Research Report

Cryptographic Protocols of the Identity Mixer Library

Contributors:
Patrik Bichsel, Carl Binding, Jan Camenisch, Thomas Groß, Tom Heydt-Benjamin, Dieter Sommer, and Greg Zaverucha

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

LIMITED DISTRIBUTION NOTICE

This report will be distributed outside of IBM up to one year after the IBM publication date. Some reports are available at http://domino.watson.ibm.com/library/Cyberdig.nsf/home.
Cryptographic Protocols of the Identity Mixer
Library

Contents

1 About this document 2
  1.1 Contributors ........................................... 2
  1.2 Naming Conventions .................................... 3

2 Introduction 3
  2.1 Private Certificate Systems .......................... 4
  2.2 Building Anonymous Credential Systems ............. 5
  2.3 Certificates on a Master Secret ....................... 5

3 Preliminaries 6
  3.1 Notation and System Parameters ..................... 6
  3.2 Attributes and Commitments .......................... 6
  3.3 Architecture Overview ............................... 9
  3.4 Specifications for Issuance and Proofs ............... 9

4 Setup and Issuer Key Generation 10
  4.1 System Setup .......................................... 10
  4.2 Issuer Key Generation ................................ 10
  4.3 Proof for Issuer Key Generation ..................... 11
    4.3.1 Protocol: pkCorrectnessProof .................... 11
    4.3.2 Protocol: pkCorrectnessVerify ................... 12
  4.4 Master Secret Generation ............................ 12

5 Details of Issuance and Proof Specifications 12
  5.1 High-level Specification $S^*$ ......................... 12
  5.2 Low-level Specification $S$ .......................... 13
## 1 About this document

The goal of this document is to detail the cryptographic protocols of the IBM idemix library. The idemix (identity mixer) library serves as the cryptographic component in a larger anonymous credential system.

Wherever possible, names and/or numbers are given to protocols, constraints, etc. appearing in this document. This is to allow the source code to reference a particular protocol or part of a protocol. For this reason, much of the numbering is done by hand (instead of using \LaTeX’s numbering) so that changes to this document do not cause renumbering, invalidating the references in the source.

The source code may refer to automatically numbered objects by using the name given to \LaTeX’s \texttt{\LaTeX} command.

### 1.1 Contributors

Jan Camenisch (\texttt{jca}) and Dieter Sommer (\texttt{dso}) wrote the initial draft of this document. Greg Zaverucha (\texttt{gza}) adapted it, adding sufficient detail and clarifications to specify idemix v. 1.0.

Patrik Bichsel (\texttt{pbi}), Carl Binding (\texttt{cbd}), Jan Camenisch (\texttt{jca}), Thomas Gross (\texttt{tgr}), Tom Heydt-Benjamin (\texttt{hey}), and Dieter Sommer (\texttt{dso}) provided help and feedback on earlier drafts of this document.

---

### 6 Certificate Issuance

- 6.1 Issuance at the Cryptographic Layer

- 6.2 Issuance at the Credsystem Layer
  - 6.2.1 Issuance Protocol

- 6.3 Library API

### 7 Proving Holdership of a Certificate

- 7.1 Proving Holdership at the Cryptographic Layer
  - 7.1.1 Protocol: \texttt{ProveProtocol}
  - 7.1.2 Protocol: \texttt{ProveCL}
  - 7.1.3 Protocol: \texttt{ProveCommitment}
  - 7.1.4 Protocol: \texttt{ProveRange}

- 7.2 Verifying Holdership at the Cryptographic Layer
  - 7.2.1 Protocol: \texttt{VerifyProtocol}

- 7.3 Proving Holdership at the Credsystem Layer
  - 7.3.1 Proof Protocol

- 7.4 Library API

### 8 Efficiency and Advanced Features

- 8.1 Implementation Considerations
- 8.2 Efficiency
- 8.3 Advanced and Planned Features
1.2 Naming Conventions

The implementation follows the protocols presented in this document closely. To facilitate reading both the documentation and the code, the following variable naming conventions have been adopted.

New code which requires a formal description (such as new cryptographic protocols) should follow these rules when naming variables.

The convention is described in Table 1 by example, but here are some general rules.

1. Capitals are prefixed with “cap” when the Java naming convention would be violated.
2. Subscripts are denoted by an underscore, “_”.
3. Use the latex names for Greek letters.
4. In cases when a variable requires multiple prefixes, use the following order tilde, hat, bold, cap. For example, $T$ should be named hatCapT, not capHatT.
5. List variables may optionally include the the suffix Tup to indicate that the variable is a tuple. For example, if $R$ is a list, it may be named capRTup.
6. If you encounter a case not covered by this guide, update the team’s decision here.

2 Introduction

In cryptography, a certificate is a list of information about the certificate holder which is signed by a trusted authority. The process of obtaining a certificate is called issuing, therefore the authority is referred to as the issuer. The recipient of the certificate in this transaction is called the user or the receiver.

After obtaining a certificate, the user may then present the certificate to another party, which we call the verifier. The verifier checks that the signature on the information is valid. If so, the verifier is then assured that it is accurate, since the issuer is trusted. The most common use of certificates is with public key cryptography, where a certificate is a signed copy of an identity and a public key. This binds the public key to the identity, assuring the encrypting party of the decrypting party’s identity.

Private certificates are a special kind of certificate, issued to a user and then kept secret. A private certificate also contains a list of attributes. The interesting feature of private certificates is that the certificate holder can assert parts of the attribute information to a verifier. Further, this is done in such a way that the certificate itself is kept private. For example, the user is able to convince the verifier that the value of the age attribute is greater than 18, without revealing the actual value, or any other information contained in the...
2.1 Private Certificate Systems

Table 1: Examples describing the naming convention for mathematical variable names in the idemix library. This describes how the variable in the \texttt{\LaTeX} column should be name in the source code.

<table>
<thead>
<tr>
<th>\texttt{\LaTeX}</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>a</td>
</tr>
<tr>
<td>$a_1$</td>
<td>a_1</td>
</tr>
<tr>
<td>$T$</td>
<td>capT</td>
</tr>
<tr>
<td>$R_i$</td>
<td>capR_i</td>
</tr>
<tr>
<td>$b_{4_2}$</td>
<td>b.4_2</td>
</tr>
<tr>
<td>$t_r$</td>
<td>t_r_i</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>gamma</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>capGamma</td>
</tr>
<tr>
<td>$\ell_k$</td>
<td>l_k</td>
</tr>
<tr>
<td>$\ell_{\rho}$</td>
<td>l_rho</td>
</tr>
<tr>
<td>$\ell_H$</td>
<td>l_H</td>
</tr>
<tr>
<td>$\hat{T}$</td>
<td>hatT</td>
</tr>
<tr>
<td>$\hat{T}$</td>
<td>hatCapT</td>
</tr>
<tr>
<td>$A_{CO}$</td>
<td>capA_CO</td>
</tr>
<tr>
<td>$S_v'$</td>
<td>S_vPrime</td>
</tr>
<tr>
<td>$v''$</td>
<td>vPrimePrime</td>
</tr>
<tr>
<td>$\tilde{X}$</td>
<td>tildeCapX</td>
</tr>
<tr>
<td>$T$</td>
<td>boldCapT</td>
</tr>
<tr>
<td>$\hat{T}_2$</td>
<td>hatCapT_Delta</td>
</tr>
<tr>
<td>$S^*$</td>
<td>capSStar</td>
</tr>
</tbody>
</table>

certificate. This is the key difference between private certificates and ordinary (public) certificates.

In this operation the user plays the role of the \textit{prover}; they will \textit{prove} or assert something to the verifier. In an anonymous credential system, this operation is called a \textit{show} operation, since the user is showing a credential. It is also called a \textit{proof} operation, as the user is proving some fact to the verifier. A traditional credential is written evidence of authority, status, rights or entitlement to a product or service. Loosely speaking, a \textit{private credential system} is a collection of protocols which allow credentials to be issued, shown and verified in a privacy-preserving way. The amount of information divulged is typically limited to the minimum required in a transaction.

Anonymous credential systems can be obtained by using \textit{private certificates} such as ones based on the Camenisch and Lysyanskaya signature protocols [3].

2.1 Private Certificate Systems

A \textit{private certificate system} is defined by a set of protocols to operate on private certificates. A basic system includes the protocols \textit{issue} and \textit{prove}. Additional
protocols that found in extended systems include protocols for revoking certificates, revoking anonymity under well-defined circumstances or limiting the number of times a user can execute the prove protocol. A private certificate system can be obtained from primitives providing two basic functions:

1. Protocols for issuing signatures on cryptographically committed values (the signer does not learn the message), and
2. the ability to prove knowledge of certificates without revealing them.

We provide detailed protocol specifications for a private certificate system based on the signature scheme of Camenisch and Lysyanskaya (CL signatures) [3]. There are two variants of CL signatures, one based on bilinear maps, the other on the strong RSA assumption (SRSA). The implementation described herein uses only the SRSA version, and any reference to CL signatures is the SRSA version. For a description of both signature schemes, see Bangerter et al. [5].

2.2 Building Anonymous Credential Systems

Private certificate systems provide a foundation for building anonymous credential systems. An anonymous credential system offers certain features that go beyond that of a private certificate system. This includes, but is not limited to consistency of credentials, binding credentials to a user, and supporting the concept of pseudonyms. See, for example, Lysyanskaya et al. [7] for a definition of anonymous credential systems.

2.3 Certificates on a Master Secret

The idemix credential system issues private certificates on a master secret of users. To achieve such a binding of a private certificate to a master secret, the first attribute, $m_1$, is dedicated to containing the user’s master secret. The certificate gets issued on the key without the issuer learning the secret. To perform a proof of knowledge using the certificate, the user must prove knowledge of the master secret.

Both the issuance and proof of knowledge of a certificate use CL signatures to issue certificates on committed values, the master secret of the user, and create proofs of knowledge of the signatures. In those protocols the master secret acts as the identity of the user and helps bind certificates to this identity. Using general zero-knowledge proof techniques, this could be realized with any signature scheme, but such a construction would not be practically efficient (see Yao [11], for a general zero-knowledge methods).

In cases when the user may possess multiple certificates, subsequent issuers may require that the master secret in the new certificate match the master secret from a previous certificate. For example, the department of motor vehicles could require that the master secret be the same as the one in the government issued...
birth certificate credential. During issuance, the user would prove possession of a birth certificate credential, and equality of the master secret.

To prevent misuse of the system by the user, the user is discouraged from sharing their master secret. This can be done either by making sharing technically difficult, e.g., by encapsulating it inside a secure computing device or by binding an external valuable secret of the user to the key. Then this external secret would be shared as well when sharing the master secret.

3 Preliminaries

We give some preliminaries necessary for the presentation of the protocols. This involves both notational conventions as well as the definition of commonly-used parameters.

3.1 Notation and System Parameters

Let \( H : \{0,1\}^* \rightarrow \{0,1\}^{\ell_H} \) be a cryptographic hash function. The current implementation uses SHA-256 [8]. Let “||” denote the operator for concatenation of numbers or strings. For an integer or a data field \( x \), \( \ell_x \) is used to denote the maximum number of bits allowed to represent \( x \). 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 \( S \).

When presenting protocols, we use the notation of Camenisch and Stadler [4] to specify zero-knowledge 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.

Table 2 lists the notation used in this document. Wherever possible, variables in the source code should use the same name. Conventions for naming variables used from this document in the source code are given in Section 1.2.

Table 3 lists recommended sizes for system parameters, and lists the constraints between parameters. The use of these parameters will be explained below.

Note that attributes can be from the interval \([-2^{\ell_m} + 1, 2^{\ell_m} - 1]\) and the attribute holding the user’s master secret is of size \( \ell'_m \).

3.2 Attributes and Commitments

We now take a moment to describe the notation used to discuss attributes, which are also seen as messages to be signed in the CL signature scheme. The message space is

\[
\mathcal{M} = \{0,1\}^{\ell_m} \times \left(\{0,1\}^{\ell_m}\right)^{l-1},
\]

which corresponds to the first attribute being the user’s master secret, and the remaining \( l - 1 \) attributes being arbitrary strings of length \( \ell_m \).
### 3.2 Attributes and Commitments

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>RSA modulus for CL signatures</td>
<td>p. 10</td>
</tr>
<tr>
<td>$p, q$</td>
<td>prime factors of $n$</td>
<td>p. 10</td>
</tr>
<tr>
<td>$QR_n$</td>
<td>group of quadratic residues mod $n$</td>
<td>p. 10</td>
</tr>
<tr>
<td>$pk_B, sk_B$</td>
<td>public key, secret key of entity $B$</td>
<td>p. 11</td>
</tr>
<tr>
<td>$S, Z, R_i$</td>
<td>part of the issuer’s public key (for CL sigs.)</td>
<td>p. 10</td>
</tr>
<tr>
<td>$\Gamma$</td>
<td>modulus of the commitment group</td>
<td>p. 10</td>
</tr>
<tr>
<td>$\rho$</td>
<td>prime order of a large subgroup of $\mathbb{Z}_\Gamma^*$</td>
<td>p. 10</td>
</tr>
<tr>
<td>$b$</td>
<td>cofactor of $\Gamma - 1$</td>
<td>p. 10</td>
</tr>
<tr>
<td>$g, h$</td>
<td>generators of the order $\rho$ subgroup of $\mathbb{Z}_\Gamma^*$</td>
<td>p. 10</td>
</tr>
<tr>
<td>$l$</td>
<td>total number of attributes in a certificate (bases in the issuer’s CL public key)</td>
<td>p. 10</td>
</tr>
<tr>
<td>$m_1$</td>
<td>the master secret</td>
<td>p. 5, 9, 12</td>
</tr>
<tr>
<td>$A$</td>
<td>ordered set of attributes</td>
<td>p. 9</td>
</tr>
<tr>
<td>$(m_1, \ldots, m_l)$</td>
<td>attributes in $A$</td>
<td>p. 9</td>
</tr>
<tr>
<td>$A_{\text{known}}$ or $A_k$</td>
<td>indices of attributes which are public during a certificate issue</td>
<td>p. 9</td>
</tr>
<tr>
<td>$A_{\text{hidden}}$ or $A_h$</td>
<td>indices of attributes which are hidden/committed during an issue</td>
<td>p. 9</td>
</tr>
<tr>
<td>$A_{\text{revealed}}$ or $A_r$</td>
<td>indices of attributes revealed during a proof</td>
<td>p. 9</td>
</tr>
<tr>
<td>$c_i$</td>
<td>commitment of attribute $i$, i.e. $\text{commit}(m_i)$</td>
<td>p. 9</td>
</tr>
</tbody>
</table>

Table 2: 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.)
### Attributes and Commitments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Bitlength</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell_n$</td>
<td>size of RSA modulus</td>
<td>2048</td>
</tr>
<tr>
<td>$\ell_\Gamma$</td>
<td>size of the commitment group modulus</td>
<td>1632</td>
</tr>
<tr>
<td>$\ell_\rho$</td>
<td>size of the prime order subgroup of $\Gamma$</td>
<td>256</td>
</tr>
<tr>
<td>$\ell_m$</td>
<td>size of attributes</td>
<td>256</td>
</tr>
<tr>
<td>$\ell_m'$</td>
<td>size of first attribute, the master secret</td>
<td>$\ell_\rho$</td>
</tr>
<tr>
<td>$\ell_{res}$</td>
<td>number reserved attributes in a certificate</td>
<td>1</td>
</tr>
<tr>
<td>$\ell_e$</td>
<td>size of $e$ values of certificates</td>
<td>596</td>
</tr>
<tr>
<td>$\ell_e'$</td>
<td>size of the interval the $e$ values are taken from</td>
<td>120</td>
</tr>
<tr>
<td>$\ell_v$</td>
<td>size of the $v$ values of the certificates</td>
<td>2723</td>
</tr>
<tr>
<td>$\ell_o$</td>
<td>security parameter that governs the statistical zero-knowledge property (source name l_statzk)</td>
<td>80</td>
</tr>
<tr>
<td>$\ell_k$</td>
<td>security parameter</td>
<td>160</td>
</tr>
<tr>
<td>$\ell_H$</td>
<td>domain of the hash function $H$ used for the Fiat-Shamir heuristic</td>
<td>256</td>
</tr>
<tr>
<td>$\ell_r$</td>
<td>security parameter required in the proof of security of the credential system</td>
<td>80</td>
</tr>
<tr>
<td>$\ell_{pt}$</td>
<td>prime number generation returns composites with probability $1 - 1/2^{\ell_{pt}}$</td>
<td>80†</td>
</tr>
</tbody>
</table>

Table 3: 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 bitlength.)

<table>
<thead>
<tr>
<th>Number</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\ell_e &gt; \ell_o + \ell_H + \max{\ell_m + 4, \ell_e' + 2}$</td>
</tr>
<tr>
<td>2</td>
<td>$\ell_v &gt; \ell_n + \ell_o + \ell_H + \max{\ell_m + \ell_e + 3, \ell_o + 2}$</td>
</tr>
<tr>
<td>3</td>
<td>$\ell_H \geq \ell_k$</td>
</tr>
<tr>
<td>4</td>
<td>$\rho \mid b$</td>
</tr>
<tr>
<td>5</td>
<td>$\ell_H &lt; \ell_e$</td>
</tr>
<tr>
<td>6</td>
<td>$\ell_e' &lt; \ell_e - \ell_o - \ell_H - 3$</td>
</tr>
<tr>
<td>7</td>
<td>$\ell_m' = \ell_o$</td>
</tr>
</tbody>
</table>

Table 4: Constraints which parameter choices must satisfy to ensure security and soundness. (Source code may refer to this table as `table:constraints`.)
The ordered set of attributes for a user will be denoted $A = (m_1, \ldots, m_l)$, and the indices of $A$ can be partitioned $A = A_{\text{known}} \cup A_{\text{hidden}}$. $A_{\text{known}}$ are the attributes which are public, while $A_{\text{hidden}}$ are private attributes which the Issuer does not learn. During a proof, all the attributes which the Prover shares with the Verifier will form the set $A_{\text{revealed}}$, and attributes which remain private are those in $A_{\text{revealed}}$ (set complement with respect to $A$). The shorthand $A_{k}, A_{h}, A_{r}$ for $A_{\text{known}}, A_{\text{hidden}}$ and $A_{\text{revealed}}$ will occasionally be used for space efficiency. Recall that $m_1$ is the master secret, therefore, we always have $1 \in A_{h}$ and $1 \in A_{r}$. The sets of integer indices are ordered in the natural way. More detail on the different types of attributes will be given with the description of IssueCertificateProtocol in §6.1.

A Pedersen commitment of a value $m$ is computed $C = g^m h^r \mod \Gamma$ where $g, h$ are the generators of $\Gamma$ and $r \in_R [1, \rho]$. The commitment can be opened by revealing $m$ and $r$. The creator of $C$ may prove knowledge of $m$ using $PK \{(\alpha, \beta) : C = g^\alpha h^\beta \mod \Gamma\}$. The notation $\text{commit}(m_i) = c_i$ is used to denote the commitment of attribute $i$. We also use $\text{open}(c_i)$ to denote the opening information of commitment $c_i$. For a Pedersen commitment, $\text{open}(c_i) = (m, r)$. Unless otherwise specified, all commitments are Pedersen commitments as described here.

### 3.3 Architecture Overview

The system design splits all protocols into a cryptographic layer and a credential system layer. The former is also referred to as the low-level protocol layer or simply “protocols”, the latter as the Credsystem layer or high-level protocol layer. The low-level cryptographic protocols are used to construct a set of high-level credential system protocols with richer semantics in terms of credentials such as different attribute types.

Names of low level protocols will always end with Protocol.

### 3.4 Specifications for Issuance and Proofs

Before a certificate can be issued, or before a proof of holdership is possible, both parties must share the same description of what is to be issued or proven. This information is contained in the specification.

At the high-level specifications are denoted $S^*$. The high level specification for issuance determines the attribute values, the certificate structure, and the way attributes are issued. The structure comprises the ontology and data types, an attribute mapping function, the issuer and the certificate type. At the low level, a similar specification $S$ contains attribute values and the method each attribute should be issued.

Neither specification may make reference to the master secret $m_1$, only references to $m_2, \ldots, m_l$. Therefore, issuance and proof operations which use $m_1$ must be “hard-coded” and happen automatically, for each issuance or show.

The primary task of the higher level is to transform high-level specifications to low-level specifications. Once complete, the low-level protocol is used directly.
4 Setup and Issuer Key Generation

4.1 System Setup

In addition to all the system parameters given in Table 3, which must be fixed and made public, we must generate a group to use for commitments. This group must be set up initially and made known to all parties taking part in the system.

The group $\mathbb{Z}_\Gamma^*$ will be used, where the group order is $\Gamma - 1 = \rho \cdot b$ for some large prime $\rho$. This ensures that $\mathbb{Z}_\Gamma^*$ has a large subgroup of prime order $\rho$, and that discrete logarithms are hard to compute. The bitlengths of $\Gamma$ and $\rho$ are given by $\ell_{\Gamma}$ and $\ell_{\rho}$ respectively. Preferably, the cofactor $b$ is small, so constraint 4 (see Table 4) is trivial.

A generator $g$ of the group is computed by choosing a random $g' \in R \mathbb{Z}_\Gamma^*$ with $g'^b \not\equiv 1 \pmod{\Gamma}$. Then the generator $g = g'^b \pmod{\Gamma}$. A second generator, $h$, will also be needed, computed by taking a random power of $g$.

The group parameters $\Gamma, \rho, g$ and $h$ are provided to all parties as public parameters. A user verifies the system parameters by checking that $\rho$ and $\Gamma$ are prime, and that $\rho \mid (\Gamma - 1), \rho \nmid \Gamma - 1$, $g^\rho \equiv h^\rho \equiv 1 \pmod{\Gamma}$.

4.2 Issuer Key Generation

The issuer’s key pair is used for issuing certificates, that is, issuing signatures on lists of attributes. The total number $l$ of possible attributes of the credential is determined by the public key. The number of attributes available to users is $l - \ell_{\text{res}}$ since some attributes will be reserved.

The issuer generates 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 $\ell_n$ bits, $p$ and $q$ must have bitlength $\ell_n/2$.

Then the issuer generates parameters for the CL signature scheme. The issuer chooses

$$S \in R \mathbb{Q}R_n, \quad Z, R_1, \ldots, R_l \in_R \langle S \rangle$$

(where $\mathbb{Q}R_n$ is the group of quadratic residues mod $n$ and $\langle S \rangle$ is the subgroup generated by $S$). $S$ must also have order $\# \mathbb{Q}R_n = p'q'$. The second step is done by choosing $x, x_R, \ldots, x_R \in R [2, p'q' - 1]$ and computing $Z = S^{x_Z}, R_i = S^{x_{R_i}}$, for $1 \leq i \leq l$.

The issuer must also compute a non-interactive zero-knowledge proof of knowledge $P$, to prove that the public key was generated correctly. This proof will convince a verifier that $Z, R_i \in \langle S \rangle$ for $1 \leq i \leq l$. Details of the proof appear in §4.3.

Each user who interacts with this issuer (or verifier) should verify $P$ once on the key to be sure that the key was generated honestly. This involves verifying $P$ and checking that the public key parameters have the required lengths. It is also possible for the user to delegate verification of $P$ to a trusted party.
4.3 Proof for Issuer Key Generation

The issuer’s public key is \( pk_I = (n, S, Z, R_1, \ldots, R_l, P) \) and the private key is \( sk_I = (p, q) \).

4.3 Proof for Issuer Key Generation

**NOTE: this is currently not implemented**

The issuer must prove that his public key was generated correctly, with the following proof

\[
PK \{ (\alpha_Z, \alpha_1, \ldots, \alpha_l) : Z = S^{\alpha_Z}, R_1 = S^{\alpha_1}, \ldots, R_l = S^{\alpha_l} \}
\]

where all equalities are mod \( n \).

The parameter \( k \) is the number of single-bit challenge proofs required, where the prover can deceive with probability \( \frac{1}{2^k} \). In order to get random challenge bits to use the Fiat-Shamir heuristic, we will need to compute the first part of all \( k \) rounds in parallel. All this information together is hashed, and each bit of the digest is used as a challenge bit.

4.3.1 Protocol: pkCorrectnessProof

1. Define length \( k \) vectors \( t_Z, t_{R_1}, \ldots, t_{R_l} \).
2. Compute (mod \( n \))

   \[
   \begin{align*}
   t_Z(i) &= S^{r_{Z,i}}  \\
   t_{R_1}(i) &= S^{r_{R_1,i}}  \\
   &\vdots  \\
   t_{R_l}(i) &= S^{r_{R_l,i}}
   \end{align*}
   \]

   for \( i = 1, \ldots, k \), and \( r_{Z,i}, r_{R_1,i}, \ldots, r_{R_l,i} \in \mathbb{Z}_{n}^* \).
3. (Challenge) Compute

   \[
   c = H(S||Z||R_1||\ldots||R_l||t_Z||t_{R_1}||\ldots||t_{R_l})
   \]

4. (Responses) Define length \( k \) vectors \( s_Z, s_{R_1}, \ldots, s_{R_l} \).
5. Compute (mod \( p'q' \))

   \[
   \begin{align*}
   s_Z(i) &= r_{Z,i} - c_i \alpha_Z  \\
   s_{R_1}(i) &= r_{R_1,i} - c_i \alpha_{R_1}  \\
   &\vdots  \\
   s_{R_l}(i) &= r_{R_l,i} - c_i \alpha_{R_l}
   \end{align*}
   \]

   for \( i = 1, \ldots, k \), where \( c_i \) is the \( i \)-th bit of \( c \).
6. Output \( (c, s_Z, s_{R_1}, \ldots, s_{R_l}) \).
4.3.2 Protocol: pkCorrectnessVerify

The input \((c, s_Z, s_{R_1}, \ldots, s_{R_l})\) is verified as follows.

1. Define length \(k\) vectors \(\hat{t}_Z, \hat{t}_{R_1}, \ldots, \hat{t}_{R_l}\).

2. Compute (mod \(n\))
   \[
   \hat{t}_Z(i) = Z^{c_i} S_z^{s_Z(i)} \\
   \hat{t}_{R_1}(i) = R_1^{c_i} S_{R_1}^{s_{R_1}(i)} \\
   \vdots \\
   \hat{t}_{R_l}(i) = R_l^{c_i} S_{R_l}^{s_{R_l}(i)}
   \]
   for \(i = 1, \ldots, k\).

3. Compute
   \[
   \hat{c} = H(S||Z||R_1||\ldots||R_l||\hat{t}_Z||\hat{t}_{R_1}||\ldots||\hat{t}_{R_l}).
   \]

4. Accept if \(\hat{c} = c\), reject otherwise.

4.4 Master Secret Generation

The user’s master secret \(m_1\) is an integer chosen uniformly at random from the interval \([1, 2^{\ell'_m}]\). Depending on the issuance specification, \(m_1\) may be new or re-used from a previous certificate. The current version supports only new master secrets in an issue.

5 Details of Issuance and Proof Specifications

The structure of the specifications held by both parties in a issuance or prove protocol is the same. However, values which are private to one party will be omitted from the specification held by the other. For instance, the issuance specification of the Recipient will contain the values of attributes indexed by \(A_{hidden}\), but not the secret key of the Issuer. Likewise, the Issuer’s specification will contain the secret key but not the hidden attributes.

Since the issuance specification is a special case of the more general specification, a single description is sufficient.

5.1 High-level Specification \(S^*\)

Complete details of \(S^*\) are given in the ICL (idemix claims language) specification [2]. We give a brief description, focusing on the parts required for this document.

\[
S^* = (certs, statement, certificatespecifications)
\]

where
5.2 Low-level Specification $S$

- $certs$ are the certificates to be used in the proof ($certs$ is empty during issuance), as output by $\text{IssueCertificateProtocol}$,
- $statement$ is a list of $\land$ connected $predicates$, and
- $certificate\ specification$ (in 1:1 correspondence with $certs$) describe the certificate issuer, the certificate type and what attributes are in the certificate.

The $predicates$ are of the form $attribute \ rel \ constant$.

- $attribute$ contains the name, value, type, issuance mode, ontology of the attribute as well as a reference to certificate containing it.
- $rel$ is a relation between the $attribute$ and the $constant$. For issuance, this will always be equality, i.e. the $attribute$ must have the value specified by $constant$. During a proof, $rel$ may be one of $=, >, \geq, <, \leq$. (Note that inequality is currently not supported.)
- $constant$ is the literal value of a date, string or integer (see [2] for valid formats).

6 Certificate Issuance

We first present the issuing protocol for a private certificate. We begin with a presentation of the low-level protocol and then give the high-level protocol (which is built on top of the low-level protocol).
6.1 Issuance at the Cryptographic Layer

In this subsection we present IssueCertificateProtocol, the protocol used to issue a certificate. The participants are the Recipient of the certificate (user) and the an Issuer of the certificate. Throughout, the terms “User” and “Recipient” will be used synonymously.


The following specification defines the interface for establishing protocol endpoints that implement the Recipient and Issuer sides of the protocol.

Protocol parameters

$S$ is an issuing specification determining the attribute values and method of issuing each attribute. Both the Issuer and Recipient are given $S$ as input. $S$ consists of predicates, each specifying a single attribute. All predicates are of the form $eq(attribute, value, method)$ that specifies $attribute = value$ and is issued using $method$. The $method$ field specifies how the attribute is chosen. There are three possible methods.

KN means known by the Issuer and the Recipient, attributes in $A_{known}$ are issued by this method. For simplicity we say that these attribute values were chosen by the Issuer.

CO means the Recipient has committed to the value, which is kept secret from the Issuer. Let $A_{CO}$ denote the set of indices of committed attributes.

UC means the attribute is chosen by the Recipient and not known by the Issuer, and not committed. Let $A_{UC}$ denote the set of indices of user chosen attributes. The master secret is always a user chosen attribute.

The ordered set $A_{hidden}$ contains indices of attributes that are issued using the method UC or CO, and 1, the index of the master secret. Let $A_{known}$ be the complement of $A_{hidden}$ with respect to $\{1, \ldots, #S\}$ where $#S$ is the number of predicates in $S$. $A_{known}$ must also be ordered. Note that $A_{hidden} = A_{UC} \cup A_{CO}$. The sets $A_{known}, A_{CO}, A_{UC}$ form a partition of $\{1, \ldots, #S\}$.

Example 6.1. $eq(m_3, 12, KN)$ specifies that the value of attribute $m_3$ has been chosen as 12 by the Issuer. $eq(m_4, UC)$ specifies $m_4$ to be chosen by the Recipient and thus is not known by the Issuer. $eq(m_2, c_2, CO)$ means that attribute $m_2$ is committed to by the Recipient as $c_2$.

The context string We also assume that both the Issuer and Recipient have computed a common string context which is a list of all public parameters and the issuer public key. This string will be included in the hash which computes the challenge non-interactively using the Fiat-Shamir heuristic. This prevents values generated during the proof from being re-used in some other context.
6.1 Issuance at the Cryptographic Layer


<table>
<thead>
<tr>
<th>Common input:</th>
<th>pk, S, {ck = g^{mk}h^r_k : k \in A_{CO}}, {mi : i \in A_k}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recipient private input:</td>
<td>m_i, {open(ck) = (m_k, r_k) : k \in A_{CO}}, {mj : j \in A_h}</td>
</tr>
<tr>
<td>Issuer private input:</td>
<td>sk_i</td>
</tr>
<tr>
<td>Recipient output:</td>
<td>(A, e, v), m_1, ..., m_l', l' \leq l or \bot</td>
</tr>
</tbody>
</table>

Round 0

0.1 Issuer chooses a random nonce \( n_1 \in_R \{0, 1\}^{\ell_R} \).

0.2 Issuer \rightarrow Recipient: \( n_1 \).

0.3 Issuer and Recipient derive \( A_{UC}, A_{CO}, A_{KN} \) from \( S \).

foreach attribute \( m_i \) in \( S \).getAttributes()
  if \( m_i \).getMode() == UC, add \( i \) to \( A_{UC} \),
  if \( m_i \).getMode() == CO, add \( i \) to \( A_{CO} \),
  if \( m_i \).getMode() == KN, add \( i \) to \( A_{KN} \).

Round 1

1.1 Recipient chooses a random integer \( v' \in_R \{0, 1\}^{\ell_v + \ell_R} \).

1.2 Recipient computes

\[
U = S^{v'} \cdot X \cdot Y \pmod{n}
\]

where

\[
X = \prod_{j \in A_{UC}} R_j^{m_j}, \text{ and}
\]

\[
Y = \prod_{k \in A_{CO}} R_k^{m_k}.
\]

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.)

\[
SPK\{(A_{hidden}, v', \{open(ck) = (m_k, r_k) : k \in A_{CO}\}) : U \equiv \pm S^{v'} \prod_{k \in A_h} R_k^{m_k} \land \\
m_i \in \{0, 1\}^{\ell_m + \ell_R + \ell_R + 2} \forall i \in A_{hidden} \land \\
c_k = g^{m_k}h^{r_k} \forall k \in A_{CO}\}.
\] (2)

Aside. In some cases the issuer may require that the master secret \( m_1 \) be the same as the master secret of a previous certificate. If this is desired, the proof above should be augmented to prove that \( U \) contains the same value as the previous certificate. Note that version 1.0 of the library focuses on single certificate proofs and will not prove cross-certificate equality.
1.3 Recipient computes SPK (2).

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

\[ \tilde{U} = S^{r_v} \cdot \tilde{X} \cdot \tilde{Y} \]

where

\[ \tilde{X} = \prod_{j \in A_{UC}} R_j^{r_j}, \]
\[ \tilde{Y} = \prod_{k \in A_{CO}} R_k^{r_k}, \]

and where

\[ r_v, r_j, r_k \in R \{0, 1\}^{f_n + 2f_R + f_H}, \]

Store all random values.

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

\[ \tilde{C} = (g^{r_k} h^{r_k})_{k \in A_{CO}} \]

where all \( r_{r_k} \) \( \in \) \( R \{0, 1\}^{f_c} \). Note that \( r_k \) are the same values as in Step 1.3.1.

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

\[ c = H(\text{context} || U || c_1 || \ldots || c_k || \tilde{U} || \tilde{C} || n_1). \]

(The ordered list \( \tilde{C} \) is catenated.)

1.3.4 (*responses to challenge*) The first response is \( s_v = r_v + cv' \), and the others, as ordered lists, are:

\[ S_{UC} = (s_j = r_j + cm_j)_{j \in A_{UC}} \]
\[ S_{CO} = (s_k = r_k + cm_k)_{k \in A_{CO}} \]
\[ S_g = (s_{g_k} = r_{g_k} + cm_k)_{k \in A_{CO}} \]
\[ S_h = (s_{h_k} = r_{h_k} + cr_k)_{k \in A_{CO}} \]
\[ S_{gh} = S_g \cup S_h \]

*Note: \( S_{CO} \) and \( S_g \) are the same.*

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

\[ P_1 = (c, s_v, S_{UC}, S_{CO}, S_{gh}). \]
6.1 Issuance at the Cryptographic Layer

1.4 Recipient → Issuer: $U, P_1, n_2 \in R \{0, 1\}^{k_0}$.

Round 2 (Signature Generation)

2.0 Issuer verifies $P_1$.

2.0.1 (representation of $U$) Compute

$$\hat{U} = U^{-c}(S^{s_i'}) \left( \prod_{j \in A_{UC}} R_j^m \right) \left( \prod_{k \in A_{CO}} R_k^n \right).$$

2.0.2 (knowledge of committed values) Compute the ordered list

$$\hat{C} = ((c_k)^{-c}(g^{s_k})(h^{s_k})))_{k \in A_{CO}}.$$

2.0.3 (verify challenge) Compute

$$\hat{c} = H(\text{context}||U||c_1||\ldots||c_k||\hat{U}||\hat{C}||n_1)$$

(The ordered list $\hat{C}$ is catenated.) If $\hat{c} \neq c$, verification fails, abort IssueCertificateProtocol and return $\perp$.

2.0.4 (length checks) Check that

$$s_{v'} \in \{0, 1\}^{f_o + 2f_h + \ell + 1},$$

$$s_i \in \{0, 1\}^{f_m + \ell + \ell + 2}, \text{ for all } s_i \in S_{UC}, S_{CO}.$$

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

$$e \in R [2^{f_e - 1}, 2^{f_e - 1} + 2^{f_e} - 1].$$

2.1.2 Choose a random integer $\tilde{v} \in R \{0, 1\}^{f_e - 1}$, and compute $v'' = 2^{f_e} + \tilde{v}$.

2.1.3 Compute

$$Q = \frac{Z}{U(\prod_{i \in A_k} R_i^{m_i})^{s_{v''}}} \text{ and } A = Q^{-1} \mod p'q' \mod n.$$

$(A, c, v'')$ will be sent to the Recipient. Recall that $A_k$ contains the known, or issuer-chosen attributes $A_{KN}$.

2.2 Issuer creates the following proof of correctness.

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

17 Licensed Material.
6.2 Issuance at the Credsystem Layer

At the Credsystem layer, issuing has more semantics. The protocol is based on the low-level protocol given above and adds the additional semantics on top as explained in detail in this section. We continue using the notation from the previous section.

The proof specification $S^*$ used in Credsystem is structurally similar to $S$ from the lower level. The key differences are that $S^*$ refers to attributes by ontology instead of by integer index, and that attributes have richer semantics at this level, e.g., they have data types assigned and can span multiple low-level attributes. Additionally, features of the certificates used in the protocol can be referred to in $S^*$ while they are mapped to more basic attributes at the cryptographic layer.

6.2.1 Issuance Protocol

The protocol is simply:

1. Issuer and Recipient transform $S^*$ to $S$ (high-level to low-level).

Details of the transformation from $S^*$ to $S$ are found in Appendix ??.

6.3 Library API

We now describe the interface for issuance provided by idemix for programmers. First the protocol API for the Issuer.
7 PROVING HOLDERSHIP OF A CERTIFICATE

API: IssueCertificateIssuer

<table>
<thead>
<tr>
<th>Input</th>
<th>All system and public parameters, ( pk_I, sk_I ) and ( S ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>nothing (success) or ( \perp ) (failure)</td>
</tr>
</tbody>
</table>

We now present the API for the Recipient. Recall that the specification \( S \) received by each party will differ to respect the private inputs specified by \( \text{IssueCertificateProtocol} \). For example, the specification of the issuer will not include the values attributes in \( A_{\text{hidden}} \).

API: IssueCertificateRecipient

<table>
<thead>
<tr>
<th>Input</th>
<th>All system and public parameters, ( pk_I ) and ( S ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>The certificate specified by ( S ) (as output by ( \text{IssueCertificateProtocol} )) or ( \perp ) (failure)</td>
</tr>
</tbody>
</table>

7 Proving Holdership of a Certificate

In this section we describe how a certificate holder generates a (mostly) non-interactive proof of an assertion. Since there is little interaction between the prover and verifier, the presentation of the protocols for both roles are easily separated. See Section 7.2 for protocols used for verification.

7.1 Proving Holdership at the Cryptographic Layer

For creating a proof of holdership of a private certificate, we use the basic proof system of the underlying private certificate system of Camenisch and Lysyanskaya [3, 6]. Our protocols are denoted \( \text{ProveProtocol} \) and \( \text{VerifyProtocol} \). The participant holding the certificate will be named Prover, and the other will be named Verifier.

A note on the applicable trust model. At some points in the proof, the Prover will prove knowledge of a value using the group \( \mathbb{Z}_n^* \) where \( n \) is the issuer’s public key. Therefore, the Verifier must trust the issuer not to share their secret key with the Prover. An alternative is to have the Prover use the Verifier’s public key, however this requires additional infrastructure.

ProveProtocol Overview

\( \text{ProveProtocol} \) extends the proof of holdership of a CL-signature (the certificate) to multiple CL-signatures. Many useful assertions require multiple certificates, for example a merchant might want to ensure that the name field of the credit card certificate matches the name field on the government ID card certificate. Currently, only proofs involving a single certificate are possible. When multiple certificates are used, we must ensure that the master secret of each certificate is the same.
Abstractly, the proof is the composition of the following proofs.

\[
SPK\{ (certs) : \text{certs are valid (ProveCL)} \land \text{knowledge of committed values (ProveCommitment)} \land \text{predicates on attributes in certs (ProveRange, etc...)} \}
\]

Implicit in (3) is that the attributes in ProveCommitment and ProveRange are the same as those certified in ProveCL. Thus, in addition to proving an attribute \( m \) lies in a given range or a given commitment, the verifier is also assured that \( m \) appears in a particular certificate.

The basic proof structure for an individual certificate 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 computes a challenge \( c = H(\ldots || t) \), and computes a response of the form \( s = r - ca \), to prove knowledge of \( a \). We will refer to the value(s) in the first step as \( t \)-value(s), and the response(s) as \( s \)-value(s).

The three major steps in (3) will be described as separate protocols. For each certificate involved in the specification, we must prove possession of a CL-signature, using ProveCL. Depending on the specification, we will also need ProveCommitment to prove knowledge of committed attributes and ProveRange to prove hidden attributes lie in some interval. Further sub-protocols, or “provers” can be added as well to extend the features provided by the library (for example, adding \( k \)-show credentials).

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.

**Data Structures**

The following three data structures are used for bookkeeping in ProveProtocol. They will be populated at the beginning of the proof from \( S \).

**Revealed attributes** Both parties use \( S \) to compute the set of attributes which will be revealed during ProveProtocol. The set \( cert.A_{\text{revealed}} \) is defined as the indices of attributes in \( cert \) which appear in a predicate using the equals relation. The remaining attributes in the certificate, represented as \( A_{\text{revealed}} \), remain hidden during the proof. Note that 1 as attribute index is never in \( A_r \) since the master secret \( m_1 \) is stored at attribute base 1 and is never revealed.

**Certificate Map** The certificate map is a table which links attributes to certificates. For an instance certMap, we may call

```
certMap.add(cert, m_i)
```

```
setExp(cert, m_i, r).
```

\[ 20 \text{ Licensed Material.} \]
7.1 Proving Holdership at the Cryptographic Layer

**Commitment Map** The commitment map is a table keeping track of which attributes appear in which commitments. To add an entry to the instance `commMap`, use

```
commMap.add(commitment, commitmentBaseIndex, m_i).
```

This is interpreted as the attribute `m_i` is contained in `commitment`, at index `commitmentBaseIndex`. For example, if `commitment = g^{m_i} h^r`, the index of `m_i` in `commitment` is 1. The commitment map also has a method `commMap.getAttribute(commitment, commitmentBaseIndex)`, which returns the attribute associated with the specified base. In our example `commMap.getAttribute(commitment, 1)` would return `m_i` and `commMap.getAttribute(commitment, 2)` would return `null` (since `r` is not an attribute).

**Exponent Relations** This is a simple table which maps attributes to random values used in the proof. For an instance `expRelations`, call

```
expRelations.add(m_i, r)
```

to indicate that `r` is to be the random value used for `m_i` in all predicates of the proof involving `m_i`. The method `expRelations.contains(m_i)` returns true if `m_i` was added, and `expRelations.get(m_i)` returns the corresponding random value.

Our presentation of `ProveProtocol` first describes how the `Prover` generates the proof, and then how the `Verifier` performs verification. Recall that the specification each party receives as input differs, the certificates are not known to the `Verifier`.

### 7.1.1 Protocol: `ProveProtocol`

<table>
<thead>
<tr>
<th>Input:</th>
<th>( m_1, S, ) nonce ( n_1 ) (from <code>Verifier</code>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>non-interactive proof of statement in ( S ): ((c, s, \text{Common}, T))</td>
</tr>
</tbody>
</table>

0. (**Setup**) Initialize a certificate map `certMap`, a commitment map `commMap`, and a table `expRelations`. The table `expRelations` is a temporary variable for Step 0.

0.1 **Populate bookkeeping data structures using \( S \)

0.1.1 **foreach** predicate `pred` in `S.getPredicates()`

if `rel` is invalid, return \( \bot \).

if `m_i` belongs to a certificate `cert` in `S.getCertificates()`

`certMap.add(cert, m_i)`

if `m_i` is committed, **then**
expRelations.add($m_i, 0$).
// this just records the fact that $m_i$ will require shared randomness.

if $m_i$ is committed, then
    commMap.add($c_i$, baseIndex, $m_i$).
    // a reference to $c_i$ and the commitment base index are stored in $m_i$.

0.1.2 foreach commitment $c \in S$
    foreach base index $b \in c$
        if ($b, c$) not in commMap then
            commMap.add($c$, $b$, null)
            // null indicates that this ($b, c$) is not used

0.1.3 foreach certificate $cert \in S$
    foreach attribute $m_i \in cert$
        if $m_i \notin cert.A_{revealed}$ then
            choose $r \in_R \{0, 1\}^{\ell_m + \ell_\emptyset + \ell_H}$
            certMap.setExp($cert$, $i$, $r$)
        if $certMap.get(cert, i) \neq null \&\& expRel.contains(m_i)$ then
            $expRelations.put(m_i, r)$

0.2.2 foreach commitment $c \in S$
    foreach base index $b \in c$
        $m_i = commMap.getAttribute(c, b)$
        if $m_i \neq null \&\& expRelations.contains(m_i)$
            $r = expRelations.get(m_i)$
        else
            choose $r \in_R \{0, 1\}^{\ell_m + \ell_\emptyset + \ell_H}$
            commMap.setExp($c$, $b$, $r$)

1. Compute $t$-values

1.1 Initialized a list $r$.

1.2 Initialize a list $T$ to store the $t$-values for each proof, and a list $Common$ to store outputs from sub-protocols which are be common to both the prover and verifier.
7.1 Proving Holdership of a Certificate

1.3 (Call ProveCL.) For each certificate $cert \in S$, do the following.

1.3.1 Let $A_r$ be the set of non-revealed attributes in $cert$, then set
$$r = (\text{certMap.getExp}(cert, m_i) | m_i \in A_r).$$

1.3.2 Call ProveCL($A_r$, $r$, $cert$) and add the output $t$-value to $T$, and the output common value to $\text{Common}$.

1.4 (Call ProveCommitment.) For each commitment $c_i$ in $S$, do the following.

1.4.1 Set $r = \text{commMap.getExps}(c_i)$.

1.4.2 Call ProveCommitment($c_i$, $r$) and add the output $t$-value to $T$.

1.4.3 Add $c_i$ to $\text{CO}$.

1.5 (Call ProveRange.) For each predicate $\text{pred}$ in $S$ where $\text{pred.rel}$ is valid, do the following.

1.5.1 Let $m_i$ be the attribute in $\text{pred}$, and $cert$ be the certificate containing $m_i$. Set $r = \text{certMap.getExp}(m_i)$.

1.5.2 Call ProveRange($r$, $\text{pred}$) and add output $t$-values to $T$, common values to $\text{Common}$.

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

$$c = H(\text{context}, \text{CO}, \text{Common}, T, n_1),$$

where

- $\text{context}$ is a string representing the context (or environment) of the proof, defined in §6.1,
- $\text{CO}$ is a list of all commitments in $S$ (e.g. values of the form $g^r h^m$ in the case of Pedersen commitments), and
- $\text{Common}$ is a list of common inputs to the zero knowledge proofs.

3. Compute the responses

3.1 Initialize a list $s$ to store the $s$-values from each proof.

3.2 For each certificate, add the $s$-values from each proof ProveCL($c$) to $s$.

3.3 Add the $s$-values from ProveCommitment($c$) to $s$ (for the proofs started in Step 1.4).

3.4 Add the $s$-values from ProveRange($c$), to $s$ (for the proofs started in Step 1.5).

4. Output the proof ($c$, $s$, $\text{Common}$). Note that version of the Identity Mixer library relies on the order of the list $s$ to determine the association of the response values with the respective prove modules.

We now give the details of the three sub protocols.
7.1 Proving Holdership at the Cryptographic Layer

7.1.2 Protocol: ProveCL

ProveCL proves the following,

\[ SPK \{ (e, \{ m_i : i \in A_{\text{hidden}} \}, v) : \]
\[ Z \prod_{i \in A_h} R_{m_i}^{m_i} \equiv \pm A^e S^v \prod_{j \in A_h} R_j^{m_j} \pmod{n} \]
\[ \land m_i \in \{0, 1\}^{\ell_{\alpha} + \ell_{\beta} + \ell_{H} + 2} \quad \forall \ i \in A_h \]
\[ \land e - 2^{\ell_{e} - 1} \in \{0, 1\}^{\ell_{\alpha} + \ell_{\beta} + \ell_{H} + 2} \pmod{n_1} \]

which expands to the following protocol.

**Protocol:** ProveCL

<table>
<thead>
<tr>
<th>Input:</th>
<th>$A_{\gamma}, r, \text{cert}, [c]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>if $c = \text{null}$, outputs one $t$-value, and a common value $A'$</td>
</tr>
<tr>
<td></td>
<td>otherwise outputs $s$-values</td>
</tr>
</tbody>
</table>

Let $(A, e, v)$ be the CL signature for `cert`. If $c$ is not null, then steps 1-3 have already been executed, skip to Step 4.

1. (randomize signature)
   1.1 Choose $r_A \in \{0, 1\}^{\ell_{\alpha} + \ell_{\beta}}$
   1.2 Compute the blinded CL signature $(A', e, v')$, where
   \[ A' = A S^r_A \pmod{n}, \]
   \[ v' = v - e r_A \pmod{n}. \]
   Additionally compute $e' = e - 2^{\ell_{e} - 1}$.

2. (compute t-values)
   2.1 Choose random integers
   \[ r_e \in \pm \{0, 1\}^{\ell_{e} + \ell_{\beta} + \ell_{H}} \]
   \[ r_{v'} \in \pm \{0, 1\}^{\ell_{e} + \ell_{\beta} + \ell_{H}} \]
   2.2 Compute
   \[ T = (A')^{r_e} \left( \prod_{i \in A_h} R_i^{m_i} \right) (S^{r_{v'}}) \]

3. Output $t$-value $T$, common value $A'$. 

24 Licensed Material.
4. (Compute the s-values.) Compute the following in $\mathbb{Z}$:

4.1 $s_c = r_c + ce'$,
4.2 $s_{m_i} = r_{m_i} + cm_i$ for $i \in A_h$, and
4.3 $s_v = r_v' + cv'$.

7.1.3 Protocol: ProveCommitment

The ProveCommitment protocol proves knowledge of the committed attributes.

**Protocol: ProveCommitment**

| Input: | $r, c_j, [c]$ |
| Output: | if $c = \text{null}$, outputs one $t$-value otherwise outputs s-values |

Note: the following supports Pedersen commitments, of the form $g^m h^r$. In idemix version 1.0 non-randomized commitments are explicitly forbidden here.

If $c$ is not null, skip to Step 2.

1. (Compute $t$-values) Output

$$T = g^{r(1)} h^{r(2)},$$

where $r(j)$ are the corresponding random values chosen in Step 1.1 of ProveProtocol.

2. (Compute $s$-values) Output $s_j = r(j) + ce_j$ for $j = 1, 2$, where $r(j)$ is the corresponding random value chosen in Step 1.1 of ProveProtocol, and $e_j$ is the exponent of base $j$ (the $j$-th value of $\text{open}(c_j)$).

7.1.4 Protocol: ProveRange

Note that the input $r_m$ corresponds to the random value chosen for $m$ in Step 1.1 of ProveProtocol.

**Protocol: ProveRange**

| Input: | $r_m, \text{pred} = (m \ \text{rel} \ b), [c]$ |
| Output: | if $c = \text{null}$, outputs $t$-values and common values otherwise outputs s-values |

If $c$ is not null, Steps 1-2 have already been executed, skip to Step 3.

1. (Proof Setup)

1.1 Define

$$\Delta = \begin{cases} 
  b - m & \text{if } \text{rel} = "\leq" \\
  b - m - 1 & \text{if } \text{rel} = "<" \\
  m - b & \text{if } \text{rel} = "\geq" \\
  m - b - 1 & \text{if } \text{rel} = ">"
\end{cases}$$
7.1 Proving Holdership at the Cryptographic Layer

7.1 PROVING HOLDERSHIP OF A CERTIFICATE

and

\[ a = \begin{cases} 
-1 & \text{if } \text{rel} = \leq \text{ or } < \\
1 & \text{if } \text{rel} = \geq \text{ or } > 
\end{cases} \]

Note that \( \Delta \) will always be non-negative if the predicate is true. If \( \Delta < 0 \), the proof should fail, return \( \bot \).

1.2 Express \( \Delta \) 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 [9, 10]. Note that this step accounts for a substantial fraction of the computation time of this protocol, and increases non-linearly with increasing values of \( \Delta \).

1.3 Compute mod \( n \):

\[
\begin{align*}
T_1 &= Z^{u_1} S^{r_1} \\
T_3 &= Z^{u_3} S^{r_3} \\
T_\Delta &= Z^{\Delta} S^{r_\Delta},
\end{align*}
\]

where \( r_\Delta, r_i \in R \{0, 1\}^{\ell_m + \ell_8} \).

The following steps implement the proof below.

\[
\text{SPK } \{(m, r_\Delta, \{u_1, \ldots, u_4\}, \{r_1, \ldots, r_4\}, \alpha) : \}
\]

\[
\begin{align*}
\land & \ T_\Delta^* Z^b \equiv \pm Z^m (S^\alpha)^{r_\Delta} \\
\land & \ T_j \equiv \pm Z^{u_j} S^{r_j}, \text{ for } j = 1, 2, 3, 4 \\
\land & \ T_\Delta \equiv T_1^{u_1} \cdots T_4^{u_4} S^{r_\Delta}(n_1)
\end{align*}
\]

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

2. (Compute \( t \)-values)

2.1 Compute mod \( n \):

\[
\begin{align*}
\hat{T}_1 &= Z^{q_1} S^{z_1} \\
\hat{T}_3 &= Z^{q_3} S^{z_3} \\
\hat{T}_\Delta &= Z^{r}(S^\alpha)^{z_\Delta}.
\end{align*}
\]

where \( q_i \in R \{0, 1\}^{\ell_m + \ell_n + \ell_8} \) and \( z_i, z_\Delta \in R \{0, 1\}^{\ell_m + \ell_n + 2\ell_8} \).

2.2 Compute

\[ Q = T_1^{z_1} T_2^{z_2} T_3^{z_3} T_\Delta^{z_\Delta} S^{r} \pmod{n} \]

where \( r_\rho \in R \{0, 1\}^{\ell_n + \ell_m + 2\ell_n + 2\ell_8 + 3} \).

2.3 Output \( t \)-values (to be hashed): \( \hat{T}_1, \ldots, \hat{T}_\Delta, Q \).

2.4 Output Common values \( T_\Delta, T_1, \ldots, T_4 \).
7.2 Verifying Holdership at the Cryptographic Layer

Proving Holdership of a Certificate

3. (Compute \( s \)-values)
   
   3.1 For \( i = 1, 2, 3, 4 \), compute
   \[
   s_{u_i} = q_i + cu_i ,
   \]
   
   3.2 and
   \[
   s_{r_i} = z_i + cr_i .
   \]
   
   3.3 Compute and output \( s_{r_\Delta} = z_\Delta + cr_\Delta . \)
   
   3.4 Compute \( U = \sum_{i=1}^{4} u_i r_i , \) and output \( s_\rho = r_\rho + c(r_\Delta - U) . \)

7.2 Verifying Holdership at the Cryptographic Layer

The verification of the proof will have a similar structure to its generation. The main protocol, \texttt{VerifyProtocol} will inspect \( S \) to determine what should be verified, then make calls to the appropriate verification sub-protocols. These are \texttt{VerifyCL}, \texttt{VerifyCommitment} and \texttt{VerifyRange}, which compute the verification values (\( \hat{t} \)-values) to be hashed (together) and compared against \( c \).

The Verifier uses \( S \) to derive \( cert.A_{\text{revealed}} \) as the Prover does. The number and order of verification sub-protocols must also match the proof sub-protocols used by the Prover.

7.2.1 Protocol: \texttt{VerifyProtocol}

| Input: \( S, P = (c, s, \text{Common}), \) nonce \( n_1 \) |
| Output: accept or reject \( P \) |

1. (Collect \( s \)-values)
   
   1.1 Use \( S \) to determine which \( s \)-values in \( s \) correspond to which sub-protocol.
   
   1.2 Equality of \( s \)-values need only be checked for committed attributes.

2. (Compute \( \hat{t} \)-values)
   
   2.1 Initialize a list \( \hat{T} \) to store the \( \hat{t} \)-values from the sub-protocols.
   
   2.2 For each certificate \( cert \in S \) call \texttt{VerifyCL}(s', cert) where \( s' \) is the appropriate set of \( s \)-values. Add the output of \texttt{VerifyCL} to \( \hat{T} \).
   
   2.3 As required by \( S \) call \texttt{VerifyCommitment}(s', cert), with the appropriate \( s \)-values and store the output in \( \hat{T} \).
   
   2.4 As required by \( S \), call \texttt{VerifyRange}(S, s', cert), with the appropriate \( s \)-values and store the output in \( \hat{T} \).
3. \((\text{Compute the verification challenge})\)

\[ \hat{c} = H(\text{context, CO, Common, } T, n_1), \]

where \text{context, CO and Common} are as defined in Step 2 of ProveProtocol.

4. \((\text{Check})\) If \(c = \hat{c}\) accept \(P\) and reject otherwise.

**Protocol: VerifyCL**

**Input:** \(s, \) output of ProveCL: \(A'\)

**Output:** \(\hat{T}\)

1. Compute:

\[ \hat{T} = \left( \frac{Z}{\prod_{i \in A_r} R_{i}^{s_m}} \right)^{\text{mod } n} \left( \prod_{i \in A_r} R_{i}^{s_m} \right)^{s_e} (A')^{s_e} \left( \prod_{i \in A_r} R_{i}^{s_m} \right)^{s_e} \mod n \]

2. Check lengths:

\[ s_{m_i} \in \{0, 1\}^{\ell_m + \ell_H + 1}, \text{ for } i \in A_r, \]

\[ s_e \in \{0, 1\}^{\ell_e + \ell_H + 1}. \]

If any of these checks fail, reject the proof.

3. Output \(\hat{T}\).

**Protocol: VerifyCommitment**

**Input:** \(c_i, s, \) output of ProveCommitment

**Output:** \(\hat{T}\)

1. Compute

\[ \hat{T} = c_i^{-\epsilon} \hat{s}^i \hat{s}^{i+1} \]

2. Output \(\hat{T}\).

**Protocol: VerifyRange**

**Input:** \(s_m, \) output of ProveRange: \(T_\Delta, T_1, T_2, T_3, T_4, Q\)

**Output:** \(\hat{T}_\Delta, \hat{T}_1, \ldots, \hat{T}_4, Q\)
7.3 Proving Holdership at the Credsystem Layer

1. Compute $\Delta'$ and update $c$ if necessary.

$$
\Delta' = \begin{cases} 
  b & \text{ if rel} = "\leq" \\
  b - 1 & \text{ if rel} = "<" \\
  b & \text{ if rel} = "\geq" \\
  b + 1 & \text{ if rel} = ">"
\end{cases}
$$

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

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

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

2.1 Compute

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

2.2 For $i = 1, 2, 3, 4$, compute

$$
\hat{T}_i = T_i^{-c} Z^{s_{u_i}} S^{s_{r_i}} \pmod{n}.
$$

2.3 Compute

$$
\hat{Q} = (T_{\Delta})^{-c} T_{1}^{s_{u_1}} T_{2}^{s_{u_2}} T_{3}^{s_{u_3}} T_{4}^{s_{u_4}} S^{s_{p}} \pmod{n}.
$$

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

7.3 Proving Holdership at the Credsystem Layer

The proof protocol at the Credsystem layer is based on the proof protocol at the lower layer (ProveProtocol). The Credsystem layer adds additional semantics in the space of a credential system. These semantics are stored in $S^*$, the high-level specification.

7.3.1 Proof Protocol

The protocol is simply:

1. **Prover** and **Verifier** transform $S^*$ to $S$ (high-level to low-level).

2. Using $S$, **ProveProtocol** and **VerifyProtocol** are executed.

Transformation of the specification is done in the same way as during certificate issuance. See Appendix ?? for details of the transformation.

7.4 Library API

We now describe the interface for proofs provided by idemix for programmers.

First the protocol API for the **Prover**.

29 Licensed Material.
8 EFFICIENCY AND ADVANCED FEATURES

API: ProveProver

<table>
<thead>
<tr>
<th>Input:</th>
<th>All system and public parameters, $m_1$ and $S$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>non-interactive proof $P$ of $S$ or ⊥</td>
</tr>
</tbody>
</table>

We now present the API for the Verifier. Recall that the specification $S$ received by each party will differ to respect the private inputs.

API: ProveVerifier

<table>
<thead>
<tr>
<th>Input:</th>
<th>All system and public parameters, $S$ and $P$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>accept $P$, reject $P$ or ⊥ (failure)</td>
</tr>
</tbody>
</table>

8 Efficiency and Advanced Features

8.1 Implementation Considerations

- hash function padding
- memory overwriting

8.2 Efficiency

Once this version is complete some benchmarks would be nice here, currently we are not using fast multi-exponentiation algorithms. See [1] for a survey of such methods.

8.3 Advanced and Planned Features

In this section we discuss some possible extensions to the system just described.

Anonymity revocation is realized using verifiable encryption of the user’s pseudonym with the certificate issuer.

Credentials with the $k$-show feature or $k$-show in a time period feature. A $k$-show credential can be shown $k$-times without the user being traceable, but when overspending, the user’s identity can be recovered. This is useful in realizing e-cash schemes. A credential that can be shown only up to $k$ times in a certain time period is useful for preventing sharing of credentials with large groups of users, e.g., by posting the credentials on the Web. Both are fully cryptographic solution and can be combined with hardware solutions for even stronger protection in a real-world deployment.

Long string attributes

It may be desirable for some applications to allow attribute strings longer than $\ell_H$. In this case, a long string can be specified in $S^*$, and is then divided into blocks of size $\ell_H$ and stored in multiple low level attributes.
Structure of the certificate

During issuance, a feature predicate $eq(m_2, \text{structure})$ is added that represents the structure of the certificate within attribute $m_2$.

Note that placing the structure into an attribute serves to authenticate the structure information to a party accepting certificate proofs. If the prover claims the certificate has an alternate structure (for example by claiming the 6th attribute contains their age instead of their birthyear) the predicate $eq(m_2, \text{structure})$ will become false.

Algorithm: ComputeCertificateStructure

This algorithm computes the cryptographic hash value of the certificate structure; this value will later be used in the proof to show authenticity of the structure. This is important for overall security in order to have a proper binding from the ontology and data types to low-level attributes. Recall that $H$ is a cryptographic hash function (our implementation uses SHA-256).

\[
\text{Input: } \text{a certificate } \text{cert}, \text{ with } l \text{ attributes} \\
\text{Output: } \text{struct}, \text{ a hash of cert's structure}
\]

\[
\text{struct} = H([1\leq i \leq l] \text{cert.types}[i]) \\
\quad ( [1\leq i \leq l] \text{cert.dataTypes}[i]) \\
\quad [\text{cert.map}\|\text{cert.certType}\|\text{cert.issuer})
\]

In the ComputeSpec algorithm, used to derive $S$ from $S^*$, we must add an extra step as well. A predicate must be added to verify the structure of each certificate involved in the proof. For each $\text{cert} \in S^*$, add a predicate

\[
eq (m_{\text{cert.map('Structure')}}), \text{ComputeCertificateStructure}\left(\text{cert}\right)
\]

for each $j$ to $S$. Note that the Prover can just use the $\text{struct}$ value encoded in the appropriate attribute of each certificate, but it is crucial that the Verifier compute the $\text{struct}$ from the structure information sent by Prover.

References


REFERENCES


