hat{\sigma}_1, \hat{\sigma}_2), \mathsf{opk})$ and $\mathsf{token} = ((\sigma_1, \sigma_2), \mathsf{item}, \mathsf{rpk})$.

  If $\Pi_{\text{sign}}.\mathsf{Vrfy}(\mathsf{opk}, \mathsf{usk}, (\hat{\sigma}_1, \hat{\sigma}_2)) = 0$ or $\Pi_{\text{sign}}.\mathsf{Vrfy}(\mathsf{rpk}, (\mathsf{usk}, H_{\mathbb{Z}_p}(\mathsf{item})), (\sigma_1, \sigma_2)) = 0$ it outputs $\perp$ and stops. Otherwise, it chooses $s \leftarrow \mathbb{Z}_p^*$, sets $\hat{\sigma}' := (\hat{\sigma}_1^s, \hat{\sigma}_2^s)$. Next, it chooses $(u, r) \leftarrow \mathbb{Z}_p^* \times \mathbb{Z}_p$, sets $\sigma' := (\sigma'_1, \sigma'_2) := (\sigma_1^u, (\sigma_2 \cdot \sigma_1^r)^u)$. For public linkability, it chooses $\zeta \leftarrow \mathbb{Z}_p$ and computes values $L_1 := H_{\mathbb{G}_1}(\mathsf{rpk}, \mathsf{item})^{\zeta + \mathsf{usk}}$ and $L_2 := b^{\zeta}$.

  To create the rating itself it obtains signature scheme $(\mathsf{FS.Sign}, \mathsf{FS.Vrfy})$ by applying the Fiat-Shamir heuristic applied to the following Σ-protocol:

$$\mathsf{PK}\left\{(\mathsf{usk}, \zeta, r) : \begin{array}{l} \dfrac{e(\hat{\sigma}'_2, \tilde{g}_{\text{opk}})}{e(\hat{\sigma}'_1, \tilde{X}_{\text{opk}})} = e(\hat{\sigma}'_1, \tilde{Y}_{\text{opk}})^{\mathsf{usk}} \wedge L_1 = H_{\mathbb{G}_1}(\mathsf{rpk}, \mathsf{item})^{\zeta + \mathsf{usk}} \wedge L_2 = b^{\zeta} \\ \wedge \dfrac{e(\sigma'_2, \tilde{g})}{e(\sigma'_1, \tilde{X})\, e(\sigma'_1, \tilde{Y}_1)^{H_{\mathbb{Z}_p}(\mathsf{item})}} = e(\sigma'_1, \tilde{g})^r\, e(\sigma'_1, \tilde{Y}_0)^{\mathsf{usk}} \end{array}\right\}.$$

  Note that $w := (\mathsf{usk}, \zeta, r)$ is a witness for instance $x := (\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{item}, \hat{\sigma}', \sigma', L_1, L_2)$ of the relation underlying the Σ-protocol given above. Finally, it computes $\rho \leftarrow \mathsf{FS.Sign}(x, w, m)$ for x, w defined above and outputs a rating $\mathsf{rating} = (m, \mathsf{item}, \mathsf{mpk}, \mathsf{rpk}, \hat{\sigma}', \sigma', \rho, L_1, L_2)$.

- $\mathsf{Vrfy}(\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{item}, \mathsf{rating}, m)$ is an non-interactive algorithm with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$, $\mathsf{pp}_{\text{nym}} = (g_{\text{nym}}, h_{\text{nym}}, p, \mathbb{G}_1)$, master public key $\mathsf{mpk} = (\mathsf{opk}, b)$ with opener public key $\mathsf{opk} = (g_{\text{opk}}, Y_{\text{opk}}, \tilde{g}_{\text{opk}}, \tilde{X}_{\text{opk}}, \tilde{Y}_{\text{opk}})$, review token issuer public key $\mathsf{rpk} = (g, Y_0, Y_1, \tilde{g}, \tilde{X}, \tilde{Y}_0, \tilde{Y}_1)$, and a rating $\mathsf{rating} = ((m, \mathsf{item}), (\mathsf{mpk}, \mathsf{rpk}, \hat{\sigma}', \sigma', \rho, L_1, L_2))$ for item.

  Vrfy applies the Fiat-Shamir heuristic to the Σ-protocol described in Rate. We denote the resulting signature scheme by $(\mathsf{FS.Sign}, \mathsf{FS.Vrfy})$.

  Finally, it runs $\mathsf{FS.Vrfy}((\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{item}, \hat{\sigma}', \sigma', L_1, L_2), m, \rho)$ and outputs the same as $\mathsf{FS.Vrfy}$.

- $\mathsf{Link}(\mathsf{pp}, \mathsf{mpk}, \mathsf{rpk}, \mathsf{rating}, \mathsf{rating}^*)$ is an non-interactive algorithm with public parameters $\mathsf{pp} = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathsf{pp}_{\text{nym}})$, $\mathsf{pp}_{\text{nym}} = (g_{\text{nym}}, h_{\text{nym}}, p, \mathbb{G}_1)$, master public key $\mathsf{mpk} = (\mathsf{opk}, b)$ with opener public key $\mathsf{opk} = (g_{\text{opk}}, Y_{\text{opk}}, \tilde{g}_{\text{opk}}, \tilde{X}_{\text{opk}}, \tilde{Y}_{\text{opk}})$, review token issuer public key $\mathsf{rpk} = (g, Y_0, Y_1, \tilde{g}, \tilde{X}, \tilde{Y}_0, \tilde{Y}_1)$, and ratings $\mathsf{rating} = ((m, \mathsf{item}), (\mathsf{mpk}, \mathsf{rpk}, \hat{\sigma}', \sigma', \rho, L_1, L_2))$ and $\mathsf{rating}^* = ((m^*, \mathsf{item}), (\mathsf{mpk}, \mathsf{rpk}, \hat{\sigma}'^*, \sigma'^*, \rho^*, L_1^*, L_2^*))$ for item. First, it checks whether rating and rating^* are valid ratings using the Vrfy algorithm of this system. If one of these check fails, output $\perp$ and stop. Otherwise, return 1 if and only if the following equation holds:

$$e\left(\frac{L_1}{L_1^*}, b\right) = e\left(H_{\mathbb{G}_1}(\mathsf{rpk}, \mathsf{item}), \frac{L_2}{L_2^*}\right).$$

Usage of the System Let us give further information about the usage of these additional features. The system developed in Section 4.3 provides us with all tools for access control and even an identity escrow mechanism in case of misuse. However, in practice it is also desirable to rate the service a user used. Construction 4.52 extends this system by these functionalities.

For illustration, we consider a typical usage of our system: Suppose a user already obtained some credentials including the registration information, and wants to use some service that grants access via our system. Further, assume that the user's credentials are sufficient to satisfy the service's policy. As described before, the user runs the protocol (ProveCred, VrfyCred) to show that she is allowed to use the service. After successful execution, access is granted to the user. The service provider in general is interested in the opinion of the user after using the service. Therefore, the service provider triggers an execution of the new protocol (RcvToken, IssToken) to issue a review token for the used service and the user it was granted for. The review token is realized by a credential with one attribute namely the item bought.

Now, a user can use her registration information and the issued token to create a review using Rate. The review essentially is a signature of knowledge on an arbitrary rating text $m \in \{0,1\}^*$ asserting that the signer is in possession of both, a valid registration information (with respect to some master public key) and a valid review token (with respect to some review token public key). Consequently, everyone that sees a review can verify (using algorithm Vrfy) whether the signer is actually a registered user in the system and whether she actually was granted access to the service.

A user publishes a created rating on the *reputation board*. The reputation board is a system-wide storage for ratings containing entries of the form (rpk, item, rating). Here, rpk denotes the service providers public key, item denotes the service/item the rating is about and rating denotes the rating output by a run of the algorithm Rate. Usually, a reputation board has no logic at all. However, we assume for convenience that the reputation board verifies a rating before publishing, i. e. runs the algorithm Vrfy.

Besides these basic rating system, users can *publicly link* two ratings using the algorithm Link. Every user includes linking information (values L_1 and L_2) in every rating created. This feature enables every entity with access to the reputation board, probably even outsiders, to check whether two ratings were created by the same user for the same item. This is not permitted in our system; each user is only allowed to give a single rating for an item of a specific service provider.

Remarks on Security Since further extensions to the security model given in Section 4.3.2 would complicate it even further, we do not explicitly adapt it for the extensions made in this section. Nevertheless, we describe how the model needs to be adapted for the new system.

Our definition of anonymity (Definition 4.39) roughly needs to be extended by two properties. First, we need to make sure that a user stays anonymous while receiving a review token. This includes that RcvToken should not reveal anything about the user, in particular her usk. Formally, this is captured analogously to *anonymity when executing* RcvCred (Definition 4.20.3). Therefore, anonymity for RcvToken holds by similiar arguments as given in Lemma 4.30. Second, we need to assure that ratings maintain the user's anonymity. Formally, two ratings output by Rate for the same pp, mpk, rpk, m and item but different reginfo, token and usk should be (computational) indistinguishable.

The soundness definition (Theorem 4.44) needs to be extended by two additional games. First, we talk about the unforgeability of ratings. A user, or a group of users, should not be able to give a rating without joining the group first, as well as giving a rating without the token to rate the specific item. To prove this, we would construct a game similar to those in Section 4.3.2.2. Second, we explain how public linkability of ratings changes the security definitions. A user should not be able to give more than one rating on the same item without those being publicly linkable. Analogously a group of n users should not be able to give $n+1$ ratings on the same

item where no pair of those ratings is publicly linkable. To prove this we would adapt the experiment based proof given in [BJK15].

Furthermore, we need to require certain properties of the hash functions $H_{\mathbb{Z}_p}$ and $H_{\mathbb{G}_1}$ used in Construction 4.52. To guarantee security, $H_{\mathbb{Z}_p}$ theoretically needs to be collision-resistant and $H_{\mathbb{G}_1}$ theoretically needs to be modeled as a random oracle. However, in practice both functions can be replaced by good cryptographic hash functions, such as SHA-2 or SHA-3. Let us argue why this properties are crucial from a theoretical point of view. The collision resistance of $H_{\mathbb{Z}_p}$ ensures with high probability that the hashes of the attribute values for two different items do not collide. This results in different review tokens for different items for the same review token issuer and user. To model $H_{\mathbb{G}_1}$ as a random oracle is a technical detail to ensure anonymity. To ensure that the Σ-Protocol given in algorithm Rate of Construction 4.52 is zero-knowledge, we need to be able to simulate it. Without $H_{\mathbb{G}_1}$ being modeled as a random oracle the values L_1 and L_2 would be distributed dependently on each other since ζ is used in both values. This makes it impossible to simulate the protocol properly. The random oracle fixes this issue by randomizing the base of L_1.

4.5 Further Extensions

We proceed with possible extensions to our anonymous credential and reputation system. We did not include them into definitions and constructions as they would have complicated our security model and proofs unnecessarily, although they are rather simple to add practically. However, this means we have not proven security of the following extensions. Therefore we argue briefly, when needed, why they intuitively do not interfere with our security notions.

4.5.1 Revocation

We already introduced the Open algorithm, which allows a trusted system manager to identify a user who misbehaves, e. g. rates the same item many times. The system manager should be able to revoke this user from the system. Furthermore, it is reasonable that credential issuers want to revoke distinct credentials from the system. Both should happen verifier-locally, i. e. verifiers who fetched the most recent list of revocation information can check for themselves whether a user or credential has been revoked. Especially there is no need to interact with a trusted third party during a verification process.

4.5.1.1 Revocation of Users

Revoking a user is a valuable mean when a user acts maliciously within the system or loses her secret key to an attacker. To revoke a user in our system the manager simply can add the tracing information $(\mathsf{upk}, \tilde{\tau})$ to a revocation list $\mathcal{RL}$. Then the deterministic VrfyCred algorithm additionally needs to check whether Open with $\mathcal{RL}$ instead of reg as input does not output $\perp$ for the transcript at hand. This check can be performed on already accepted proofs and therefore especially on ratings. For security, intuitively spoken, an accepted proof/rating from a revoked user suffices to break traceability, since the system manager could not open it correctly, i. e. would output $\perp$ or a wrong user.

This integration is simple and allows revocation in hindsight, but is not very efficient for upcoming verification protocols. For example a verifier needs to iterate through all entries in $\mathcal{RL}$ and compute several pairings when *interacting* with a non-revoked user. If many users are revoked over time, this check may be impractical as users need to wait too long until some confirmation. An intuitive way to overcome this is that the system manager issues a personalized "non-revoked" credential to a joining user. When revoking a user, this credential needs to be revoked (Section 4.5.1.2) *additionally* to publishing the tracing information. ProveCred is extended by a proof that the "non-revoked" is valid *at this point in time*, i. e. with respect to

the current information on $\mathcal{RL}$. A verifier who fetched $\mathcal{RL}$ is able to check this proof more efficient than using Open. Nevertheless, using Open seems inevitable for our construction when checking already accepted transcripts and ratings for revocation.

4.5.1.2 Revocation of Credentials

Besides the revocation of users by the system manager, some issuers may need the option to revoke single credentials they have issued. This can already be achieved via a *unique identifier* chosen by the issuer, fixed to the credential and on revocation added to a black list or deleted from a white list. This approach demands that $(\mathsf{ProveCred}, \mathsf{VrfyCred})$ contains a proof, stating "identifier in cred is (is not) on white list (black list)". The white list approach is simple to integrate via the predicate ϕ and a set membership proof (Construction 4.14). A black list could efficiently be integrated via a non-membership protocol, possible via AND-concatenated inequality proofs (Section 4.1.3.2) or probably a universal accumulator like from Li, Li, and Xue [LLX07]. Note, that the anonymity of a user is disclosed to the point that the credential contains an identifier (not) on the list. If a white list contains only a single identifier which has *really* been issued within a credential a user might be tracked. Therefore, this approach should only be used with a trusted issuer, as users cannot check which identifiers were issued.

4.5.1.3 Expiration of Credentials

Expiration or a time to live of credentials can be seen as a kind of revocation happening automatically after some time span. This feature can be integrated using the predicates of our system. An issuer can add an expiration date in form of a timestamp to issued credentials. Users would then need to show the timestamp lies within the required interval via a range proof (Section 4.1.3.3).

4.5.2 Non-Frameability of Users via Judge Algorithm

Regarding accountability, no party should be able to falsely accuse a user of having performed some action within the system. Such a property is called non-frameability in the field of group signatures. This especially targets the system manager, whose outputs of Open are not verifiable in our actual construction.

We can use the Judge-mechanism given in the group signature scheme of Pointcheval and Sanders [PS16, Appendix B]. In Join a user additionally generates $(\mathsf{usk}', \mathsf{upk}')$ of some secure signature scheme and sends a signature on upk to the system manager. The Open procedure, in case of success, outputs a non-interactive proof of knowledge π (via the Fiat-Shamir heuristic (Construction 3.55)) of the correct tracing information of user with upk. Additionally, the correct signature of upk under upk' is output. This ensures, that the user with upk' verifiably joined the system with upk, as else a signature under upk' would have been forged. Note, that the tracing information is not disclosed, hence only the single opened transcript can be linked to the user. Therefore, the anonymity proof still works and our system remains anonymous. The Judge algorithm only checks the signature and runs the verifier part of the Fiat-Shamir heuristic on π.

4.5.3 Removing the Random Oracle

In the security proofs of our extended ACS we rely on the random oracle model (ROM). The ROM is not always favorable and we give an intuition how to remove it as an assumption for our extended system's security. The random oracle is used within the group signature scheme, which we employ to assure traceability of users in our system while remaining their anonymity at a high degree. In particular we used the group signature scheme proposed by Pointcheval and Sanders [PS16, Appendix B] and combined its Sign method with the $(\mathsf{ProveCred}, \mathsf{VrfyCred})$-protocol from

our basic ACS (Construction 4.25). However, we could have extended the basic ACS using any other group signature scheme with the same security properties by signing a completed transcript at the end of an interactive, basic $(\mathsf{ProveCred}, \mathsf{VrfyCred})$ execution. For example we could have used the efficient scheme proposed by Ateniese et al. [Ate+05] which is an adaption of [Bic+10] without random oracle. Since the latter scheme is very similar to the one we use, an analogue adaption seems possible, minimizing the changes for Construction 4.45. Note, that this modularity of replacing the group signature scheme in order to remove the random oracle is only meaningful in the extended ACS. This is due to the fact that we rely on the random oracle to create non-interactive arguments within the Rate method.

4.5.4 Credentials for k-time Use

Another interesting feature is restricting credentials to at most k uses. The most relevant case might be the one-time use, for example when using credentials as a kind of voucher. Intuitively one could say the public linkability information achieves exactly this behavior. Remember, it implies using several review tokens to output two or more unlinkable ratings for the same item is infeasible for a single user. This is desirable for ratings, however, a user might receive two different vouchers for the same item and should be able to use both without being linked.

The idea is analogue to Shamir's secret sharing scheme [Sha79]. The user chooses a polynomial P of degree k with $P(0) = \mathsf{usk}$ uniformly at random. In particular, the user chooses coefficients $a_i \leftarrow \mathbb{Z}_p$ for $i = 1, \ldots, k$, which are blindly added to the credential by the issuer. When executing ProveCred the user is asked to evaluate P at the challenge c and output the result R. The user can prove $R = P(c) = \mathsf{usk} + \sum_{i=1}^{k} a_i c^i$ with a generalized Schnorr protocol. After a successful proof, the verifier interpolates for each combination of (c, R) with k previously accepted points (c_i, R_i) a polynomial P' and computes $\mathsf{usk}' = P'(0)$. If usk' is valid, i. e. $\tilde{g}^{\mathsf{usk}}$ is the correct tracing information for the transcript, the credential has been used more than k times and the user disclosed her identity. For large values of k this is not very efficient, because many (wrong) combinations of points from other users with different polynomials might be tested. However, the risk and cost of disclosing usk should deter users from using such credentials too often.

4.5.5 Disable Rating Own Products

A problem inherent to reputation systems is the intent of rating products worse or better than they are, e. g. as competitor or seller respectively. We deliberately allow rating issuers to join the system as user, e. g. to buy and resell items, for which a rating token is provided. However, our current construction provides no mean to prevent issuing a rating token to the own user identity. By public linkability this method allows at most one undetectable rating, assuming the issuer can join the system as user only once. We considered this as minor problem and did not cover it in the construction.

To avoid such scenarios we need to require that an issuer is always bound to a user identity, i. e. a public key upk of form g^{usk} in our system. Particularly, an issuer with rpk should only be accepted by users or verifiers, if she possesses a verifiable certificate stating that upk and rpk belong to the same entity. In our system the issuer could first join via Join using a upk from U.Init and then generate rpk via R.Init. Afterwards she could go to the system manager, prove the public keys belong to her and for example obtain an ordinary signature signing both together. Next, Rate needs to be changed, such that a proof $\log_g(\mathsf{upk}) \neq \mathsf{usk}$ is included, where usk is included in the rating token. This can be handled via an inequality proof as Construction 4.12. Assuming that users are restricted to have only one upk within the system, this method prevents the usage of rating tokens given from an issuer to her own user identity.

4.5.6 Invalidation of Ratings

At some point in time a user might decide a previous rating of herself should be invalidated, for example because she changed her mind about a product.

Our first proposal assumes a central reputation board managed by an honest party. For invalidation the user authenticates herself as the rating's author, e. g. by proving knowledge of usk related to $(\hat{\sigma}'_1, \hat{\sigma}'_2)$ from the rating. This can exactly be done as in the Schnorr proof of the Rate algorithm. If verification succeeds, the manager should delete the rating from its board. The approach obviously demands a manager, who acts honestly and deletes the rating. Furthermore, no copies of this rating should exist at other places, because they remain valid. Managers would need to store deleted ratings to check whether an incoming rating is not an old copy. However, usually we cannot assume a single board where all ratings are stored. In fact a feature of our system is that ratings can be published anywhere and checked verifier-locally, possibly supported by an up-to-date revocation list.

The second idea is to have a central, *lightweight* invalidation list allowing for verifier-local invalidation checks, which circumvents to have a single (trusted) reputation board. An entry of the list needs to contain a verifiable proof that the original author demanded invalidation of her rating. For example, the user can use the Fiat-Shamir heuristic to output a proof over usk, such that $e(\hat{\sigma}'_2, \tilde{g}')e(\hat{\sigma}'_1, \tilde{X}')^{-1} = e(\hat{\sigma}'_1, \tilde{Y})^{\mathsf{usk}}$ holds for $(\hat{\sigma}'_1, \hat{\sigma}'_2)$ from the rating. This proof should additionally sign the old rating and perhaps a message like "rating invalid". Soundness, i. e. the infeasibility of deleting foreign ratings, follows because knowledge of usk is proven. Anonymity still holds due to the proof's zero-knowledge property. A more efficient, yet *potentially* insecure method is to store ζ from the rating creation and publish g^ζ to the list in case of invalidation. A verifier only needs to check whether $e(g^\zeta, b) = e(g, L_2)$ holds for L_2 of a rating. This approach might harm the anonymity of the user who invalidates her rating and requires deeper security analysis.

4.5.7 Editability of Ratings

A user who changed her mind about some product may favor to edit an old rating instead of deleting it completely. This can essentially be done by deleting/invalidating the old rating (Section 4.5.6) and outputting a new rating bound to the invalidation information. Concretely, the user could compute a signature of knowledge of usk as above, which signs the old rating alongside the text of a new rating message. Again, it potentially suffices to compute g^ζ and output a signature of knowledge of ζ, signing the edited rating and its new message. Both approaches can be applied several times by editing the most recent rating on the central list of edited ratings.

Part II

Practical Realization

5 From Theory to Practice

On the theoretical side, there are several security assumptions that need to be reflected in the implementation of our system to meet the required security goals (cf. Section 4.2.2). This section will emphasize that not all of those assumptions can be directly translated into a practical implementation and how those have been adapted to fulfill the required security goals.

Random Oracle Model In our security proofs we sometimes assume the existence of a random oracle, the so called *random oracle model*. To realize this in practice, one would have to first choose a random function out of all possible random functions and then share it among all participating parties. But not every truly random function can be encoded in polynomial size, thus there cannot exist an efficient way to share every possible random function. Therefore, we need to use another method, which is efficient and closely resembles a random function. What we do instead is using a cryptographic hash function. By using a good cryptographic hash function such as SHA-256 [15], we achieve efficient generation of seemingly random outputs.

The outputs are not truly random since functions like SHA-256 are deterministic. But, by definition of a cryptographic hash function, there is no way of getting any information about the input given only the output. Therefore, it has all properties of a random oracle for our scenario. We have to be careful to only use state-of-the-art hash functions though, since using outdated hash functions such as SHA-1 would lead to possible security breaches [WYY05].

Secure Channels Since we are constructing an *anonymous* credential system, we want to make sure that every entity can only obtain the relevant information that is needed for him. Thus, if two participants of the ACS communicate, a third entity should not be able to know the contents of the communication. Therefore, we want to have confidential channels to send all messages over. To be more specific, we also require forward secrecy [Gün90], else our security proof for credential unforgeability (cf. Lemma 4.35) does not hold.

Furthermore, we want our channels to be authenticated. Else, there is the possibility of *impersonation* through a so called *man-in-the-middle attack* (MTM). There, an adversary intercepts the encrypted messages between two parties A and B. To A, the adversary poses as B and to B he poses as A. Thus, both A and B think they speak directly with each other, while in reality they talk to the adversary. Since we only have confidentiality between direct communication partners, the adversary is able to read every message.

To provide secure channels, we can simply use TLS. It is important to encrypt everything with TLS though, since we want to prevent a leakage of disclosed attributes or any data manipulation. This means that every protocol and every other communication has to be encrypted.

Since we can use existing tools to encrypt with TLS, this approach is easily implemented. Another thing we get for free with this is forward secrecy as well. This ensures that even if a key is compromised, stored data from before can still not be decrypted.

Public Key Infrastructure For the prevention of MTM attacks we assumed the existence of public keys to authenticate the communication. Since in practice such keys do not just exist, they need to be generated and distributed. Furthermore, users need some way to make sure that the distributed keys are not compromised. We do this by using a *public-key infrastructure*. In such a structure, every public key is certified by a trusted authority. Therefore, an adversary

would have to forge such a certificate or compromise a participant that both users trust in order to do the MTM, which is hard or not probable respectively.

Optimally, we would use TLS public keys and build our PKI that way. Since this would be a lot of overhead for our simple example application, we instead provide unique identities for all entities ourselves. We do this by putting all public information of an entity into an identity object. If an entity has no such information, we simply choose a random element.

Note that this approach has some flaws. Choosing a random element might result in duplicate entities since a public identity is not necessarily unique if it is chosen at random. But since we only showcase the library functionalities within the example application in a simple manner, this approach ensures the desired properties. It is important to point out that a real-world application should use TLS public keys and a real PKI though. Furthermore, the user public key should *not* be publicly known (the system manager knows it though), since we want users to be anonymous. The public keys of verifier or issuer instances *must* be publicly known though, since users need these public keys to initialize the TLS handshake.

(Non-)Interactive Protocols With the ACS being constructed as it currently is, there exists a rather trivial attack to break the credential unforgeability.

An adversary seeing some message(s) from a prover P to a verifier V can send the same message(s) to V to gain access herself. Note that this attack is not possible if interactive protocols are used, since a verifier chooses a random challenge for every execution of the protocol and the exact same message(s) would therefore most likely not work. For non-interactive protocols, this attack would work if no auxiliary information is used though.

Similarly, an adversary can play the role of a verifier in a protocol with some prover P and *concurrently* play the prover in a protocol against some verifier V. Then, the adversary could simply send the messages sent by P to V and vice-versa. Thus, V believes that the adversary possesses credentials to access V's service. This attack is called a *relay attack*. Even for interactive protocols, the adversary could choose the same challenge as the verifier to successfully prove the possession of the prover's credentials just by using his messages.

In our implementation, we use only non-interactive protocols. Furthermore, we use the identity of the verifier together with the policy information as auxiliary information. We do this specifically by applying the fiat-shamir-heuristic on interactive protocols to transform it into a non-interactive proof (cf. Section 3.10) and then adding the public identity together with the policy information as auxiliary information in these proofs. By including the public identity of the verifier within the proof, she can check if the proof was meant for her or not. Note that with this approach, we prevent replay **and** relay attacks at once. In both cases, the verifier would check if she is the correct recipient for the proof and reject if that is not the case. Additionally, by including the policy information within the auxiliary information within the proof, the adversary also can not change the policy to always accept anything the prover sends and then use this to initialize the opening with the system manager. The usage of non-interactive protocols also enables us to utilize the RESTful API for the example application, since this would not be possible with stateful protocols.

Support for More Predicates An important remark is that our security model does *not* support proving knowledge of attributes with credentials issued by different issuers. It only covers proofs of possession of a single credential from a single issuer. Through an application-dependent approach, this is extendable to an AND-composition of (threshold-)sub policies, by expecting multiple executions of the $(\mathsf{ProveCred}, \mathsf{VrfyCred})$-protocol with the same pseudonym. But adding OR-gates this way is not possible without letting the verifier know which policies the prover fulfills, which goes against our security goals. However, in our implementation we support the more complicated predicates (cf. Section 6.4.4.2). There, proving to fulfill a policy with AND-gates, OR-gates and threshold-gates has to be done in a single proof. Still, our security model

can be extended to cover this functionality. For example, instead of using a single credential in ProveCred a vector of credentials is used, while VrfyCred gets a vector of issuer public keys. Furthermore, credential unforgeability would require that no ppt adversary is able to convince a verifier that he fulfills a policy if he was not issued credentials that fulfill the policy. Since covering this functionality would result in more complicated definitions and especially more complicated proofs of security, we decided to only use the simple security model.

6 Implementation

Contents

6.1 Introduction **118**
6.2 Architecture **118**
6.3 Building Blocks **119**
6.3.1 Commitment Schemes 119
6.3.1.1 Pedersen Commitment Scheme 121
6.3.1.2 HashThenCommit Commitment Scheme 121
6.3.2 Arguments of Knowledge 122
6.3.2.1 Sigma Protocols 122
6.3.2.2 Generalized Schnorr Protocol 124
6.3.2.3 Damgård's Technique 125
6.3.2.4 Fiat-Shamir Heuristic 126
6.3.3 Signature Schemes 127
6.3.3.1 Fiat-Shamir Signature Scheme 127
6.3.3.2 Pointcheval-Sanders Signature Scheme 127
6.3.4 Accumulators 128
6.3.4.1 Nguyen Accumulators 130
6.4 Zero-Knowledge Component **131**
6.4.1 Protocol Overview: 132
6.4.2 Building Blocks of the Zero-Knowledge Component 133
6.4.2.1 Generation of Generalized Schnorr Protocols 133
6.4.2.2 Secret-Sharing 135
6.4.2.3 Proofs of Partial Knowledge 135
6.4.3 Protocol Overview: 137
6.4.3.1 Join/MJoin 137
6.4.3.2 ProveNym/VrfyNym 138
6.4.3.3 IssueCred/RcvCred 138
6.4.3.4 ProveCred/VrfyCred 138
6.4.4 ProveCred/VrfyCred 138
6.4.4.1 Policies 138
6.4.4.2 ProveCred/VrfyCred Implementation 139
6.4.5 Common Input 140
6.5 Reputation System **141**
6.5.1 Ratings/Reviews 141
6.5.2 Reusing the Credential Issuing Structure 142
6.5.3 Reviewing 142
6.5.4 Verifying and Linking 142
6.6 API **142**
6.6.1 Setup 143

6.6.2 Actor creation . . . 143
6.6.2.1 System Manager . . . 143
6.6.2.2 Issuer . . . 143
6.6.2.3 ReviewTokenIssuer . . . 144
6.6.2.4 User . . . 144
6.6.2.5 CredentialVerifier . . . 144
6.6.2.6 ReactReviewVerifier . . . 145
6.6.3 Actor interaction . . . 145
6.6.3.1 Issuing of credentials . . . 145
6.6.3.2 Proving of credentials . . . 146
6.6.3.3 Reviewing of items . . . 147
6.6.3.4 Verification of reviews . . . 148
6.7 Example Application . . . 148
6.7.1 Architecture . . . 148
6.7.2 Use Cases . . . 150
6.7.2.1 Initialization . . . 150
6.7.2.2 Pseudonym Creation and Product Hosting Credentials Issuing . . . 151
6.7.2.3 Hosting A New Product . . . 153
6.7.2.4 Buying and Rating A Product . . . 153

6.1 Introduction

In this chapter, we give further documentation for the library additional to the `JavaDoc` that is provided for the code itself. We start by showing an overview of the architecture before elaborating the individual modules. One of the larger modules is the zero-knowledge component, which is explained in detail. Furthermore, we explain how the different entities interact with each other and how the library has to be used to develop a secure application. At last, we showcase our implementation with our example application, illustrating a possible implementation.

6.2 Architecture

In order to ease the use and extendability of the library it was separated into multiple modules. This section will explain the purpose and the dependencies between these individual modules.

To be able to reuse a set of lower-level cryptographic functions the modules make use of the two libraries CrACo and PBC[1]. From the CrACo library features like the Pointcheval-Sanders signature scheme, secret sharing and low-level policies are used. The PBC library provides primitives for pairing-based cryptography, but also common groups from algebra. It also comes with a serialization framework which is also being used in the CrACo library. Thus, for consistency, the modules developed as part of the ACRS were also developed utilizing these serialization methods.

The individual modules are realized as Maven sub-modules[2] which are aggregated using a parent `pom.xml`. This parent module references the following sub-modules:

[1] `https://sfb901.uni-paderborn.de/de/projects/tools-and-demonstration-systems/tools-from-the-1st-funding-period/craco/`
[2] `https://maven.apache.org/`

building-blocks: provides the implementation for standalone primitives like commitment schemes (cf. Section 6.3.1) or signature schemes (cf. Section 6.3.3). This sub-module only depends on the CrACo and PBC libraries and thus is independent of the actual ACRS implementation.

protocols: provides primitives which allow the implementation of the Zero-Knowledge Component (cf. Section 6.4). Implementations for the Schnorr Protocol, Damgard or Accumulators reuse the elements from the `building-blocks` module while still not mandating specific predicates.

predicate-protocols: provides the implementation for the predicates which are required in order to construct the ACRS with the anticipated features. Namely, equality-, inequality-, range and set membership-proofs are implemented here.

acs: provides the actual implementation of the ACRS. It contains the high-level actor classes which expose the functionality of the system in a consumer-friendly form. This module is the interface as described in Section 6.6 and is the API against which an user of the library should implement.

support: provides optional functionality which may not be required for an actual ACRS use-case, but still might be useful for consumers of the library. Currently it only exports utility functions for encrypting serializable objects to arbitrary streams.

As illustrated in Figure 6.1, the upper layers in the architecture utilize lower-level modules and thus depend on them.

While the intended interface for applications consuming the library is the `acs` module, the other modules can still be used in order to either, achieve greater control about the internals of the credential system, or also to reuse parts for completely credential system unrelated use-cases. For example, the commitment or signature schemes from the `building-blocks` module can be reused in a different context with this architecture.

6.3 Building Blocks

In this section we describe the basic building blocks (cf. Chapter 3) we implemented and reason the design decisions that we took. One challenge in realizing the building blocks was to achieve interoperability. For achieving this we oriented on the design criteria of CrACo (cf. Section 6.2) and decided to use interfaces for reflecting the theoretical properties of cryptographic constructions. A more detailed example of this approach is described in Section 6.3.1. Later sections rather use class diagrams to describe the important parts of the implementation.

6.3.1 Commitment Schemes

In our library we want the option that a sender does not have to reveal the content of her message immediately, but is able to do so at a later point. This can be achieved through commitment schemes. A commitment scheme is a non-interactive protocol between a sender and a receiver. The sender first commits to a message and then sends this commitment to the receiver. When this commitment arrives, the receiver will not be able to gain any information about the content inside the commitment.

For representing the functionality and theoretical properties of commitment schemes (cf. Section 3.5) we created the interfaces which need to be implemented by concrete commitment schemes realizations (like the Pedersen commitment scheme, cf. Section 6.3.1.1). The interfaces can be seen in Table 6.1 with a brief description:

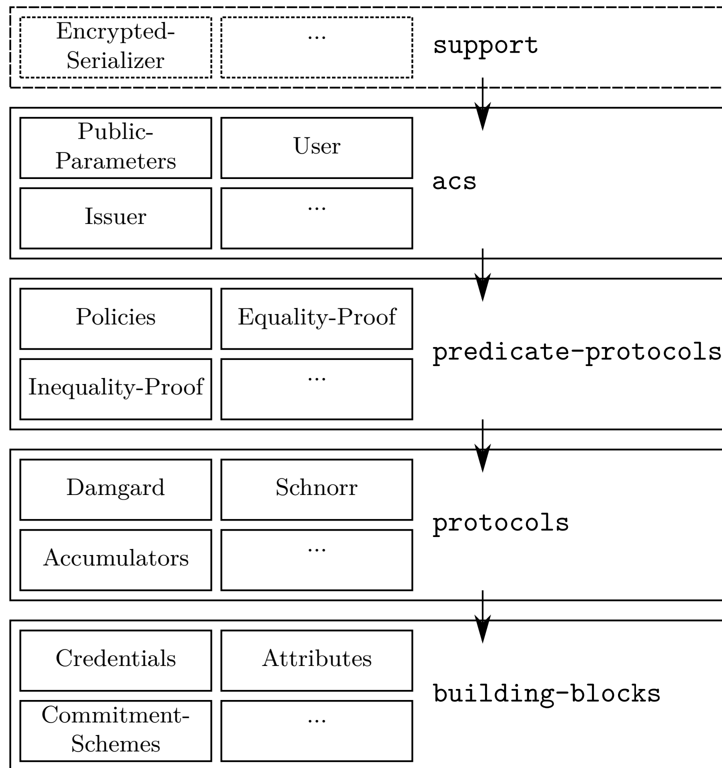


Figure 6.1: The layer based architecture of the developed library.

Table 6.1: Overview of interfaces for a commitment scheme

Interface	**Short description**
`CommitmentScheme`	Main class handling most functionality
`CommitmentSchemePublicParameters`	Encapsulates the public parameters
`CommitmentSchemePublicParametersGen`	Generator for the public parameters
`CommitmentPair`	Encapsulates the commitment return values
`CommitmentValue`	Encapsulates the commitment value
`OpenValue`	Encapsulates the open value

In order to allow consumers of commitment schemes like Damgård's technique (cf. Section 6.3.2.3) to only work on the general interfaces it is necessary to use decoupled value types. Hence, concrete data types are encapsulated in interfaces as well. The provided interfaces for data types used in a commitment scheme are shown in Table 6.2, mapping the theoretical constructs to our implemented interfaces.

Table 6.2: Overview of interfaces for data types

Interface	**Theoretical Construct**
`CommitmentPair`	(c, d)
`CommitmentValue`	c
`OpenValue`	d

We decided to use a `PlainText` implementation from CrACo for the message space of a `CommitmentScheme` in general as it can handle/encapsulate most possible forms of input.

Following CrACo's design, the `setup()` of a commitment scheme's public parameters is realized by the `CommitmentSchemePublicParametersGen`; all other functionality of a commitment scheme is provided in the `CommitmentScheme`-interface.

The CommitmentScheme interface covers the following functionality:

- commit()
- open()
- verify()
- mapToPlainText()

In addition to the functions from the theoretical definition (cf. Section 3.5) we added the methods verify() and mapToPlainText(). The method verify() receives a PlainText of the plain message and compares it to the result of opening the CommitmentValue with the OpenValue; if this is the case it returns true. Having this method simplifies the usage of commitment schemes in combination with hashing (cf. Section 6.3.1.2) which enables enlarging the message space to any bit string (cf. Construction 3.29). Hash-functions as used in HashThenCommitCommitmentScheme (cf. Section 6.3.1.2) return the hash in the form of a byte-array. In order to use commitment schemes in combination with such byte-arrays it is necessary to provide a method mapping a byte-array to a PlainText. We decided to use the mapToPlainText() method as is defined for CrACo's SignatureScheme. The method mapToPlainText() generates an injective mapping (injectivity is only guaranteed for arrays of the same length) from a byte-array to a PlainText corresponding to the input space of the concrete CommitmentScheme implementation. The mapping has to be injective in order to prevent collisions or different mappings for the same input-array.

6.3.1.1 Pedersen Commitment Scheme

The Pedersen commitment scheme is one of the major building blocks in our library as it is the concrete commitment scheme implementation that many other building blocks such as Damgård's technique (cf. Section 6.3.2.3) depend on. Therefore, each commitment scheme interface (cf. Section 6.3.1) is implemented in our Pedersen commitment scheme realization, thus achieving loose coupling such that consumers can work on the interfaces only. The input space of the Pedersen commitment scheme as defined in our theory part (cf. Construction 3.23) consists of ZpElement(s). Since the PedersenCommitmentScheme is required to expose its message space as PlainText (cf. Section 6.3.1), a single ZpElement is put into one RingElementPlainText and multiple ZpElement(s) can be concatenated as a MessageBlock (both implement/extend PlainText respectively). Hence, the Pedersen commitment scheme's message space is realized with the classes RingElementPlainText and MessageBlock.

As described in Section 6.3.1, mapToPlainText() needs to be implemented. The method provides an injective mapping of a byte-array to a MessageBlock containing the same number of messages as the PedersenCommitmentScheme-instance expects according to its PedersenPublicParameters. Yet, only the first element of the MessageBlock is an injective mapped RingElementPlainText; all other elements are RingElementPlainText containing the respective zero element. However this implementation could be improved to find injective mappings for a larger input space if a more sophisticated approach can be found. Such an approach would be using all elements of the MessageBlock (instead of only the first) while still guaranteeing injective mappings for arrays of the same length.

6.3.1.2 HashThenCommit Commitment Scheme

For enabling commitment schemes to handle inputs of arbitrary size and form we use commitment schemes in combination with hashing (cf. Construction 3.29). Following the design of HashThenSign we therefore implemented a HashThenCommitCommitmentScheme consuming

CommitmentScheme and a HashFunction as input and resulting in a commitment scheme itself. The HashFunction is required to hash its input into byte[] with a maximum size that is not larger than the CommitmentScheme's mapToPlainText() can find injective mappings for.

The HashThenCommitCommitmentScheme implements the CommitmentScheme-interface and applies a HashFunction in commit() and open() utilizing the encapsulated commitment scheme's mapToPlainText() to map the resulting hash to a valid PlainText correlating to the encapsulated commitment scheme. Hence, HashThenCommitCommitmentScheme is a wrapper applying a HashFunction onto a CommitmentScheme. When using HashThenCommitCommitmentScheme the HashFunction has to be chosen in such a way that its output is of the correct input size (in form of a byte[]) for the encapsulated CommitmentScheme. Otherwise mapToPlainText() of the wrapped CommitmentScheme might not be able to find an injective PlainText.

6.3.2 Arguments of Knowledge

Σ-protocols are three way protocols guaranteeing *honest verifier zero-knowledgeness* (cf. Section 3.8). Thereby, a prover can convince another honest party, called verifier, that he knows a secret during the interactive protocol execution. Afterwards, it is guaranteed that the verifier has not learned anything about the secret. The realization of Σ-protocols is described in Section 6.3.2.1, the most important Σ-protocol we are using, the generalized Schnorr protocol, is described in Section 6.3.2.2.

To achive security even against dishonest or concurrently running verifiers, we apply Damgård's technique, introduced in Section 3.9. Applying this technique to our created Σ-protocols ensures that even a cheating verifier, who is using different distributions to choose the challenge or interacting with different provers in parallel, does not learn anything about the secret. Our realization of Damgårds technique is described in Section 6.3.2.3.

To prove knowledge of a secret without an interactive communication, the prover can use non-interactive arguments (cf. Definition 3.37). By applying the Fiat-Shamir heuristic to a Σ-protocol, we obtain a non-interactive argument. The realization is explained in Section 6.3.2.4. We make use of non-interactive arguments to realize the reputation system in Section 6.5.

6.3.2.1 Sigma Protocols

As shown in Section 4.1.3, all protocol primitives can be realized using Σ-protocols. Thus, the class SigmaProtocol is extended by all protocols that can be instantiated by the zero-knowledge component, described in Section 6.4. A SigmaProtocol is uniquely describable via an Problem-Array and PublicParameter. They are publicly known and in combination equivalent to the problem of a zero-knowledge argument of knowledge defined in Definition 3.36. The realization of publicly known information is described in Section 6.4.5.

The class diagram of the class SigmaProtocol and the classes extended by SigmaProtocol are presented in Figure 6.2. The SigmaProtocol inherits methods to generate the announcement, the challenge and the response from the InteractiveThreeWayAoK, as well as a method to verify. A correct interactive protocol execution between a prover and a verifier consists of three messages:

- Announcement
- Challenge
- Response

A prover sends an Announcement, then the verifier replies with a Challenge. The prover calculates a Response for the first two messages (Announcement and Challenge) and sends it to the verifier. At last the verifier performs verify(), which accepts only for a correct protocol

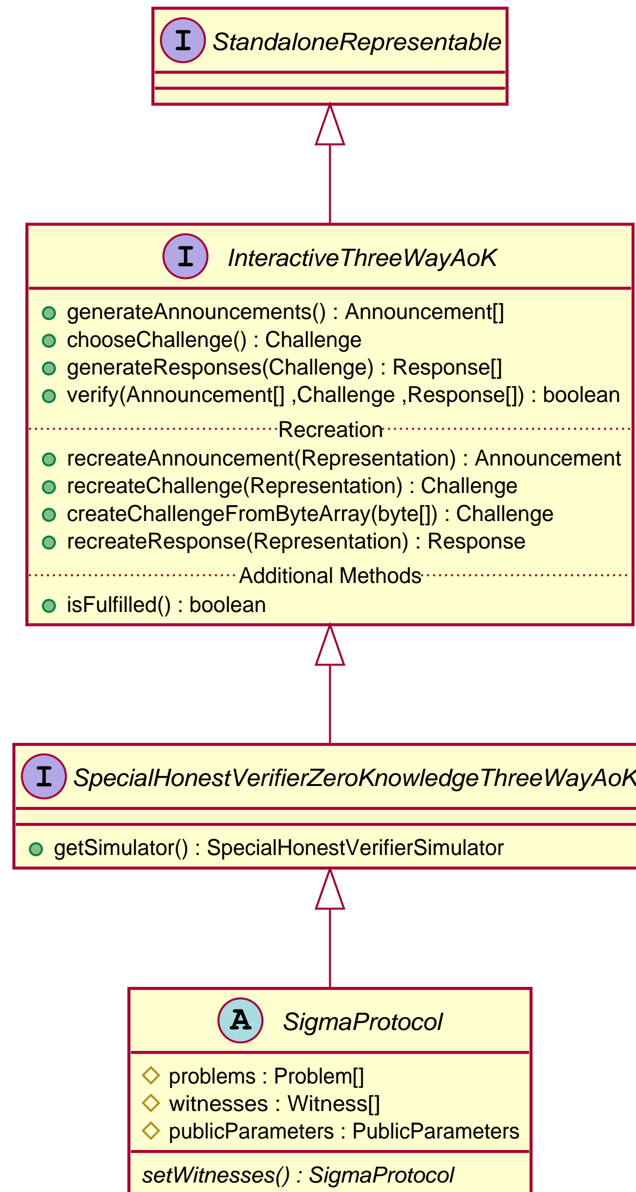


Figure 6.2: Class diagram of a Σ-protocol

execution. The methods `generateAnnouncements()` is a realization of the ppt algorithm α of the Σ-protocol, the method `generateResponses(Challenge)` realizes the ppt algortihm γ. The method `chooseChallenge()` chooses a challenge uniformly at random from the defined challenge space C.

Additionally, the `SigmaProtocol` exposes methods to recreate the three protocol messages. The additional method `isFulfilled` checks if a given protocol, defined through problem-description and public parameters, can be fulfilled with the given witnesses. A `Witness` object contains a unique name and protocol-specific data. The name needs to be unique in the protocol context, meaning that two different protocols may have witnesses with the same name. If the names are unique in the protocol context, an automatic mapping of witnesses to the problem description is possible, if protocols should be build automatically. This approach is for example used in the `GeneralizedSchnorrProtocolFactory` (cf. Section 6.4.2.1).

A method to get a `SpecialHonestVerifierSimulator` is inherited from the interface `Special-HonestVerifierZeroKnowledgeThreeWayAoK`. Since no default implementation for the simula-

tor is given, every concrete implementation is enforced to provide the simulator. Thereby, every Σ-protocol can be simulated. This fact is used during proof of partial knowledge, described in Section 6.4.2.3.

Simulation of Σ-protocols Since Σ-protocols have the Special Honest-Verifier Zero-Knowledge property, there exists a simulator for every Σ-protocol (cf. Definition 3.43). Most implemented protocols are based on a generalized Schnorr protocol and therefore make use of the class `GeneralizedSchnorrSimulator`. The construction for the generalized Schnorr protocol and the corresponding simulator is given in Section 3.8.1. We have proven that all protocols that are used in our system are Σ-protocols. To prove the Special Honest-Verifier Zero-Knowledge property, a construction for a simulator for the Σ-protocols is given. Such a construction is used as base for the implementation of the simulator.

The `SpecialHonestVerifierSimulator` gets a challenge for the protocol and outputs an accepting transcript for the protocol, containing an `Announcement`-Array, the given `Challenge` and a `Response`-Array. Transcripts will potentially be exchanged between two parties or stored independently, thus is extends the interface `StandaloneRepresentable`. Thereby, it is guaranteed that the transcript can be restored without additional information. Additionally, the protocol instance itself is stored in the transcript, since the messages of the protocol can only be restored using the protocol. Thus, a `SigmaProtocol` needs to be Standalone-representable as well.

6.3.2.2 Generalized Schnorr Protocol

To realize Σ-protocols in our system, we are able to use only one protocol implementation, namely the generalized Schnorr protocol (cf. Section 3.8.1), since all protocols used are based on generalized Schnorr protocols (cf. Section 4.2.3). The generalized Schnorr protocols corresponds to proving knowledge of the following equation:

$$\bigwedge_{j=1}^{m} A_j = \prod_{i=1}^{n} g_{i,j}^{x_i}$$

for cyclic groups $\mathbb{G}_1, \ldots, \mathbb{G}_m := \mathbb{G}$ of prime order p, witnesses $(x_1, ..., x_n) \in \mathbb{Z}_p^n$ and $A_j, g_{i,j} \in \mathbb{G}_j$ for all $i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}$.

The class `GeneralizedSchnorrProtocol` is implemented as a subclass of the abstract class `SigmaProtocol`, described in Figure 6.2. A class diagram for the `GeneralizedSchnorrProtocol` is given in Figure 6.3.

To simplify the usage of a `GeneralizedSchnorrProtocol`, the random values, computed during the announcement phase, are stored in the protocol object as a state. Thereby, the prover does not need to store the random values outside of the protocol object. In case a user wants to use specific random values, an array of `ZPElements` can be passed as input for the method `generateAnnouncements(ZpElement[])`. It is not recommended to use this method, except an overlaying construction enforces the knowledge of the random values.

To create an instance of the `GeneralizedSchnorrProtocol` it is sufficient to state the problem equations in a notation that is similar to the Camenisch-Stadler Notation Camenisch and Stadler [CS97]. The generation of generalized Schnorr protocols using problem equations is described in Section 6.4.2.1. A single problem equation is represented by an instance of a `Generalized-SchnorrProblem` and has the following form:

$$A = \prod_{i=1}^{n} g_i^{x_i}$$

To check the validity of the represented equation, the class `GeneralizedSchnorrProtocol` offers the static method `isInvalidProblem(Problem)`. A problem is valid if it fulfills the following properties:

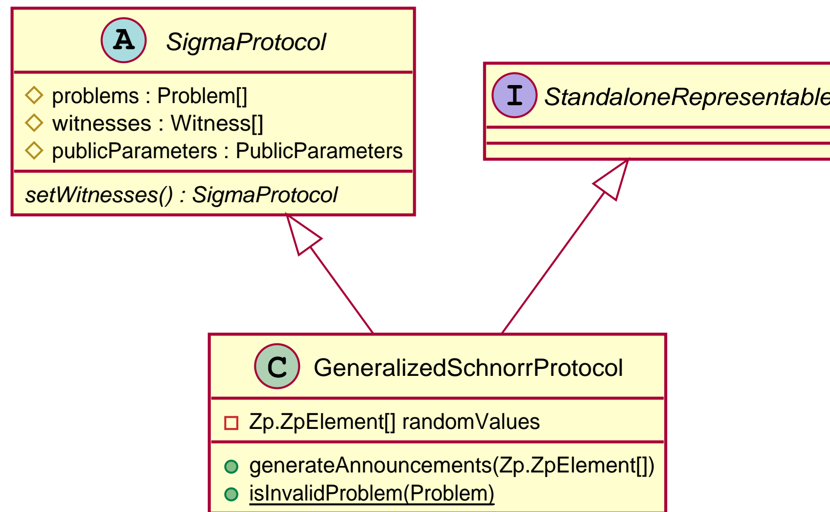


Figure 6.3: Class diagram of a generalized Schnorr protocol

- It is an instance of `GeneralizedSchnorrProblem`
- The left-hand-side of the problem equation that is stored inside the `GeneralizedSchnorrProblem` object is fixed. Therefore, it has either a constant value or a value is assigned to the variable used.
- The right-hand-side of the problem equation is one product-expression. Each factor of the product represents an exponentiation-expression.

The problem equations can be represented using `ArithExpressions`.

6.3.2.3 Damgård's Technique

Damgård's technique is a construction used for improvements to Σ-protocols in order to provide security against concurrent adversaries. The resulting protocol is a "Concurrent black-box zero knowledge three-way interactive argument of knowledge" (and therefore implements the interface `InteractiveThreeWayAoK`, cf. Section 6.3.2.1).

The theory part (cf. Section 3.9) defines that Damgård's technique consumes a Σ-protocol (cf. Construction 3.42) and a commitment scheme (cf. Definition 3.20). So the goal of our implementation was to be able to handle any Σ-protocol and commitment scheme. Hence, the implementation of Damgård's technique, the class `DamgardTechnique`, does only work on the interfaces `SigmaProtocol` (cf. Section 6.3.2.1) and `CommitmentScheme`(cf. Section 6.3.1) as well as their related interfaces. Thus, `DamgardTechnique` is able to handle any Σ-protocol in combination with any commitment scheme (being able to handle the Σ-protocol's announcements as input) implementing/extending the necessary interfaces. Table 6.3 lists the implemented classes (implementing interfaces for `InteractiveThreeWayAoK`).

Damgård's technique basically encapsulates a `SigmaProtocol` in combination with a `CommitmentScheme` which is required to be able to handle, and thus committing to, the Σ-protocol's announcements. A protocol execution of Damgård's technique according to the definition (cf. Construction 3.51) works like this:

A prover using `DamgardTechnique` sends a `DamgardAnnouncement` (which is a commitment to the encapsulated `SigmaProtocol`'s `Announcement`) to the verifier.

The verifier using `DamgardTechnique` replies with the `Challenge` (of the encapsulated `SigmaProtocol`).

Table 6.3: Overview of classes in the implementation of Damgård's Technique

Class	Short description
`DamgardTechnique`	Handling all functionality
`DamgardAnnouncement`	Commitment to the encapsulated Σ-protocol's announcements
`DamgardResponse`	Container for exchanged messages of the Σ-protocol's execution (Encapsulated Σ-protocol's announcements and response and the open value for the `DamgardAnnouncement`)

The prover then calculates the `Response` (according to the encapsulated `SigmaProtocol` for the encapsulated `SigmaProtocol`'s `Announcement`) and sends a `DamgardResponse` containing the encapsulated `SigmaProtocol`'s `Announcement`, the encapsulated `SigmaProtocol`'s `Response` and the `OpenValue` for the `CommitmentValue` inside of the `DamgardAnnouncement`.

The verifier using `DamgardTechnique` then perform in her `verify()`-method:

- The `CommitmentScheme`'s `verify()` for `CommitmentValue` inside of the `DamgardAnnouncement`, `OpenValue` and the encapsulated `SigmaProtocol`'s `Announcement` (as original message).
- The `SigmaProtocol`'s `verify()` for encapsulated `SigmaProtocol`'s `Announcement`, `Challenge` and `Response`.

`DamgardTechnique`'s `verify()`-method returns `true` only if both `verify()`-methods return `true`.

Overall, `DamgardTechnique` is able to handle any `SigmaProtocol` and `CommitmentScheme` implementing/extending the necessary interfaces by implementing `DamgardAnnouncement` and `DamgardResponse` which implement the interfaces `Announcement` and `Response`. They handle and encapsulate the necessary values and messages to reflect the theoretical construction of Damgard's technique, but this only works assuming that the `CommitmentScheme` is chosen in such a way that it can handle the `SigmaProtocol`'s `Announcement`'s message space and size. An intuitive way to achieve this is assuming that the input space of the `CommitmentScheme` is $\{0,1\}^*$. This can be achieved by using hashing in combination with the `CommitmentScheme`. As a simplification for this approach we implemented the wrapper `HashThenCommitCommitmentScheme` (cf. Section 6.3.1.2) which applies a `HashFunction` onto the encapsulated `CommitmentScheme`. In the concrete implementation used in our library, the wrapper is always applied onto the `PedersenCommitmentScheme` (cf. Section 6.3.1.1).

6.3.2.4 Fiat-Shamir Heuristic

The *Fiat-Shamir heuristic* is a cryptographic construct that transforms a Σ-protocol into a non-interactive zero-knowledge argument of knowledge (cf. Section 3.10). It becomes non-interactive, because instead getting a randomly-chosen challenge from the verifier, the prover uses a random oracle to derive a challenge from the announcements. The returned value is a fixed length bit string and needs to be transformed into a correct challenge. Since the challenge space C may differ between several Σ-protocols, every Σ-protocol has the ability to transform a bit string into an element c form the challenge space C.

Based on Construction 3.55 and following the design conventions in our library, the class `FiatShamirHeuristic` implements the general interface `NonInteractiveArgument` which, conceptually, represents an abstract non-interactive proof of knowledge. This interface provides the two public methods `prove()` and `verify()`. The former method is the one that is used by the

prover in a non-interactive proof of knowledge, to generate the non-interactive `Proof` which, later on, serves as an input to the `verify()` method on the verifier side. In `FiatShamirHeuristic`, the `FiatShamirProof` (an actual implementation of `Proof`), returned by the `prove()` method, comprises three values: the generated `Announcement[]`, a `Challenge` self-computed by the prover and the generated `Response[]`.

The class `FiatShamirHeuristic` functionally relies on the implementation of the interface `InteractiveThreeWayAoK` in order to gain the ability to generate the non-interactive proof and to verify it. In other words, it delegates the responsibilities of generating the announcements and responses to a concrete `InteractiveThreeWayAoK` object which does the actual computation of those values.

The component also makes use of one concrete implementation of the interfaces `HashFunction`. Since random oracles are difficult in practice, we make use of a good cryptographic hash function, as stated in (cf. chapter 5).

6.3.3 Signature Schemes

Digital signatures are used for assuring authenticity, integrity and non-repudiation. For instance they allow an issuer to sign the attributes of an user and therefore form a credential, the cornerstone of our ACS. Our implementations of signature schemes follow the primitives given by CrACo, we added the Fiat-Shamir signature scheme (cf. Section 6.3.3.1) and extended the Pointcheval-Sanders signature scheme (cf. Section 6.3.3.2).

6.3.3.1 Fiat-Shamir Signature Scheme

The Fiat-Shamir heuristic can additionally be used as a signature scheme as defined in Construction 3.57. The implementation of the signature scheme is based on the `MultiMessage-SignatureScheme` interface and provides the two methods `sign()` and `verify()`.

To construct a `FiatShamirSignatureScheme` a `ProtocolProvider` and a `HashFunction` are needed. The `ProtocolProvider` can generate a Σ-protocol, given an instance of the protocol and a witness. This is neccessary to ensure that the realization of the signature scheme is stateless. Since we will use the `FiatShamirSignatureScheme` only for Generalized Schnorr Protocols, we only provide a `GeneralizedSchnorrProtocolProvider`.

The implementation of the signature scheme is based on the theoretical construction Construction 3.57. The realization of `sign()` makes use of the Fiat-Shamir heuristic, described in Section 6.3.2.4. During `verify()`, the signature scheme creates a protocol for the used protocol-instance to restore the announcement from the proof. The restored announcement is needed to recreate the challenge, which is equal to the hash of the restored announcement and the message committed on. Therefore, the announcement and the message are hashed and compared to the challenge contained in the proof. Only if the hash is equal to the challenge and the proof is accepted, the verify method returns true.

6.3.3.2 Pointcheval-Sanders Signature Scheme

We use the Pointcheval-Sanders signature scheme in our library, because of its nice properties allowing the randomization of a Poincheval-Sanders Signature (cf. page 16), thus enabling the combined usage with the Pedersen commitment scheme (cf. Construction 4.4) for blindly signing messages and unblinding signatures on blinded messages. The CrACo-library that we use already provides a working multi-message implementation (called `PSSignatureScheme`) of the Pointcheval-Sanders signature scheme which follows our code and design constraints. Hence, we decided to use this implementation and to extend it where necessary to use the advantages of the Pointcheval-Sanders signature scheme we need in our library.

Following the definition for the Pointcheval-Sanders signature scheme variation in our theory part (cf. Section 3.4.1), the public key is adapted to contain: Generator g and group elements $Y_1, \ldots, Y_\ell$, where $g \leftarrow \mathbb{G}_1 \setminus \{1\}$ and $Y_i \coloneqq g^{y_i}$ for $y_1, \ldots, y_\ell \leftarrow \mathbb{Z}_p$ for $\ell \in \mathbb{N}$ (number of messages) and $i = 1, \ldots, \ell$ (for the complete definition, cf. Construction 3.17).

For using CrACo's Pointcheval-Sanders signature scheme implementation with our definition (cf. Section 3.4.1) two classes needed to be extended, those are:

- `PSVerificationKey` for containing the additional parameters in the public key
- `PSSignatureScheme` for generating the additional parameters

The reason for extending the public key `PSVerificationkey` of the `PSSignatureScheme` is to store further variables in the `PSExtendedVerificationkey` allowing the possibility of a combined usage of the `PSExtendedSignatureScheme` with the `PedersenCommitmentScheme` for being able to blind and unblind messages ("commit" and "open") before and after signing them. This is achieved by using the same generator g and group elements Y_i in the `PedersenPublicParameters` (as generator g and group elements h_i, cf. Definition 4.3) as provided by the `PSExtendedSignatureScheme`. This allows a user to receive a signature on a commitment for a message and to then calculate the signature for the 'uncommited' message and thereby receiving a signature from a signer for a message without revealing the content of the plain message to the signer.

In `PSSignatureScheme` the key pair containing the private and public key is generated with the method `generateKeyPair()`. The resulting key pair contains all necessary parameters ($\mathbb{G}_1$ and $y_1, \ldots, y_\ell$) to generate the additional parameters for the public key's extension.
Therefore we implemented `PSExtendedSignatureScheme` extending `PSSignatureScheme`. In the extended key pair generation process first the regular key pair is generated. Using this key pair's parameters the additional parameters for the extended public key (`PSExtendedVerificationKey`) are calculated and a new key pair containing the private key (`PSSigningKey`) and `PSExtendedVerificationKey` is returned.

Furthermore we extended the `PSExtendedSignatureScheme` to provide methods for randomization of a Poincheval-Sanders Signature, for blindly signing messages and unblinding signatures on blinded messages. Those are:

- `randomizeExistingSignature()`
- `blindSign()`
- `unblindSignature()`

The method `randomizeExistingSignature()` randomizes a signature based on input randomness (cf. Section 3.4.1, 16), `blindSign()` signs a messages and blinds the resulting signature (cf. Construction 4.4) and `unblindSignature()` unblinds a signature which was previously blinded (cf. Construction 4.4).
An overview of the extended classes is given in Figure 6.4. Note that only overwritten and additional methods and member variables are included.

6.3.4 Accumulators

Accumulators are introduced for our extension of the ACS (cf. Section 4.3). They can be used to prove that a value x is part of a set X. Using an accumulator value which is independent

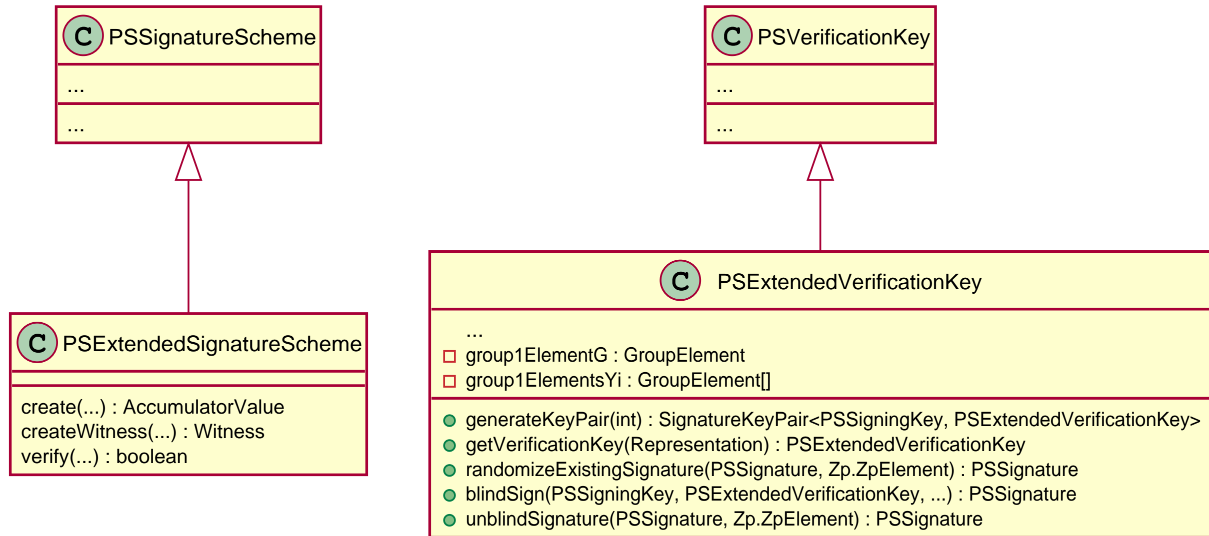


Figure 6.4: Overview of extensions as class diagram

of the set's size, this set membership can be proven efficiently (cf. Section 3.12), thus offering performance advantages.

Similarly to e.g. commitment schemes (cf. Section 6.3.1), we created the following interfaces representing the theoretical properties of accumulators (cf. Section 3.12) which need to be implemented by concrete accumulators (like the Nguyen accumulator, cf. Section 6.3.4.1). An overview of interfaces for accumulators is given in Table 6.4:

Table 6.4: Overview of interfaces for accumulators

Interface	**Short description**
`StaticAccumulator`	Main class for static accumulator
`DynamicAccumulator`	Main class for dynamic accumulator
`AccumulatorPublicParameters`	Encapsulates public parameters
`AccumulatorPublicParametersGen`	Generator for public parameters
`AccumulatorValue`	Encapsulates the accumulator value
`AccumulatorIdentity`	Encapsulates an accumulator identity
`Witness`	Encapsulates a witness for an accumulator identity It is the same interface as for arguments of knowledge (cf. Section 6.3.2)

Two interfaces for accumulators are needed as there are two types of accumulators, the static accumulator (cf. Definition 3.64) and the dynamic accumulator (cf. Section 3.12.2, Definition 3.66) which extends the static one to be dynamic. Dynamic in this context means that insertions and deletions of identities i are possible, so the set of accumulated identities can be changed dynamically including the update of witnesses w_j.

The `setup()` of the accumulator's public parameters is realized by the `AccumulatorPublicParametersGen` (following CrACo's design); all other functionality of an accumulator is provided in the `StaticAccumulator`-interface or `DynamicAccumulator`-interface respectively.

Data types for accumulators are reflected in interfaces as listed in Table 6.5:

The `StaticAccumulator`-interface provides the following methods (cf. Definition 3.64):

- `create()` (for AccCreate)
- `createWitness()` (for WitCreate)

Table 6.5: Overview of interfaces for data types

Interface	Theoretical Construct
`AccumulatorValue`	Accumulator value V
`AccumulatorIdentity`	Identity i which can be part of the set
`Witness`	Witness w_i for the identity i

- `verify()` (for Vrfy)

The `DynamicAccumulator`-interface additionaly provides the following methods (cf. Definition 3.66):

- `insert()` (for AccInsert)
- `delete()` (for AccDelete)
- `update()` (for WitUpdate)

The `AccumulatorPublicParameters` contain the universe U of identities and $q \in \mathbb{N}$ the upper bound for the number of identities that can be accumulated in the accumulator (cf. Definition 3.64). We decided to in general return the universe U as a `List` of elements extending the `AccumulatorIdentity`. In this way concrete accumulator implementations can model their universe relative freely, yet it already can be reflected in the interface.

6.3.4.1 Nguyen Accumulators

In our library we use the Nguyen accumulator (cf. Section 3.12.3) which is a dynamic accumulator (cf. Section 3.12.2). Hence, a `NguyenAccumulator` has to implement the `DynamicAccumulator`-interface. The class diagram of the class `SigmaProtocol` and the classes extended by `SigmaProtocol` are presented in Figure 6.5.

As defined in the theory part (cf. Section 3.12.3), the `NguyenAccumulator` accumulates a set of `Zp.ZpElement`(s). Thus, the `NguyenAccumulatorIdentity` encapsulates and contains a `Zp.ZpElement`. This has to be reflected in the universe which is part of the `NguyenAccumulatorPublicParameters`. For implementing the `AccumulatorPublicParameters`-interface there are two intuitive options to achieve this as the universe is the structure `Zp`. We chose to use one `NguyenAccumulatorIdentity` encapsulating a `Zp.ZpElement` and its structure `Zp` and to provide a convenient method `getUniverseStructure()` in `NguyenAccumulatorPublicParameters` returning the universe-structure as `Zp`. Thus, adding another class is need necessary as in the second approach which would be to introduce a new class extending the `AccumulatorIdentity` containing the `Zp` (universe).

The set of `NguyenAccumulatorIdentity`s are accumulated in the `NguyenAccumulatorValue` which is a `GroupElement`. The `NguyenWitness` for a `NguyenAccumulatorIdentity` is a `GroupElement` too.

All methods required in the `DynamicAccumulator`-interface are implemented in the `NguyenAccumulator` and work as defined in the theory part (cf. Section 3.12.3). In this part modifications for `update()` are described to compute the updated `NguyenAccumulatorValue` more efficiently than using `create()` for the changed set. Nevertheless, this more efficient computation for updating the `NguyenAccumulatorValue` is only defined for the change of a single element in the accumulated set, not for several. For only using the efficient `update()` computation, it would be necessary to track all changes of the accumulated set and the accumulator value resulting in an enormous overhead in "tracking-infrastructure".

Thus, we decided to compromise in the `update()` implementation. In `update()` we first check if just a single element in the accumulated set changed. In this case, the updated `NguyenAccumulatorValue` is computed by using the efficient, `private` implementation `updateInsert()`

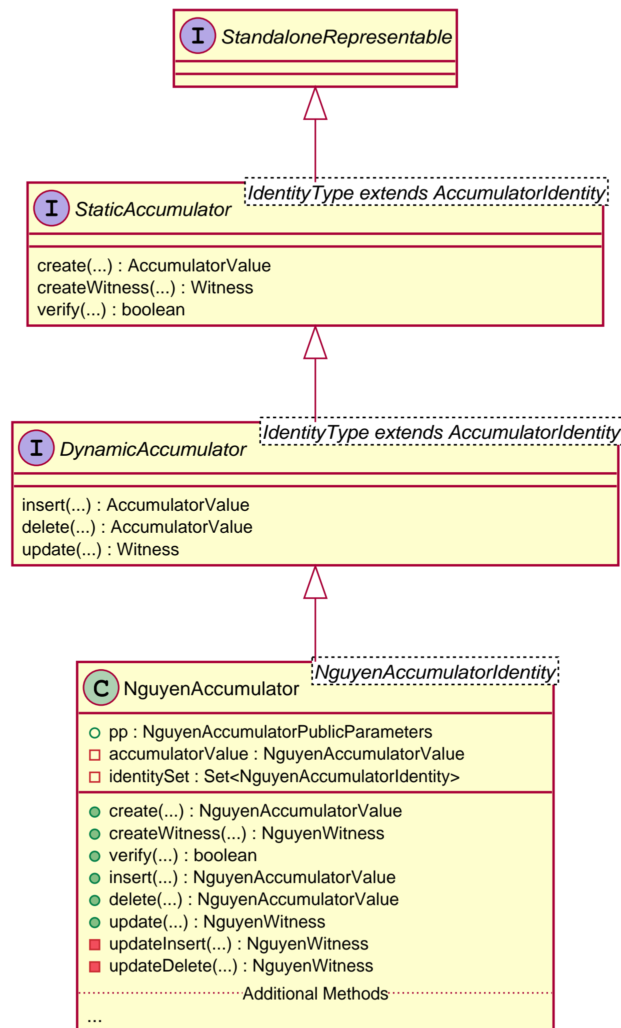


Figure 6.5: Class diagram of a `NguyenAccumulator`

(for one inserted element) or `updateDelete()` (for one deleted element) respectively. If more than one element in the accumulated set changed, the updated `NguyenAccumulatorValue` is computed by using the `create()` method. In this way we achieve an easy-to-use `update()`-method using the modified update computation when possible (and deciding between `update-Insert()` and `updateDelete()` automatically).

Yet, if a "tracking-infrastructure" for changes to the accumulated set would be provided, our `update()`-implementation could simply be called iteratively for only using the efficient update computation.

6.4 Zero-Knowledge Component

The key feature of the ACS is to grant a user anonymous access based on a set of attributes belonging to the user certified by credentials. This proceeding requires a zero-knowledge proof (cf. Definition 3.39) to prove possession of a set of attributes, issued by some issuer. The zero-knowledge proofs are necessary to prove that the issued attributes match a certain policy, while hiding the concrete attributes and especially the user's identity. These proofs can be done using interactive and non-interactive protocols.

While implementing our system from the theoretical constructions given in this document, some parts have to be extended to be implemented in an understandable and secure manner. Im-

plementation of the interactive protocols were impacted the most by these required extensions. Implementing Σ-protocol with the same functionality as described in the theoretical constructs resulted in a hierarchy of protocols, which was required to realize more complex constructions such as proofs of partial knowledge. This hierarchy offers clean modularity and levels of abstraction for the implementation.

The Zero-Knowledge Component is responsible for generating all needed protocols. Our goal is to build a protocol with just the CS notation given, limiting this to only generalized Schnorr protocols though. It uses two building blocks, one is the `GeneralizedSchnorrProtocolFactory` (cf. Section 6.4.2.1) allowing generation of generalized Schnorr protocols from a construction similar to the Camenisch-Stadler Notation Camenisch and Stadler [CS97]. The second building block is the general implementation of proofs of partial knowledge, using boolean formulas over Σ-protocols (cf. Section 6.4.2.3). These two building blocks simplify the generation of complex protocols and prevent redundant code.

Guaranteeing that every generated protocol is a Σ-protocol is one big advantage of the Zero-Knowledge Component. Thereby, techniques like the Fiat-Shamir heuristic or Damgård's technique (cf. Section 6.3.2.4, Section 6.3.2.3) can easily be applied to create non-interactive AOKs or strengthen the security properties.

6.4.1 Protocol Overview:

All protocols to prove fulfillment of a predicate (cf. Section 4.1.3) can be build on generalized Schnorr protocols. A short overview of the methods provided by the `GeneralizedSchnorr-Protocol` can be found in Section 6.3.2.1. Since some protocols used require some additional public information that are computed on the prover's side in advance (e.g. range proofs in Section 4.1.3.3), an extension for the Σ-protocols is needed. Additionally, some protocols contain several different generalized Schnorr protocols or a proof of partial knowledge (cf. Section 6.4.2.3). In summary, a Σ-protocol may need to handle additional public information or may contain nested protocols. We came up with a general approach to face both problems using a "wrapper protocol", that is shown in Figure 6.6. Important to mention that this general approach is still a Σ-protocol.
Both parties will generate the Σ-protocols using factories that are introduced in this chapter. The factories ensure that the same protocol is generated on both sides independently, so no protocol needs to be exchanged. Thereby, neither the prover needs to trust the verifier to generate the protocol honestly nor vice versa.
In the first phase, the `Initialization`, the prover initializes her nested protocols and computes additional information, if needed. They will be send alongside with the announcement. Additionally, the prover initializes the nested protocols on her side. During the `Announcement` phase, the prover collects the announcements of the nested protocols (if existing) and sends them alongside with the announcement of the "wrapper protocol". Afterwards, the verifier will choose a challenge, according to the given distribution, and send it to the prover. At this point the verifier may not have any information about the public values computed by the prover, for example if Damgård's technique (cf. Section 3.9)is applied. This is no problem, since the challenge is chosen uniformly at random form the set C (cf. Construction 3.42).

Afterwards, in the `Response` phase, the prover may share the challenge (in case of a proof of partial knowledge) and forward the challenge to the nested protocols. Finally, the prover collects all computed responses and sends them to the verifier. The verifier initializes the nested protocols during verification phase, after extracting the public information from the announcements. In the end the verifier checks the nested protocols for fulfillment (if needed) and then returns the result of the verify-method. This extension is used in a more enhanced way during proofs of partial knowledge (cf. Section 6.4.2.3).

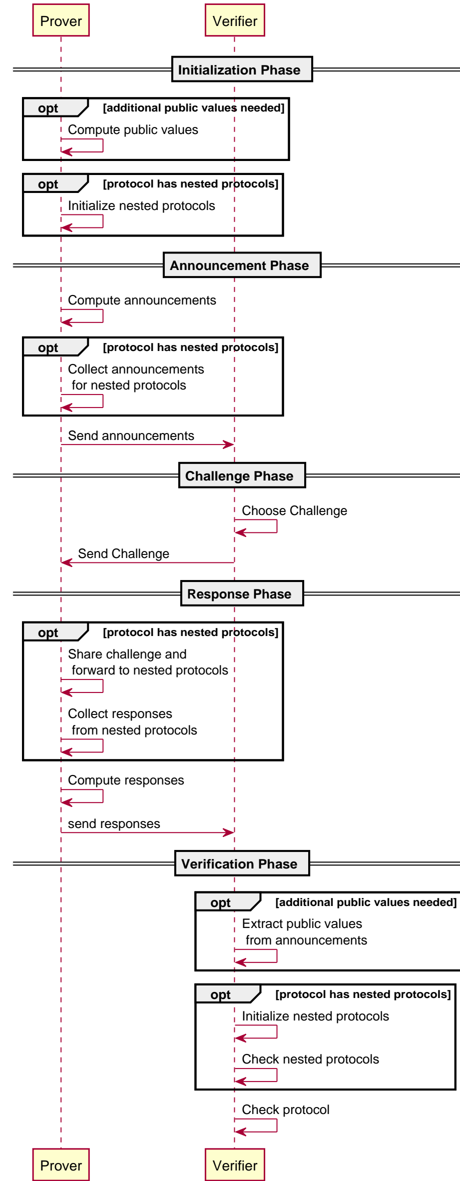


Figure 6.6: Sequence diagram for a extended Σ-protocol

6.4.2 Building Blocks of the Zero-Knowledge Component

Next, we will introduce the two building blocks used for the Zero-Knowledge Component. Firstly, we describe how generalized Schnorr protocols are generated. Afterwards, we explain how the proofs of partial knowledge are realized.

6.4.2.1 Generation of Generalized Schnorr Protocols

The Camenisch-Stadler Notation is an easy and compact way to denote a Σ-protocol. In this section we will focus on the generation of generalized Schnorr protocols, introduced in Section 6.3.2.2, which correspond to proving knowledge of the following equation:

$$\bigwedge_{j=1}^{m} A_j = \prod_{i=1}^{n} g_{i,j}^{x_i}$$

for cyclic groups $\mathbb{G}_1, \ldots, \mathbb{G}_m := \mathbb{G}$ of prime order p, witnesses $(x_1, ..., x_n) \in \mathbb{Z}_p^n$ and $A_j, g_{i,j} \in \mathbb{G}_j$ for all $i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\}$.

The `GeneralizedSchnorrProtocolFactory` offers two methods, one for creating a protocol for a prover and one to create a protocol for a verifier. Both methods will result in equivalent protocols (without witnesses for the verifier) and can be executed independently. Both methods are similar, the only difference is the additional input of witnesses fulfilling the problem equations on the prover site. To generate a complete instance of the `GeneralizedSchnorrProtocol`, several elements need to be generated:

- A problem description
- An array of witnesses
- An array of the groups
- A table of group elements used as bases for the exponentiations
- The number of witnesses n

The `GeneralizedSchnorrProtocolFactory` is given the problem description represented using an array of `ComparisonExpression`. It is required that the left hand side of the equations contains the fixed value A_j and the right hand side the product of the power expressions. These expressions are first of all checked for correctness, meaning that they in fact are a problem description for a generalized Schnorr protocol.

After the problem equations are fixed, the witnesses are stored in an array. The witnesses are given as input to the `GeneralizedSchnorrProtocolFactory` in the form of a mapping from the unique witness name to their value. The ordering of the witnesses is given by the ordering in the first problem equation. Witnesses that are not present in the first equation are added at the end of the witness-array in the same order they can be found in the other equations. Although the right hand side of a problem equation contains a product expression that is commutative, an ordering of the witnesses is enforced to ensure a deterministic witness ordering during all generations. Since the verifier does not know the witnesses, only the unique names of the witnesses are extracted from the problem equations, the values are left empty.

Finally, the so-called public parameters of the protocol are generated. The groups $G_1, \ldots, G_m$ are extracted from the fixed values $A_1, \ldots, A_m$. The base elements g_i are extracted from the problem-equations and ordered in the same ordering of the witnesses. If a witness is not present in one equation, the base element is set to the neutral element of the current group. For example, if a protocol has three witnesses w_1, w_2 and w_3 and the first problem equation is

$$A = g_1^{w_1}\ g_2^{w_2}$$

for some $A, g_1, g_2 \in \mathcal{G}$, the problem equation will be used internally as

$$A = g_1^{w_1}\ g_2^{w_2}\ 1^{w_3}$$

where 1 is the neutral element of $\mathcal{G}$. This trick allows an easier handling of problem equations represented as a table of generators and an array of witnesses, since no special cases (e.g. empty generator) need to be thought of. Note that the problem description is not changed and still contains the "minimal" form without any neutral elements as bases of an power-expression. Finally, the values p and n are computed, p as size of G_1 and n as the number of the witnesses extracted from the problem equations.

6.4.2.2 Secret-Sharing

To be able to apply the proofs of partial knowledge defined in Section 3.11, we first need to implement the underlying secret-sharing scheme(s). As already highlighted in the aforementioned section, we followed the approach of Cramer, Damgård, and Schoenmakers [CDS94] and implemented the semi-smooth secret-sharing scheme of Shamir [Sha79] in such a way that it is applicable for an inner node of the given `ThresholdPolicy` as described in Section 6.4.4.1.

The resulting `ShamirSecretSharing` is therefore a (threshold) secret sharing scheme based on polynomial interpolation. $\mathsf{Share}(s)$ calculates a polynomial P of degree $t-1$ (t being the threshold required for reconstruction) with $P(0) = s$ and outputs a list of shares $s_1, \ldots, s_n$, which correspond to points $(i, s_i = P(i))$. As a result, any set S of share-receivers with $|S| \geq t$ will be able to reconstruct s since Recon tries to reconstruct the polynomial P via polynomial interpolation and outputs $P(0)$, if successful.

To be able to execute the secret-sharing operations on the whole policy the `ThresholdTreeSecretSharing` was implemented which takes a (semi-)smooth secret-sharing scheme, `ShamirSecretSharing` in our case, and applies its operations recursively on each node of the `ThresholdPolicy`. This approach allows to substitute the actual secret-sharing scheme used, as long as it is publicly known to all parties involved.

6.4.2.3 Proofs of Partial Knowledge

For a prover to be able to prove arbitrary boolean formulas over her attributes in a zero-knowledge proof, it is necessary to define a protocol which can prove the fulfillment of threshold policies without revealing information to the verifier. The concept of threshold policies is described in Section 6.4.4.1.

The proof of partial knowledge technique (popk, cf. Construction 3.59) yields a special kind of Σ-protocol, as defined in Section 3.8, which enables proofs on boolean formulas over the satisfaction of (internal) Σ-protocol(s). The internal secret sharing scheme is the `ThresholdTreeSecretSharing`, backed by `ShamirSecretSharing`, as described in Section 6.4.2.2 The implementation details of the actual protocols can be found in Figure 6.7

The premise of the shown process is that both parties exchanged the public parameters, containing the $\mathbb{Z}_p$ to execute the secret sharing on and the (semi-)smooth secret sharing scheme (cf. Section 3.6.1) to be used for the threshold secret sharing, as well as the boolean formula (threshold policy) over the Σ-protocol(s) to be proven.

As defined in Construction 3.59 we denote the components of protocol $(\mathcal{P}_i, \mathcal{V}_i)$ by $(\alpha_i, C_i, \gamma_i, \psi_i, \mathcal{S}_i)$, where α_i is a ppt computing the announcement, C_i is the finite challenge space, γ_i is a ppt computing the responses, ψ_i is polynomial-time verifiable predicate and $\mathcal{S}_i$ is the special honest-verifier zero-knowledge simulator. For simplicity we omit the $(x_i, w_i) \in R_i$ here, as those are in practice only "known" to the internal protocols and not by the popk protocol instance.

Both parties start with their respective initialization phase. During this both participants set up the dual structure (cf. Definition 3.31) of the given policy to be proven/verified and utilize this to set up their threshold secret sharing scheme. The prover additionally collects all Σ-protocol(s) which can be fulfilled to be able to construct the set of protocols to be simulated later on.

The actual protocol execution follows the usual three way approach of a Σ-protocol.

1. Announcement (α): The prover first constructs the set of all protocols which need to be simulated and then constructs the announcement for all protocols based on this set. All of those protocols get a share assigned after sharing the (arbitrarily chosen) value 1 via the secret sharing scheme, i. e. $((s_1, \ldots, s_n) \leftarrow \mathsf{Share}(1)$. If a protocol is not fulfilled the transcript $t_i = (a_i, c_i, r_i) \leftarrow \mathcal{S}_i(s_i)$ is computed, stored internally in the set T and the transcript's announcements a_i will be added to the popk-announcement. Otherwise the pro-

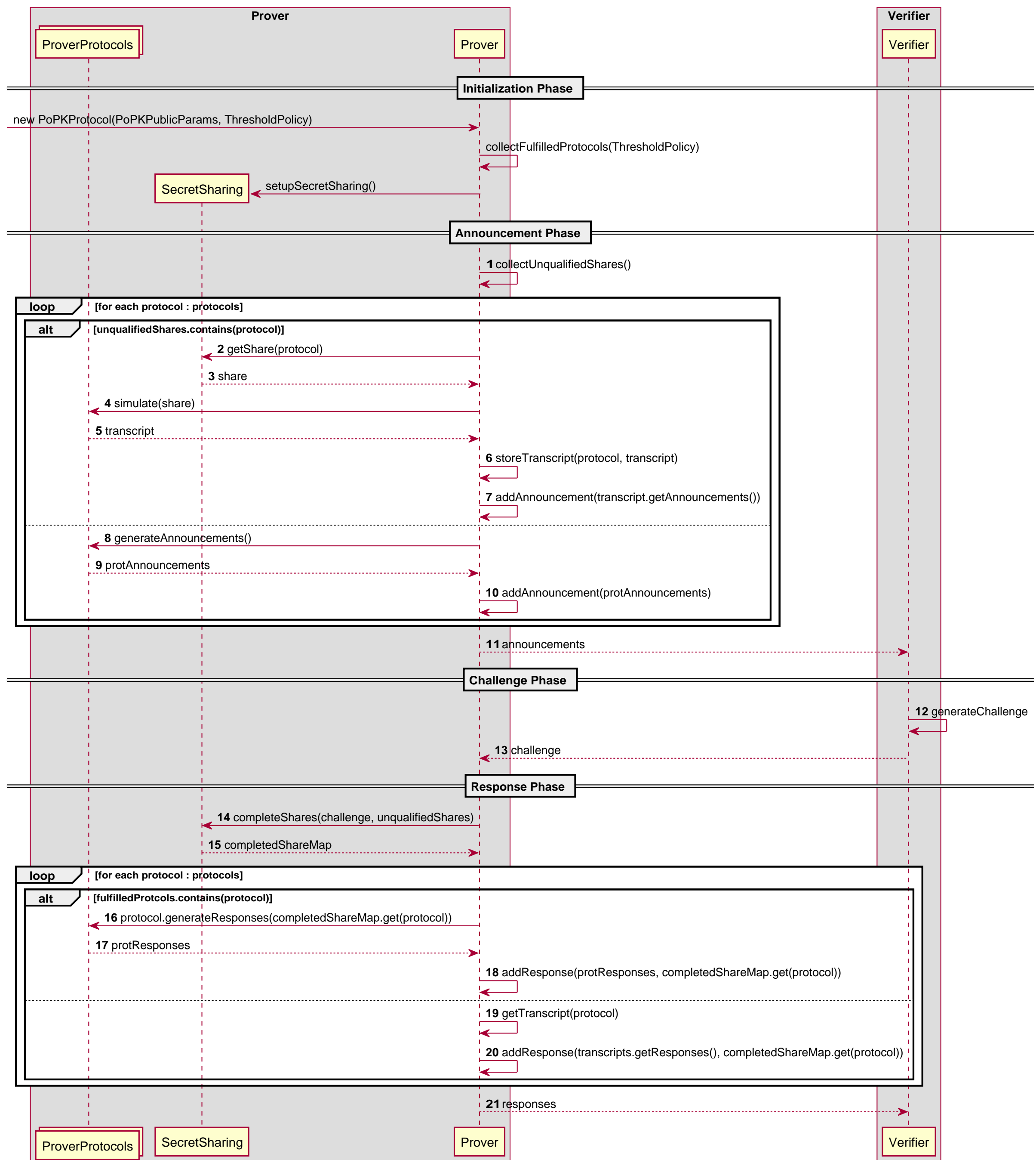


Figure 6.7: Proof of partial knowledge protocol - prover side

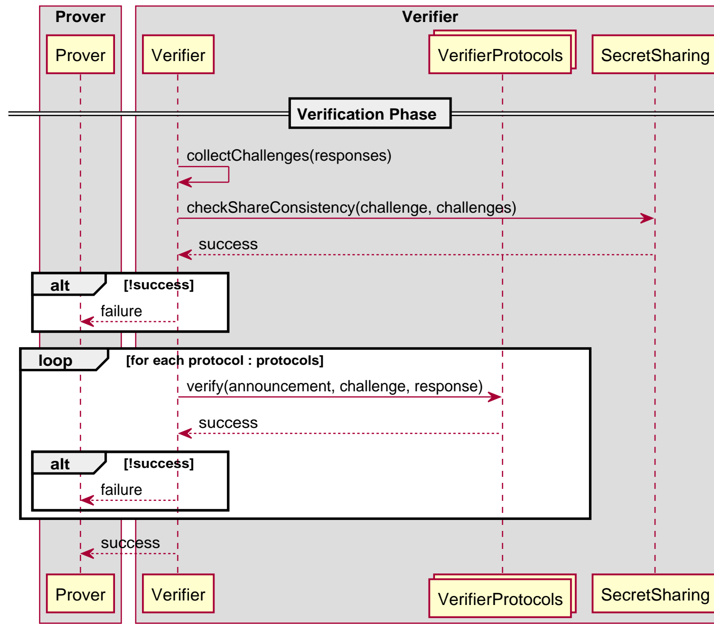


Figure 6.7: Proof of partial knowledge protocol - verifier side

tocol's announcements are computed by $a_i := \alpha_i()$ and added to the popk-announcement. The collected announcements $a := a_1, \ldots, a_n$ are then sent to the verifier.

2. Challenge: The verifier chooses a challenge c uniformly at random from its challenge space C (the publicly known $\mathbb{Z}_p$). This challenge is sent to the prover.

3. Response (γ): The prover uses her secret sharing scheme to complete the shares of the simulated protocol execution to the given challenge, i. e. $((c_1, \ldots, c_n) \leftarrow \mathsf{Complete}(c, \{c_j \mid t_j \in T\})$. If a protocol is fulfilled add $(r_i := \gamma_i(a_i, c_i), c_i)$ to the popk-responses. Otherwise the responses (r_i, c_i) from the previously simulated transcript t_i is added to the popk-response. The collected responses $r = (r_1, c_1), \ldots, (r_n, c_n)$ are then sent to the verifier.

4. Verify (ψ): After collecting all (a_i, c_i, r_i)-tuples for all given inner protocols $\mathcal{V}_i$, the verifier accepts iff

   - The shares $(c_1, \ldots, c_n)$ are all consistent with c, i. e. $\mathsf{CheckConsistency}(c, (c_1, \ldots, c_n)) = 1$.
   - $\forall i = 1, \ldots, n : \psi_i(a_i, c_i, r_i) = 1$.

6.4.3 Protocol Overview:

In the construction of the (extended) ACS (cf. Section 4.2.3, Section 4.3.3) one will find several interactive protocols. The main goal of the Zero-Knowledge Component is to simplify the generation of these interactive protocols. In the following sections we will explain the realization of the protocols Join/MJoin, ProveNym/VrfyNym, IssCred/RcvCred and ProveCred/VrfyCred.

6.4.3.1 Join/MJoin

To join the system, the user has to register at the system manager. A theoretical description of the protocol is given in Section 4.4.1. During this registration, a master credential is issued to the user which is needed for the ProveCred/VrfyCred (cf. Section 6.4.4.2) later on. The important thing happening during this interaction is that the system manager stores the user within her registry, which includes the user public key, the issued master credential and an

additional element τ which is used during the joining. At the end, the user only has to save the master credential from the system manager to finish the joining.

To use this protocol in combination with a RESTful API as done in the example application, stateful messages as in interactive protocol executions have to be prevented. Therefore we provide this protocol in a non-interactive form as well. So instead of using processes for the user and the system manager, a proof is computed by the user using the Fiat-Shamir heuristic (cf. Section 6.3.2.4), taking the protocol as input. After verifying this proof, the system manager provides the master credential and saves the user in her registry.

6.4.3.2 ProveNym/VrfyNym

The `ProveNymProtocol` encapsulates a `GeneralizedSchnorrProtocol` for the relation described in Section 4.2.3. To construct the protocols, it makes use of the `GeneralizedSchnorrProtocol-Factory`.

6.4.3.3 IssueCred/RcvCred

To actually receive a credential, the user has to interact with an issuer. A theoretical description of this protocol is given in Section 4.4.1. The user performs the proof of knowledge given in Section 4.4.2 with the issuer, using the `GeneralizedSchnorrProtocol(Factory)`. If the verification is successful, the issuer computes a blinded signature and sends it to the user, who unblinds it. If the unblinded credential is a valid signature the user saves it alongside the signed attributes and issuer public key as credential. This whole interaction is encapsulated in processes for both sides, ensuring that they only have the functionality fitting to their role within this interaction.

As for all other interactive protocols, our library also provides a non-interactive version realized with the Fiat-Shamir heuristic (cf. Section 6.3.2.4). The non-interactive version of the protocol will be used in our example application.

6.4.3.4 ProveCred/VrfyCred

Since the ProveCred/VrfyCred protocol is rather complex, it is described in Section 6.4.4.2.

6.4.4 ProveCred/VrfyCred

To get access to a service, a user need to convince a verifier that his credentials fulfill the service's access policy without revealing her attributes or her identity. Therefore we will first describe how we realized the access policy described by the predicate $\phi \in \mathcal{U}_\Phi$ and afterwards how we use it to implement ProveCred/VrfyCred.

6.4.4.1 Policies

The theoretical construction defines predicates in Section 4.1.3 to express properties a user needs to fulfill to get access to a service. These predicates will form the basis for the constructed predicate $\phi \in \mathcal{U}_\Phi$, which additionally specifies how the access requirements are connected, e. g. only need one or another attribute.

To accomplish this, we introduce the idea of *policies* and *sub-policies*. A sub-policy contains a boolean formula over predicates, which only relate to a set of attributes issued by exactly one issuer. Therefore, it contains the ipk and the attribute definitions of that issuer for the issued credential. All needed information to prove the fulfillment of a sub-policy are stored in the `PolicyInformation` object which is constructed by the verifier to describe the requirements for a prover to fulfill.

Both, policy and sub-policy, are represented by a structure referred to as *threshold policy*. A threshold policy is thereby a boolean formula containing *AND*-, *OR*- and *Threshold*-gates and are handled based on the information contained in their leaves, the so called *policy facts*. Following this definition, a policy is a threshold policy over one or more sub-policies, whereby a sub-policy is a threshold policy over predicates which can be fulfilled with a single credential.

This approach enables us to encapsulate the different stages needed to prove fulfillment of the verifiers requirement which may require the possession of credentials of several issuers. Additionally we can use the `PolicyInformation` to store additional data regarding the threshold policy which are required to transform the leaves during the protocol executions. A more detailed explanation of the underlying mechanisms is given in Section 6.4.4.2.

Another problem that needed to be faced during implementation was the selective disclosure. As described in Section 4.2, the user may need to disclose some attribute values during ProveCred/VrfyCred. To realize this, the verifier can add these elements to the `Policy-Information`. As will be described in Section 6.4.4.2, disclosing attributes changes the protocol executed during ProveCred, since the disclosed elements need to be transmitted from the prover to the verifier and are no longer required to be part of the zero-knowledge proof.

6.4.4.2 ProveCred/VrfyCred Implementation

To be able to prove fulfillment of a policy, and therefore of the predicates defined in Section 4.1.3, we decided to use a hierarchical structure of protocols. These protocols encapsulate each other and enable "blackbox"-like behavior as well as easier simulation during the proofs of partial knowledge as described in Section 6.4.2.3. A graphical and hierarchical overview of all used protocols is given in Figure 6.8. The "root" protocol is the `PolicyProvingWithMasterCred-`

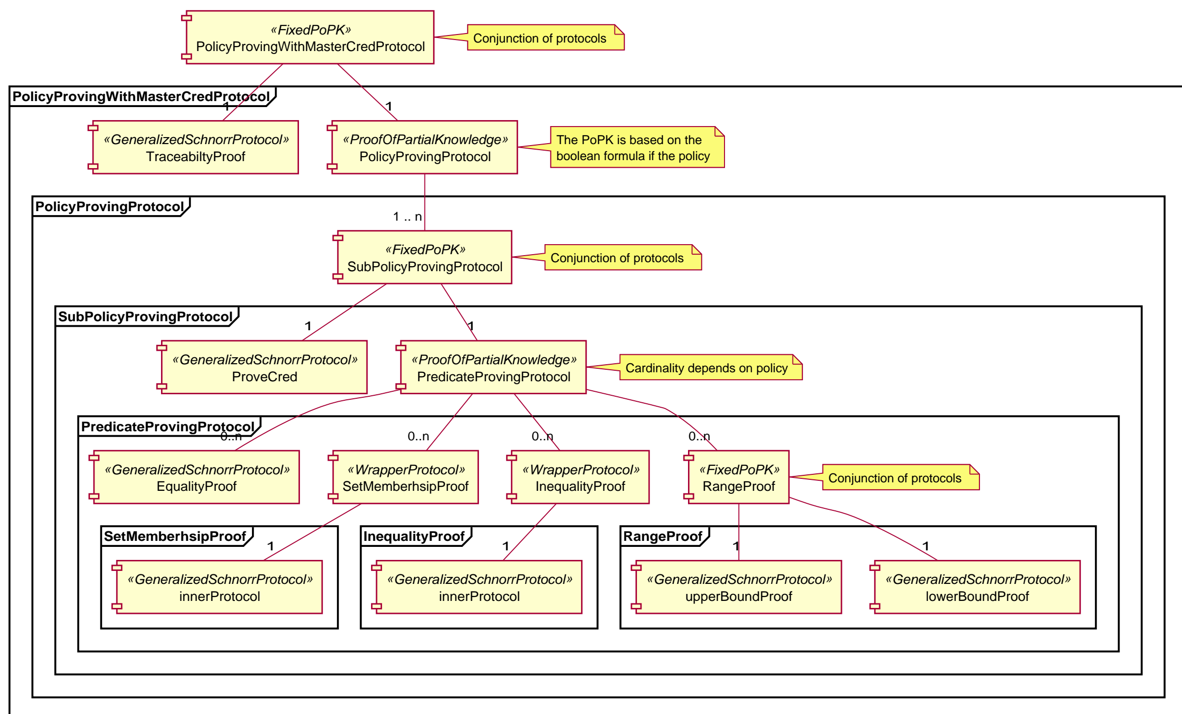


Figure 6.8: Overview of protocols participating in ProveCred

`Protocol` or `PolicyProvingProtocol`. This depends on the fact, whether the proof, that the user has successfully joined the system is included. As outlined in Section 6.4.5 the generation of the outermost protocol depends on the exchange of the common input. At this point the policy is described as threshold policy over sub-policies (c.f. Section 6.4.4.1), so a policy of

which fulfillment can be proven with a single credential of the associated issuer. The common input for the generation of the `PolicyProvingProtocol` (with or without the master credential proof) contains the following elements:

- Defined by the prover:
  - The prover's pseudonym
  - Optional: The prover's randomized master credential
- Defined by the verifier (encapsulated in the `PolicyInformation`):
  - AttributeSpace(s) of the involved issuer(s)
  - The actual policy to be proven
  - Attributes to be disclosed

Additionally the prover needs to supply her `usk`, the open-value for her pseudonym and the credential(s) to be used for proving fulfillment of the policy to the factory generating the protocol. She does not send any of this information to the verifier, since the values are her witnesses. The `PolicyProvingProtocol` contains a proof of partial knowledge as inner protocol as well as a generalized Schnorr protocol for proving the possession of a valid master credential as described in ProveCred of the extended ACS defined in Section 4.3. To initialize the proof of partial knowledge, the policy needs to be transformed to contain Σ-protocol as leaves. Therefore, for each sub-policy a `SubPolicyProvingProtocol` is generated and passed to the `ProofOfPartial-Knowledge`.

For the prover side, the generation of a `SubPolicyProvingProtocol` takes the prover's `usk` as well as the credential used to fulfill the sub-policy as input. During its initialization the protocol will create commitments on each attribute inside the credential and for randomizing the signature of the given credential. During the announcement phase, the `SubPolicyProvingProtocol` sends the onuted commitments on the attributes and the randomized signature. Additionally, the proof of fulfillment of the policy as well as knowledge of the signature containing the necessary attributes and the correct `usk` are send(cf.Section 4.2.3). The proof of fulfillment is delegated to a proof of partial knowledge. As described in Section 6.4.5 the announcement send to the verifier contain the announcements of the inner protocol, the information computed by this protocol and the randomized signature.

To construct the proof of partial knowledge protocol the policy needs to be transformed again. For each leaf in the tree, defining some predicate over an attribute as described in Section 4.1.3, the corresponding Σ-protocol is created. The prover adds her available witnesses and the attributes signed in the credential used to the appropriate protocols.

During the verification phase of the `PolicyProvingProtocol` the call is delegated to the inner protocol(s), where the generalized Schnorr protocol, if needed, verifies the validity of the provided master credential. The proof of partial knowledge also delegates the verify call to its inner `SubPolicyProvingProtocol`(s), which is the information contained in the corresponding announcements and responses to instantiate another proof of partial knowledge over the verifier protocol(s) for the predicates to be proven. Afterwards the whole hierarchy of protocols is executed, where each layer's fulfillment depends on the fulfillment of their respective inner protocols. If all protocols are successfully verified, the fulfillment of the policy has been proven.

6.4.5 Common Input

The theoretical constructions for the various protocols needed during a (zero-knowledge) proof of knowledge (cf. Section 3.7, Section 4.1.3) define a set of "publicly known information" (common input), which is known to both parties prior to the actual protocol execution. In practice we need to somehow ensure that both parties get access to this common input. We developed two

ways to solve this problem of exchanging "publicly known" information which will be described in this section.

The first way is a communication step prior to the protocol generation during which the public information used for generation itself is exchanged. Since communication between (two) parties is not part of the ACS implementation the actual exchange of information has to be provided by the consumer of the exposed API. This option is used to generate the `PolicyProvingProtocol` described detailed in Section 6.4.4.2.

The second approach works by encapsulating the protocol's common input inside of the announcement of the next outer protocol in the hierarchy (cf. Section 6.4.1). It is used in all inner protocols during `ProveCred`. This approach enables us to generate the common input during the announcement phase of the prover side. Afterwards, it is send as part of the protocol's announcements, alongside the announcements of the inner protocol(s), as shown in Figure 6.6 . On the verifier side the common input for the inner protocol is extracted from the announcement during the verify phase and delegates the corresponding verification to the protocol created from it. A more detailed explanation of this approach can be found in Section 6.4.1

6.5 Reputation System

The reputation system enables the user to create ratings for the acquired services. It is important to note that we renamed the ratings to reviews in our library, since the actual rating is a message and not a rating on a scale, especially because our example application showcasing these functionalities has the form of an online shop offering items. This is why our reviews have a specific `Item` within them, instead of a service description. The user creates reviews via `ReviewTokens` which are given to the user once he successfully proves the necessary attributes for the transaction. A token allows the user to write a single review for the `Item`. Even if she successfully requests the `Item` multiple times, only one review is allowed. If the user attempts to publish a second review, the verifier will notice this and simply delete the second review. We achieve this by giving the verifier a link algorithm which can link two reviews from the same issuer on the same item additionally to being able to verify reviews due to public linkability. For this implementation, we reuse the existing structures and protocols of the ACS. The result is that the issuing is almost identical to the issuing of credentials and the rate algorithm also reuses existing parts of protocols while only slightly extending them.

6.5.1 Ratings/Reviews

Within a `ReactReview` there are a multitude of elements which need to be stored. Corresponding to the definition in Section 4.4.2 these are:

- The `Item` the review is written for
- The actual review message, given as `ByteArrayImplementation`
- The public key of the `SystemManager`
- A `GroupElement` for the `linkabilityBasis`
- The public key of the `ReviewTokenIssuer`
- The master credential `PSSignature` (blinded)
- The `ReviewToken` `PSSignature` (blinded)
- The `FiatShamirSignature` for the review
- Two `GroupElements` `L1` and `L2` depending on the `linkabilityBasis` and the usk

All of these elements except for the message need to be passed to the verifier in order to create the verifier side of the protocol for the `FiatShamirSignatureScheme` to verify the review signature. When this is done, the message can be published on the reputation board.

6.5.2 Reusing the Credential Issuing Structure

When designing the reputation system, we can reuse the existing structures of the ACS. Basically, the issuing of `ReviewTokens` is the exact same procedure as issuing credentials. This means that we can reuse all existing protocols and processes used for the credential issuing and only need to extend it to not only work with `Attributes`, but also with Items. The user requests a `ReviewToken` on a specific `Item`, which relates to requesting a credential on a single `Attribute`. In order to do this, we need to hash the `Item` into $\mathbb{Z}_p$ to get the same structure that was used for an `Attribute`. Everything else can then be executed in the exact same way by wrapping the `HashOfItem` and the `Attribute` into a single class `Issuable` and using this structure in a designated `ReviewTokenIssuer` as counterpart for the `CredentialIssuer`.

6.5.3 Reviewing

If the user got the `ReviewToken`, she can write the review she wants to publish. This is done by providing a `FiatShamirSignature` to the `ReviewTokenVerifier`. The computation of the review is similar to the interactive protocol ProveCred/ VrfyCred, but non-interactive. Instead of the part of the protocol where the pseudonym is proven, the user proves the correct values for `L1` and `L2`, which lets the verifier make sure that she can link the review to other reviews by the same user, provided the reviews are on the same `Item`. `L1` contains the public key of the `ReviewTokenIssuer`, the `Item` and the `usk` together with a random element ζ and `L2` contains the linkability basis together with the same random element ζ. To convert the public key of the `ReviewTokenIssuer` and the `Item` to a `GroupElement`, we need to provide a hash function hashing the concatenation of these two elements into $\mathbb{G}_1$. This ζ is only known to the user though. After calculating the `FiatShamirSignature`, `L1` and `L2`, the user fills the rating with all other necessary values described in Section 6.5.1.

6.5.4 Verifying and Linking

The verification algorithm checks if the signature is a valid signature for the `Item` provided in the review. This is done by computing the verifier side of the protocol used for the computation of the `FiatShamirSignature` during the reviewing and then using the `FiatShamirSignatureScheme` with that protocol to verify the signature. Again, this is similar to the ProveCred/ VrfyCred protocol, but instead of doing it interactive it is non-interactive.

The linking algorithm however is a completely new algorithm since it was not needed for credentials. This algorithm works because of the linkability basis from the system manager and the two extra elements `L1` and `L2`, which he can use to link two reviews (cf. Section 4.4.2). In a more sophisticated appplication, the `ReviewVerifier` would run the link algorithm every time a new review is going to be published on the reputation board. She would then check if the new review and any of the other reviews for the same `Item` are from the same user and delete the new one if this is the case.

6.6 API

For simplifying the usage of the credential system, the `acs` module exposes the features of the ACRS in an object-oriented API (Application Programming Interface). This is the layer in the architecture which an application which consumes the library should develop against. Each actor of the system is represented by an interface, namely being:

- `User`
- `Issuer`
- `ReviewTokenIssuer`
- `SystemManager`
- `Verifier`
- `ReviewVerifier`

However, being only interfaces they do not implement any of the actual logic. Therefore, the ACRS, as described in this document, is implemented by classes prefixed with a “React” followed by the actor name. For example, the class `ReactUser` implements the generic interface `User` for the described system. This extra interface layer decouples the API from the actual ACRS construction and allows exchanging the construction without changing the API usage.

Since the full details of the classes and their methods are given in the JavaDoc documentation, this section only gives an introduction into the API and outlines the most important use-cases.

6.6.1 Setup

Before any actor can be created, the public parameters of the system need to be generated. This task is handled by the `ReactPublicParametersFactory` class. The `create()` method of that class will generate the billinear map, the pseudonym public parameters and the required hashing functionality for hashing arbitrary values into the respective groups. Usually this function will be called once by a trusted party and the resulting `ReactPublicParameters` are serialized in order to be shared with the individual actors of the system.

6.6.2 Actor creation

Assuming the availability of the public parameters, the actors of the system can be generated. As they have varying dependencies on others, a short introduction is given for creating each of the actors.

6.6.2.1 System Manager

The system manager does not depend on the information of any other actors. Thus, the only information required for its creation is the public parameter object:

```
1 SystemManager systemManager = new ReactSystemManager(pp);
```

The `systemManager` object then offers methods for handling join requests and retrieving the public key of registered users. Since the object internally holds a registry containing the joined users, the object will need to be serialized and restored if the consuming application is restarted.

6.6.2.2 Issuer

In order to initiate an `Issuer` actor, first the attributes issuable by the issuer need to be determined:

```
1 final List<AttributeDefinition> attributeDefinitions =
2     Arrays.asList(
3         new StringAttributeDefinition("country", ""),
4         new BigIntegerAttributeDefinition(
5             "age", BigInteger.ONE, BigInteger.valueOf(200))
6     );
```

The above code will create a list consisting of the two attributes `country` and `age`, where `country` has no value restriction and `age` is restricted to a value between 1 and 200.

The creation of the issuer then happens using the public parameters and the `attributeDefinitions`:

```
1 Issuer<Attributes, PSCredential> issuer =
2     new ReactCredentialIssuer(pp, attributeDefinitions);
```

6.6.2.3 ReviewTokenIssuer

Since the `ReviewTokenIssuer` fundamentally functions the same as an `Issuer` where the attribute space is a single review token, the class constructor does not require any attribute information:

```
1 Issuer<HashOfItem, ReactRepresentableReviewToken>
2     reviewTokenIssuer = new ReactReviewTokenIssuer(pp);
```

6.6.2.4 User

Creating a `User` also requires to register this `User` at the `SystemManager`. This requires an invocation of the `createNonInteractiveJoinRequest()` method on the `User` and an invocation of `nonInteractiveJoinVerification` on the `SystemManager`:

```
1 ReactUser user = new ReactUser(pp);
2
3 final ReactNonInteractiveJoinRequest joinRequest =
4         user.createNonInteractiveJoinRequest(
5             systemManager.getPublicIdentity()
6         );
7 ReactJoinResponse joinResponse =
8         systemManager.nonInteractiveJoinVerification(joinRequest);
9 user.finishRegistration(joinResponse);
```

In this example, the `systemManager` is directly known by the code which also created the `user`. In a real-world scenario, this usually will not be the case since the `systemManager` is likely to be a self-contained service. In that case, the code which initializes the `User` object then needs the value of `systemManager.getPublicIdentity()` and can invoke the join verification using a remote service call.

After the basic setup of the user is done, it is possible to create pseudonyms:

```
1 Identity identity = user.createIdentity();
```

Here, the `Identity` class represents the shareable pseudonym, as well as the secret open value. When requesting or proving credentials such an identity object has to be passed in order to determine under which pseudonym the proofs should be executed.

6.6.2.5 CredentialVerifier

The `CredentialVerifier` only requires the public parameters and the system manager public identity:

```
1 CredentialVerifier verifier = new ReactCredentialVerifier(
2     pp, systemManager.getPublicIdentity()
3 );
```

The public identity of the system manager is required for verifying policies which mandate that the user registered on the system manager and transmitted the necessary data in order to perform the open operation. The open is of course done by the `systemManager` and not the `Verifier` itself, but the `Verifier` has to check the provided policy proof whether it would allow the open operation.

6.6.2.6 ReactReviewVerifier

The `ReactReviewVerifier` additionally requires the public identity of the `reviewTokenIssuer`:

```
1 ReviewVerifier reviewVerifier = new ReactReviewVerifier(
2     pp,
3     systemManager.getPublicIdentity(),
4     reviewTokenIssuer.getPublicIdentity()
5 );
```

The information about the `reviewTokenIssuer` is required at construction time because, unlike the credential verification, there is no additional policy information being passed for verifying reviews. In the credential verification case, this policy information data includes the public identities of the involved issuers and thus is not required to be known at construction time.

The `ReviewVerifier` still requires the system manager public identity since it includes the parameters for performing linking checks between two reviews.

6.6.3 Actor interaction

After the actors got instantiated, they are usually running in different processes or even entirely different machines. This makes serialization of the objects to transmit mandatory. The PBC library provides the required features for this:

```
1 final JSONConverter converter = new JSONConverter();
2
3 // serialize
4 final String json = converter.serialize(
5     new RepresentableRepresentation(<object to serialize>));
6
7 // deserialize
8 Representation representation = converter.deserialize(json);
9 StandaloneRepresentable object =
10     (StandaloneRepresentable)representation.repr()
11         .recreateRepresentable();
```

Since all the objects which need to be transited as part of invoking the library are derived from the `StandaloneRepresentable` class, the above code snippet should be enough to handle all the serialization requirements for invoking the API. However, for simplicity serialization will be left out in the following examples and it is assumed that the code has direct access to all the actor instances.

Futhermore, only the “non-interactive” variant of issuing and proving credentials is introduced here. The library also supports interactive variants which do not utilize the Fiat-Shamir heuristicSection 6.3.2.4 but require an additional interaction in order to query the challenge of the issuer or verifier. Details for the interactive variants are given in the JavaDoc of the respective methods.

6.6.3.1 Issuing of credentials

The process of issuing credentials for attributes requires the two actors `ReactUser` and `ReactCredentialIssuer` who execute the algorithms IssCred and RcvCred as described in Construction 4.25. Due to user centric control in the credential system, the process needs to be initiated by the user. She has to pick the set of requested attribute values and creates a credential request for the issuer under a specific pseudonym identity:

```
1 final StringAttributeDefinition countryDef =
2     (StringAttributeDefinition)issuer.getPublicIdentity()
3         .getAttributeSpace().get("age");
```

```
4
5 final BigIntegerAttributeDefinition ageDef =
6     (BigIntegerAttributeDefinition)issuer.getPublicIdentity()
7         .getAttributeSpace().get("age");
8
9 AttributeNameValuePair country =
10    countryDef.createAttribute("Germany");
11
12 AttributeNameValuePair age =
13    ageDef.createAttribute(BigInteger.valueOf(20));
14
15 attributes = new Attributes(
16    new AttributeNameValuePair[]{country, age}
17 );
18
19 final ReactCredentialNonInteractiveResponseHandler
20 credentialResponseHandler =
21    reactUser.createNonInteractiveIssueCredentialRequest(
22        issuer.getPublicIdentity(),
23        identity, attributes);
```

Since the issue process requires the user to hold state until the answer of the issuer arrives, the `createNonInteractiveIssueCredentialRequest` returns a `ResponseHandler`. This object contains the necessary information for unblinding the credential issued by the issuer and thus needs to be available at the time the issue response is received:

```
1 final ReactCredentialIssueResponse
2     nonInteractiveCredentialResponse =
3         issuer.issueNonInteractively(
4             credentialResponseHandler.getRequest()
5         );
6 reactUser.receiveCredentialNonInteractively(
7     credentialResponseHandler,
8     nonInteractiveCredentialResponse);
```

The newly issued credential is now registered within the `user` and can be used in order to fulfill policies which require a credential containing attributes issued by this issuer. Since each issuer is only able to issue credentials for a single attribute space, the user can maintain a simple mapping between the attribute space and the issued credential.

The code example, as it is, currently allows the user to create credentials with arbitrary attribute values. In a real system, the application layer of the issuer side needs to check that the client only requested permissable attributes before forwarding the credential request to the actual `issuer.issueNonInteractively()` call.

6.6.3.2 Proving of credentials

In order to use the newly acquired credentials for proving the fulfillment of a policy, a `Policy-Information` object has to be created. For this a fluent api[3] is shipped as part of the library:

```
1 policyInformation =
2     policy(pp).forIssuer(issuer.getPublicIdentity())
3         .attribute("age").isInRange(18, 200)
4         .attribute("country").isInSet("Germany", "USA")
5         .build();
```

The resulting `policyInformation` object can then be used by the `user` in order to create a proof for the given policy:

[3] https://martinfowler.com/bliki/FluentInterface.html

```
1 final NonInteractivePolicyProof proof =
2     reactUser.createNonInteractivePolicyProof(
3         reactIdentity,
4         policyInformation,
5         verifier.getIdentity()
6     );
```

Notable here is the referencing of a `verifier` as part of the proof. This makes sure that the created `proof` is only positively verified by the pinned `verifier`. Chapter 5 can be consulted for the security background on this.

The verifier side can then verify the `proof` together with the `policyInformation`:

```
1 VerificationResult verificationResult =
    verifier.verifyNonInteractiveProof(proof, policyInformation);
2     if (verificationResult.isVerify()) {
3         // perform application dependent logic
4     }
```

The `verificationResult`, given that the verification was successful, also contains information about the used pseudonym and information for system manager, namely being the blinded master credential. Especially the pseudonym can be interesting to query by the application layer in order to provide services which are linked to other requests under the same pseudonym.

6.6.3.3 Reviewing of items

In order to be able to issue reviews for an item, the user first has to request a review token from a review token issuer. This process is very similar to the credential request procedure:

```
1 final ReactReviewTokenNonInteractiveResponseHandler
2     reviewTokenResponseHandler =
3     reactUser.createNonInteractiveIssueReviewTokenRequest(
4         reviewTokenIssuer.getPublicIdentity(),
5         identity, "Game of Thrones".getBytes()
6     );
7
8 final ReactReviewTokenIssueResponse
9     nonInteractiveReviewTokenResponse =
10            reviewTokenIssuer.issueNonInteractively(
11                reviewTokenResponseHandler.getRequest()
12            );
13
14 reactUser.receiveReviewTokenNonInteractively(
15     reviewTokenResponseHandler,
16     nonInteractiveReviewTokenResponse
17 );
```

The main notable difference is, that instead of attributes an arbitrary review subject has to be passed as a byte array. This byte array has to represent the specific item which is supposed to be rated. The review token will only be valid for that item. Again, checks for whether the user is actually allowed to request this review token are left out here for brevity but need to be handled by the application layer. In real scenarios the token issuing could, for example, be combined with a verifier which first checks for a fulfillment of a policy before issuing a review token.

After the review token is registered, the user is able to issue a review:

```
1 Review review = reactUser.createReview(
2     "5 of 5 stars!".getBytes(),
3     reviewTokenIssuer.getPublicIdentity(),
4     "Game of Thrones".getBytes()
5 );
```

This `review` object can then be shared with other parties of the application. The `React-RepresentableReview` can be a helpful utility for simplifying the serialization and deserialization process. However, details for this class are left out here and can be found in the JavaDoc.

6.6.3.4 Verification of reviews

Created reviews can be verified under two different aspects. One being that the reviewer had the necessary review token for issuing a review, the other that one reviewer did not author multiple reviews under the same review token.

The first verification is handled by the `verify()` method of the `reviewVerifier`:

```
1 boolean isValid = reviewVerifier.verify(review);
```

The usage should be clear, a `true` implies a valid review, `false` an invalid one.

Checking for multiple reviews under the same token has to be done by a pairwise compare of reviews:

```
1 boolean sameUser =
2     reviewVerifier.areFromSameUser(review1, review2);
```

If the two compared reviews are authored by the same reviewer under the same review token the method would return `true`. Thus, checking for duplicate reviews in lists has to be done by comparing each combination of reviews with each other.

6.7 Example Application

As part of the practical work, we also developed a simple demo application which conceptually serves as a mean of proving a wide range of technical concepts and features of our library. The application represents a basic shop system where different types of customers can sell, buy and rate custom products.

The following sub-sections provide an overview of the example application, showing the primary use cases from a user perspective and discussing its architecture.

6.7.1 Architecture

We chose microservice architecture to be the application architectural style. In real-world scenarios, any role in our system (issuer, verifier, shop owner, etc.) will probably reside in a separate entity (service). Figure 6.9 gives an overview of the example application architecture.

The following illustration shows the implemented micro-services and their main role in the example application:

System Manager Service This service hosts one instance of the `ReactSystemManager`. The service exposes `ReactSystemManager.nonInteractiveJoinVerification()` API to the `/join` end-point, which enables clients to perform non-interactive join to the system.

Base Credential Issuing Service This service embodies one `ReactCredentialIssuer` object. That issuer has the following attribute definitions in her the attributes space: birth-date, country, body-size and legal-entity. This service is used to issue a credential upon a given set of values for those defined attributes.

Shop Hosting Credential Issuing Service Another `ReactCredentialIssuer` object resides in this service. The owner of a credential, issued by this issuer, is granted to host new shop items to the shop system.

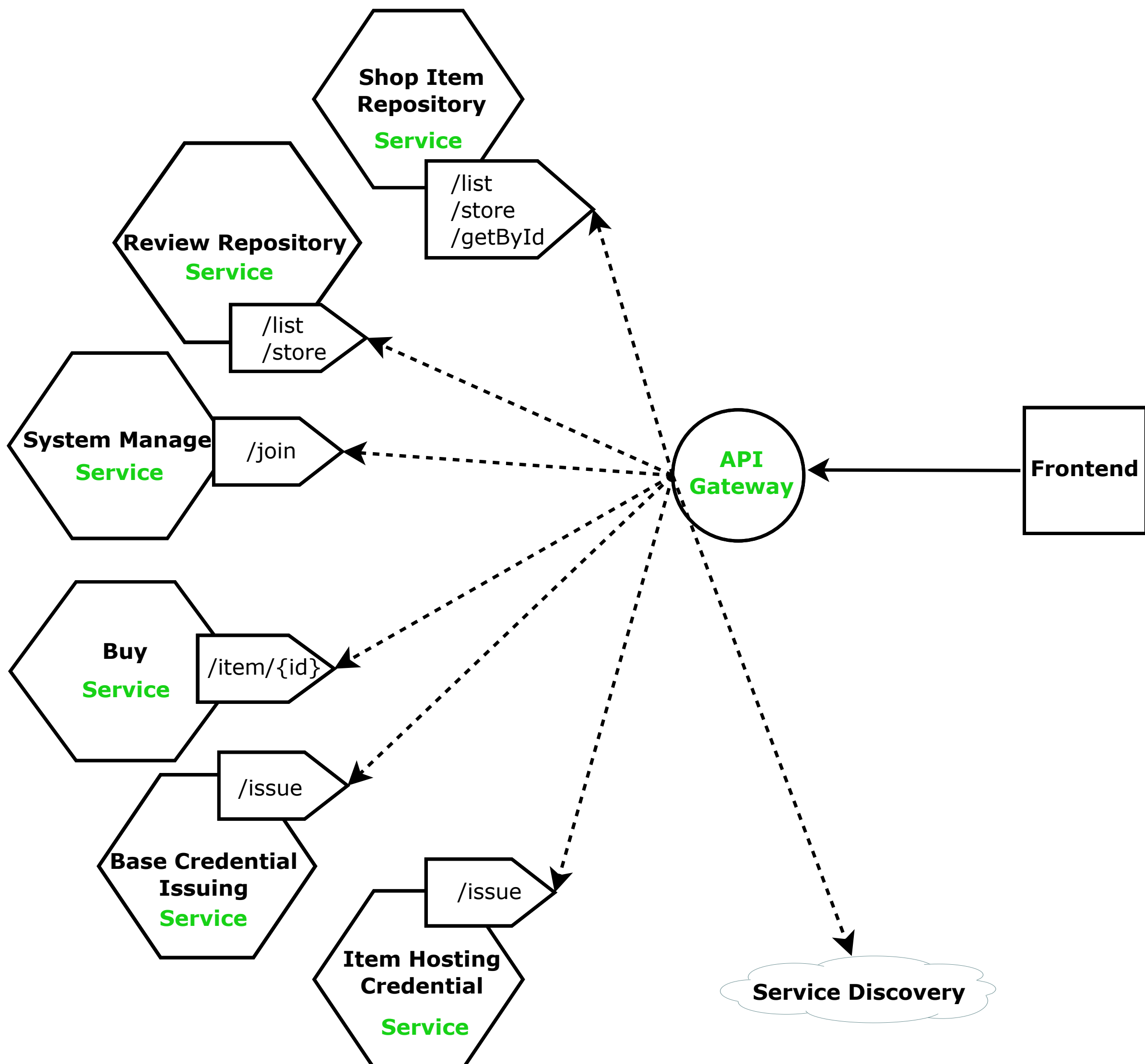


Figure 6.9: The micro-service architecture of the example application

Review Repository Service The review repository service can be seen as a storage object for the product reviews. It offers two public API end-points; `/store` and `/list`. As the names indicate, the former is used to store a new review for a product while the latter is used to fetch one existing product reviews.

Buy Service The buy service has two main entities; one `ReactCredentialVerifier` instance and one `ReactReviewTokenIssuer` instance. It allows buyers to buy a certain product depending on the `NonInteractivePolicyProof` object they send along with the buying request. The buying request also contains one instance of `ReactNonInteractiveReviewTokenRequest`. The buying verifier verifies the sent proof against the product policy. After a successful verification process, the service computes a `ReactReviewTokenIssueResponse` object using the review token request via the review token issuer and sends it back to the buyer.

Shop Item Repository Service This service represents the product management component in our shop system. Through this service, clients can fetch all hosted products as well as a single product with a particular `ID`. Additionally, service clients with a valid product hosting credential can use the `/store` public API for hosting a new product. The service contains a different instance of the `ReactCredentialVerifier` which is mainly responsible for validating the proof that is sent along with the product hosting request.

6.7.2 Use Cases

Before we start discussing the main scenarios, a good point to highlight is that one instance of `ReactPublicParameters` object is common and shared among all parties (components and services) involved in the example application.

6.7.2.1 Initialization

When the application is first started, a new instance of `ReactUser` is created. The initialization phase comprises three steps:

1. Setting up the user
2. Issuing a base credential
3. Listing shop products

User Setup The frontend application communicates with the *System Manager Service* via its Rest API for joining the system by sending a join request. When the join succeeds, an initial pseudonym is created and stored by the `ReactUser`.

Issuing The Base Credential After the user setup, the frontend application prompts a modal dialog to the user containing an input form for some specific attribute values[4] upon which the user wants to have a credential (cf. Figure 6.10). When the user clicks the register button, the client creates a `NonInteractiveIssuableRequest` with the chosen attribute values and sends it to the *Base Credential Issuing Service* to retrieve and store the first credential.

Listing Shop Products As part of the initialization, the shop frontend also communicates with *The Shop Item Repository Service* via its REST API to fetch the information of all hosted products and lists them in the main product list view.

[4]In this case, the values conform to the attribute space of the issuer from which the user wants to issue a credential

Figure 6.10: Issuing a base credential form

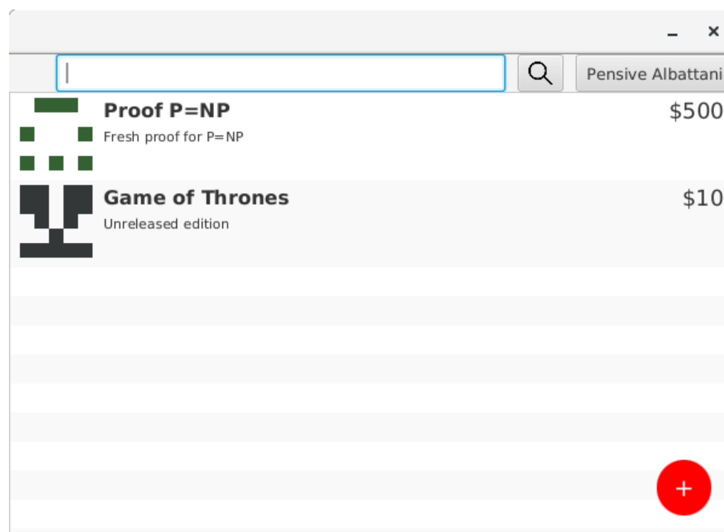


Figure 6.11: The main window in the example application after a successful initialization

After a successful initialization, the user is redirected to a new window with a list of all hosted products as shown in Figure 6.11.

6.7.2.2 Pseudonym Creation and Product Hosting Credentials Issuing

The product list window populates all available shop products along with options to create new pseudonyms, host new products in the shop and request a credential for products hosting as shown in Figure 6.12.

On the top right corner of the main window view, the user can click on the pseudonym switcher button. As a result, a pop-up window with all available pseudonyms appears (Figure 6.12). In the window, the user can create new pseudonyms, as a result of pushing the "Create Pseudonym" button, and delete already existing Pseudonyms. The user can select one of the available pseudonyms in the list to attach it in the process of issuing a "shop item hosting" credential which happens after clicking on the register button.

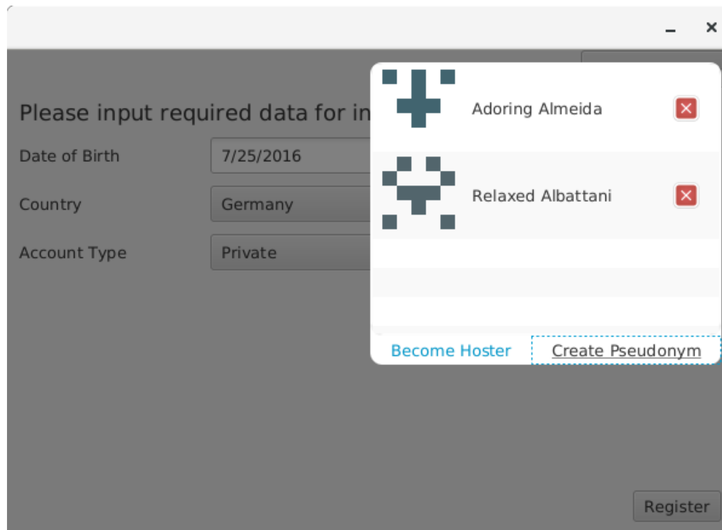


Figure 6.12: Pseudonym switcher pop-up view

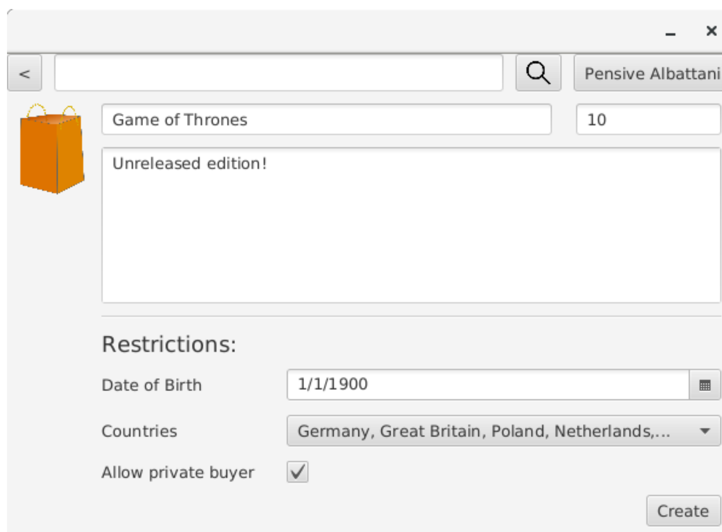


Figure 6.13: View for creating new product listings

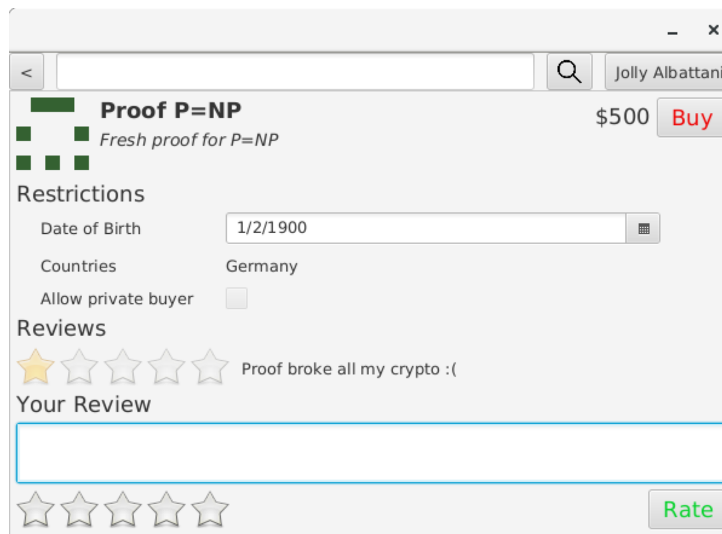


Figure 6.14: View for buying and rating products

6.7.2.3 Hosting A New Product

When the shop item hosting credential is obtained, the user can start hosting new shop items on the shop system. Figure 6.13 shows this product creation form. Using the form, user can input the product information (name, description, ...) along with the buying policy information values (age, country, ...). Clicking the “Create” button, the client app constructs a new proof which confirms the client ability to host new shop items. In other words, it proves the possession of a valid shop hosting credential. Then it sends the new product information along with the proof object to the *Shop Item Repository Service* where the shop hosting credential verifier resides.

6.7.2.4 Buying and Rating A Product

When the user clicks on one of the products in the products list, she navigates to the product details view. There, the customer can buy and, later on, rate that product (cf. Figure 6.14). Based on the buying credential she obtained from the *Base Credential Issuing Service*, the buyer constructs the proof along with a review token request and sends them as a buying request to the *Buy Service*. In response, the buyer receives a review token to be used, later on, in the product rating. The review token is unblinded and stored by `ReactUser`.

7 Conclusion

In this document, we have given a theoretical construction for a combination of an anonymous credential system and a reputation system.

This combination offers anonymous access control while also providing a rating mechanism. Within the system, we allow almost arbitrary predicates as access policy by using concatenations of different policies and various different proof mechanics such as range proofs and proofs of partial knowledge.

Additionally, we also provide documentation and guidelines how to use the library implementing this system. This library can then be used for any application using anonymous credentials for access control while our guidelines ensure the security of the application.

The library was developed in a generic way by using interfaces throughout the whole project, separating our specific implementation from general concepts. As a consequence, consuming our library is not coupled with the explicit constructions we are using. Only the zero-knowledge component exclusively supports generalized Schnorr protocols and would need a substantial expansion to support other protocols as well. However, supporting all forms of generalized Schnorr protocols leaves the zero-knowledge component with a huge set of problems that can be solved and is sufficient for our system. This is illustrated by the different compositions of policies that we are able to use (cf. Section 6.4.4.1).

There are still functionalities which are not fully integrated in the construction or the implementation. This includes revocation of users, revocation of credentials as well as editability and deletability of ratings. While they are not integrated, at least the revocation is mentioned in Section 4.5. Therefore this would be a good point to start an extension of our current system.

Bibliography

[AAS16] Hiroaki Anada, Seiko Arita, and Kouichi Sakurai. *Proof of Knowledge on Monotone Predicates and its Application to Attribute-Based Identifications and Signatures.* Cryptology ePrint Archive, Report 2016/483. http://eprint.iacr.org/2016/483. 2016.

[Ate+05] Giuseppe Ateniese, Jan Camenisch, Susan Hohenberger, and Breno de Medeiros. *Practical Group Signatures without Random Oracles.* Cryptology ePrint Archive, Report 2005/385. http://eprint.iacr.org/2005/385. 2005.

[BC93] Amos Beimel and Benny Chor. "Universally Ideal Secret Sharing Schemes (Preliminary Version)". In: *Advances in Cryptology – CRYPTO'92.* Ed. by Ernest F. Brickell. Vol. 740. Lecture Notes in Computer Science. Springer, Heidelberg, August 1993, pp. 183–195.

[BN06] Mihir Bellare and Gregory Neven. "Multi-signatures in the plain public-Key model and a general forking lemma". In: *ACM CCS 06: 13th Conference on Computer and Communications Security.* Ed. by Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati. ACM Press, October 2006, pp. 390–399.

[BR93] Mihir Bellare and Phillip Rogaway. "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols". In: *ACM CCS 93: 1st Conference on Computer and Communications Security.* Ed. by V. Ashby. ACM Press, November 1993, pp. 62–73.

[BL90] Josh Cohen Benaloh and Jerry Leichter. "Generalized Secret Sharing and Monotone Functions". In: *Advances in Cryptology – CRYPTO'88.* Ed. by Shafi Goldwasser. Vol. 403. Lecture Notes in Computer Science. Springer, Heidelberg, August 1990, pp. 27–35.

[BPW12] David Bernhard, Olivier Pereira, and Bogdan Warinschi. "How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios". In: *Advances in Cryptology – ASIACRYPT 2012.* Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Springer, Heidelberg, December 2012, pp. 626–643. DOI: 10.1007/978-3-642-34961-4_38.

[Bic+10] Patrik Bichsel, Jan Camenisch, Gregory Neven, Nigel P. Smart, and Bogdan Warinschi. "Get Shorty via Group Signatures without Encryption". In: *SCN 10: 7th International Conference on Security in Communication Networks.* Ed. by Juan A. Garay and Roberto De Prisco. Vol. 6280. Lecture Notes in Computer Science. Springer, Heidelberg, September 2010, pp. 381–398.

[Bla97] Kelly Black. "Classroom note: Putting constraints in optimization for first-year calculus students". In: *SIAM review* 39.2 (1997), pp. 310–312.

[BB17] Johannes Blömer and Jan Bobolz. "Delegatable Attribute-based Anonymous Credentials from Dynamically Malleable Signatures". Unpublished paper as of 14.03.18. 2017.

[BJK15] Johannes Blömer, Jakob Juhnke, and Christina Kolb. "Anonymous and Publicly Linkable Reputation Systems". In: *FC 2015: 19th International Conference on Financial Cryptography and Data Security.* Ed. by Rainer Böhme and Tatsuaki Okamoto. Vol. 8975. Lecture Notes in Computer Science. Springer, Heidelberg, January 2015, pp. 478–488. DOI: 10.1007/978-3-662-47854-7_29.

[Bon98] Dan Boneh. “The decision Diffie-Hellman problem”. In: *Third Algorithmic Number Theory Symposium (ANTS)*. Vol. 1423. Lecture Notes in Computer Science. Invited paper. Springer, Heidelberg, 1998.

[BB04] Dan Boneh and Xavier Boyen. “Short Signatures Without Random Oracles”. In: *Advances in Cryptology – EUROCRYPT 2004*. Ed. by Christian Cachin and Jan Camenisch. Vol. 3027. Lecture Notes in Computer Science. Springer, Heidelberg, May 2004, pp. 56–73.

[CCs08] Jan Camenisch, Rafik Chaabouni, and abhi shelat. “Efficient Protocols for Set Membership and Range Proofs”. In: *Advances in Cryptology – ASIACRYPT 2008*. Ed. by Josef Pieprzyk. Vol. 5350. Lecture Notes in Computer Science. Springer, Heidelberg, December 2008, pp. 234–252.

[CL01] Jan Camenisch and Anna Lysyanskaya. “An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation”. In: *Advances in Cryptology – EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Vol. 2045. Lecture Notes in Computer Science. Springer, Heidelberg, May 2001, pp. 93–118.

[CL03] Jan Camenisch and Anna Lysyanskaya. “A Signature Scheme with Efficient Protocols”. In: *SCN 02: 3rd International Conference on Security in Communication Networks*. Ed. by Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano. Vol. 2576. Lecture Notes in Computer Science. Springer, Heidelberg, September 2003, pp. 268–289.

[CL04] Jan Camenisch and Anna Lysyanskaya. “Signature Schemes and Anonymous Credentials from Bilinear Maps”. In: *Advances in Cryptology – CRYPTO 2004*. Ed. by Matthew Franklin. Vol. 3152. Lecture Notes in Computer Science. Springer, Heidelberg, August 2004, pp. 56–72.

[CS03] Jan Camenisch and Victor Shoup. “Practical Verifiable Encryption and Decryption of Discrete Logarithms”. In: *Advances in Cryptology – CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. Lecture Notes in Computer Science. Springer, Heidelberg, August 2003, pp. 126–144.

[CS97] Jan Camenisch and Markus Stadler. “Efficient Group Signature Schemes for Large Groups (Extended Abstract)”. In: *Advances in Cryptology – CRYPTO’97*. Ed. by Burton S. Kaliski Jr. Vol. 1294. Lecture Notes in Computer Science. Springer, Heidelberg, August 1997, pp. 410–424.

[CLs10] Rafik Chaabouni, Helger Lipmaa, and abhi shelat. “Additive Combinatorics and Discrete Logarithm Based Range Protocols”. In: *ACISP 10: 15th Australasian Conference on Information Security and Privacy*. Ed. by Ron Steinfeld and Philip Hawkes. Vol. 6168. Lecture Notes in Computer Science. Springer, Heidelberg, July 2010, pp. 336–351.

[CL06] Melissa Chase and Anna Lysyanskaya. “On Signatures of Knowledge”. In: *Advances in Cryptology – CRYPTO 2006*. Ed. by Cynthia Dwork. Vol. 4117. Lecture Notes in Computer Science. Springer, Heidelberg, August 2006, pp. 78–96.

[CK93] Benny Chor and Eyal Kushilevitz. “Secret Sharing Over Infinite Domains”. In: *Journal of Cryptology* 6.2 (1993), pp. 87–95.

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. “Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols”. In: *Advances in Cryptology – CRYPTO’94*. Ed. by Yvo Desmedt. Vol. 839. Lecture Notes in Computer Science. Springer, Heidelberg, August 1994, pp. 174–187.

[Dam10] Ivan Damgård. “On Σ-protocols”. `http://www.cs.au.dk/~ivan/Sigma.pdf`. 2010.

[FS90a] Uriel Feige and Adi Shamir. “Witness Indistinguishable and Witness Hiding Protocols”. In: *22nd Annual ACM Symposium on Theory of Computing*. ACM Press, May 1990, pp. 416–426.

[FS90b] Uriel Feige and Adi Shamir. “Zero Knowledge Proofs of Knowledge in Two Rounds”. In: *Advances in Cryptology – CRYPTO'89*. Ed. by Gilles Brassard. Vol. 435. Lecture Notes in Computer Science. Springer, Heidelberg, August 1990, pp. 526–544.

[FS87] Amos Fiat and Adi Shamir. “How to Prove Yourself: Practical Solutions to Identification and Signature Problems”. In: *Advances in Cryptology – CRYPTO'86*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Springer, Heidelberg, August 1987, pp. 186–194.

[GPS08] Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. “Pairings for cryptographers”. In: *Discrete Applied Mathematics* 156.16 (2008), pp. 3113–3121. DOI: `10.1016/j.dam.2007.12.010`.

[Gol01] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Vol. 1. Cambridge, UK: Cambridge University Press, 2001, pp. xix + 372.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. “A Digital Signature Scheme Secure Against Adaptive Chosen-message Attacks”. In: *SIAM Journal on Computing* 17.2 (April 1988), pp. 281–308.

[Gün90] Christoph G. Günther. “An Identity-Based Key-Exchange Protocol”. In: *Advances in Cryptology – EUROCRYPT'89*. Ed. by Jean-Jacques Quisquater and Joos Vandewalle. Vol. 434. Lecture Notes in Computer Science. Springer, Heidelberg, April 1990, pp. 29–37.

[LLX07] Jiangtao Li, Ninghui Li, and Rui Xue. “Universal Accumulators with Efficient Nonmembership Proofs”. In: *ACNS 07: 5th International Conference on Applied Cryptography and Network Security*. Ed. by Jonathan Katz and Moti Yung. Vol. 4521. Lecture Notes in Computer Science. Springer, Heidelberg, June 2007, pp. 253–269.

[Ngu05] Lan Nguyen. “Accumulators from Bilinear Pairings and Applications”. In: *Topics in Cryptology – CT-RSA 2005*. Ed. by Alfred Menezes. Vol. 3376. Lecture Notes in Computer Science. Springer, Heidelberg, February 2005, pp. 275–292.

[PM04] Andreas Pashalidis and Chris J. Mitchell. “A Security Model for Anonymous Credential Systems”. In: *Information Security Management, Education and Privacy*. Ed. by Yves Deswarte, Frédéric Cuppens, Sushil Jajodia, and Lingyu Wang. Boston, MA: Springer US, 2004, pp. 183–199.

[Ped92] Torben P. Pedersen. “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing”. In: *Advances in Cryptology – CRYPTO'91*. Ed. by Joan Feigenbaum. Vol. 576. Lecture Notes in Computer Science. Springer, Heidelberg, August 1992, pp. 129–140.

[PS15] David Pointcheval and Olivier Sanders. *Short Randomizable Signatures*. Cryptology ePrint Archive, Report 2015/525. `http://eprint.iacr.org/2015/525`. 2015.

[PS16] David Pointcheval and Olivier Sanders. “Short Randomizable Signatures”. In: *Topics in Cryptology – CT-RSA 2016*. Ed. by Kazue Sako. Vol. 9610. Lecture Notes in Computer Science. Springer, Heidelberg, February 2016, pp. 111–126. DOI: `10.1007/978-3-319-29485-8_7`.

[PS96] David Pointcheval and Jacques Stern. “Security Proofs for Signature Schemes”. In: *Advances in Cryptology – EUROCRYPT'96*. Ed. by Ueli M. Maurer. Vol. 1070. Lecture Notes in Computer Science. Springer, Heidelberg, May 1996, pp. 387–398.

[Sch91] Claus-Peter Schnorr. “Efficient Signature Generation by Smart Cards”. In: *Journal of Cryptology* 4.3 (1991), pp. 161–174.

[Sch01] Berry Schoenmakers. “Some efficient zeroknowledge proof techniques”. Slides presented at the *International Workshop on Cryptographic Protocols*. March 2001.

[Sch05] Berry Schoenmakers. “Interval proofs revisited”. Slides presented at the *International Workshop on Frontiers in Electronic Elections*. September 2005.

[15] *Secure Hash Standard (SHS)*. National Institute of Standards and Technology, NIST FIPS PUB 180, U.S. Department of Commerce. 2015.

[Sha79] Adi Shamir. “How to Share a Secret”. In: *Communications of the Association for Computing Machinery* 22.11 (November 1979), pp. 612–613.

[WYY05] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. “Finding Collisions in the Full SHA-1”. In: *Advances in Cryptology – CRYPTO 2005*. Ed. by Victor Shoup. Vol. 3621. Lecture Notes in Computer Science. Springer, Heidelberg, August 2005, pp. 17–36.

[Zho+16] Zhe Zhou, Tao Zhang, Sherman S. M. Chow, Yupeng Zhang, and Kehuan Zhang. “Efficient Authenticated Multi-Pattern Matching”. In: *ASIACCS 16: 11th ACM Symposium on Information, Computer and Communications Security*. Ed. by Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang. ACM Press, May 2016, pp. 593–604.