Specification of the Identity Mixer Cryptographic Library

Version 2.3.2

IBM Research – Zurich

December 15, 2010

Abstract

As we are transforming into an digital society, it is vital that we protect our data in all of our transactions. This requires that transactions are securely authenticated, and that we protect privacy by not revealing more about ourselves than necessary. Anonymous credentials promise to address both of these seemingly opposing requirements at the same time. Anonymous credentials are essentially a privacy-enhancing public-key infrastructure which require standardization to be widely used. Anonymous credential systems are far more complex than ordinary signature schemes since they provide more functionality in order to address all of the requirements of a public key infrastructure with privacy-protection. Unfortunately, the description of these features are spread over many research papers and it is often not clear how they could all be securely integrated into a single system. This paper describes the Identity Mixer anonymous credential system that integrates cryptographic techniques from many sources to build an anonymous credential system with a rich feature set. The aim of this paper is to stimulate standardization effort towards a privacy-enhancing public-key infrastructure.

Contents

1 Introduction

The amount of our daily transactions that we perform electronically is rising. Many of us use the Internet on a daily basis for purposes ranging from accessing information to electronic commerce and e-banking to interactions with government bodies. Securing these transactions requires the use of strong authentication. Electronic authentication tokens and mechanisms that provide authentication become common, not only for the use with the Internet. Indeed, electronic identity cards, authentication by mobile phone, and RFID tokens are spreading fast.

These authentication mechanisms unfortunately have the shortcoming that they label users with a unique identifier. This is a risk to users' privacy because transactions by the same user can be linked together. This lack of privacy is typically not a problem for e-government applications. However, such a government issued strong root of trust is very attractive for so-called secondary use by the commercial sector. In this application area, unique identification is often inappropriate, attribute-based authentication mostly desired and privacy important to make the services sustainable. A position paper issued in February 2009 by ENISA[1] on "Privacy Features of European eID Card Specifications" underlines the need for "privacy-respecting use of unique identifiers" in emerging European eID cards, and explicitly refers to the emerging anonymous credentials technologies ("privacy-enhanced PKI tokens" in their terminology), as having significant potential in this arena.

Anonymous credentials [Bra95a, Bra95b, CL01, Cha85, Dam90], allow an identity provider to issue a credential (or certificate) to a user. This credential contains attributes of the user such as address or date of birth but also the user's rights or roles. Using the credential, the user can later prove to a third party that she possesses a credential containing a given attribute or role without revealing any other information stored in the credential. For instance, the user could use a government-issued anonymous ID credential to prove that she is of age, i.e., that she possesses a credential that contains a date of birth which is further in the past than 21 years. Thus, anonymous credentials promise to be an important technology in protecting users' privacy in an electronic environment.

There is a large body of research on anonymous credential systems and a number of different methods or algorithms are described in the literature. In addition to the basic functionality of an anonymous credential system, i.e., to issue a credential and then later to selectively reveal attributes contained in credentials, there are extensions proposed in the literature to meet the requirements of real world deployment. These extensions include the revocation of credentials [BDD07, CKS09, CL04, NFHF09], revocation of anonymity [CL01], encoding binary attributes efficiently [CG08], or to verifiably encrypt attributes under some third party's encryption key [CD00, CS03]. These important features were described somewhat independently, with different setup assumptions and trust models.

Based on these papers we have implemented a unified system called the the *Identity Mixer Anonymous Credential System*, which takes the form of a cryptographic library. This paper presents a high level specification of the system, based on the full specification with all cryptographic details [IBM09]. It is similar to a cryptographic library that offers, for example, implementations of the RSA or DSA signature schemes: it offers all the functionalities required to establish a pseudonym, issue a credential containing different attributes to a pseudonym, and different ways of proving possession of a credential. The ways to prove possession of a credential offered are the disclosure of a selected subset of the attributes contained in the credential, to prove that an (integer) attribute lies in a given range, to prove that an attribute is verifiably encrypted under some third party's public key, or that a cryptographic commitment contains a specific attribute. The source code of the Identity Mixer library has been made publicly available.

2 Overview of an Anonymous Credential System

An anonymous credential system involves the roles of *issuers*, *recipients*, *provers* and *verifiers* (or *relying parties*). Parties acting in those roles execute the issuing protocol, where a credential for the recipient is created by the issuer, or the proving protocol, where the owner creates a proof on behalf of the verifier. An entity (for example, user, company, government) can assume any role during each protocol run. For instance, a company can act as verifier and run the proof protocol with a user before assuming the role of the issuer and running the issuance protocol (possibly with the same user). Finally, an extended

[1]ENISA: European Network and Information Security Agency `http://www.enisa.europa.eu`

credential system requires the role of trusted third parties who performs tasks such as anonymity revocation, credential revocation, or decryption of (verifiably) encrypted attributes. Usually organizations or governments assume the roles the issuer, verifier and trusted party, and natural persons the ones of recipient and prover.

Note, all parties in an anonymous credential system agree on general system parameters that define the bit length of all relevant parameters as well as the groups that will be used. In practice, these parameters can be distributed together with the code and they must be authenticated.

To participate a user needs to choose her *master secret key* based on the group parameters of the system. This secret allows her to derive pseudonyms, which she can use similar to a session identifier, that is, it allows the communication partner to link the actions of the user. However, the user can create new pseudonyms at her discretion and all pseudonyms are unlinkable unless the user proves that they are based on the same master secret key. Certain scenarios require one user only having one pseudonym with an organization, where we call such pseudonym a domain pseudonym. In addition to being used for pseudonym generation, the master secret will be encoded into every credential. This constitutes a sharing prevention mechanism as sharing one credential implies sharing all credentials of a user.

The setup procedure for issuers and trusted parties consists of generating public key pairs, create a specification of the services they offer and publish the specification as well as the public key. As an example, an issuer runs the issuer key generation and publishes the structure(s) of the credential(s) it is willing to issue together with its public key.

Let us now elaborate on the issuing and the proving protocol. The credential *issuance protocol* is carried out between an issuer and a recipient with the result of the recipient having a credential. The credential consists of a set of attribute values as well as cryptographic information that allows the owner of the credential (i.e., the recipient) to create a *proof of possession* (also called 'proof of ownership' or 'proof'). When encoding the values into a credential, the issuer and recipient agree on which values the issuer learns and which will remain unknown to it, that is, they agree on a credential structure. In addition, they agree on the values that will be encoded.

The *proving protocol* requires a prover and a verifier to interact, that is, the owner of one or several credentials acts as prover in the communication with a verifier. Firstly, the entities define (interactively) what statement will be proved about which attribute value. Secondly, the prover compiles a cryptographic proof that complies with the statements negotiated before. Thirdly, the verifier checks if the given proof is compiled correctly. The first step is a very elaborate process that is outside of the scope of this paper. To indicate the complexity remember that a proof can range from merely proving possession of a credential issued by some issuer to proving detailed statements about the individual attributes. Our specification focuses on the language that expresses the results from the negotiation phase as well as the second and third step from before. The difficulties here lie in the fact that a proof may be linked to a pseudonym of the user's choice or it may release a verifiable encryption of some attribute value under a third party's public key. In addition, we need to be able to express statements about attributes that will be proved. Finally, the protocols for proving possession of credentials and issuing credentials may be combined. In particular, before issuing a new credential, the issuer may require the recipient to release certified attribute values, that is, prove that she holds a credential issued by another party.

3 Architecture & Specifications

In this section we first discuss the components of *idemix*, then we show how the components are used in the protocols, and finally we provide the specification of the objects used in those protocols. In particular, we introduce the specification languages for the information that needs to be passed between participants.

3.1 Components of *idemix*

An extended anonymous credential system consists of many components. We will introduce them starting with the attributes that are contained in credentials. Continuing with the credentials we will finish the discussion with the optional components such as commitments and pseudonyms, which are used to implement extensions.

3.1.1 Attributes.

We denote an attribute a_i as the tuple consisting of *name*, *value* and *type*, that is, $a_i = \{n_i, v_i, t_i\}$. The name must be unique within its scope (e.g., a credential structure or a commitment), which will allow us to refer to the attribute using that name and the scope. The value refers to the content of the attribute, which is encoded as defined by the type. For each type we define a mapping from the content value to a value that can be used in the cryptographic constructions. Currently, *idemix* supports the attribute types *string*, *int*, *date1900s*, and *enum*. Encoding a string to be used in a group $\mathbb{G}$ with generator g can be achieved by use of a collision-resistant hash function $\mathcal{H}_{\mathsf{hk}}(:)\{0,1\}^* \rightarrow \mathbb{G}$. Integers do not require such mapping unless they are larger than the order of the group used by *idemix*. In such case, the value will be encoded into several attributes. We chose the granularity of the currently implemented date type as a second and set the origin to 1.1.1900. Enumerated attributes are mapped using a distinct prime according to the description in [CG08].

3.1.2 Credentials.

We denote the set of attributes together with the corresponding cryptographic information as credentials. We classify attributes contained in credentials depending on which party knows the value of an attribute. More concretely, the owner of a credential always knows all attribute values but the issuer or the verifier might not be aware of certain values. During the issuance of a credential we distinguish three sets of attributes as the issuer might *know* a value, have a *commitment* of the value, or the value might be completely *hidden* to him. Let us denote the these sets of attributes by A_k, A_c, and A_h, respectively. Note that the user's master secret, as introduced in Section 2, is always contained in A_h.

When creating a proof of possession of credentials, the user has the possibility to reveal only a selected set of attributes. Therefore, we distinguish the *revealed* attributes, which will be learned by the verifier, from the *unrevealed* attributes. We call the two sets of attributes during the proving protocol A_r and $A_{\overline{r}}$. Note, that each attribute can be assigned to either A_r or $A_{\overline{r}}$ independently of all previous protocols and, in particular, independently of the issuing protocol.

3.1.3 Commitments and Representations of Group Elements.

With commitments [DF02] a user can commit to a value v, which we denote as $C \leftarrow comm(v)$. The commitment has a hiding and a binding property, where hiding refers to the recipient not being able to infer information about v given C and binding refers to the committer not being able to convince a recipient that $C = comm(v')$ for a $v' \neq v$. Either of the two properties can be information theoretically achieved where the other will hold computationally.

In our context the bases of a commitment are selected from the bases of the group parameters. When we need the more general version of arbitrarily chosen bases, we call the corresponding object a representation. Where the name is chosen because such objects are representations of group elements w.r.t. other group elements. Representations enable the integration of almost arbitrary proof statements, for example, they are building blocks for building e-cash schemes or (more generally) cloning prevention for credentials.

3.1.4 Pseudonyms and Domain Pseudonyms.

We denote randomized commitments to the master secret as pseudonyms. Thus, a pseudonym is similar to a public key in a traditional PKI and can be used to establish a relation with an organization, for example, in case a user wants an organization to recognize her as a returning user. In contrast to an ordinary public-secret key pair, however, the user can generate an unlimited number of pseudonyms based on the same master secret without the link between those pseudonyms (i.e., the master secret key) becoming apparent.

A domain pseudonym is a special kind of pseudonym in the sense that a user can create exactly one pseudonym w.r.t. one domain. The domain is specified by a verifier, which allows it to enforce usage control for its domain. Note that no two pseudonyms (be it domain or ordinary) are linkable unless a user proves that the underlying master secret key is the same.

3.2 Protocols

The basic building block of *idemix* is the Camenisch-Lysyanskaya (CL) signature scheme [CL01, CL03] which largely determines the protocols. The signature scheme supports blocks of messages, that is, with a single signature many messages can be signed. In a simple credential, thus, each attribute value is handled as a separate message. A more elaborate idea is to use a compact encoding as in [CG08] to combine several attribute values into one message. The signature scheme also supports "blind" signing, where the recipient provides the issuer only with a commitment of the attribute value that will be included in the credential. This is used for attributes of the set A_c. Credentials are always issued to a recipient authenticated with a pseudonym, which ensures that the user's master secret gets "blindly" embedded into the credential.

The distinguishing feature of a CL signature is that it allows a user to prove possession of a signature without revealing the underlying messages or even the signature itself using efficient zero-knowledge proofs of knowledge. Thus, when a prover wants to convince a verifier that she has obtained a credential from an issuer and selectively reveal some of the messages of the credential, she employs a zero-knowledge proof stating that she "knows" a signature by the issuing organization and messages such that signature is valid. As the proof is "zero-knowledge", the user can repeat such a proof as many times as she wants and still it is not possible to link the individual proofs. This statement even holds if the verifier and the issuer pool their information. Of course, a user can also prove possession of several credentials (acquired from different issuers) at once to a verifier and then prove that these credentials share some messages (without revealing the messages).

Let us specify the inputs of the protocols. The issuance protocol requires two inputs for either participant, namely an issuance specification and a set of values. The former is the same for both participants as it defines the issuance process, that is, it links to the definition of the structure of the credential to be issued or the system parameters. The latter are the values assigned to the attributes of the newly created credential. As we pointed out already, the issuer may operate on a set of the values that differs from the one used by the receiver as A_h are not know to it and for values in A_c the issuer only knows a commitment. Note, the issuer may additionally input cryptographic components into the protocol. This is useful when combining the issuance and the proving protocol, for example, the issuer can input a commitment received during a previous run of the proving protocol. It can use the value "sealed" in the commitment as the value of an attribute from the set A_c.

The proving protocol most notably makes use of the proof specification, which the prover and the verifier both must provide as input to the protocol. This specification defines all details of the proof. In addition, it links to the necessary elements for compiling and verifying such proof. The prover provides all credentials referenced in the proof specification as input and the verifier uses the credential structures (cf. Section 3.3) to verify the proof. The cryptographic proof object will be provided to the verifier during the protocol run.

3.2.1 Extensions to the Issuing Protocol.

The issuing protocol has fewer degrees of freedom compared to the proving protocol. This results from the credential structure setting many limitations on the protocol. For instance, the structure defines which attributes belong to which set (i.e., A_k, A_c, or A_h). Still we provide a mechanism for extending the issuing protocol and use it for implementing a feature that enables efficient updates of the attribute values (A_k) contained in a credential.

Credential Updates. As the issuing protocol is interactive (and for security reasons might need to be executed in a particularly protected environment) re-running it would be impractical in many cases. Rather, *idemix* offers an non-interactive method to update credentials where the issuer publishes update information for credentials such that attribute values are updated if necessary.

This feature can, for example, be used to implement credential revocation. The mechanism that we have implemented employs epochs for specifying the life time. A credential thus expires and can be re-validated when updating the expiration date (given that the issuer provides such) [CKS09].

3.2.2 Extensions to the Proving Protocol.

The proving protocol requires the prover and the verifier to agree on the attribute values that will be revealed during the proof, that is, all attributes a_i are contained in either A_r or $A_{\overline{r}}$ such that $A_r \cap A_{\overline{r}} = \emptyset$. In addition, the verifier may define what partial information about the attributes $a_i \in A_{\overline{r}}$ has to be proved, where partial information denotes:

Equality. A user can prove equality of attribute values, where the values may be contained in different credentials. In particular, equality proofs can be created among values that are contained in any cryptographic object such as credentials or commitments. As an example, a user can compute a commitment to a value v, with $C \leftarrow \mathit{comm}(v)$. Assuming a value v' is contained in a credential, the user can prove that $v = v'$.

Inequality. Allows a user to prove that an attribute value is larger or smaller than a specified constant or another attribute value.

Set Membership. Each attribute that is contained as a compact encoding as described in [CG08] enables the user to prove that the attribute value does or does not lie in a given set of values.

Pseudonym. A pseudonym allows a user to establish a linkable connection with a verifier. Furthermore, domain pseudonyms allow a verifier to guarantee that each user only registers one pseudonyms w.r.t. his domain.

Verifiable Encryption. A user can specify an encryption public key under which an attribute value contained in a credential shall be (verifiably) encrypted.

3.3 Specification Languages

As pointed out in Section 1, one challenge when designing the specification languages is to abstract from the underlying cryptography while allowing access to flexible primitives that enable developers to build a broad range of systems. The necessity of both parties having certain information (e.g., the credential structure) in order to extract the semantic of a proof presents another difficulty. For instance, a verifier needs to know the issuer of a credential, the attributes names, their order or their encoding within a credential used in a proof. Thus, it is essential to separate the structural information from the data, where the latter may remain unknown to one communication partner. We will not introduce such separation for objects that do not require it (e.g., public keys). Our specifications are in XML and each component uses and XML schema to define its general structure. Note that the information acquired through unsecured channels needs to be authenticated, which can be attained using a traditional PKI.

System and Group Parameters. The system and group parameters are specified as a list of their elements. In addition, the group parameters contain a link to the system parameters. Both system and group parameters need to be authenticated.

Issuer Key Pair. The issuer key pair consists of a public key and a private key, where mostly the specification of the public key is of interest as the private key as it is never communicated. The public key links to the group parameters with respect to which it has been created. Note that apart from the public key, an issuer needs to publish the structures of the credentials it issues. Even though this information might be included in the public key, we suggest to create a designated file.

Third Party Information. A third party offering some service, such as being trusted party for verifiably encrypting values, must publish a description of the services it offers.

Credentials. As mentioned earlier, we decompose credentials into a *credential structure*, which is the public part, and the *credential data*, which is private to the owner of the credential. In addition a credential data object is partially populated and sent to the verifier during the proving protocol. This decomposition is needed in the issuing process, when the credential data has not been created, as well as in the verification protocol, where the verifier does only get to know a selected subset of the credential data.

```
References{
   Schema = http://www.zurich.ibm.com/security/idemix/pubKey.xsd
   GroupParams = http://www.zurich.ibm.com/security/idemix/gp.xml
}
Elements{
   S = { 9328...4423 }, Z = { 9058...2857 }, n = { 1109...7843 }
   R = { 3287...4359, 8384...1035, 8475...1101, 5837...5752,
         3285...0932, 9438...3218 }
}
```

Figure 1: Example of an issuer public key. The values of the public key are abbreviated for readability reasons.

```
References{
   Schema = http://www.zurich.ibm.com/security/idemix/credStruct.xsd
   IssuerPublicKey = http:www.ch.ch/passport/ipk/chPassport10.xml
}
Attributes{
   Attribute { FirstName, known, type:string }
   Attribute { LastName, known, type:string }
   Attribute { CivilStatus, known, type:enum }
      {
          Marriage, NeverMarried, Widowed, LegallySeparated,
          AnnulledMarriage, Divorced, Common-lawPartner
      }
   Attribute { SocialSecurityNumber, known, type:int }
   Attribute { BirthDate, known, type:date1900s }
   Attribute { Diet, committed, type:string }
   Attribute { Epoch, known, type:int }
}
Features{
   Domain { http://www.ch.ch/passport/v2010 }
   Update  { http://www.ch.ch/passport/v2010/update.xml }
}
Implementation{
   PrimeFactor { CivilStatus:Marriage = 3 }
   PrimeFactor { CivilStatus:NeverMarried = 5 }
   PrimeFactor { CivilStatus:Widowed = 7 }
   PrimeFactor { CivilStatus:LegallySeparated = 11 }
   PrimeFactor { CivilStatus:AnnulledMarriage = 13 }
   PrimeFactor { CivilStatus:Divorced = 17 }
   PrimeFactor { CivilStatus:Common-lawPartner = 19 }
   AttributeOrder { FirstName, LastName, CivilStatus,
                    SocialSecurityNumber, BirthDate, Diet, Epoch }
}
```

Figure 2: Example credential structure where we assume this structure being located at `http://www.ch.ch/passport/v2010/chPassport10.xml` and corresponding to a Swiss passport. For the XML version refer to [?].

In Fig. 2 we describe the credential structure. It contains (1) references to the XML schema and the issuer public key and (2) information about the structure of a credential, which is needed to extract the semantics of a proof. We partition the latter into the attribute, feature, and implementation specific information.

The attribute information defines name, issuance mode (cf. Section 3.1), and type (e.g., string, enumeration) of each attribute. The feature section contains all relevant information about extensions such as domain pseudonyms. Finally, the implementation specific information is mapping general concepts to the actual implementation. As an example, enumerated attributes are implemented using prime encoded attributes [CG08], which requires the assignment of a distinct prime to each possible attribute value.

The credential data most importantly refers to the credential structure that it is based on. In addition, it contains the (randomized) signature and the values of the attributes. Figure 3 shows a credential created according to the structure provided in Fig. 2 and corresponding to the proof specification given in Fig. 9.

```
References{
   Schema = http://www.zurich.ibm.com/security/idemix/cred.xsd
   Structure = http://www.ch.ch/passport/v2010/chPassport10.xml
}
Elements{
   Signature { A:4923...8422, v:3892...3718, e:8439...9239 }
   Features { Update:http://www.ch.ch/passport/v2010/update/7a3i449.xml }
   Values { FirstName:Patrik; LastName:Bichsel; ... }
}
```

Figure 3: This example shows a Swiss passport credential. Note that the owner who will act as prover knows all the attribute values.

Credential Updates. Credential update information is twofold: it consists of (1) general information detailing, e.g., which attributes will be updated, and (2) the information specific to each credential. The former is linked from the credential structure (see Fig. 2), the latter is referenced from the credential (see Fig. 3). Only attributes from the set A_k can be updated.

Commitment and Representation. A commitment and a representation, similar to a credential, consist of a set of values. We assume that the bases for the commitments are listed in the same file as the group parameters. Thus, they use a reference to link to the corresponding parameters. The representations, however, list their bases in addition to the list of exponents.

Pseudonym and Domain Pseudonym. As pseudonyms are a special case of commitments, they also contain a reference to the group parameters they make use of. In addition, at the user's side pseudonyms contain the randomization exponent value (r in Fig. 7). Domain pseudonyms additionally link to their domain.

Verifiable Encryption. A verifiable encryption is transferred to a verifier and (if necessary) to the trusted party for decryption. It does not need to be stored at the prover's side as it usually is not repeatedly needed. It contains the public key used for the encryption as well as the name used in the proof specification, the label and the ciphertext of the encryption.

```
References{
   Schema = http://www.zurich.ibm.com/security/idemix/update.xsd
   Type = http://www.ch.ch/passport/v2010/update.xml
}
Elements{
   Signature { A:5930...8120, v:3221...8221, e:8934...2911 }
       Values { Epoch:3284..2342 }
}
```

Figure 4: Example of the specific credential update information, which in this case consists of the update of the epoch value for the credential given in Fig. 3.

```
References{
   Schema = http://www.zurich.ibm.com/security/idemix/comm.xsd
   GroupParams = http://www.zurich.ibm.com/security/idemix/gp.xml
}
Elements{
   Name = o2
   Value = 2622...8343
}
```

Figure 5: This example shows the commitment that the prover of the example in Fig. 9 issued and that can be used in the issuance as specified in Fig. 8.

```
References{
   Schema = http://www.zurich.ibm.com/security/idemix/nym.xsd
   GroupParams = http://www.zurich.ibm.com/security/idemix/gp.xml
}
Elements{
   Name = nym1
   Values { Commitment:9874...3298; r:2832...2938 }
}
```

Figure 6: Example of a pseudonym as it is stored by a user (i.e., it contains the secret randomization exponent.

```
References{
   Schema = http://www.zurich.ibm.com/security/idemix/verEnc.xsd
   PublicKey = http://www.insurance.com/trustedParty/publicKey.xml
}
Elements{
   Name = verifiableEncryption1
   Label = 3842...2384
   Values { ciphertext:9874...3298 }
}
```

Figure 7: Example of a verifiable encryption as sent to a verifier.

Protocol Messages. When running the protocols, there are several messages that are passed between the communication partners. The specification of those objects contains the reference to the schema and the cryptographic values. Each cryptographic value is assigned a name such that the communication partner can retrieve the values easily.

Issuance Specification. Issuing a credential most importantly requires a credential structure and a set of attribute values. As introduced in Section 3.1.1, the set of values from the issuer may differ from the set of the recipient. More specifically, values of attributes in A_k are known to both recipient and issuer and values of attributes $a_i \in A_h$ are only known to the recipient. For each attribute $a_i \in A_c$ the recipient knows the corresponding value v_i and the issuer only knows a commitment $C \leftarrow comm(v_i)$. We define the issuance modes *known*, *hidden*, and *committed* in the credential structure to denote the set an attribute belongs to. The reason for defining the issuance mode in the credential structure is to unify the issuance modes between different recipients.

As the majority of the information used in the issuance protocol is defined by the credential structure, the issuance specification is only needed to implement advanced features (e.g., binding a proving and an issuing protocol).

```
References{
   CredStruct = http://www.ch.ch/passport/v2010/chPassport10.xml
   Commitments{ Commitment randCommName2 }
}
Elements {
   Attributes{ Diet:randCommName2 }
}
```

Figure 8: Example of an issuance specification defining that a commitment is expected. This commitment will be used to write the value of the attribute called "Diet". The issuer will not get to know the value of this attribute.

Proof Specification. The proof specification is more elaborate than the issuing specification as the *idemix* anonymous credential system supports many features that require specification. Thus, even when using a specific credential we can imagine a broad range of different proofs to be compiled. We start by specifying an identifier for each distinct value that will be included in a proof. Also, we specify the attribute type of each identifier, where the protocol aborts if the type of the identifier and the type of an attribute that it identifies do not match. In addition to identifiers, we allow for constants in the proof specification.

We start the definition of the statements to be proved with a list of credentials that the user proves ownership of (i.e., the user proves knowledge of the underlying master secret key). Next, we assign attribute identifiers or constants to the attributes, where the constants will cause an equality proof w.r.t. the constant. Using the same identifier several times creates an equality proof among those attributes (e.g., `id2` is used within two credentials). More technically, one identifier is tied to one T-value and one S-value of ZKP, thus, if the values assigned to one identifier do not match, the ZKP will fail. Note that we only need to assign an identifier to attributes that are either revealed or partial information is proved. Attributes with no corresponding identifier are added to the set of unrevealed attributes $A_{\overline{r}}$.

Let us elaborate the examples of property proof provided in the specification in Fig. 9 before describing the property proofs more generally. Using two commitments and a representation that have already been communicated to the verifier, the user proves that (1) the last name in all credentials are identical, (2) the employee is in band 5, (3) the prover is widowed or married as certified by the Swiss government, (4) the employee is at least 5 years with IBM (where this proof uses values from the public key of the issuer of the IBM employee credential), (5) the commitment with name `randCommName1` contains the first and last name as certified by the Swiss government. Here, the commitment `randCommName2` could be used for an issuance protocol as explained in Section 3.3.

Note that the proof specification does not contain any implementation specific parts. We define the *idemix* specific details in the credential structure specification (see Fig. 2).

More generally, we allow for proofs of set membership for enumerated attributes. We support the *and*, *or*, and *not* operators on a given set of values and w.r.t. an attribute identifier. Similar to set membership proofs, we allow for inequality proofs, that is, proofs for statements of the form $v_i \circ \hat{v}$, where

```
Declaration{ id1:unrevealed:string; id2:unrevealed:string;
             id3:unrevealed:int; id4:unrevealed:enum;
             id5:revealed:string; id6:unrevealed:enum }
ProvenStatements{
   Credentials{
      randName1:http://www.ch.ch/passport/v2010/chPassport10.xml =
          { FirstName:id1, LastName:id2, CivilStatus:id4 }
      randName2:http://www.ibm.com/employee/employeeCred.xml =
          { LastName:id2, Position:id5, Band:5, YearsOfEmployment:id3 }
      randName3:http://www.ch.ch/health/v2010/healthCred10.xml =
          { FirstName:id1, LastName:id2, Diet:id6 } }
   Enums{
      randName1:CivilStatus = or[Marriage, Widowed]
      randName3:Diet = or[Diabetes, Lactose-Intolerance] }
   Inequalities{    {http://www.ibm.com/employee/ipk.xml, geq[id3,4]} }
   Commitments{     randCommName1 = {id1,id2}; randCommName2 = {id6} }
   Representations{ randRepName = {id5,id2; base1,base2} }
   Pseudonyms{      randNymName; http://www.ibm.com/employee/ }
   VerifiableEncryptions{ {PublicKey1, Label, id2} }
   Message { randMsgName = "Term 1:We will use this data only for ..." }
}
```

Figure 9: Example proof specification using a Swiss passport, an IBM employee credential, and a Swiss health credential.

v_i is an attribute value, $\circ$ is the operator, and $\hat{v}$ can be a constant or another attribute value. Currently, the following operators are implemented: $<$, $>$, $\leq$, and $\geq$. Consequently, we also support proofs that an attribute lies within a specified range.

Relating to the components that we describe in Section 3.1, we need to describe how commitments, representations, pseudonyms and domain pseudonyms are handled. For each exponent of any of those components, the proof specification defines the identifier that it relates, or the constant that it is equivalent to. In addition, all the components of a proof specification are assigned random names, which allow for the identification of the corresponding object in the context of a proof but prevent different proofs from becoming trivially linkable.

The corresponding objects contain the cryptographic values such as the signature on a credential, the commitment value or the bases and the value of a representation.

4 Preliminaries

We give some preliminaries necessary for the presentation of the protocols. In addition, we define commonly-used parameters that are necessary to have a mapping from this specification to the implementation (i.e., Java).

4.1 Notation and System Parameters

Let $H : \{0,1\}^* \rightarrow \{0,1\}^{\ell_H}$ be a cryptographic hash function. The current implementation uses SHA-256 [Nat93]. Let "$\|$" denote the operator for concatenation of numbers or strings. By $\{0,1\}^\ell$ we denote the set of integers $\{0,\ldots,2^\ell-1\}$ and by $\pm\{0,1\}^\ell$ the set of integers $\{-2^\ell+1,\ldots,2^\ell-1\}$. There is no explicit relations between this notation and the number of bits needed to represent these integers. Having said this, the set $\{0,1\}^\ell$ can be mapped to the set of all bit strings with ℓ binary digits in a straightforward manner. Note that, however, $\{-2^\ell+1,\ldots,2^\ell-1\}$ does not so easily map to the set of all $\ell+1$ bit strings. The notation $x \in_R S$ means x is chosen uniformly at random from the set S, and $\#S$ denotes the number of elements in S. The Tables 1, 2 and 3 in Appendix A list the notation used in this document.

4.2 Zero-Knowledge Proofs

When presenting protocols, we use the notation of Camenisch and Stadler [CS97] to specify zero-knowledge (ZK) proofs in an abstract way. This allows the reader to quickly determine what the protocol will accomplish, before looking through the details of how it is accomplished. For instance,

$$PK\{(\alpha, \beta, \delta) : y = g^{\alpha} h^{\beta} \wedge \tilde{y} = \tilde{g}^{\alpha} \tilde{h}^{\delta}\}$$

denotes a "*zero-knowledge Proof of Knowledge of integers* α, β, *and* δ *such that* $y = g^{\alpha} h^{\beta}$ *and* $\tilde{y} = \tilde{g}^{\alpha} \tilde{h}^{\delta}$ *holds,*" where $y, g, h, \tilde{y}, \tilde{g}$, and $\tilde{h}$ are elements of some groups $G = \langle g \rangle = \langle h \rangle$ and $\tilde{G} = \langle \tilde{g} \rangle = \langle \tilde{h} \rangle$ that have the same order. (Note, that some elements in the representation of y and $\tilde{y}$ are equal.) The convention is that values (α, β, δ) denote quantities of which knowledge is being proven (and which are kept secret), while all other values are known to the verifier. For prime-order groups, which include all groups we consider in this paper, it is well-known that there exists a knowledge extractor which can extract these quantities from a successful prover.

All of the zero-knowledge proofs in the *idemix* library are implemented as a common three-move ZK protocol, made non-interactive using the Fiat-Shamir heuristic [FS87] (three-move ZK protocols are similar to Schnorr signatures [Sch91] and are also called sigma protocols). The values in the first flow of the protocol (of the form $t = g^r$) will be referred to as "t-values", while the responses computed in the third flow (of the form $s = r - c\alpha$) will be called "s-values". The challenge, c, is computed has the hash of the t-values, common inputs, and also a common string we call the `context` string, consisting of a list of all public parameters and the issuer public key. This prevents values generated during the proof from being re-used in some other context.

We refer to Appendix C for more details.

4.3 The CL Signature Scheme

We recall this signature scheme (the CL signature scheme) and the related protocols here.

Key generation. On input ℓ_n, choose an ℓ_n-bit RSA modulus n such that $n \leftarrow pq$, $p \leftarrow 2p' + 1$, $q \leftarrow 2q' + 1$, where p, q, p', and q' are primes. Choose, uniformly at random, $R_0, \ldots, R_{L-1}, S, Z \in QR_n$. Output the public key $(n, R_0, \ldots, R_{L-1}, S, Z)$ and the secret key p.

Message space. Let ℓ_m be a parameter. The message space is the set

$$\{(m_0, \ldots, m_{L-1}) : m_i \in \pm\{0,1\}^{\ell_m}\} \ .$$

Signing algorithm. On input $m_0, \ldots, m_{L-1}$, choose a random prime number e of length $\ell_e > \ell_m + 2$, and a random number v of length $\ell_v \leftarrow \ell_n + \ell_m + \ell_r$, where ℓ_r is a security parameter. Compute the value A such that

$$A \leftarrow \left(\frac{Z}{R_0^{m_0} \ldots R_{L-1}^{m_{L-1}} S^v} \right)^{1/e} \bmod n \ .$$

The signature on the message $(m_0, \ldots, m_{L-1})$ consists of (A, e, v).

Verification algorithm. To verify that the tuple (A, e, v) is a signature on message $(m_0, \ldots, m_{L-1})$, check that

$$Z \equiv A^e R_0^{m_0} \ldots R_{L-1}^{m_{L-1}} S^v \pmod{n} \ , \quad m_i \in \pm\{0,1\}^{\ell_m}, \text{ and } 2^{\ell_e} > e > 2^{\ell_e - 1}$$

all holds.

Theorem 4.1 ([CL03]). *The signature scheme is secure against adaptive chosen message attacks [GMR88] under the strong RSA assumption.*

The original scheme considered messages in the interval $[0, 2^{\ell_m} - 1]$. Here, however, we allow messages to be from $[-2^{\ell_m} + 1, 2^{\ell_m} - 1]$. The only consequence of this is that we need to require that $\ell_e > \ell_m + 2$ holds instead of $\ell_e > \ell_m + 1$.

4.4 The CS Encryption Scheme

This text is taken from Camenisch-Shoup [CS03] and is a variation of an encryption scheme put forth in [CS02].

4.4.1 Background

Let p, q, p', q' be distinct odd primes with $p \leftarrow 2p' + 1$ and $q \leftarrow 2q' + 1$, and where p' and q' are both ℓ bits in length. Let $n \leftarrow pq$ and $n' \leftarrow p'q'$. Consider the group $\mathbb{Z}^*_{n^2}$ and the subgroup $\mathbf{P}$ of $\mathbb{Z}^*_{n^2}$ consisting of all n-th powers of elements in $\mathbb{Z}^*_{n^2}$.

Paillier's Decision Composite Residuosity (DCR) assumption [Pai99] is that given only n, it is hard to distinguish random elements of $\mathbb{Z}^*_{n^2}$ from random elements of $\mathbf{P}$.

To be completely formal, one should specify a sequence of bit lengths $\ell(\lambda)$, parameterized by a security parameter $\lambda \geq 0$, and to generate an instance of the problem for security parameter λ, the primes p' and q' should be distinct, random primes of length $\ell \leftarrow \ell(\lambda)$, such that $p \leftarrow 2p' + 1$ and $q \leftarrow 2q' + 1$ are also primes.

The primes p' and q' are called Sophie Germain primes and the primes p and q are called safe primes. It has never been proven that there are infinitely many Sophie Germain primes. Nevertheless, it is widely conjectured, and amply supported by empirical evidence, that the probability that a random ℓ-bit number is Sophie Germain prime is $\Omega(1/\ell^2)$. We shall assume that this conjecture holds, so that we can assume that problem instances can be efficiently generated.

Note that Paillier did not make the restriction to safe primes in originally formulating the DCR assumption. As will become evident, we need to restrict ourselves to safe primes for technical reasons. However, it is easy to see that the DCR assumption without this restriction implies the DCR assumption with this restriction, assuming that safe primes are sufficiently dense, as we are here.

We can decompose $\mathbb{Z}^*_{n^2}$ as an internal direct product

$$\mathbb{Z}^*_{n^2} \equiv \mathbf{G}_n \cdot \mathbf{G}_{n'} \cdot \mathbf{G}_2 \cdot \mathbf{T},$$

where each group $\mathbf{G}_\tau$ is a cyclic group of order τ, and $\mathbf{T}$ is the subgroup of $\mathbb{Z}^*_{n^2}$ generated by $(-1 \bmod n^2)$. This decomposition is unique, except for the choice of $\mathbf{G}_2$ (there are two possible choices). For any $x \in \mathbb{Z}^*_{n^2}$, we can express x uniquely as $x \equiv x(\mathbf{G}_n)x(\mathbf{G}_{n'})x(\mathbf{G}_2)x(\mathbf{T})$, where for each $\mathbf{G}_\tau$, $x(\mathbf{G}_\tau) \in \mathbf{G}_\tau$, and $x(\mathbf{T}) \in \mathbf{T}$.

Note that the element $h \leftarrow (1 + n \bmod n^2) \in \mathbb{Z}^*_{n^2}$ has order n, i.e., it generates $\mathbf{G}_n$, and that $h^a \leftarrow (1 + an \bmod n^2)$ for $0 \leq a < n$. Observe that $\mathbf{P} \leftarrow \mathbf{G}_{n'}\mathbf{G}_2\mathbf{T}$.

4.4.2 The Scheme

For a security parameter $\lambda \geq 0$, $\ell \leftarrow \ell(\lambda)$ is an auxiliary parameter.

The scheme makes use of a keyed hash scheme $\mathcal{H}$ that uses a key hk, chosen at random from an appropriate key space associated with the security parameter λ; the resulting hash function $\mathcal{H}_{\mathsf{hk}}(\cdot)$ maps a triple (u, e, L) to a number in the set $[2^\ell]$. We shall assume that $\mathcal{H}$ is collision resistant, i.e., given a randomly chosen hash key hk, it is computationally infeasible to find two triples $(u, e, L) \neq (u', e', L')$ such that $\mathcal{H}_{\mathsf{hk}}(u, e, L) \equiv \mathcal{H}_{\mathsf{hk}}(u', e', L')$.

Let $\text{abs} : \mathbb{Z}^*_{n^2} \to \mathbb{Z}^*_{n^2}$ map $(a \bmod n^2)$, where $0 < a < n^2$, to $(n^2 - a \bmod n^2)$ if $a > n^2/2$, and to $(a \bmod n^2)$, otherwise. Note that $v^2 \equiv (\text{abs}(v))^2$ holds for all $v \in \mathbb{Z}^*_{n^2}$.

We now describe the key generation, encryption, and decryption algorithms of the encryption scheme, as they behave for a given value of the security parameter λ.

Key Generation. Select two random ℓ-bit Sophie Germain primes p' and q', with $\mathsf{p}' \neq \mathsf{q}'$, and compute $\mathsf{p} := (2\mathsf{p}' + 1)$, $\mathsf{q} := (2\mathsf{q}' + 1)$, $\mathsf{n} := \mathsf{pq}$, and $\mathsf{n}' := \mathsf{p}'\mathsf{q}'$, where $\ell \leftarrow \ell(\lambda)$ is an auxiliary security parameter. Choose random x_1, x_2, $x_3 \in_R [\mathsf{n}^2/4]$, choose a random $\mathsf{g}' \in_R \mathbb{Z}^*_{\mathsf{n}^2}$, and compute $\mathsf{g} := (\mathsf{g}')^{2\mathsf{n}}$, $\mathsf{y}_1 := \mathsf{g}^{x_1}$, $\mathsf{y}_2 := \mathsf{g}^{x_2}$, and $\mathsf{y}_3 := \mathsf{g}^{x_3}$. Also, generate a hash key hk from the key space of the hash scheme $\mathcal{H}$ associated with the security parameter λ. The public key is $(\mathsf{hk}, \mathsf{n}, \mathsf{g}, \mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3)$. The secret key is $(\mathsf{hk}, \mathsf{n}, x_1, x_2, x_3)$.

In what follows, let $\mathsf{h} \leftarrow (1 + \mathsf{n} \bmod \mathsf{n}^2) \in \mathbb{Z}^*_{\mathsf{n}^2}$, which as discussed above, is an element of order n.

Encryption. To encrypt a message $m \in [\mathsf{n}]$ with label $L \in \{0,1\}^*$ under a public key as above, choose a random $\mathsf{r} \in_R [\mathsf{n}/4]$ and compute

$$\mathsf{u} := \mathsf{g}^{\mathsf{r}} \ , \qquad \mathsf{e} := \mathsf{y}_1^{\mathsf{r}}\mathsf{h}^m \ , \quad \text{and} \qquad \mathsf{v} := \text{abs}\left((\mathsf{y}_2\mathsf{y}_3^{\mathcal{H}_{\mathsf{hk}}(\mathsf{u},\mathsf{e},L)})^{\mathsf{r}}\right) \ .$$

The ciphertext is $(\mathsf{u}, \mathsf{e}, \mathsf{v})$.

Decryption. To decrypt a ciphertext $(\mathsf{u}, \mathsf{e}, \mathsf{v}) \in \mathbb{Z}^*_{\mathsf{n}^2} \times \mathbb{Z}^*_{\mathsf{n}^2} \times \mathbb{Z}^*_{\mathsf{n}^2}$ with label L under a secret key as above, first check that $\text{abs}(\mathsf{v}) \equiv \mathsf{v}$ and $\mathsf{u}^{2(x_2+\mathcal{H}_{\mathsf{hk}}(\mathsf{u},e,L)x_3)} \equiv \mathsf{v}^2$. If this does not hold, then output reject and halt. Next, let $\mathsf{t} \leftarrow 2^{-1} \bmod \mathsf{n}$, and compute $\hat{m} := (\mathsf{e}/\mathsf{u}^{x_1})^{2\mathsf{t}}$. If $\hat{m}$ is of the form h^m for some $m \in [\mathsf{n}]$, then output m; otherwise, output reject.

4.5 Integer Commitments

We require integer commitments to implement protocol extensions such as Inequality proofs or to allow an issuer to make the issuance dependent on a credential proof. Finally, integer commitments can be used to link to application level protocols. As a commitment scheme we use the so-called Damgård-Fujisaki-Okamoto scheme [DF02], which is essentially the Pedersen commitment scheme [Ped92] in a group of unknown order.

Assuming Z, S, n from the public key of an issuer generated as described above, committing to an *arbitrarily large* integer m is done by

1. choosing a random $r \in_R [0, \lfloor n/4 \rfloor]$ and
2. computing the commitment as $C := Z^m S^r \bmod n$.

We note that it is important that the committing entity is not privy of the factorization of n. Thus, it is preferable to use the Z, S, n from the public key of an issuer.

5 System Setup

The anonymous credential system requires general parameters, which we separate into system parameters (consisting of bit lengths as well as prime probabilities) and group parameters, which define the groups that are used within the system. In addition, the issuers, trusted third parties and users must generate parameters (e.g., public keys, master secret) to be able to participate in the credential system.

5.1 System and Group Parameters

In order to generate issuer public and private keys, and thus to enable all subsequent operations, a set of public group parameters must be either generated or re-used. Note that those parameters define the system environment, thus, generating new group parameters creates a new environment which is incompatible (in the sense that credentials of the two environments cannot be shown in the same proving protocol) with any other system. The *idemix* library is able to any environment but it will fail if it is provided inputs that are incompatible.

The system parameters as given in Table 2 must be fixed and made public. In addition, we must generate and publish a group to be used for commitments. We denote this group $\mathbb{Z}^*_\Gamma$ with order $\Gamma - 1 = \rho \cdot b$ for some large prime ρ . This ensures that $\mathbb{Z}^*_\Gamma$ has a large subgroup of prime order ρ, and that discrete logarithms are hard to compute. The bit lengths of Γ and ρ are given by ℓ_Γ and ℓ_ρ respectively. Preferably, the co-factor b is small, which will render constraint 4 trivial (see Table 3).

A generator g of the group is computed by choosing a random $g' \in_R \mathbb{Z}^*_\Gamma$ with $g'^b \not\equiv 1 \pmod{\Gamma}$ and computing $g = g'^b \pmod{\Gamma}$. The necessary second generator h can be computed by choosing $r \in_R [0..\rho]$ and computing $h = g^r$. The group parameters Γ, ρ, g and h are provided to all parties as public parameters. A user verifies the system parameters by checking that ρ and Γ are prime, and that $\rho \mid (\Gamma - 1)$, $\rho \nmid \frac{\Gamma-1}{\rho}$, and $g^\rho \equiv h^\rho \equiv 1 \pmod{\Gamma}$.

5.2 Issuer Key Generation (CL Signature Scheme)

The issuer's key pair is foremost used for issuing certificates, that is, issuing signatures on lists of attributes. In addition, the bases of an issuer public key are also used for proving inequalities. Let us present the key generation step of the CL signature scheme.

The maximum number l of attributes of the credential is determined by the number of bases contained in the public key. Note, that ℓ_{res} bases are reserved (e.g., for the master secret) system wide and cannot be used for attributes defined by the issuer. After defining those lengths the issuer starts the process of generating public key by creating a safe RSA key pair. To this effect he first generates the safe primes p and q , $p = 2p' + 1$ and $q = 2q' + 1$, then computes $n = pq$. For security, n should be ℓ_n bits, p and q must have bit length $\ell_n/2$. In addition, the issuer generates parameters for the CL signature scheme by choosing

$$S \in_R QR_n, \quad \text{and} \quad Z, R_1, \ldots, R_l \in_R \langle S \rangle$$

(where QR_n is the group of quadratic residues $\pmod n$ and $\langle S \rangle$ is the subgroup generated by S). S must have order $\#QR_n = p'q'$. Furthermore, the issuer chooses $x_Z, x_{R_1}, \ldots, x_{R_l} \in_R [2, p'q' - 1]$ and computes $Z = S^{x_Z}$, $R_i = S^{x_{R_i}}$ for $1 \le i \le l$. The issuer's public key is $pk_I = (n, S, Z, R_1, \ldots, R_l, P)$ and the private key is $sk_I = (p, q)$.

The following non-interactive zero-knowledge proof of knowledge assures every user of the key about its correct generation, that is, that $Z, R_i \in \langle S \rangle$ for $1 \le i \le l$.

$$SPK\{(\alpha_Z, \alpha_1, \ldots, \alpha_l) : Z = S^{\alpha_Z}, R_1 = S^{\alpha_1}, \ldots, R_l = S^{\alpha_l}\}$$

where all equalities are mod n. In addition to verifying the proof, the lengths of the the public key parameters must be verified.

5.3 Trustee Key Generation (CS Encryption Scheme)

Select two random ℓ-bit Sophie Germain primes p' and q', with $\mathsf{p}' \neq \mathsf{q}'$, and compute $\mathsf{p} := (2\mathsf{p}' + 1)$, $\mathsf{q} := (2\mathsf{q}' + 1)$, $\mathsf{n} := \mathsf{pq}$, and $\mathsf{n}]' := \mathsf{p}'\mathsf{q}'$, where $\ell = \ell(\lambda)$ is an auxiliary security parameter. Choose random x_1, x_2, $x_3 \in_R [\mathsf{n}^2/4]$, choose a random $\mathsf{g}' \in_R \mathbb{Z}^*_{\mathsf{n}^2}$, and compute $\mathsf{g} := (\mathsf{g}')^{2\mathsf{n}}$, $\mathsf{y}_1 := \mathsf{g}^{x_1}$, $\mathsf{y}_2 := \mathsf{g}^{x_2}$, and $\mathsf{y}_3 := \mathsf{g}^{x_3}$. Also, generate a hash key hk from the key space of the hash scheme $\mathcal{H}$ associated with the security parameter λ. The public key is $(\mathsf{hk}, \mathsf{n}, \mathsf{g}, \mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3)$. The secret key is $(\mathsf{hk}, \mathsf{n}, x_1, x_2, x_3)$.

In what follows, let $\mathsf{h} = (1 + \mathsf{n} \bmod \mathsf{n}^2) \in \mathbb{Z}^*_{\mathsf{n}^2}$, which as discussed above, is an element of order n.

5.4 User Master Secret Generation

The user master secret is used to several credentials together. This allows *idemix* to create proofs involving several credentials and to achieve a meaningful semantic (i.e., the proof is not issued using credentials from different people). Thus, the issuing and proving protocol take the master secret as input argument. As the name implies, this data item must be kept secret and is bound to a specic user.

Let us denote the user's master secret as m_1 . It is an integer chosen uniformly at random from the interval $[1, \rho]$. Depending on the issuance specification, m_1 may be new or re-used from a previous credential.

5.5 Pseudonyms and Domain Pseudonyms

Having seen the intuition behind pseudonyms (cf. § 3.1.4) we will discuss the technical details here. Domain pseudonyms enforce that a user can only generate one pseudonym per domain by requiring that all pseudonyms are computed in the subgroup $\langle g \rangle$ of $\mathbb{Z}^*_\Gamma$ which has order ρ.

A pseudonym is computed as $\mathsf{Nym} := \mathit{commit}(m_1)$. Let $\mathtt{dom}$ be an arbitrary string describing a domain. A domain pseudonym for $\mathtt{dom}$ is computed as $\mathsf{DNym} := g_{\mathtt{dom}}^{m_1}$, where $g_{\mathtt{dom}} := \mathcal{H}(\mathtt{dom})^{(\Gamma-1)/\rho} \bmod \Gamma$, where $\mathcal{H}$ is a hash function mapping $\{0,1\}^* \to \mathbb{Z}_\Gamma$ (see [IBM09] for description of $\mathcal{H}$).

6 Protocol Specification of *idemix*

Let us recall the inputs of the *idemix* protocols as discussed in Section 3.2. The protocols are run interactively between the participants, which we describe in a sequential fashion in the issuance protocol. The proving protocol is less interactive, however, it is essential to note that the prover first computes cryptographic values for all sub-proofs. Then she is able to compute the challenge used to make the zero-knowledge proof non-interactive. Using this challenge, the prover (again) calls all sub-provers. This methodology can be seen as there is an optional value $[c]$ specified as input of all sub-provers.

6.1 Issuing Protocol

Let us begin by giving an overview of the protocol used to issue a credential. The roles in this protocol are the RECIPIENT of the credential, which is executed by a user, and the ISSUER of the credential, which is typically run by an organization (e.g., a company, a government). During the issuance protocol, the ISSUER and RECIPIENT interactively create a CL-signature for the RECIPIENT, which is the cryptographic part of a credential. At each step of this three flow protocol, a zero knowledge proof ensures that both parties are correctly implementing the protocol.

6.1.1 Protocol: IssueCertificateProtocol

Input	**Common:**	$\mathcal{S}, \{nym\}, \{dNym\}, \{C_k\}_{\forall k \in A_c}, \{m_i\}_{\forall i \in A_k}$
	RECIPIENT:	$m_1, \{(m_k, r_k)\}_{k \in A_c}, \{m_j\}_{j \in A_h}, \{r_i\}_{i \in \{nym\}}$
	ISSUER:	sk_I
Output	**Common:**	The CL-signature (A, e, v) or $\perp$ if the protocol fails.

$\mathcal{S}$ is the issuing specification, which is an input to the ISSUER and the RECIPIENT, and it most importantly determines the credential structure. Thereby, the issuer public key and the attribute structures are implicitly defined. Note that the credential structure specifies the attributes, that is, their data type, issuance mode (see 3.1.1) and the index with respect to the bases in the issuer public key. In addition, $\mathcal{S}$ allows both parties to compute a common string `context` which is a list of all public parameters. This string will be included in the hash, which binds the zero-knowledge proof to the current context. This prevents values generated during the proof from being re-used in some other context.

The committed values ($\{m_k\}_{k \in A_c}$) are given to the ISSUER through the commitment $C_k \leftarrow Z_{(k)}^{m_k} S_{(k)}^{r_k}$ (mod $n_{(k)}$). Subsequently, the issuer has as private input the integer values r_k. We remark that the commitment bases need to be chosen such that the issuer is ensured that the recipient (the committer) does not know the factorization of the modulus $n_{(k)}$ (see § 4.5).

Similarly, the set of values r_i are the integers s.t. $nym \leftarrow g^{m_1} h^{r_i}$, $\forall i \in \{nym\}$. If the protocol is run without any pseudonym, we set $nym := \perp$, similarly if not domain pseudonym is present $dNym := \perp$.

A note on enumerated attributes. Enumerated attributes are specified as a set of names. Using the attribute structure, the set can be translated into the value of the enumerated attribute. An enumerated attribute can contain the values of several attributes.

All possible values for enumerated attributes are specified in the credential structure. To indicate which of the enumerated attributes are contained in the credential, there is a list in the attribute. This list is built of *attributeName:enumValue* entries. Using this list and the primes assigned in the credential structure, the attribute value can be calculated as the product of the corresponding primes.

Example 6.1. Let us assume that there is an attribute named "sex", which can assume the values "male" or "female", and an attribute named "language", which can have the values "german", "french" or "english". Those attributes are encoded in the attribute "enumEncoding" and the following primes are associated to the attributes:

$$\begin{aligned} \textit{sex:male} &= 3 \\ \textit{sex:female} &= 5 \\ \textit{language:german} &= 7 \\ \textit{language:french} &= 11 \\ \textit{language:english} &= 13 \end{aligned}$$

A male user speaking German and French would consequently get assigned the values *sex:male*, *language:german*, and *language:french* which results in an effective attribute value m_j of 231.

Round 0

0.1 Issuer chooses a random nonce $n_1 \in_R \{0,1\}^{\ell_\varnothing}$.

0.2 Issuer $\rightarrow$ Recipient: n_1.

0.3 Issuer and Recipient load attribute structures A from $\mathcal{S}$.

Round 1

1.1 Recipient chooses a random integer $v' \in_R \pm\{0,1\}^{\ell_n+\ell_\varnothing}$.

1.2 Recipient computes

$$U := S^{v'} \cdot \prod_{j \in A} R_j^{m_j} \bmod n$$

Next, Recipient computes a non-interactive proof that this was done correctly. (We slightly abuse the CS notation by *not* replacing all the values the prover is proving knowledge of with Greek letters.)

$$\begin{aligned} SPK\{\{m_i\}_{i\in A_h}, v', \{(m_k, r_k)\}_{k\in A_c} :& \\ & U \equiv \pm S^{v'} \prod_{k\in A_h} R_k^{m_k} \pmod n \\ & \wedge_{nym\neq\perp}\ nym \equiv g^{m_1} h^r \pmod \Gamma \\ & \wedge_{dNym\neq\perp}\ dNym \equiv g_{\mathsf{dom}}^{m_1} \pmod \Gamma \\ & \wedge_{k\in A_c} (C_k \equiv \pm Z_{(k)}^{m_k} S_{(k)}^{r_k} \pmod{n_{(k)}}) \\ & \wedge\ m_i \in \pm\{0,1\}^{\ell_m+\ell_\varnothing+\ell_H+1} \quad \forall\, i \in A_h\} \end{aligned} \tag{1}$$

Note. In some cases the issuer may require that the master secret m_1 be the same as the master secret of a credential previously used during a proof protocol. This is enforced by using the same pseudonym *nym* or *dNym* in both protocols. Moreover, the master secret can only ever be used for pseudonym, i.e., the library will not allow its use for computation anywhere else than for *nym*, *dNym* and U.

1.3 Recipient computes SPK (1).

1.3.0.0 Choose $\tilde{m}_j \in_R \pm\{0,1\}^{\ell_m+\ell_\varnothing+\ell_H+1}$ for $j \in \{1 \cup A_h \cup A_c\}$.

1.3.0.1 (*knowledge of pseudonym and master secret key*) If $nym \neq \perp$ compute

$$\widetilde{nym} := g^{\tilde{m}_1} h^{\tilde{r}_i} \bmod \Gamma$$

where $\tilde{r}_i \in_R [1, \rho]$, otherwise set $\widetilde{nym} := \perp$.

1.3.0.2 (*knowledge of domain pseudonym and master secret key*) If $dNym \neq \perp$ compute

$$\widetilde{dNym} := g_{\mathsf{dom}}^{\tilde{m}_1} \bmod \Gamma$$

otherwise set $\widetilde{dNym} := \perp$.

1.3.1 (*knowledge of U's representation*) Compute

$$\tilde{U} := S^{\tilde{v}'} \cdot \prod_{j\in A} R_j^{\tilde{m}_j} \bmod n$$

where

$$\tilde{v}' \in_R \pm\{0,1\}^{\ell_n+2\ell_\varnothing+\ell_H}\quad .$$

Store all random values.

1.3.2 (*knowledge of committed values*) Compute the map from attribute names to $\tilde{C}_j$ where

$$\tilde{C}_j := (Z_{(j)}^{\tilde{m}_j} S_{(j)}^{\tilde{r}_j} \bmod n_{(j)})_{j \in A_c}$$

with all $\tilde{r}_j \in_R \pm \{0,1\}^{\ell_n + 2\ell_\varnothing + \ell_H}$.

1.3.3 (*challenge via Fiat-Shamir*) Compute the challenge as:

$$\begin{aligned} c := & H(\texttt{context} \| U \| C_1 \| \dots \| C_k \| nym \| dNym \| \\ & \tilde{U} \| \tilde{C}_1 \| \dots \| \tilde{C}_k \| n\tilde{y}m \| d\tilde{N}ym \| n_1) \ . \end{aligned}$$

1.3.4 (*responses to challenge*) The responses are (some denoted as ordered lists):

$$\begin{aligned} \hat{v}' &:= \tilde{v}' + cv' \\ s_A &:= (\hat{m}_j := \tilde{m}_j + cm_j)_{j \in A} \end{aligned}$$

If $nym \neq \perp$ compute $\hat{r}_j := \tilde{r}_j + cr_j \bmod \rho$ for all $j \in \{nym\}$ otherwise set $\hat{r} := \perp$.

1.3.5 (*output SPK*$\{\dots\}(n_1)$) The complete proof signature is

$$P_1 := (c, \hat{v}', s_A, \hat{r}) \ .$$

1.4 Recipient $\to$ Issuer: U, P_1, $n_2 \in_R \{0,1\}^{\ell_\varnothing}$.

1.5 Recipient stores the following elements to file (for use in credential epoch update)

- A_k (values and structure)
- v'
- `context`

Round 2 (*Signature Generation*)

2.0 Issuer verifies P_1.

2.0.0.1 (*knowledge of pseudonym and master secret key*) If $nym \neq \perp$ compute

$$n\hat{y}m := nym^{-c} g^{\hat{m}_1} h^{\hat{r}} \bmod \Gamma$$

otherwise set $n\hat{y}m := \perp$.

2.0.0.2 (*knowledge of pseudonym and master secret key*) If $dNym \neq \perp$ compute

$$d\hat{N}ym := dNym^{-c} g_{\mathsf{dom}}^{\hat{m}_1} \bmod \Gamma$$

otherwise set $d\hat{N}ym := \perp$.

2.0.1 (*representation of U*) Compute

$$\hat{U} := U^{-c}(S^{\hat{v}'}) \prod_{j \in A} R_j^{\hat{m}_j} \bmod n \ .$$

2.0.2 (*knowledge of committed values*) Compute the ordered list

$$\hat{C}_j := \left(c_j^{-c} Z_{(j)}^{\hat{m}_j} S_{(j)}^{\hat{r}_j} \bmod n_{(j)} \right)_{j \in A_c} \ .$$

2.0.3 (*verify challenge*) Compute

$$\begin{aligned} \hat{c} := & H(\texttt{context} \| U \| C_1 \| \dots \| C_k \| nym \| dNym \| \\ & \hat{U} \| \hat{C}_1 \| \dots \| \hat{C}_k \| n\hat{y}m \| d\hat{N}ym \| n_1) \end{aligned}$$

If $\hat{c} \neq c$, verification fails, abort $\mathsf{IssueCertificateProtocol}$ and return $\perp$.

2.0.4 (*length checks*) Check that

$$\hat{v}' \in \pm\{0,1\}^{\ell_n + 2\ell_\varnothing + \ell_H + 1} \ ,$$
$$\hat{m}_i \in \pm\{0,1\}^{\ell_m + \ell_\varnothing + \ell_H + 2} \ , \text{ for all } i \in \{1 \cup A_h \cup A_c\} \ .$$

If any length check fails; abort IssueCertificateProtocol and return $\perp$.

2.1 Issuer generates a CL signature on the attributes.

2.1.1 Choose a random prime

$$e \in_R [2^{\ell_e - 1}, 2^{\ell_e - 1} + 2^{\ell'_e - 1}] \ .$$

2.1.2 Choose a random integer $\tilde{v} \in_R \{0,1\}^{\ell_v - 1}$, and compute $v'' := 2^{\ell_v - 1} + \tilde{v}$.

2.1.3 Compute

$$Q := \frac{Z}{US^{v''} \prod_{i \in A_k} R_i^{m_i}} \bmod n \ \text{ and } \ A := Q^{e^{-1} \bmod p'q'} \bmod n \ .$$

(A, e, v'') will be sent to the Recipient. Recall A_k that contains the known attributes.

2.1.4 Issuer stores the following elements RecipientRecord to file (for use in credential updates):

- Q
- A_k (values and structure)
- v''
- `context`

2.2 Issuer creates the following proof of correctness.

$$SPK\{(e^{-1}) : A \equiv \pm Q^{e^{-1}} \pmod{n}\}(n_2) \ .$$

2.2.1 Compute $\tilde{A} := Q^r \pmod{n}$, for $r \in_R \mathbb{Z}^*_{p'q'}$.

2.2.2 Compute $c' := H(\texttt{context} \| Q \| A \| \tilde{A} \| n_2)$.

2.2.3 Compute $s_e := r - c'e^{-1} \pmod{p'q'}$. The proof is $P_2 \leftarrow (s_e, c')$.

2.3 Issuer sends (A, e, v''), P_2, $(m_i)_{i \in A_k}$ to the Recipient.

Round 3

3.0 Compute $v := v'' + v'$.

3.1 Recipient verifies (A, e, v), using the CL-sig verification algorithm:

3.1.0 Check that e is prime, and $e \in [2^{\ell_e - 1}, 2^{\ell_e - 1} + 2^{\ell'_e - 1}]$.

3.1.1 Compute $Q := \frac{Z}{S^v \prod_{i \in \mathcal{S}} R_i^{m_i}} \bmod n$.

3.1.2 Compute $\hat{Q} := A^e \bmod n$.

3.1.3 If $\hat{Q} \not\equiv Q \pmod{n}$, abort IssueCertificateProtocol.

3.2 Recipient verifies P_2.

3.2.1 Compute $\hat{A} := A^{c' + s_e \cdot e} S^{v' s_e} \bmod n$.

3.2.2 Compute $\hat{c} := H(\texttt{context} \| Q \| A \| \hat{A} \| n_2)$.

3.2.3 If $\hat{c} \neq c'$, abort IssueCertificateProtocol.

3.3 (*output*) If steps 3.1 and 3.2 succeed, store the credential $(m_i)_{iinA}, (A, e, v))$.

6.1.2 Protocol: UpdateCredential

To update a credential, the issuer runs some parts of the issuance protocol again. However, he includes the new message values for the updated attributes. The issuer has stored A_k, $(m_i)_{A_k}$, Q and v'' on file for the credential. Now let $(\bar{m}_i)_{A_k}$ be the values of the attributes as they should appear in updated credential and let $\Delta m_i = \bar{m}_i - m_i$. We assume that one of these m_i encodes the epoch.

Round 1 (*Signature Generation*)

1.1 Issuer reads previously saved elements from update file.

1.2 Issuer generates a CL signature on the attributes.

1.2.1 Choose a random prime

$$e \in_R [2^{\ell_e - 1}, 2^{\ell_e - 1} + 2^{\ell'_e - 1}] .$$

1.2.2 Choose a random integer $\tilde{v} \in_R \{0,1\}^{\ell_v - 1}$, and compute $\bar{v}'' := 2^{\ell_v - 1} + \tilde{v}$ and $\Delta v'' := \bar{v}'' - v''$

1.2.3 Compute

$$\bar{Q} := \frac{Q}{(\prod_{i \in A_{\mathrm{KN}}} R_i^{\Delta m_i}) S^{\Delta v''}} \bmod n \quad \text{and} \quad \bar{A} := \bar{Q}^{e^{-1} \bmod p'q'} \bmod n .$$

1.2.4 Set $Q := \bar{Q}$, $v'' := \bar{v}''$, $m_i := \bar{m}_i$, and $A := \bar{A}$.

1.3 Issuer creates the following proof of correctness.

$$P_2 := SPK \left\{ (e^{-1}) : A \equiv \pm Q^{e^{-1}} \pmod n \right\} (n_2) .$$

1.4 Issuer sends (A, e, v''), P_2, and $(m_i)_{i \in A_k}$ to the Recipient.

1.5 Issuer updates the following elements in file (for use in credential update)

- Q
- v''
- $\{m_i : i \in A_k\}$

Round 2

2.1 Recipient reads previously saved elements from update file.

2.2 Recipient verifies P_2.

2.2.1 Compute Q and $\hat{Q} = A^{c'} Q^{s_e}$.

2.2.2 Compute $\hat{c} = H(\texttt{context} \| Q \| A \| \hat{Q} \| n_2)$.

2.2.3 If $\hat{c} \neq c'$, abort IssueCredentialProtocol.

2.3 Recipient computes and stores $v = v' + v''$.

2.4 Recipient verifies (A, e, v), using the CL-signature verification algorithm (see issuing Step 2.2.

2.4.2 Check that e is prime and that $e \in [2^{\ell_e - 1}, 2^{\ell_e - 1} + 2^{\ell'_e - 1}]$.

2.4.1 Check $Z \equiv A^e R_1^{m_1} \ldots R_l^{m_l} S^v \pmod n$.

2.5 (*output*) If steps 2.2 and 2.4 succeed, the credential $(m_1, \ldots, m_l, (A, e, v))$ is output and the recipient update its record for this credential.

6.2 Proving Protocol

Let us describe how a user acting as the Prover can use a proof specification to create a proof of possession of a credential. Such a proof can be verified by the Verifier, which could be a relying party basing its access control decision on such a proof. The most basic proof is called a proof of possession, which means that a user proves knowledge of a Camenisch-Lysyanskaya signature [CL03] on a set of attributes without disclosing the latter. This basic proof can be extended by protocols for verifiable encryption of Camenisch and Shoup [CS03] and for enumerated attributes described by Camenisch and Gross [CG08].

We refer to the example of a proof specification given in Fig. 9 where where we also explained the components of the proof specification. For the detailed notation of the object see Appendix D.2.

A note on the trust model is applicable here. A proof of knowledge of a value in the group $\mathbb{Z}_n^*$ where n is from the issuer's public key, requires that the VERIFIER trusts the issuer not to share the secret key with the PROVER (see § 4.5). An alternative would be to have the PROVER use the VERIFIER's public key, however this requires additional infrastructure.

As there is little interaction between PROVER and VERIFIER, the presentation of the protocols for both roles are easily separated. The protocols for the prover and the verifier are denoted ProveProtocol and VerifyProtocol, respectively. We start with the description of the ProveProtocol.

6.2.1 The Prove Protocol

The ProveProtocol realizes a non-interactive proof of the statements defined in the proof specification.

Note that many assertions require multiple credentials, for example, a merchant might want to ensure that the `name` field of the credit card credential matches the `name` field on the government ID credential. We allow for such proofs, however, the library requires the master secret of all credentials used in one proof to be the same (this is proven transparently to the user of the library).

Abstractly, the proof is the composition of the following proofs. Let $V = \{v_1, \ldots, v_t\}$ be the set of values in the credentials held by the prover.

$$
\begin{aligned}
SPK\{(V) : v_i \in V & \text{ are certified, same master secret used}(\mathsf{ProveCL}) \\
\wedge & \text{ pseudonyms matching master secret } (\mathsf{ProvePseudonym}) \\
\wedge & \text{ commitments to some } v_i \ (\mathsf{ProveCommitment}) \\
\wedge & \ v_i \text{ lies satisfies some inequality } (\mathsf{ProveInequality}, \text{etc}\ldots) \\
\wedge & \text{ some } v_i \text{ are encrypted } (\mathsf{ProveVerEnc}, \text{etc}\ldots) \\
\wedge & \ldots \text{and other predicates}\}
\end{aligned}
\tag{2}
$$

Implicit in (2) is that the attributes in the predicates ProveCommitment, ProveInequality, and ProveVerEnc, etc. are the same as those certified in ProveCL. Thus, in addition to proving an attribute m_i lies in a given range or is contained in a given commitment, the verifier is also assured that m_i appears in a particular credential. Finally, note that some certified attributes are revealed to the verifier in the clear (A_r). Here, the verifier is assured that the attributes were signed by the issuer, in addition to learning their values.

The basic proof structure for an individual credential proof is similar to a Schnorr signature, the non-interactive version of a common three-move ZK proof. First, the PROVER computes random values of the form $t := g^r$, then she computes a challenge $c := H(\ldots \| t)$, and finally she computes a response of the form $s := r - c\alpha$, to prove knowledge of α. We will refer to the value(s) in the first step as t-value(s) and the response(s) as s-value(s).

The major steps in (2) will be described as separate protocols. For each credential in the specification, we must prove possession of a CL-signature, using ProveCL. We may also need various other sub-protocols (or "sub-provers"), typically one for each predicate type. The modularity of implementing each predicate by a sub-protocol, has been helped manage the complexity of ProveProtocol, both in documentation and implementation.

Since all sub-proofs will share a challenge value, they must run in two steps. First, each sub-protocol outputs the t-values which are all included in a hash to form the challenge. Then, given the challenge, the sub-protocols output the s-values. The table below gives the names of each predicate and the corresponding sub-provers for proof and verification.

Predicate Name	ProveProtocol	VerifyProtocol
CLPredicate	ProveCL	VerifyCL
InequalityPredicate	ProveInequality	VerifyInequality
PrimeEncodePredicate	ProvePrimeEncode	VerifyPrimeEncode
CommitmentPredicate	ProveCommitment	VerifyCommitment
RepresentationPredicate	ProveRepresentation	VerifyRepresentation
PseudonymPredicate	ProvePseudonym	VerifyPseudonym
DomainNymPredicate	ProveDomainNym	VerifyDomainNym
VerEncPredicate	ProveVerEnc	VerifyVerEnc
EpochPredidcate	reveal the epoch attribute	

Our presentation of ProveProtocol first describes how the PROVER generates the proof, and then how the VERIFIER performs verification. Recall that the PROVER and VERIFIER augment the specification with additional, possibly private, values.

Validation The specification is (partially) validated at three occasions: first, when it is loaded through the Parser; second, the specification is built; third, when it is passed to the Prover. This method in validating allows us to fail as early as possible. However, assume that the proof specification defines attribute value v_i and attribute value v_j to be the same. If $v_i \neq v_j$, the validation will not fail but the proof cannot be compiled. Let us summarize the validations:

Parser validations done when parsing a proof specification from XML

- attribute identifiers i have unique names
- $\forall dNym$: domain is not null
- all identifiers i referenced in CLPredicate, InequalityPredicate, RepresentationPredicate, CommitmentPredicate, EnumPredicate, or VerEncPredicate are found, i.e., $i \in \mathcal{I}_r \cup \mathcal{I}_{\overline{r}}$.

ProofSpec validations done when building a proof specification object

- all attributes have corresponding attribute identifiers
- data type of attributes and identifiers match
- group parameters among all CLPredicates match

Prover validations carried out when creating a prover object

- all credentials referenced from a CLPredicate are provided to the Prover
- all referenced pseudonyms are provided to the Prover

Prover **and** Verifier validations

- the bases in the proof specification match the bases in the representation object.
- messages in proof specification matches library internal message.

6.2.2 Protocol: buildProof

Input: m_1, $\{cred\}$, $\mathcal{S}$, n_1, $\{comm\}$, $\{rep\}$, $\{nym\}$, $\{dNym\}$, $\{verEnc\}$, $\{msg\}$.
Output: non-interactive proof of statements in $\mathcal{S}$: $(\mathbf{Common}, c, \mathbf{s})$

Here, $\{cred\}$ are the credential(s), $\{comm\}$ the commitments, $\{rep\}$ the representations, $\{nym\}$ the pseudonyms, $\{dNym\}$ the domain pseudonyms, $\{verEnc\}$ the verifiable encryptions, and $\{msg\}$ the message(s) that are needed to compile the proof. The nonce n_1 is generated by the VERIFIER and sent to the PROVER.

Let $\mathcal{P}$ be the set of predicates as specified in $\mathcal{S}$, $\mathcal{I}_{\overline{r}}$ the set of non-revealed identifiers, and $\mathcal{I}_r$ the set of revealed identifiers. This protocol coordinates the use of the sub-provers and passes the relevant inputs of the Prover to them.

0. *Setup*

0.1 For each hidden identifier $v_i \in V_h$, generate $\hat{v_i} \in_R \{0,1\}^{\ell_m+\ell_\varnothing+\ell_H}$. These values are stored in the respective identifier.

1. *Compute t-values*

1.1 For each predicate p in $\mathcal{P}$ call the appropriate sub-prover. Each sub-prover has access to a list **T** to store the t-values, and a list **Common** to store common values. Sub-provers can access the random values from step 0.1 through the identifiers.

We assume the sub-protocols maintain state until the end of the proof (i.e., until both the t-values and s-values have been computed).

2. *Compute the challenge*

2.1 Challenge c:

$$c := H(\texttt{context}, \mathbf{Common}, \mathbf{T}, \{comm\}, \{rep\}, \{nym\}, \{dNym\}, \{verEnc\}, \{msg\}, n_1),$$

where `context` is a string representing the context (or environment), defined in § 4.2,

3. *Compute the responses*

3.1 For each predicate $p \in \mathcal{P}$, call the corresponding sub-prover. Sub-provers output s-values to $\mathbf{s}$ as required, indexed by predicate or identifier name.

4. *Output proof*

4.1 Output the proof $(c, \mathbf{s}, \mathbf{Common}, \mathcal{R})$. The structure of $\mathbf{s}$ is such that given $\mathbf{s}$ and $\mathcal{S}$ the verifier can easily determine the correct s-values to use during verification.

6.2.3 Protocol: ProveCL

For each credential, the sub-protocol ProveCL is invoked which proves the following,

$$\begin{aligned} SPK\{(e, \{m_i : i \in A_{\mathrm{h}}\}, v) :& \\ \frac{Z}{\prod_{i \notin A_{\mathrm{h}}} R_i^{m_i}} &\equiv \pm A^e S^v \prod_{i \in A_{\mathrm{h}}} R_i^{m_i} \pmod{n} \\ \wedge \quad & m_i \in \{0,1\}^{\ell_m+\ell_\varnothing+\ell_H+2} \;\; \forall \;\; i \in A_{\mathrm{h}} \\ \wedge \quad & e - 2^{\ell_e-1} \in \{0,1\}^{\ell'_e+\ell_\varnothing+\ell_H+2} \}(n_1) \end{aligned}$$

and expands to the following protocol.

Input	*cred*, CLPredicate p, $[c]$
Output	if $c \equiv \perp$, outputs one t-value, and a common value A' otherwise outputs three s-values

Let (A, e, v) be the CL signature for *cred*. Let I be the list of identifiers in p. If $c \not\equiv \perp$, then steps 1-3 have already been executed, skip to Step 4.

1. *Randomize signature*

1.1 Choose $r_A \in_R \{0,1\}^{\ell_n+\ell_\varnothing}$

1.2 Compute the randomized CL signature (A', e, v'), where

$$\begin{aligned} A' &:= AS^{r_A} \pmod{n}, \\ v' &:= v - er_A \text{ (in } \mathbb{Z}). \end{aligned}$$

Additionally compute $e' := e - 2^{\ell_e-1}$.

2. *Compute t-values*

2.1 Choose random integers

$$\tilde{e} \in_R \pm \{0,1\}^{\ell'_e+\ell_\varnothing+\ell_H}$$
$$\tilde{v}' \in_R \pm \{0,1\}^{\ell_v+\ell_\varnothing+\ell_H} \ .$$

For each identifiers $i \in I$, recover the corresponding random value $\tilde{m}_i$, computed in step 0.1 of buildProof.

2.2 Compute

$$\tilde{Z} := (A')^{\tilde{e}} \left(\prod_{i \in I} R_i^{\tilde{m}_i} \right) (S^{\tilde{v}'}) \bmod n \ .$$

3. Output t-value $\tilde{Z}$, common value A'.

4. *Compute the s-values.* Compute the following in $\mathbb{Z}$:

4.1 $\hat{e} := \tilde{e} + ce' \quad (= \tilde{e} + c(e - 2^{\ell_e - 1}))$,

4.2 $\hat{v}' := \tilde{v}' + cv'$,

4.3 $\hat{m}_i := \tilde{m}_i + cm_i$ for $i \notin A_r$.

6.2.4 Proof Prime Encoding

The protocol ProvePrimeEncode is implemented by three other protocols, ProveCGAND, ProveCGOR, and ProveCGNOT, the Camenisch-Gross [CG08] protocols for the AND, OR and NOT operators applicable to PrimeEncodePredicate. These three protocols are described separately.

The prime encoding protocols require a modulus n (mostly for computing commitments). This is taken from the issuer's public key in the first credential certifying the attribute that appears in the predicate being proven.

Protocol: ProveCGAND Here we describe the procotol for predicates with the AND operator. Let m be the value specified by the attribute, c_i the primes corresponding to the attributes revealed during the proof, and $m_r := \prod_i c_i$ their product. The product of the primes not revealed during the proof are denoted $m_h := m/m_r$.

Let $C := Z^m S^r$ be a commitment to m. The protocol ProveCGAND proves

$$SPK\{(m,r,m_h) : C \equiv \pm Z^m S^r \pmod{n} \ \wedge \ C \equiv \pm (Z^{m_r})^{m_h} S^r \pmod{n}\} \ .$$

Input	PrimeEncodePredicate p (with AND operator), $[c]$
Output	if $c \equiv \perp$, outputs two t-values, and a common value otherwise outputs two s-values

Recover m from the identifier, obtain the value m_r from the predicate, and compute $m_h := m/m_r$. The random value $\tilde{m}$, is the shared value computed in step 0.1 of buildProof.

1. *Commit to m*

1.1 Choose $r \in_R \{0,1\}^{\ell_n}$

1.2 Compute the commitment $C = Z^m S^r \pmod{n}$.

2. *Compute t-value*

2.1 Choose the random integers

$$\tilde{m}_h \in_R \pm \{0,1\}^{\ell_m+\ell_\varnothing+\ell_H+1-\ell_{m_r}}$$
$$\tilde{r} \in_R \pm \{0,1\}^{\ell_n+\ell_\varnothing+\ell_H+1}$$

where ℓ_{m_r} is the bitlength of m_r.

2.2 Compute

$$\tilde{C} := (Z^{m_r})^{\tilde{m}_h} S^{\tilde{r}} \pmod n,$$
$$\tilde{C_0} := Z^{\tilde{m}} S^{\tilde{r}} \pmod n.$$

3. Output t-values $\tilde{C}$ and $\tilde{C_0}$, common value C.

4. *Compute and output the s-values.* Compute the following in $\mathbb{Z}$:

4.1 $\hat{m}_h := \tilde{m}_h + cm_h$

4.2 $\hat{r} := \tilde{r} + cr$.

Protocol: ProveCGNOT Let m_i be the factors of the attribute m, and m_r be the constant we want to show that for all i such that $m = \prod_i m_i$, $m_i \neq m_r$ holds. The sub-protocol ProveCGNOT proves the following,

$$SPK\{(m, r, a, b, r'):$$
$$Z \equiv C^a (Z^{m_r})^b S^{r'} \pmod n \qquad \wedge\ C \equiv Z^m S^r \pmod n$$
$$\wedge\ m, r \in \{0,1\}^{\ell_m + \ell_\varnothing + \ell_H + 2} \qquad \wedge\ a, b \in \{0,1\}^{\ell_m + \ell_\varnothing + \ell_H + 2}\ \}$$

and expands to the following protocol.

Input PrimeEncodePredicate p (with NOT operator), $[c]$
Output if $c \equiv \perp$, outputs two t-values, and a common value otherwise outputs four s-values

If m is not a comitted attribute, and if i is not in $\mathcal{E}$, then we need to commit to it, otherwise we skip to step 1.3.

1. *Commit to m and compute exponents*

1.1 Choose $r \in_R \{0,1\}^{\ell_n}$

1.2 Compute the commitment C, where

$$C := Z^m S^r \pmod n\ ,$$

1.3 Choose random integers

$$\tilde{r} \in_R \pm\{0,1\}^{\ell_m + \ell_\varnothing + \ell_H + 1}\ ,$$

and note that we use the randomness $\tilde{m}$ that we used as randomness for attribute m in the ProveCL protocol.

1.4 Compute a and b with extended Euclid's algorithm, s.t. $ma + m_r b = 1$

1.5 compute r'

$$r' := -ra$$

2. *Compute t-values*

2.1 Choose random values

$$\tilde{r}', \tilde{a}, \tilde{b} \in_R \pm\{0,1\}^{\ell_m + \ell_\varnothing + \ell_H + 1 - n_i \ell_t}$$

2.2 Compute

$$\tilde{C} := C^{\tilde{a}}(Z^{m_r})^{\tilde{b}} S^{\tilde{r}'} \pmod{n} \ .$$
$$\tilde{C}_c := Z^{\tilde{m}} S^{\tilde{r}} \pmod{n} \ .$$

3. Output t-values $\tilde{C}$ and $\tilde{C}_c$, common value C.

4. *Compute and output the s-values.* Compute the following in $\mathbb{Z}$:

4.1 $\hat{r}' := \tilde{r}' + cr'$

4.2 $\hat{a} := \tilde{a} + ca$.

4.3 $\hat{b} := \tilde{b} + cb$.

4.4 If the commitment to m has been computed add also $\hat{r} := \tilde{r} + cr$ as s-value.

Protocol: ProveCGOR The proof specification specifies a number of constants. The goal of the OR proof is to show that any of those constants is contained in an attribute of the credential. Let m_r be the product of those constants and m the value of the attribute (which may also be a product of prime numbers).

For each predicate with an OR relation the sub-protocol ProveCGOR is invoked which proves the following,

$$\begin{aligned} SPK\{(m, m_i, \alpha, \beta, \gamma, \delta, r_0, r_1, r_2, \rho_0, \rho_1, \rho_2) : \ & D \equiv \pm Z^{m_i} S^{r_0} \pmod{n} \\ & \wedge \ Z^{m_r} \equiv \pm D^{\alpha} S^{r_1} \pmod{n} \\ & \wedge \ 1 \equiv \pm D^{\beta} Z^{m} S^{r_2} \pmod{n} \\ & \wedge \ \mathfrak{D} \equiv \pm \mathfrak{g}^{m_i} \mathfrak{h}^{\rho_0} \pmod{n} \\ & \wedge \ \mathfrak{g} \equiv \pm (\mathfrak{D}/\mathfrak{g})^{\gamma} h^{\rho_1} \pmod{n} \\ & \wedge \ \mathfrak{g} \equiv \pm (\mathfrak{D} \cdot \mathfrak{g})^{\delta} \mathfrak{h}^{\rho_2} \pmod{n} \\ & \wedge \ m_i \in \{0,1\}^{\ell_m + \ell_\varnothing + \ell_H + 2} \ \} \ . \end{aligned}$$

The intuition behind this proof is that the prover knows a factor m_i that divides both m as well as m_r. To show this fact, the prover shows knowledge of a factor δ with $\delta m_i \equiv m_r$. Furthermore, he proves that he knows β such that $\beta m_i \equiv m$. Note that he uses the random value chosen during the ProveCL protocol to ensure the equality of m to an attribute in the credential. As this proof would allow a prover to chose m_i as ± 1 the prover must add the commitments from an additional group with two generators $\mathfrak{g}$ and $\mathfrak{h}$ with unknown $\log_{\mathfrak{h}}(\mathfrak{g})$. The proof expands to the following protocol.

Input	PrimeEncodePredicate p (with OR operator), $[c]$
Output	if $c \equiv \perp$, outputs six t-values, and two common values otherwise outputs eleven s-values

1. *Compute exponents*

1.1 compute $\alpha := m_r/m_i$

1.2 compute $\beta := -m/m_i$

1.3 choose random $r_0 \in_R \pm \{0,1\}^{\ell_n}$

1.4 compute $r_1 := -r_0 \alpha$

1.5 compute $r_2 := -r_0 \beta$

1.6 compute $\gamma := (m_i - 1)^{-1}$

1.7 compute $\delta := -(m_i + 1)^{-1}$

1.8 choose random $\rho_0 \in_R \mathbb{Z}_{\mathfrak{q}}$

1.9 compute $\rho_1 := -\rho_0 \gamma$

1.10 compute $\rho_2 := -\rho_0 \delta$

2. *Compute commitments*

$$D := Z^{m_i} S^{r_0} \pmod{n}, \qquad \mathfrak{D} := \mathfrak{g}^{m_i} \mathfrak{h}^{\rho_0} \pmod{n} .$$

3. *Compute t-values*

3.1 Choose random integers

$$\begin{aligned} \tilde{m}_i &\in_R \pm \{0,1\}^{\ell_t+\ell_\varnothing+\ell_H+1} \\ \tilde{\alpha}, \tilde{\beta} &\in_R \pm \{0,1\}^{\ell_m+\ell_\varnothing+\ell_H+1} \\ \tilde{r}_0, \tilde{r}_1, \tilde{r}_2, &\in_R \pm \{0,1\}^{\ell_n+\ell_\varnothing+\ell_H+1} \\ \tilde{\gamma}, \tilde{\delta}, \tilde{\rho}_0, \tilde{\rho}_1, \tilde{\rho}_2 &\in_R \mathbb{Z}_{\mathfrak{q}} \end{aligned}$$

3.2 Compute

$$\begin{aligned} \tilde{T}_1 &:= Z^{\tilde{m}_i} S^{\tilde{r}_0} \pmod{n} \\ \tilde{T}_2 &:= D^{\tilde{\alpha}} S^{\tilde{r}_1} \pmod{n} \\ \tilde{T}_3 &:= D^{\tilde{\beta}} Z^{\tilde{m}} S^{\tilde{r}_2} \pmod{n} , \end{aligned}$$

where $\tilde{m}$ is the randomness chosen for attribute m in the ProveCL protocol.

3.4 Compute

$$\begin{aligned} \tilde{T}_4 &:= \mathfrak{g}^{\tilde{m}_i} \mathfrak{h}^{\tilde{\rho}_0} \pmod{n} \\ \tilde{T}_5 &:= (\mathfrak{D}/\mathfrak{g})^{\tilde{\gamma}} \mathfrak{h}^{\tilde{\rho}_1} \pmod{n} \\ \tilde{T}_6 &:= (\mathfrak{g}\mathfrak{D})^{\tilde{\delta}} \mathfrak{h}^{\tilde{\rho}_2} \pmod{n} \end{aligned}$$

4. Output t-values $\tilde{T}_1, \tilde{T}_2, \tilde{T}_3, \tilde{T}_4, \tilde{T}_5, \tilde{T}_6,$, common values $D, \mathfrak{D}$.

5. *Compute and output the s-values.* Compute the following in $\mathbb{Z}$:

5.1 $\hat{m}_i := \tilde{m}_i + c m_i$

5.2 $\hat{\alpha} := \tilde{\alpha} + c\alpha$.

5.3 $\hat{\beta} := \tilde{\beta} + c\beta$

5.4 $\hat{r}_0 := \tilde{r}_0 + c r_0$.

5.5 $\hat{r}_1 := \tilde{r}_1 + c r_1$.

5.6 $\hat{r}_2 := \tilde{r}_2 + c r_2$.

5.7 $\hat{\gamma} := \tilde{\gamma} + c\gamma$.

5.8 $\hat{\delta} := \tilde{\delta} + c\delta$.

5.9 $\hat{\rho}_0 := \tilde{\rho}_0 + c\rho_0$.

5.10 $\hat{\rho}_1 := \tilde{\rho}_1 + c\rho_1$.

5.11 $\hat{\rho}_2 := \tilde{\rho}_2 + c\rho_2$.

6.2.5 Protocol: ProveInequality

An inequality predicate specifies an attribute value m, an operator $*$ and a constant m_r and proves that a $m * m_r$ holds. Instead of using a constant to define m_r the proof specification can indicate a revealed attribute. It's value will define m_r. The operator $*$ is one of $>, \geq, <, \leq$. The proof idea is to calculate the delta value from m and m_r depending on the operator. Then the delta value is split into the sum of four squares $u_i, i \in \{1, \ldots, 4\}$ and creates the following proof. The proof guarantees that the delta value is a positive number.

$$
\begin{aligned}
SPK\{(m, r_\Delta, \{u_1, \ldots, u_4\}, \{r_1, \ldots, r_4\}, \alpha) :& \\
&\wedge\ T_\Delta^a Z^b \equiv \pm Z^m (S^a)^{r_\Delta} \pmod{n} \\
&\wedge\ T_j \equiv \pm Z^{u_j} S^{r_j} \pmod{n}, \text{ for } j = 1, 2, 3, 4 \\
&\wedge\ T_\Delta \equiv T_1^{u_1} \ldots T_4^{u_4} S^\alpha \pmod{n}\}(n_1)
\end{aligned}
$$

where $\alpha = r_\Delta - \sum_{j=1}^{4} u_j r_j$.

Input	InequalityPredicate p, $[c]$
Output	if $c \equiv \perp$, outputs six t-values, and five common values otherwise outputs ten s-values

Let r_m be the random value for m chosen in Step 0.1 of buildProof. If $c \not\equiv \perp$, Steps 1-2 have already been executed, skip to Step 3.

1. *Proof Setup*

1.1 Define

$$
\Delta := \begin{cases} m_r - m & \text{if } * \equiv \leq \\ m_r - m - 1 & \text{if } * \equiv < \\ m - m_r & \text{if } * \equiv \geq \\ m - m_r - 1 & \text{if } * \equiv > \end{cases}
$$

and

$$
a := \begin{cases} -1 & \text{if } * \equiv \leq \text{ or } < \\ 1 & \text{if } * \equiv \geq \text{ or } >. \end{cases}
$$

Note that Δ will always be non-negative if the predicate is true. If $\Delta < 0$, the proof fails and returns $\perp$.

1.2 Express Δ as the sum of four squares,

$$
\Delta := {u_1}^2 + {u_2}^2 + {u_3}^2 + {u_4}^2 .
$$

The current implementation uses an algorithm of Rabin and Shallit, described in [RS86, Sch]. Note that this step accounts for a substantial fraction of the computation time of this protocol, and increases non-linearly with increasing values of Δ.

1.3 Compute:

$$
\begin{aligned}
T_1 &:= Z^{u_1} S^{r_1} \pmod{n} & T_2 &:= Z^{u_2} S^{r_2} \pmod{n} \\
T_3 &:= Z^{u_3} S^{r_3} \pmod{n} & T_4 &:= Z^{u_4} S^{r_4} \pmod{n} \\
T_\Delta &:= Z^\Delta S^{r_\Delta} \pmod{n} ,
\end{aligned}
$$

where $r_\Delta, r_i \in_R \{0,1\}^{\ell_n + \ell_\varnothing}$. Record $r_\Delta, r_i, T_\Delta, T_i$ for later use.

2. *Compute t-values*

2.1 Compute :

$$
\begin{aligned}
\hat{T}_1 &:= Z^{\tilde{u}_1} S^{\tilde{r}_1} \pmod{n} & \hat{T}_2 &:= Z^{\tilde{u}_2} S^{\tilde{r}_2} \pmod{n} \\
\hat{T}_3 &:= Z^{\tilde{u}_3} S^{\tilde{r}_3} \pmod{n} & \hat{T}_4 &:= Z^{\tilde{u}_4} S^{\tilde{r}_4} \pmod{n} \\
\hat{T}_\Delta &:= Z^{\tilde{m}} (S^a)^{\tilde{r}_\Delta} \pmod{n} .
\end{aligned}
$$

where $\tilde{u}_i \in_R \{0,1\}^{\ell_m+\ell_H+\ell_\varnothing}$ and $\tilde{r}_i, \tilde{r}_\Delta \in_R \{0,1\}^{\ell_m+\ell_H+2\ell_\varnothing}$. Note, $\tilde{m}$ is the randomness shared accross sub-provers.

2.2 Compute

$$Q := T_1^{\tilde{u}_1} T_2^{\tilde{u}_2} T_3^{\tilde{u}_3} T_4^{\tilde{u}_4} S^{\tilde{\alpha}} \pmod{n}$$

where $\tilde{\alpha} \in_R \{0,1\}^{\ell_n+\ell_m+2\ell_k+2\ell_\varnothing+3}$.

2.3 Output t-values: $\hat{T}_1, \ldots, \hat{T}_4, \hat{T}_\Delta, Q$.

2.4 Output **Common** values $T_\Delta, T_1, \ldots, T_4$.

3. *Compute s-values*

3.1 For $i \in \{1,2,3,4\}$, compute and output

$$\begin{array}{l} \hat{u}_i := \tilde{u}_i + cu_i \\ \hat{r}_i := \tilde{r}_i + cr_i \end{array} ,$$

3.2 as well as $\hat{\Delta} := \tilde{\Delta} + c\Delta$, and $\hat{r}_\Delta := \tilde{r}_\Delta + cr_\Delta$.

3.3 Compute $\alpha := r_\Delta - \sum_{j=1}^{4} u_j r_j$, and output $\hat{\alpha} := \tilde{\alpha} + c\alpha$.

6.2.6 Protocol: ProveCommitment

The ProveCommitment protocol proves knowledge of a commitment and is run for each commitment predicate. A commitment *comm* is of the form $C := R_1^{a_1} R_2^{a_2} \ldots R_L^{a_L} S^r$. The bases R_i and S are from a public key of an issuer, and $a_1, \ldots, a_L$, are the committed values. The proof works for one to L values, and each value i may either be revealed ($i \in \mathcal{I}_r$) or non-revealed ($i \in \mathcal{I}_{\overline{r}}$).

We prove:

$$SPK\{(a_1, \ldots, a_L, r) : \frac{C}{\prod_{i \in \mathcal{I}_r} R_i^{a_i}} \equiv \pm \left(\prod_{i \in \mathcal{I}_{\overline{r}}} R_i^{a_i} \right) S^r \pmod{n}\} .$$

Input	*comm*, CommitmentPredicate p, $[c]$
Output	if $c \equiv \perp$, outputs one t-value otherwise outputs one s-values

If $c \not\equiv \perp$, skip to Step 2.

1. *Compute t-values*

1.1 Choose the random integer

$$\tilde{r} \in_R \pm\{0,1\}^{\ell_n+\ell_\varnothing+\ell_H+1}$$

1.2 Compute and output

$$\tilde{C} := \left(\prod_{i \in \mathcal{I}_{\overline{r}}} R_i^{\tilde{a}_i} \right) S^{\tilde{r}} \pmod{n} ,$$

where $\tilde{a}_i$ are chosen in Step 0.1 of buildProof. Note that the commitment C is a common value.

2. *Compute response values*

2.1 Compute in $\mathbb{Z}$ and output $\hat{r} := \tilde{r} + cr$.

6.2.7 Protocol: ProveRepresentation

Let a representation *rep* be of the form $R := \prod_{i=1}^{k} g_i^{x_i} \pmod{n}$. The random values corresponding to the exponents x_i are generated in Step 0.1 of buildProof. Since all random values are generated in buildProof, this sub-protocol computes no responses based on c.

We prove:

$$SPK\{(x_1, \ldots, x_k) : R \equiv \pm \prod_{i=1}^{k} g_i^{x_i} \pmod{n}\} .$$

Input	*rep*, RepresentationPredicate p, $[c]$
Output	if $c \equiv \perp$, outputs one t-value otherwise outputs one s-values

1. *Compute and output t-Value* $\tilde{R} = \prod g_j^{\tilde{r}_j}$, for all j such that x_j corresponds to a hidden identifier. Note that $\tilde{r}_j$ corresponds to the randomness chosen for the value x_j, and that the representation R is a common value.

6.2.8 Protocol: ProvePseudonym/ProveDomainNym

The buildProof protocol proves knowledge of the secret underlying a pseudonym and that the pseudonym is based on the master secret key. Let *nym* and/or *dNym* be a pseudonym or domain pseudonym, respectively.

This implements the sub-protocol

$$\begin{aligned} SPK\{(m_1, r_1) : & \\ & nym \equiv g^{m_1} h^{r} \pmod{\Gamma} \wedge_{nym \neq \perp} \\ & dNym \equiv g_{\mathsf{dom}}^{m_1} \pmod{\Gamma} \wedge_{dNym \neq \perp} \} \end{aligned}$$

where m_1 is the master secret key.

Input	*nym* and/or *dNym*, NymPredicate and/or DomNymPredicate p, $[c]$
Output	if $c \equiv \perp$, outputs one t-value otherwise outputs one s-values

If $c \not\equiv \perp$, skip to Step 2.

1. *Compute t-value*

1.1 Choose a random integer $\tilde{r} \in_R \mathbb{Z}_\rho$.

1.2 Compute and output

$$\begin{aligned} \tilde{nym} &:= g^{\tilde{m}_1} h^{\tilde{r}} \pmod{\Gamma} \text{ if } nym \neq \perp \text{ and} \\ \tilde{dNym} &:= g_{\mathsf{dom}}^{\tilde{m}_1} \pmod{\Gamma} \text{ if } dNym \neq \perp \ . \end{aligned}$$

where $\tilde{m}_1$ is chosen in Step 0.1 of buildProof. Note that the pseudonym *nym* (or the domain pseudonym *dNym*) are common values.

2. *Compute s-values*

2.1 Compute in $\mathbb{Z}$ and output $\hat{r} := \tilde{r} + cr$.

6.2.9 Protocol: ProveVerEnc

We assume that the prover has made a Camenisch-Shoup encryption $(\mathsf{u}, \mathsf{v}, \mathsf{e})$ of some attribute m with label L and now want to prove that this is indeed the case. That is, we want to run the following subprotocol

$$PK\{(\mathsf{r}, m) : \mathsf{u}^2 \equiv \mathsf{g}^{2\mathsf{r}} \ \wedge \ \mathsf{e}^2 \equiv \mathsf{y}_1^{2\mathsf{r}}\mathsf{h}^{2m} \ \wedge \ \mathsf{v}^2 \equiv (\mathsf{y}_2\mathsf{y}_3^{\mathcal{H}_{\mathsf{hk}}(\mathsf{u},\mathsf{e},L)})^{2\mathsf{r}} \ \} \ .$$

Note that r is the randomness used to encrypt m as $(\mathsf{u}, \mathsf{v}, \mathsf{e})$

Input *verEnc*, VerEncPredicate p, $[c]$
Output if $c \equiv \perp$, outputs 3 t-values otherwise outputs s-value

If $c \not\equiv \perp$, Steps 1 has already been executed, skip to Step 2. The predicate p contains the public key, and the ciphertext $(\mathsf{u}, \mathsf{v}, \mathsf{e})$, as well as the randomness r has been loaded by the prover.

1. *Compute t-values*

1.1 Choose random random $\tilde{\mathsf{r}} \in \pm \{0,1\}^{2\ell_{\mathsf{enc}}+\ell_{\varnothing}+\ell_H+1}$

1.2 Compute

$$\hat{\mathsf{u}} := \mathsf{g}^{2\tilde{\mathsf{r}}} \pmod{n^2}, \ \ \hat{\mathsf{e}} := \mathsf{y}_1^{2\tilde{\mathsf{r}}}\mathsf{h}^{2\tilde{m}} \pmod{n^2}, \ \text{ and } \ \hat{\mathsf{v}} := (\mathsf{y}_2\mathsf{y}_3^{\mathcal{H}_{\mathsf{hk}}(\mathsf{u},\mathsf{e},L)})^{2\tilde{\mathsf{r}}} \pmod{n^2} \ .$$

1.3 Output t-values (to be hashed): $\hat{\mathsf{u}}$, $\hat{\mathsf{e}}$, and $\hat{\mathsf{v}}$

2. (*Compute s-values*)

2.1 Compute and output $\hat{\mathsf{r}} := \tilde{\mathsf{r}} + c\mathsf{r}$.

6.2.10 The Verify Protocol

The verification of the proof will have a similar structure to its generation. The main protocol, VerifyProtocol will iterate over the predicates in $\mathcal{S}$ and make calls to to the appropriate verification sub-protocols. The sub-protocols compute the verification values ($\hat{t}$-values) to be hashed (together) and compared against c.

6.2.11 Protocol: verifyProof

Input $\mathcal{S}$, $P = (c, \mathbf{s}, \mathbf{Common})$, n_1
Output accept or reject P

The data structure $\mathbf{s}$ is accesible to all sub-provers, which may retrieve the required s-values.

1. *Compute $\hat{t}$-values*

1.1 For each predicate p in $\mathcal{P}$ call the appropriate sub-verifier. Each sub-verifier has access to a list $\hat{\mathbf{T}}$ to store the $\hat{t}$-values, and the appropriate common values from **Common**. Sub-verifiers can access the s-values.

2. *Compute the verification challenge*

2.1

$$\begin{aligned} c := H(&\texttt{context}, \mathbf{Common}, \hat{\mathbf{T}}, \{comm\}, \{rep\}, \{nym\}, \{dNym\}, \\ &\{verEnc\}, \{msg\}, n_1), \end{aligned}$$

where `context` is defined in § 6.2.2.

3. *Verify equality of challenge*

3.1 If $c \equiv \hat{c}$ accept P and reject otherwise.

6.2.12 Protocol: VerifyCL

Input Credential structure, CLPredicate p
Output one $\hat{t}$-value

Using p, the verifier retrieves the common value A' and the s-values. To compute the t-value, the verifier iterates over the attributes from the credential structure and retrieves either the s-value through the attribute identifier name or the attribute value (also stored in the list of s-values).

1. Compute:

$$\hat{T} := \left(\frac{Z}{(\prod_{i \in A_r} R_i^{m_i})(A')^{2^{\ell_e - 1}}} \right)^{-c} (A')^{\hat{e}} \left(\prod_{i \in A_{\overline{r}}} R_i^{\hat{m}_i} \right) S^{\hat{v}'} \bmod n$$

2. Check lengths:

$$\begin{aligned} \hat{m}_i &\in \pm \{0,1\}^{\ell_m + \ell_\varnothing + \ell_H + 1}, \text{ for } i \in A_{\overline{r}}, \\ \hat{e} &\in \pm \{0,1\}^{\ell'_e + \ell_\varnothing + \ell_H + 1} . \end{aligned}$$

If any of these checks fail, reject the proof.

3. Output $\hat{T}$.

6.2.13 Verify Prime Encoding

Similar to ProvePrimeEncode (§ 6.2.4), separate sub-protocols are used to implement the verification of prime encoded attribute proofs. These are VerifyCGAND, VerifyCGOR and VerifyCGNOT, which are described in § 6.2.13, § 6.2.13, § 6.2.13 . Much of the computation must be done with respect to an issuer public key. For a PrimeEncodePredicate p, we use the public key of the first credential in the specification which certifies the attribute referenced by p.

Protocol: VerifyCGAND

Input PrimeEncodePredicate p (with AND operator)
Output one $\hat{t}$-value

Let $\hat{m}$ be the s-value corresponding to the identifier specified in p, m_r the constant in p, and let $\hat{m}_h, \hat{r}$ and C be the output of ProveCGAND.

1. Compute:

$$\begin{aligned} \hat{C} &:= C^{-c} (Z^{m_r})^{\hat{m}_h} S^{\hat{r}} \pmod{n}, \text{ and} \\ \hat{C}_0 &:= C^{-c} Z^{\hat{m}} S^{\hat{r}} \pmod{n}. \end{aligned}$$

2. Check length:

$$\hat{m}_h \in \pm \{0,1\}^{\ell_m + \ell_\varnothing + \ell_H + 1 - \ell_{E_r}} ,$$

where ℓ_{E_r} is the bitlength of E_r. Reject the proof if this check fails.

3. Output $\hat{C}, \hat{C}_0$.

Protocol: VerifyCGNOT

Input PrimeEncodePredicate p (with NOT operator)
Output one $\hat{t}$-value

Let $\hat{m}$ be the s-value corresponding to the identifier specified in the CL proof, m_r the constant in the proof specification, and let $\hat{r}', \hat{a}, \hat{b}, \hat{r}$ and C be the output of ProveCGNOT.

1. Compute:

$$\hat{C} := Z^{-c} C^{\hat{a}} (Z^{m_r})^{\hat{b}} S^{\hat{r}'} \pmod{n} \tag{3}$$
$$\hat{C}_c := C^{-c} Z^{\hat{m}} S^{\hat{r}} \pmod{n} \ . \tag{4}$$

2. Check lengths:

$$\begin{aligned} \hat{m} &\in \pm \{0,1\}^{\ell_m+\ell_\varnothing+\ell_H+1} \\ \hat{a} &\in \pm \{0,1\}^{\ell_m+\ell_\varnothing+\ell_H+1-n_i\ell_t} \\ \hat{b} &\in \pm \{0,1\}^{\ell_m+\ell_\varnothing+\ell_H+1-n_i\ell_t} \quad . \end{aligned}$$

If any of these checks fail, reject the proof.

3. Output $\hat{C}$ and $\hat{C}_c$.

Protocol: VerifyCGOR

Input PrimeEncodePredicate p (with OR operator)
Output six $\hat{t}$-values

1. Compute:

$$\begin{aligned} \hat{T}_1 &:= D^{-c} Z^{\hat{m}_i} S^{\hat{r}_0} \pmod{n} \\ \hat{T}_2 &:= (Z^{m_r})^{-c} D^{\hat{\alpha}} S^{\hat{r}_1} \pmod{n} \\ \hat{T}_3 &:= D^{\hat{\beta}} Z^{\hat{m}} S^{\hat{r}_2} \pmod{n} \\ \hat{T}_4 &:= \mathfrak{D}^{-c} \mathfrak{g}^{\hat{m}_i} \mathfrak{h}^{\hat{\rho}_0} \pmod{n} \\ \hat{T}_5 &:= \mathfrak{g}^{-c} (\mathfrak{D}/\mathfrak{g})^{\hat{\gamma}} \mathfrak{h}^{\hat{\rho}_1} \pmod{n} \\ \hat{T}_6 &:= \mathfrak{g}^{-c} (\mathfrak{g}\mathfrak{D})^{\hat{\delta}} \mathfrak{h}^{\hat{\rho}_2} \pmod{n} \end{aligned}$$

2. Check lengths:

$$\begin{aligned} \hat{m}_i &\in \pm \{0,1\}^{\ell_t+\ell_\varnothing+\ell_H+1} \\ \hat{\alpha}, \hat{\beta} &\in \pm \{0,1\}^{\ell_m+\ell_\varnothing+\ell_H+1} \quad . \end{aligned}$$

If any of these checks fail, reject the proof.

3. Output $\hat{T}_1, \hat{T}_2, \hat{T}_3, \hat{T}_4, \hat{T}_5, \hat{T}_6$.

6.2.14 Protocol: VerifyInequality

Input InequalityPredicate p
Output six $\hat{t}$-value

Let $T_\Delta, T_1, T_2, T_3, T_4$ be the common values from the proof.

1. Compute Δ' and update c if necessary.

$$\Delta' = \begin{cases} m_r & \text{if } * \equiv \leq \\ m_r - 1 & \text{if } * \equiv < \\ m_r & \text{if } * \equiv \geq \\ m_r + 1 & \text{if } * \equiv > \end{cases}$$

$$a = \begin{cases} -1 & \text{if } * \equiv \leq \text{ or } < \\ 1 & \text{if } * \equiv \geq \text{ or } > \end{cases}$$

Note that $\Delta' = m - a\Delta$.

2. *Compute $\hat{t}$-values*

2.1 Compute

$$\hat{T}_\Delta := \left({T_\Delta}^a Z^{\Delta'}\right)^{-c} Z^{\hat{m}} (S^a)^{\hat{r}_\Delta} \pmod{n} .$$

2.2 For $i \in \{1, \ldots, 4\}$, compute

$$\hat{T}_i := T_i^{-c} Z^{\hat{u}_i} S^{\hat{r}_i} \pmod{n} .$$

2.3 Compute

$$\hat{Q} := (T_\Delta)^{-c} T_1^{\hat{u}_1} T_2^{\hat{u}_2} T_3^{\hat{u}_3} T_4^{\hat{u}_4} S^{\hat{\alpha}} \pmod{n} .$$

3. Output $\hat{T}_\Delta, \hat{T}_1, \ldots, \hat{T}_4, \hat{Q}$.

6.2.15 Protocol: VerifyCommitment

Input *comm*, CommitmentPredicate p
Output one $\hat{t}$-value

Let $\hat{r}$ be the s-value, and let $\mathcal{I}_{\overline{r}}$ and $\mathcal{I}_r$ be as defined in ProveCommitment (see § 6.2.6).

1. Compute

$$C' := \frac{C}{\prod_{i \in \mathcal{I}_r} R_i^{a_i}} \pmod{n} ,$$

and

$$\hat{C} := (C')^{-c} \left(\prod_{i \in \mathcal{H}} R_i^{\hat{a}_i} \right) S^{\hat{r}} \pmod{n} .$$

6.2.16 Protocol: VerifyRepresentation

Input *rep*, RepresentationPredicate p
Output one $\hat{t}$-value

For each identifier in p, recover $\hat{x}_i$, from the proof. Let $\mathcal{I}$ be the set of identifiers (i.e., $\{1, \ldots k\}$ used in R, where $\mathcal{I}_r$ are revealed and $\mathcal{I}_{\overline{r}}$ are not revealed.

1. Compute $R' := \prod_{j \in \mathcal{I}_r} g_j^{x_j}$.

2. Compute and output $\hat{R} = (R/R')^{-c} \prod_{j \in \mathcal{I}_{\overline{r}}} g_j^{\hat{x}_j}$.

6.2.17 Protocol: VerifyPseudonym/VerifyDomainNym

Input *nym* or *dNym*, PseudonymPredicate of DomainNymPredicate p
Output one $\hat{t}$-value

Using p the protocol locates the value $\hat{r}$.

1. Compute

$$\begin{aligned} \hat{nym} &:= nym^{-c} g^{\hat{m}_1} h^{\hat{r}} \pmod{\Gamma} && \text{if } nym \neq \perp \text{ and} \\ d\hat{Nym} &:= dNym^{-c} g_{\mathsf{dom}}^{\hat{m}_1} \pmod{\Gamma} && \text{if } dNym \neq \perp \end{aligned}$$

2. Output $\hat{nym}$ and/or $d\hat{Nym}$, if applicable.

6.2.18 Protocol: $\mathsf{VerifyVerEnc}$

Input *verEnc*, $\mathsf{ProveVerEnc}$ p
Output three $\hat{t}$-values

Using p the protocol retrieved the values $\hat{m}_1$ and $\hat{\mathsf{r}}$.

1. (*Compute $\hat{t}$-values*)

1.1 Compute

$$\hat{\mathsf{u}} := \mathsf{u}^{-2c}\mathsf{g}^{2\hat{\mathsf{r}}},$$
$$\hat{\mathsf{e}} := \mathsf{e}^{-2c}\mathsf{y}_1^{2\hat{\mathsf{r}}}\mathsf{h}^{2\hat{m}_1}, \text{ and}$$
$$\hat{\mathsf{v}} := \mathsf{v}^{-2c}\left(\mathsf{y}_2\mathsf{y}_3^{\mathcal{H}_{\mathsf{hk}}(\mathsf{u},\mathsf{e},L)}\right)^{2\hat{\mathsf{r}}} \ .$$

2. Output $\hat{\mathsf{u}}$, $\hat{\mathsf{e}}$, and $\hat{\mathsf{v}}$.

References

[BDD07] Stefan Brands, Liesje Demuynck, and Bart De Decker. A practical system for globally revoking the unlinkable pseudonyms of unknown users. In Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson, editors, *ACISP*, volume 4586 of *Lecture Notes in Computer Science*, pages 400–415. Springer, 2007.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *First ACM Conference on Computer and Communication Security*, pages 62–73. Association for Computing Machinery, 1993.

[Bra95a] Stefan Brands. Restrictive blinding of secret-key certificates. Technical Report CS-R9509, CWI, September 1995.

[Bra95b] Stefan Brands. Secret-key certificates. Technical Report CS-R9510, CWI, September 1995.

[CD00] Jan Camenisch and Ivan Damgård. Verifiable encryption, group encryption, and their applications to group signatures and signature sharing schemes. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 331–345. Springer Verlag, 2000.

[CG08] Jan Camenisch and Thomas Groß. Efficient attributes for anonymous credentials. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 345–356, 2008.

[Cha85] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, October 1985.

[CKS09] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, pages 481–500, 2009.

[CL01] Jan Camenisch and Anna Lysyanskaya. Efficient non-transferable anonymous multi-show credential system with optional anonymity revocation. In Birgit Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 93–118. Springer Verlag, 2001.

[CL03] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *Security in Communication Networks, Third International Conference, SCN 2002*, volume 2576 of *Lecture Notes in Computer Science*, pages 268–289. Springer Verlag, 2003.

[CL04] Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In Matthew K. Franklin, editor, *Advances in Cryptology — CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 56–72. Springer Verlag, 2004.

[CS97] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In Burt Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, volume 1296 of *Lecture Notes in Computer Science*, pages 410–424. Springer Verlag, 1997.

[CS02] Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In *Advances in Cryptology — EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2002.

[CS03] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In Dan Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 126–144, 2003.

[Dam90] Ivan Bjerre Damgård. Payment systems and credential mechanism with provable security against abuse by individuals. In Shafi Goldwasser, editor, *Advances in Cryptology — CRYPTO '88*, volume 403 of *Lecture Notes in Computer Science*, pages 328–335. Springer Verlag, 1990.

[Dam00] Ivan Damgård. Efficient concurrent zero-knowledge in the auxiliary string model. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 431–444. Springer Verlag, 2000.

[DF02] Ivan Damgård and Eiichiro Fujisaki. An integer commitment scheme based on groups with hidden order. In *Advances in Cryptology — ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*. Springer, 2002.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer Verlag, 1987.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[IBM09] IBM. Cryptographic protocols of the Identity Mixer library, v. 1.0. IBM Research Report RZ3730, IBM Research, 2009. `http://domino.research.ibm.com/library/cyberdig.nsf/index.html`.

[Nat93] National Institute of Standards and Technology. NIST FIPS PUB 180: Secure hash standard, May 1993.

[NFHF09] Toru Nakanishi, Hiroki Fujii, Yuta Hira, and Nobuo Funabiki. Revocable group signature schemes with constant costs for signing and verifying. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 463–480. Springer, 2009.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite residuosity classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–239. Springer Verlag, 1999.

[Ped92] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer Verlag, 1992.

[PS96] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, volume 1070 of *Lecture Notes in Computer Science*, pages 387–398. Springer Verlag, 1996.

[RS86] Michael O. Rabin and Jeffrey O. Shallit. Randomized algorithms in number theory. *Communications in pure and applied mathematics*, 39:239–256, 1986.

[Sch] P. Schorn. Four squares. `http://schorn.ch/lagrange.html`. Accessed July 2008.

[Sch91] Claus P. Schnorr. Efficient signature generation for smart cards. *Journal of Cryptology*, 4(3):239–252, 1991.

Symbol	Description	Defined
n	RSA modulus for CL signatures	p. 17
p, q	prime factors of n	p. 17
QR_n	group of quadratic residues mod n	p. 17
pk_B, sk_B	public key, secret key of entity B	p. 17
S, Z, R_i	part of the issuer's public key (for CL sigs.)	p. 17
Γ	modulus of the commitment group	p. 16
ρ	prime order of a large subgroup of $\mathbb{Z}_\Gamma^*$	p. 16
b	cofactor of $\Gamma - 1$	p. 16
g, h	generators of the order ρ subgroup of $\mathbb{Z}_\Gamma^*$	p. 16
l	total number of attributes in a certificate (bases in the issuer's CL public key)	p. 17
m_1	the master secret	p. 17, 17
A	ordered set of attributes	p. **??**
$(m_1, \ldots, m_l)$	attributes in A	p. **??**
A_k	indices of attributes which are public during a certificate issue	p. 6
A_h	indices of attributes which are hidden/committed during an issue	p. 6
A_r	indices of attributes revealed during a proof	p. 6
c_i	commitment of attribute i, i.e. $comm(m_i)$	p. **??**
$\mathsf{n}, \mathsf{g}, \mathsf{y}_1, \mathsf{y}_2, \mathsf{y}_3$	Verifiable encryption public key	—
$\mathsf{n}, x_1, x_2, x_3$	Verifiable encryption private key	—
h	Ver. enc. param. $\mathsf{h} = (1 + \mathsf{n} \bmod \mathsf{n}^2) \in \mathbb{Z}_{\mathsf{n}^2}^*$	—

Table 1: Notation used in this document. The "Defined" column gives the page number where the symbol is defined. (The source code may refer to this table as `table:notation`.)

A Notation

Table 1 lists the notation used in this document. Table 2 lists recommended sizes for system parameters, and lists the constraints between parameters.

B Cryptographic Assumptions

The protocols discussed in this chapter rely on the strong RSA and the decisional Diffie-Hellman assumptions. We state then briefly for completeness.

Assumption 1 (Strong RSA Assumption)**.** The strong RSA (SRSA) assumption states that it is computational infeasible, on input a random RSA modulus n and a random element $u \in \mathbb{Z}_n^*$, to compute values $e > 1$ and v such that $v^e \equiv u \pmod{n}$.

The tuple (n, u) generated as above is called an *instance* of the *flexible* RSA problem.

Assumption 2 (DDH Assumption)**.** Let Γ be an ℓ_Γ-bit prime and ρ is an ℓ_ρ-bit prime such that $\rho | \Gamma - 1$. Let $\gamma \in \mathbb{Z}_\Gamma^*$ be an element of order ρ. Then, for sufficiently large values of ℓ_Γ and ℓ_ρ, the distribution $\{(\delta, \delta^a, \delta^b, \delta^{ab})\}$ is computationally indistinguishable from the distribution $\{(\delta, \delta^a, \delta^b, \delta^c)\}$, where δ is a random element from $\langle\gamma\rangle$, and a, b, and c are random elements from $[0, \rho - 1]$

The following theorem will turn out to be handy in some of our analyses.

Theorem B.1. *[CS03] Under the strong RSA assumption, given a modulus n (distributed as above), along with random elements $g, h \in (\mathbb{Z}_n^*)^2$, it is hard to compute an element $w \in \mathbb{Z}_n^*$ and integers a, b, c such that*

$$w^c \equiv g^a h^b \pmod{n} \quad \textit{and } c \textit{ does not divide } a \textit{ or } b. \tag{5}$$

Parameter	Description	Bitlength
ℓ_n	size of RSA modulus	2048
ℓ_Γ	size of the commitment group modulus	1632
ℓ_ρ	size of the prime order subgroup of Γ	256
ℓ_m	size of attributes	256
ℓ_{res}	number reserved attributes in a certificate	$1^\dagger$
ℓ_e	size of e values of certificates	597
ℓ'_e	size of the interval the e values are taken from	120
ℓ_v	size of the v values of the certificates	2724
$\ell_\varnothing$	security parameter that governs the statistical zero-knowledge property (source name `l_statzk`)	80
ℓ_k	security parameter	160
ℓ_H	domain of the hash function H used for the Fiat-Shamir heuristic	256
ℓ_r	security parameter required in the proof of security of the credential system	80
ℓ_{pt}	prime number generation returns composites with probability $1 - 1/2^{\ell_{pt}}$	$80^\dagger$
ℓ_{enc}	security parameter for the CS encryption scheme, bit length of $\sqrt{\mathsf{n}}$	1500

Table 2: System parameter sizes (in bits) used in idemix. (Source code may refer to this table as `table:params`.) († This value is an integer, not a bit length.)

Number	Constraint
1	$\ell_e > \ell_\varnothing + \ell_H + \max\{\ell_m + 4, \ell'_e + 2\}$
2	$\ell_v > \ell_n + \ell_\varnothing + \ell_H + \max\{\ell_m + \ell_r + 3, \ell_\varnothing + 2\}$
3	$\ell_H \geq \ell_k$
4	$\rho \nmid b$
5	$\ell_H < \ell_e$
6	$\ell'_e < \ell_e - \ell_\varnothing - \ell_H - 3$
7	$\ell_\rho \leq \ell_m$

Table 3: Constraints which parameter choices must satisfy to ensure security and soundness. (Source code may refer to this table as `table:constraints`.)

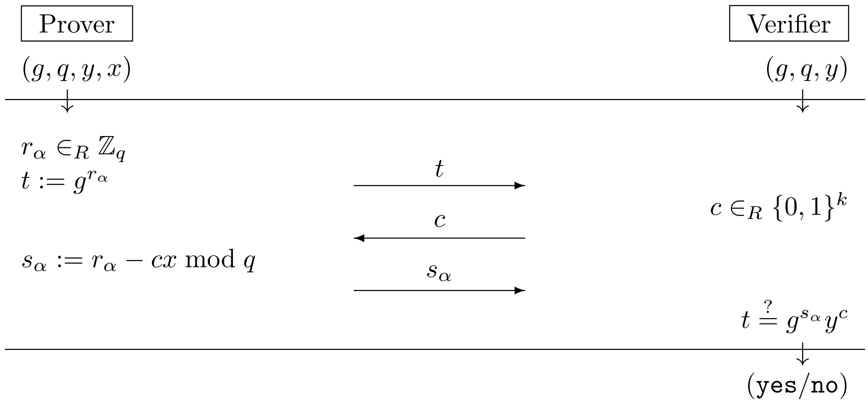

Figure 10: The protocol denoted $PK\{(\alpha) : y = g^\alpha\}$. The prover's input to the protocol is (g, q, y, x) and the verifier's input is (g, q, y). The prover has no output; the verifier's output is either `yes` or `no`, depending on whether or not he accepts the protocols, i.e., whether or not $t = g^{s_\alpha} y^c$ holds.

Intuitively, computing such a w and integers a, b, and c such that c does not divide a and b seems to require knowledge of the group's order: One computes w as $g^x h^y$ for some x and y and then raises both sides to the power of c. Now if c should not divide a, then one seems to need reducing xc modulo the order of the group. However, this is not possible as the order is not known and so it seems impossible to compute such an a that is not divisible by c.

C Protocols to Prove Knowledge of and Relations among Discrete Logarithms

In the following we will use various protocols to prove knowledge of and relations among discrete logarithms. To describe these protocols, we use notation introduced by Camenisch and Stadler [CS97] for various proofs of knowledge of discrete logarithms and proofs of the validity of statements about discrete logarithms. For instance,

$$PK\{(\alpha, \beta, \gamma) : y = g^\alpha h^\beta \ \wedge \ \tilde{y} = \tilde{g}^\alpha \tilde{h}^\gamma \ \wedge \ (v < \alpha < u)\}$$

denotes a "*zero-knowledge* P*roof of* K*nowledge of integers* α*,* β*, and* γ *such that* $y = g^\alpha h^\beta$ *and* $\tilde{y} = \tilde{g}^\alpha \tilde{h}^\gamma$ *holds, with* $v < \alpha < u$*,*" where $y, g, h, \tilde{y}, \tilde{g}$, and $\tilde{h}$ are elements of some groups $G = \langle g \rangle = \langle h \rangle$ and $\tilde{G} = \langle \tilde{g} \rangle = \langle \tilde{h} \rangle$. The convention is that Greek letters denote the quantities the knowledge of which is being proved, while all other parameters are known to the verifier. More precisely, the property of "proof of knowledge" means that there exists a *knowledge extraction* algorithm who can extract the Greek quantities from a successful prover if given rewinding and reset access to the prover (cf. Chapter **??**). Thus, using this notation, a proof protocol can be described by just pointing out its aim while hiding the protocols realization details.

In the following we will first explain how such protocols can be constructed. Unless otherwise stated, we assume a group $G = \langle g \rangle$ of prime order q.

C.1 Schnorr's Identification Scheme

The probably simplest is case is the protocol denoted $PK\{(\alpha) : y = g^\alpha\}$, where $y \in G$, and is depicted in Figure 10. This protocol is also known as Schnorr's identification protocol [Sch91]. As the first step in the protocol, the prover chooses a random integer r_α, computes the *protocol commitment* $t := g^{r_\alpha}$ and sends it to the verifier. The verifier replies with a random *protocol challenge* c. Next, the prover computes the *protocol response* $s := r - cx \bmod q$ and sends it to the verifier. Finally, the verifier accepts the protocol if the *verification equation* $t = g^{s_\alpha} y^c$ holds.

The protocol is a proof of knowledge of the discrete logarithm $\log_g y$ with cheating probability (knowledge error) of 2^{-k}. The protocol is also zero-knowledge against an honest verifier.

To achieve zero-knowledge against any verifier, one needs to choose k logarithmic in the security parameter and repeat the protocols sufficiently many times to make the knowledge error small enough, losing some efficiency by this repetition. Reasonable parameters seem to be $k = 10$ and repeating the protocol 8 times to achieve an overall cheating probability of 2^{-80}. Luckily, one can alternatively use one of the many known constructions to achieve zero-knowledge that retain efficiency, e.g., [Dam00]. We note that this discussion holds for all the protocols we consider in this chapter.

From the protocol just discussed, one can derive the Schnorr signature scheme, denoted $SPK\{(\alpha) : y = g^\alpha\}(m)$, by using the Fiat-Shamir heuristic [FS87, PS96], i.e., by replacing the verifier by a call to a hash function $\mathcal{H}$ and thus computing the challenge as $c = \mathcal{H}(g\|y\|t\|m)$, where $m \in \{0,1\}^*$ is the message that is signed. The signature of m consists of the pair (s, c). The verification equation of the signature scheme entails computing $\hat{t} := g^s y^c$ and then verifying whether $c = \mathcal{H}(g\|y\|\hat{t}\|m)$ holds. This signature scheme can be shown secure in the so-called random oracle model [BR93].

C.2 Proving Knowledge of a Representation

One generalization of Schnorr's identification scheme is to use, say, ℓ bases $g_1, \ldots, g_\ell$ with $g_i \in G$. That is, the protocol denoted $PK\{(\alpha_1, \ldots, \alpha_\ell) : y = \prod_{i=1}^{\ell} g_i^{\alpha_i}\}$ is a proof of knowledge of the representation of $y \in G$ w.r.t. the bases g_i. It is constructed as follows. The inputs of the prover and the verifier consist of y, g_1, …, g_ℓ, the order q of the group, and the system parameter k. The secret input of the prover consists of x_i's such that $y = \prod_{i=1}^{\ell} g_i^{x_i}$. Thus, each x_i corresponds to an α_i. Now, to compute the first message of the protocol, the prover chooses ℓ random integers $r_{\alpha_i} \in \mathbb{Z}_q$, computes $t := \prod_{i=1}^{\ell} g_i^{r_{\alpha_i}}$, and sends t to the verifier. The verifier replies with a c chosen as in the protocol above, i.e., randomly from $\{0,1\}^k$. Next, the prover computes $s_{\alpha_i} := r_{\alpha_i} - cx_i \bmod q$ and sends the resulting $s_{\alpha_1}, \ldots, s_{\alpha_\ell}$ to the verifier. The verifier will accept the protocol if the equation $t = y^c \prod_{i=1}^{\ell} g_i^{s_{\alpha_i}}$ holds. This protocol can be shown to be a proof of knowledge of values α_i such that $y = \prod_{i=1}^{\ell} g_i^{\alpha_i}$ with knowledge error 2^{-k} and to be honest-verifier zero-knowledge.

C.3 Combining Different Proof Protocols

One can combine different instances of the protocol described so far. The simplest combination is a protocol denoted

$$PK\{(\alpha_1, \ldots, \alpha_\ell, \beta_1, \ldots, \beta_{\ell'}) : y = \prod_{i=1}^{\ell} g_i^{\alpha_i} \ \wedge \ z = \prod_{i=1}^{\ell'} h_i^{\beta_i}\}$$

with $g_i, h_i, y, z \in G$. It is obtained by running the two protocols

$$PK\{(\alpha_1, \ldots, \alpha_\ell,) : y = \prod_{i=1}^{\ell} g_i^{\alpha_i}\} \quad \text{and} \quad PK\{(\beta_1, \ldots, \beta_{\ell'}) : z = \prod_{i=1}^{\ell'} h_i^{\beta_i}\}$$

in parallel as sub-protocols as follows. First, the prover computes and sends to the verifier the commitment messages of both protocols at the same time. Next, the verifier chooses and sends back a *single* challenge message, i.e., the verifier chooses the same challenge message for both protocols. Finally, the verifier will accept the overall protocol only if the verification equations of both (sub-)protocols hold.

In the same way one constructs a protocol that involves several terms (i.e., representations of several values) by just running the protocol for each term in parallel and by letting the challenge to be the same for each of these protocols. So, for instance, the protocol $PK\{(\alpha, \beta, \gamma) : y = g^\alpha \ \wedge \ z = h^\beta \ \wedge \ w = g^\gamma\}$ is obtained by running the three sub-protocols $PK\{(\alpha) : y = g^\alpha\}$, $PK\{(\beta) : z = h^\beta\}$, and $PK\{(\gamma) : w = g^\gamma\}$ in parallel in as just described.

C.4 Proving Equality of Discrete Logarithms

Another useful combination of the protocols discusses so far is one where, in addition to prove knowledge of discrete logarithms or representations, the prover can show that some discrete logarithms are equal. Such protocols are in principle also obtained from running the basic protocol for each term in parallel. However, now not only are the challenges the same for each of these protocols but also some of random choices of the prover as well as some of the response messages of the protocols needs to be the same. In general, when we compose protocols for arbitrary terms, the prover is to use the same random r_α in all the protocols that involve α in their term. Let us consider the protocol $PK\{(\alpha, \beta) : y = g^\alpha h^\beta \ \wedge\ z = h^\alpha\}$ as an example. The verifier's and the prover's common input to the protocol consists of y, z, g, h, q and the prover's secret input consists of x_α and x_β such that $y := g^{x_\alpha} h^{x_\beta}$ and $z := h^{x_\alpha}$. To compute the commitment message, the prover chooses random r_α and r_β from $\mathbb{Z}_q$ and computes $t_y := g^{r_\alpha} h^{r_\beta}$ and $t_z := h^{r_\alpha}$. Upon receiving the commitment messages t_y and t_z, the verifier replies with a single random challenge message $c \in_R \{0,1\}^k$. Next, the prover computes the response messages as $s_\alpha := r_\alpha + c x_\alpha \bmod q$ and $s_\beta := r_\beta + c x_\beta \bmod q$. Having received s_α and s_β, the verifier will accept the protocol if the two verification equations $t_y = y^c g^{s_\alpha} h^{s_\beta}$ and $t_z = z^c h^{s_\alpha}$ hold.

Let us explain why the verifier should be convinced that $\log_h z$ equals the first element in the representation of y w.r.t. g and h. Using standard rewinding techniques, one can obtain from a successful prover commitment and response messages for different challenge messages but the same commitment messages, i.e., two tuples $(t_y, t_z, c, s_\alpha, s_\beta)$ and $(t'_y, t'_z, c', s'_\alpha, s'_\beta)$ with $(t_y, t_z) = (t'_y, t'_z)$ and $c \neq c'$. From the verification equations one can thus conclude that

$$y^{c-c'} = g^{s'_\alpha - s_\alpha} h^{s'_\beta - s_\beta} \quad \text{and} \quad z^{c-c'} = h^{s'_\alpha - s_\alpha} \ .$$

Now we can set $\hat{x}_\alpha := (s'_\alpha - s_\alpha)(c - c')^{-1} \bmod q$ and $\hat{x}_\beta := (s'_\beta - s_\beta)(c - c')^{-1} \bmod q$ and thus we have

$$y = g^{\hat{x}_\alpha} h^{\hat{x}_\beta} \quad \text{and} \quad z = h^{\hat{x}_\alpha} \ ,$$

i.e., we see that indeed $\log_h z$ equals the first element in the representation of y w.r.t. g and h.

From what we have now seen, we are able to construct protocols that fall into the class denoted

$$PK\{(\alpha_1, \ldots \alpha_{\ell_\alpha}) : y_1 = \prod_{i \in I_1} g_i^{\alpha_{f_1(i)}} \ \wedge\ \ y_2 = \prod_{i \in I_2} g_i^{\alpha_{f_2(i)}} \ \wedge \\ \cdots \ \wedge\ \ y_{\ell_y} = \prod_{i \in I_{\ell_y}} g_i^{\alpha_{f_{\ell_y}(i)}}\} \ ,$$

where

- ℓ_α, ℓ_g, and ℓ_y are parameters denoting the number of secrets α_i, of bases g_i, and of elements y_i, respectively,
- the y_i's and g_i's can be arbitrary elements of G and do not necessarily be distinct or can be a product of other (given) elements, e.g., $y_3 = y_5 g_2^3 / y_2$,
- the sets I_j define which bases are used for the j-th term and the functions f_j define which secret α_k is used for a particular base in the term.

The protocol is depicted in Figure 11. We note that the protocol has the property that from a successful prover one can extract values $\alpha_1, \ldots, \alpha_{\ell_\alpha}$ such that the equations

$$y_1 = \prod_{i \in I_1} g_i^{\alpha_{f_1(i)}}, \quad y_2 = \prod_{i \in I_2} g_i^{\alpha_{f_2(i)}}, \quad \cdots, \quad y_{\ell_y} = \prod_{i \in I_{\ell_y}} g_i^{\alpha_{f_{\ell_y}(i)}}$$

all hold. That is, it is an honest-verifier zero-knowledge proof of knowledge of the values $\alpha_1, \ldots, \alpha_{\ell_\alpha}$ with knowledge error 2^{-k}.

Let us finally consider some example instances of this general protocol. The protocol denoted $PK\{(\alpha, \beta) : y_1 = g^\alpha \wedge y_2 = g^\beta \wedge y_3 = y_1^\beta\}$ proves that $\log_g y_3$ is the product of $\log_g y_1$ and $\log_g y_2$, the protocol denoted $PK\{(\alpha) : y_1 = g^\alpha \wedge y_2 = y_1^\alpha\}$ proves that $\log_g y_2 = (\log_g y_1)^2$, and the protocol denoted $PK\{(\alpha, \beta) : g = y^\alpha h^\beta\}$ proves that the first element of the representation of y that the prover knows w.r.t. g and h is non-zero, provided that the prover is not privy to $log_g h$ (in case the prover knows $log_g h$, he is able to compute q different representations of y w.r.t. g and h, otherwise he can only know one).

Prover		Verifier
$(g_1,\ldots,g_{\ell_g},y_1,\ldots,y_{\ell_y},$ $I_1,\ldots,I_{\ell_y},f_1,\ldots,f_{\ell_y},$ $x_1,\ldots,x_{\ell_\alpha},q)$		$(g_1,\ldots,g_{\ell_g},y_1,\ldots,y_{\ell_y},$ $I_1,\ldots,I_{\ell_y},f_1,\ldots,f_{\ell_y},$ $q)$
$r_{\alpha_1},\ldots,r_{\alpha_{\ell_\alpha}} \in_R \mathbb{Z}_q$		
For $j=1,\ldots,\ell_y$ do:		
$t_{y_j} := \prod_{i\in I_j} g_i^{r_{\alpha_{f_j(i)}}}$	$\xrightarrow{t_{y_1},\ldots,t_{y_{\ell_y}}}$	
	$\xleftarrow{c}$	$c \in_R \{0,1\}^k$
$s_{\alpha_i} := r_{\alpha_i} - cx_i \bmod q$	$\xrightarrow{s_{\alpha_1},\ldots,s_{\alpha_{\ell_\alpha}}}$	
		For $j=1,\ldots,\ell_y$ check:
		$t_{y_j} \stackrel{?}{=} y_j^c \prod_{i\in I_j} g_i^{s_{\alpha_{f_j(i)}}}$
		(yes/no)

Figure 11: The protocol denoted $PK\{(\alpha_1,\ldots\alpha_{\ell_\alpha}) : y_1 = \prod_{i\in I_1} g_i^{\alpha_{f_1(i)}} \ \wedge\ y_2 = \prod_{i\in I_2} g_i^{\alpha_{f_2(i)}} \ \wedge\ \cdots\ \wedge\ y_{\ell_y} = \prod_{i\in I_{\ell_y}} g_i^{\alpha_{f_{\ell_y}(i)}}\}$.

C.5 The Schnorr Protocol Modulo a Composite

So far we have considered proof protocols for a group of prime order where the order is known to both the prover and the verifier. However, often one would like to use these kind of protocols in groups where the order is not known to all parties as is for instance the case in RSA-groups. RSA groups are subgroups of $\mathbb{Z}_n^*$, where n is the product of two primes. Thus, if one does not know the factorization of n, one does in general not know the order of a subgroup generated by a random element of $\mathbb{Z}_n^*$.

In particular, let n be the product of two safe primes, i.e., primes p and q such that $p' = (p-1)/2$ and $q' = (q-1)/2$ are also primes. Let g be a generator of the quadratic residues modulo n (so, g will have order $p'q'$). In this case, knowing the order $p'q'$ of g is equivalent to knowing the factorization of n. Let $y = g^x$ with $x \in \{0,1\}^\ell$ for some ℓ. Now consider the protocol denoted $PK\{(\alpha) : y = g^\alpha \pmod n\}$, where at least the prover is assumed not to be privy to the factorization of n. The prover and verifier's common inputs are (y, g, n, ℓ_n) and the prover's additional input is $x = \log_g y$, where $\ell_n = \lceil \log_2 n \rceil$ (i.e., the length of n in bits). We may assume that $x \in [0, n]$. The protocol is as follows.

1. The prover chooses uniformly at random $r_\alpha \in_R \{0,1\}^{\ell_n+\ell_c+\ell_\varnothing}$, computes $t_y := g^r \bmod n$, and sends t to the verifier.
2. The verifier chooses a random $c \in_R \{0,1\}^{\ell_c}$ and sends that to the prover.
3. The prover replies with $s_\alpha := r_\alpha - xc$.
4. The verifier checks whether $t_y \stackrel{?}{\equiv} y^c g^{s_\alpha} \pmod n$ holds.

The difference to the protocol in the case the order of the group $\langle g\rangle$ is known is that here, as she does not know the order of the group, the prover can no longer choose r_α randomly from the integers modulo this order and can no longer reduce s_α modulo this order. So, the prover needs to choose r_α some how differently such that nevertheless s_α and t_y do not reveal information about x, i.e., such that the protocol remains zero-knowledge. Thus if $x \in [0, n]$ and the prover chooses $r_\alpha \in_R \{0,1\}^{\ell_n+\ell_c+\ell_\varnothing}$, then, for any $c \in \{0,1\}^{\ell_c}$ and sufficiently large $\ell_\varnothing$ (e.g., 80), the value $s_\alpha := r_\alpha - xc$ is distributed statistically close to the uniform distribution over $\{0,1\}^{\ell_n+\ell_c+\ell_\varnothing}$. Also, the value $t_y := g^r \bmod n$ is distributed statistically close to uniform over $\langle g\rangle$. Therefore, provided that $y \in \langle g\rangle$, the protocol is statistical honest-verifier zero-knowledge for sufficiently large $\ell_\varnothing$ (e.g., $\ell_\varnothing = 80$).

Now, this protocol is only a proof of knowledge of $\log_g y$ if ℓ_c equals 1 and if repeated sufficiently many, say k, times. Let us investigate this. Assume we are given a prover that can successfully run the protocol for given y, g, and n. By standard rewinding techniques, one can extract two triples (t, c, s) and (t', c', s') from the prover such that $t = t'$, $c \neq c'$, and $t \equiv y^c g^s \equiv y^{c'} g^{s'} \pmod{n}$ holds. W.l.o.g. we may assume that $c' > c$. From the last equation we can derive that $y^{c'-c} \equiv g^{s-s'} \pmod{n}$.

If $\ell_c = 1$, we must have $c' = 1$ and $c = 0$ and hence $y \equiv g^{s-s'} \pmod{n}$, i.e., $s - s'$ is a discrete logarithm of y to the base g and hence the protocol is indeed a proof of knowledge.

If $\ell_c > 1$, we are stuck with the equation $y^{c'-c} \equiv g^{s-s'} \pmod{n}$, i.e., we seem to need to compute a $(c' - c)$-th root of $g^{s-s'}$ modulo n which is assumed to be hard without knowledge of g's order. Unfortunately, Theorem B.1 provides a way out. That is, under the strong RSA assumption, we will have that $(c' - c)|(s - s')$. Let u be such that $(c' - c)u = (s - s')$. Then $y \equiv bg^u \pmod{n}$ for some b such that $b^{c'-c} \equiv 1 \pmod{n}$. Assuming that $2^{\ell}_c < \min(p', q')$, it must be that $b = \pm 1$ or $\gcd(b \pm 1, n)$ splits n (which again would counter the strong RSA assumption). Of course, if both y and g are quadratic residues, then $y \equiv g^u \pmod{n}$.

The reader might now think that the protocol is therefore a proof of knowledge under the strong RSA assumption for $\ell_c > 1$. Unfortunately this is not the case. Let us expand. The definition of a proof of knowledge (cf. Chapter **??**), requires that the inputs n, g, and y be fixed, i.e., that the knowledge extractor works for any prover that is successful for a given input n, g, and y. However, the above argument only works for n chosen at random. For instance, it does not work if the prover knew the factorization of n as he then could compute c, c', s, and s' such that $(c' - c) \nmid (s - s')$ (for instance by adding a multiple of the order of g to s). Now, if n is fixed, there always exists a prover who has the factorization encoded into herself and thus she could successfully run the protocol but the knowledge extractor cannot extract a witness. In order words, the protocol only has the proof of knowledge property for random n which contradicts the requirement of a proof of knowledge that the witness can be extracted for any given n. Nevertheless, the protocol is still useful as a building block, i.e., one only need to take into account the probability spaces of n, g, and y. That is, one has to consider the overall system of which the protocol is part and cannot just consider the protocol by itself as one could if it was a true proof of knowledge. Despite of all of this, we denote this protocol also as $PK\{(\alpha) : y \equiv \pm g^\alpha \pmod{n}\}$ or $PK\{(\alpha) : y \equiv g^\alpha \pmod{n}\}$, depending on whether the verifier is assured that y is a quadratic residue or not.

C.6 Proving that a Secret Lies in a Given Interval

One property of the protocol just described that is handy in many cases is the fact that the prover cannot reduce the response messages modulo the order of the group is to argue about the bit-length of the secret known to the prover. Let $x \in \pm\{0, 1\}^{\ell_x}$ for some ℓ_x and let $y := g^x \bmod n$, with g and n as before. Now consider the following modification of the protocol (the inputs to the prover and the verifier remain unchanged except that both parties are further given ℓ_x).

1. The prover chooses uniform at random $r_\alpha \in_R \{0, 1\}^{\ell_x + \ell_c + \ell_\varnothing}$, computes $t_y := g^r \bmod n$, and sends t to the verifier.

2. The verifier chooses a random $c \in_R \{0, 1\}^{\ell_c}$ and sends that to the prover.

3. The prover replies with $s_\alpha := r_\alpha - xc$.

4. The verifier checks whether

$$t_y \stackrel{?}{\equiv} y^c g^{s_\alpha} \pmod{n} \quad \text{and} \quad s_\alpha \stackrel{?}{\in} \pm\{0, 1\}^{\ell_x + \ell_c + \ell_\varnothing + 1}$$

   hold.

The modification is that the prover chooses r_α from a different interval and that the verifier also checks that s_α takes at most $\ell_x + \ell_c + \ell_\varnothing + 1$ bits.

The analysis of this protocols is of course almost identical to the original one, except that we are now considering the bit-lengths of s_α and r_α. First, it is not difficult to see that the protocol remains statistical honest-verifier zero-knowledge. Above we have argued that under the strong RSA assumption, one can extract a value u from a successful prover such that $y \equiv \pm g^u \pmod{n}$, where with $u = (s - s')/(c' - c)$. Now because $(c' - c)$ divides $(s - s')$ and as $(s - s') \in \pm\{0, 1\}^{\ell_x + \ell_c + \ell_\varnothing + 2}$, we must have $u \in \pm\{0, 1\}^{\ell_x + \ell_c + \ell_\varnothing + 2}$. In other words, the discrete logarithm known to the prover has at most $\ell_x + \ell_c + \ell_\varnothing + 2$

bits (neglecting its sign). Note that this length is independent of the length of the modulus n. Also note that in fact the length of the prover's secret must be shorter, (only about ℓ_x bits) for the prover to be able to run the protocol successfully with high probability; so the protocol is not an argument of the exact length of the secret. However, in many cases this is good enough. We denote this modified protocol as $PK\{(\alpha) : y \equiv g^\alpha \pmod n \quad \wedge \quad \alpha \in \pm\{0,1\}^{\ell_x+\ell_c+\ell_\varnothing+2}\}$.

The protocol can be also be used to prove that the secret known to the prover lies in any interval, say, in $[a, b]$. To this end note that $[a, b] = [\frac{a+b}{2} - \frac{b-a}{2}, \frac{a+b}{2} + \frac{b-2}{2}]$. Thus the protocol denoted

$$PK\{(\alpha) : \frac{y}{g^{(a+b)/2}} \equiv g^\alpha \pmod n \quad \wedge \quad \alpha \in [-\frac{b-a}{2}, \frac{b-a}{2}]\}$$

is an argument that the prover knows a value x such that $x \in [a, b]$ and $y \equiv g^x \pmod n$. As before, the actual value x given as input to the prover must lie in a smaller interval, namely in the interval $[\frac{a+b}{2} - \frac{b-a}{2\cdot 2^{\ell_c+\ell_\varnothing+2}}, \frac{a+b}{2} + \frac{a+b}{2\cdot 2^{\ell_c+\ell_\varnothing+2}}]$.

It is straightforward to extent and generalize the protocols just discussed in the same way as the protocols we considered for groups of known order. Moreover, the protocols over groups of known order and those over groups of unknown order can be combined. Let us consider a simple combination as an example; the constructions of general combinations is left as an exercise to the reader.

Assume a group $G = \langle g \rangle$ of order a prime q, and let n be an RSA modulus as above, h_1 be an element from $\mathbb{Z}_n^*$, h_2 be an element from $\langle h_1 \rangle$, and $y = g^x$ with $x \in \{0,1\}^{\ell_x}$ for some integer $\ell_x < (\log_2 q) - 1 - (\ell_c + \ell_\varnothing + 2)$. Finally, assume that the prover is not privy to n's factorization. Now consider the following protocol. The common input to the prover and the verifier consists of q, g, y, n, h_1, h_2, and a and the secret input to the prover consists of x.

1. First, the prover chooses a random $r \in_R [0, n2^{\ell_\varnothing}]$, computes $z := h_1^x h_2^r \bmod n$, and sends z to the verifier.
2. Next, the prover and the verifier run the protocol denoted:

$$PK\{(\alpha, \beta) : y = g^\alpha \quad \wedge \quad z \equiv h_1^\alpha h_2^\beta \pmod n \quad \wedge \quad \alpha \in \pm\{0,1\}^{\ell_x+\ell_c+\ell_\varnothing+2}\},$$

i.e., they perform the following steps.

   (a) The prover chooses a random $r_\alpha \in [-a2^{\ell_c+\ell_\varnothing}, a2^{\ell_c+\ell_\varnothing}]$ and a random $r_\beta \in \{0,1\}^{\ell_n+\ell_c+2\ell_\varnothing}]$, computes $t_y := g^{r_\alpha}$ and $t_z := h_1^{r_\alpha} h_2^{r_\beta} \bmod n$ and sends t_y and t_z to the verifier.

   (b) The verifier chooses a random $c \in_R \{0,1\}^{\ell_c}$ and sends that to the prover.

   (c) The prover replies with $s_\alpha := r_\alpha - xc$ and $s_\beta := r_\beta - rc$.

   (d) The verifier checks whether

$$t_y \stackrel{?}{=} y^c g^{s_\alpha} \quad , \qquad t_z \stackrel{?}{\equiv} z^c h_1^{s_\alpha} h_2^{s_\beta} \pmod n \quad , \text{ and}$$
$$s_\alpha \stackrel{?}{\in} \pm\{0,1\}^{\ell_x+\ell_c+\ell_\varnothing+1}$$

   hold.

Let us analyze this protocol. From our considerations above we know that one can extract from a successful prover values $(t_y, t_z, c, s_\alpha, s_\beta)$ and $(t'_y, t'_z, c', s'_\alpha, s'_\beta)$ with $(t_y, t_z) = (t'_y, t'_z)$ and $c \neq c'$. From the verification equations we have that

$$z^{c-c'} \equiv h_1^{s'_\alpha - s_\alpha} h_2^{s'_\beta - s_\beta} \pmod n \ , \qquad y^{c-c'} = g^{s'_\alpha - s_\alpha} \ , \quad \text{and}$$
$$(s'_\alpha - s_\alpha) \in \pm\{0,1\}^{\ell_x+\ell_c+\ell_\varnothing+2} \quad .$$

From the first of these equations we can conclude that under the Strong RSA assumption $(c - c')$ divides $(s'_\alpha - s_\alpha)$, i.e., there exists some u such that $(s'_\alpha - s_\alpha) = u(c - c')$. Thus, we can rewrite the second equation as $y^{c-c'} = g^{u(c-c')}$. Now, as $y \in \langle g \rangle$ (which we can test as g has prime order q), we must have $y = g^u$. From the third of the equations we can further derive that $u \in \pm\{0,1\}^{\ell_x+\ell_c+\ell_\varnothing+2}$. Thus the verifier is assured that the prover knows $\log_g y$ which lies in $\pm\{0,1\}^{\ell_x+\ell_c+\ell_\varnothing+2}$, i.e., the protocol is a proof that a discrete logarithm in a group of *known order* lies in some interval. Of course this makes sense only if the group's order is larger than $2^{\ell_x+\ell_c+\ell_\varnothing+3}$.

As the basic protocol $PK\{(\alpha) : y \equiv \pm g^\alpha \pmod{n}\}$ in group of unknown order, the protocol just described in not a proof of knowledge either as its analysis depends on the strong RSA assumption and thus is correct only if the prover can execute the protocol for a random n. As the goal of the protocol was to prove that $\log_g y$ lies in some interval, we do not need to fix n but could have the verifier generate n and send it to the prover before running the protocol. The protocol augmented like this would indeed be a true proof of knowledge. For the protocol to be zero-knowledge, the prover needs to be ensured that $h_2 \in \langle h_1 \rangle$ as otherwise z could leak information about x. Unfortunately, the only way known to prove that $h_2 \in \langle h_1 \rangle$ holds is not very efficient, i.e., is to have the verifier run the protocol $PK\{(\alpha) : h_2 \equiv \pm h_1^\alpha \pmod{n}\}$ many times with binary challenges (cf. above). However, in some cases this proof can be delegated to a set-up phase and thus needs to be done only once and for all, or sometimes n, h_1, and h_2 can be provided by a trusted third party.

The protocol just discussed can be extended to three (or more) different groups, i.e., groups $G_1 = \langle g_1 \rangle$ and $G_2 = \langle g_2 \rangle$ of known order q_1 and q_2 together with a RSA sub-group of unknown order as to show that two discrete logarithms in G_1 and G_2 are the same. I.e., let $y_1 = g_1^x$ and $y_2 = g_2^x$ with $x \in \pm\{0,1\}^{\ell_x}$ and $\ell_x < (\log_2 \min\{q_1, q_2\}) - 1 - (\ell_c + \ell_\varnothing + 2)$, and $z = h_1^x h_2^r \pmod{n}$ for some random r. Then, the protocol

$$PK\{(\alpha,\beta) : y_1 = g_1^\alpha \quad \wedge \quad y_2 = g_2^\alpha \quad \wedge z \equiv h_1^\alpha h_2^\beta \pmod{n} \quad \wedge \quad \alpha \in \pm\{0,1\}^{\ell_x+\ell_c+\ell_\varnothing+2}\} \ ,$$

will convince a verifier that $\log_{g_1} y_1 = \log_{g_2} y_1$ where we define $\log_g y$ to be the integer x for which $y = g^x$ and that closed to 0, i.e., $\log_g y$ lies in $[-q/2, q/2]$. Of course, the protocol can also be generalized to representations, i.e., to a protocol showing that an element of a representation lies in a certain interval.

D Example Specifications in XML

We provide some examples of elements specified in XML. In addition, the test cases use simple specifications that can be used as a reference when creating your own XML specifications.

D.1 Credential Structure

The following credential structure follows the one proposed in Fig. 2. In addition it shows the flexibility of the prime encodings by having several attributes encoded into one prime encoding.

```
<?xml version="1.0" encoding="UTF-8"?>
<CredentialStructure
xmlns="http://www.zurich.ibm.com/security/idemix"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xsi:schemaLocation="http://www.zurich.ibm.com/security/idemix
                                file:CredentialStructure.xsd">

<References>
  <IssuerPublicKey>
    http://www.ch.ch/passport/v2010/ipk.xml
  </IssuerPublicKey>
</References>

<Attributes>
  <Attribute name="FirstName" issuanceMode="known" type="string"/>
  <Attribute name="LastName" issuanceMode="known" type="string"/>
  <Attribute name="CivilStatus" issuanceMode="known" type="enum">
    <EnumValue name="Marriage"/>
    <EnumValue name="NeverMarried"/>
    <EnumValue name="Widowed" />
    <EnumValue name="LegallySeparated" />
    <EnumValue name="AnnulledMarriage" />
```

```
    <EnumValue name="Divorced" />
    <EnumValue name="Common-lawPartner" />
  </Attribute>
  <Attribute name="Sex" issuanceMode="known" type="enum">
    <EnumValue name="Male" />
    <EnumValue name="Female" />
  </Attribute>
  <Attribute name="OfficialLanguage" issuanceMode="known" type="enum">
    <EnumValue name="German" />
    <EnumValue name="French" />
    <EnumValue name="Italian"/>
    <EnumValue name="Rhaeto-Romanic" />
  </Attribute>
  <Attribute name="SocialSecurityNumber" issuanceMode="known" type="int"/>
  <Attribute name="BirthDate" issuanceMode="known" type="dateTime"/>
  <Attribute name="Epoch" issuanceMode="known" type="int"/>
  <Attribute name="Diet" issuanceMode="committed" type="string"/>
</Attributes>

<Features>
  <Epoch location="http://www.ch.ch/epoch file:CalculationMethod.xml"/>
  <Domain location="http://www.ch.ch "/>
</Features>

<Implementation>
  <PrimeEncoding name="PrimeEncoding1">
    <PrimeFactor attributeName="CivilStatus" enumValue="Marriage">
      3
    </PrimeFactor>
    <PrimeFactor attributeName="CivilStatus" enumValue="NeverMarried">
      5
    </PrimeFactor>
    <PrimeFactor attributeName="CivilStatus" enumValue="Widowed">
      7
    </PrimeFactor>
    <PrimeFactor attributeName="CivilStatus" enumValue="LegallySeparated">
      11
    </PrimeFactor>
    <PrimeFactor attributeName="CivilStatus" enumValue="AnnulledMarriage">
      13
    </PrimeFactor>
    <PrimeFactor attributeName="CivilStatus" enumValue="Divorced">
      17
    </PrimeFactor>
    <PrimeFactor attributeName="CivilStatus" enumValue="Common-lawPartner">
      19
    </PrimeFactor>
    <PrimeFactor attributeName="Sex" enumValue="Male">
      23
    </PrimeFactor>
    <PrimeFactor attributeName="Sex" enumValue="Female">
      29
    </PrimeFactor>
  </PrimeEncoding>
  <PrimeEncoding name="PrimeEncoding2">
    <PrimeFactor attributeName="OfficialLanguage" enumValue="German">
      2
    </PrimeFactor>
    <PrimeFactor attributeName="OfficialLanguage" enumValue="French">
      3
    </PrimeFactor>
```

```
    <PrimeFactor attributeName="OfficialLanguage" enumValue="Italian">
      5
    </PrimeFactor>
    <PrimeFactor attributeName="OfficialLanguage" enumValue="Rhaeto-Romanic">
      7
    </PrimeFactor>
  </PrimeEncoding>
  <AttributeOrder>
    <Attribute name="FirstName">1</Attribute>
    <Attribute name="LastName">2</Attribute>
    <Attribute name="primeEncoding1">3</Attribute>
    <Attribute name="primeEncoding2">4</Attribute>
    <Attribute name="SocialSecurityNumber">5</Attribute>
    <Attribute name="BirthDate">6</Attribute>
    <Attribute name="Epoch">7</Attribute>
    <Attribute name="Diet">8</Attribute>
  </AttributeOrder>
</Implementation>

</CredentialStructure>
```


D.2 Proof Specification


```
<?xml version="1.0" encoding="UTF-8"?>
<CredentialStructure xmlns="http://www.zurich.ibm.com/security/idemix"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xsi:schemaLocation="http://www.zurich.ibm.com/security/idemix CertificateStructure.xsd">

<References>
<IssuerPublicKey>
tests\com\ibm\zrl\idmx\tests\ipk.xml
</IssuerPublicKey>
</References>

<Attributes>
<Attribute issuanceMode="known" name="FirstName" type="string" />
<Attribute issuanceMode="known" name="LastName" type="string" />
<Attribute issuanceMode="known" name="CivilStatus" type="enum">
<EnumValue>Marriage</EnumValue>
<EnumValue>NeverMarried</EnumValue>
<EnumValue>Widowed</EnumValue>
<EnumValue>LegallySeparated</EnumValue>
<EnumValue>AnnulledMarriage</EnumValue>
<EnumValue>Divorced</EnumValue>
<EnumValue>Common-lawPartner</EnumValue>
</Attribute>
<Attribute issuanceMode="known" name="SocialSecurityNumber"
type="int" />
<Attribute issuanceMode="known" name="BirthDate" type="date" />
<Attribute issuanceMode="committed" name="Diet" type="string" />
<Attribute issuanceMode="known" name="Epoch" type="epoch" />
<Attribute issuanceMode="known" name="OfficialLanguage"
type="enum">
<EnumValue>German</EnumValue>
<EnumValue>French</EnumValue>
<EnumValue>Italian</EnumValue>
<EnumValue>Rhaeto-Romanic</EnumValue>
</Attribute>
<Attribute issuanceMode="committed" name="DriverCategory"
type="enum">
```

```
<EnumValue>A1</EnumValue>
<EnumValue>B</EnumValue>
<EnumValue>B1</EnumValue>
<EnumValue>C</EnumValue>
<EnumValue>C1</EnumValue>
<EnumValue>D</EnumValue>
<EnumValue>D1</EnumValue>
<EnumValue>BE</EnumValue>
<EnumValue>CE</EnumValue>
<EnumValue>DE</EnumValue>
<EnumValue>C1E</EnumValue>
<EnumValue>D1E</EnumValue>
<EnumValue>F</EnumValue>
<EnumValue>G</EnumValue>
<EnumValue>M</EnumValue>
</Attribute>
<Attribute issuanceMode="known" name="Gender" type="enum">
<EnumValue>Male</EnumValue>
<EnumValue>Female</EnumValue>
</Attribute>
</Attributes>

<Features>
<Updates>http://www.ibm.com/employee/updates.xml</Updates>
</Features>

<Implementation>
<PrimeEncoding name="PrimeEncoding1">
<PrimeFactor attName="CivilStatus" attValue="Marriage">3
</PrimeFactor>
<PrimeFactor attName="CivilStatus" attValue="NeverMarried">5
</PrimeFactor>
<PrimeFactor attName="CivilStatus" attValue="Widowed">7
</PrimeFactor>
<PrimeFactor attName="CivilStatus" attValue="LegallySeparated">11
</PrimeFactor>
<PrimeFactor attName="CivilStatus" attValue="AnnulledMarriage">13
</PrimeFactor>
<PrimeFactor attName="CivilStatus" attValue="Divorced">17
</PrimeFactor>
<PrimeFactor attName="CivilStatus" attValue="Common-lawPartner">19
</PrimeFactor>
<PrimeFactor attName="OfficialLanguage" attValue="German">23
</PrimeFactor>
<PrimeFactor attName="OfficialLanguage" attValue="French">29
</PrimeFactor>
<PrimeFactor attName="OfficialLanguage" attValue="Italian">31
</PrimeFactor>
<PrimeFactor attName="OfficialLanguage" attValue="Rhaeto-Romanic">37
</PrimeFactor>
<PrimeFactor attName="Gender" attValue="Male">41
</PrimeFactor>
<PrimeFactor attName="Gender" attValue="Female">43
</PrimeFactor>
</PrimeEncoding>
<PrimeEncoding name="PrimeEncoding2">
<PrimeFactor attName="DriverCategory" attValue="A1">3
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="B">5
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="B1">7
```

```
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="C">11
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="C1">13
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="D">17
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="D1">19
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="BE">23
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="CE">29
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="DE">31
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="C1E">37
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="D1E">41
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="F">43
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="G">47
</PrimeFactor>
<PrimeFactor attName="DriverCategory" attValue="M">53
</PrimeFactor>
</PrimeEncoding>
<AttributeOrder>
<Attribute name="FirstName">1</Attribute>
<Attribute name="LastName">2</Attribute>
<Attribute name="PrimeEncoding1">3</Attribute>
<Attribute name="SocialSecurityNumber">4</Attribute>
<Attribute name="BirthDate">5</Attribute>
<Attribute name="Diet">6</Attribute>
<Attribute name="Epoch">7</Attribute>
<Attribute name="PrimeEncoding2">8</Attribute>
</AttributeOrder>
</Implementation>
</CredentialStructure>
```