Documentation of the Topology Certification (TOPOCERT) Tool

Version 0.0.1

Newcastle University

March 26, 2018

Abstract

The TOPOCERT tool offers an approach for integrity and privacy for clouds and is a component of the PRISMACLOUD architecture. TOPOCERT yields the possibility to certify and prove properties of cloud infrastructures without disclosing the layout of the infrastructure. In this document, we describe the architecture and design of the tool, the preliminaries, the system setup, and the computations required for each library that comprises the TOPOCERT tool. TOPOCERT realizes what could be described as a credential system on cloud topologies and thereby follows the design paradigms of established anonymous credential schemes.

Contents

1 Preliminaries

1.1 Assumptions

Special RSA Modulus: A special RSA modulus has the form $N = pq$, where $p = 2p' + 1$ and $q = 2q' + 1$ are safe primes, the corresponding group is called *Special RSA group*.

Strong RSA Assumption: Given an RSA modulus N and a random element $g \in Z_N^*$, it is hard to compute $h \in Z_N^*$ and integer $e > 1$ such that $h^e \equiv \ mod\ N$. The modulus N is of special form pq, where $p = 2p' + 1$ and $q = 2q' + 1$ are safe primes.

Quadratic Residues: The set QR_N is the set of Quadratic Residues of a special RSA group with modulus N.

2 TOPOCERT Tool

3 Cryptography Utilities

In this section, we define utilities for computations in the underlying group structure, especially QR_N. Algorithms presented here are largely adapted from Victor Shoup's excellent book *A Computational Introduction to Number Theory and Algebra* [**?**].

3.1 Special RSA modulus

Algorithm 1: generateSpecialRSAModulus(): Generate Special RSA Modulus N.

Input: candidate integer a, prime factors of positive, odd integer N: $q_1, \ldots, q_r$.
Output: N, p, q, p', q'

1 $(p, p') \leftarrow generateRandomSafePrime()$
2 $(q, q') \leftarrow generateRandomSafePrime()$
3 $N \leftarrow pq$
4 **return** (N, p, q, p', q')

3.2 Generate Random Safe Prime

The algorithm for generating safe primes is adapted from [**?**, Section 4.6.1].

A fast algorithm for generating primes is presented in [**?**].

Algorithm 2: generateRandomSafePrime(): Generate random safe prime.

Input: required k bit-length of the prime.
Output: safe prime p, Sophie Germain prime p'

do
  Select a random (k-1)-bit prime p'
  $p \leftarrow 2p' + 1$
while not *isPrime(p)*
return (p, p')

Algorithm 3: generateRandomSafePrimeFast(): Generate random safe prime (fast algorithm).

Input: required k bit-length of the prime.
Output: safe prime e

Select a random (k+1)-bit prime P
do
  $R \leftarrow computeRandomNumber((2^k - 1)/2P, (2^{k+1} - 1/2P))$
  $e \leftarrow 2PR + 1$
while not *isPrime(e)*
return e

3.3 Commitment group

Algorithm 4: generateNumberSequence(): Compute number sequence [**?**, Section 9.5].

Input: integer number $m \geq 2$
Output: number sequence $(n_1, \ldots, n_k)$

$n_0 \leftarrow m$
$k \leftarrow 0$
repeat
  $k \leftarrow k + 1$
  $n_k \xleftarrow{R} \{1, \ldots, n_{k-1}\}$
until $n_k = 1$
return $(n_1, \ldots, n_k)$

Algorithm 5: generateRandomNumberWithFactors(): Compute random number in factored form [**?**, Section 9.6].

Input: integer number $m \geq 2$
Output: random number prime factorization $(p_1, \ldots, p_k)$

while *true* **do**
  $(n_1, \ldots, n_k) \leftarrow$ generateNumberSequence(m)
  let $(p_1, \ldots, p_r)$ be the subsequence of primes in $(n_1, \ldots, n_k)$
  $y \leftarrow \prod_{i=1}^{r} p_i$
  if $y \leq m$ **then**
    $x \xleftarrow{R} \{1, \ldots, m\}$
    if $x \leq y$ **then**
      return $(p_1, \ldots, p_r)$

Algorithm 6: generateRandomPrimeWithFactors(): Compute random prime number in factored form [**?**, Section 11.1].

Input: integer number $m \geq 2$
Output: prime number p factorization $(p_1, \ldots, p_k)$

do
  $(p_1, \ldots, p_k) \leftarrow generateRandomNumberWithFactors(m)$
  $p \leftarrow \prod_{i=1}^{k} p_i + 1$
while not *isPrime(p)*
return $(p_1, \ldots, p_k)$

Algorithm 7: createZPSGenerator(): Compute generator for $\mathbb{Z}_p^*$ [**?**, Section 11.1].

Input: prime factorization of the order of an odd prime p ($p - 1 = \prod_{i=1}^{r} q_i^{e_i}$)
Output: generator g for $\mathbb{Z}_p^*$

for $i \leftarrow 1$ *to* r **do**
  repeat
    choose $\alpha \in \mathbb{Z}_p^*$ at uniformally random
    compute $\beta \leftarrow \alpha^{(p-1)/q_i}$
  until $\beta \neq 1$
  $\gamma_i \leftarrow \alpha^{(p-1)/q_i^{e_i}}$
$\gamma \leftarrow \prod_{i=1}^{r} \gamma_i$
return γ

3.4 Chinese Remainder Theorem

Algorithm 8: Compute the Chinese Remainder Theorem algorithm (CRT) [**?**, Section 8.1.5].

Input: integers x_p, x_q, primes p, q with $gcd(p, q) = 1$.
Output: the integer x ($0 \leq x < p \cdot q,\ x \equiv x_p \pmod{p},\ x \equiv x_q \pmod{q}$)

Compute X, Y using Extended Euclidean algorithm such that $Xp + Yq = 1$
$1_p \leftarrow [Yq \bmod N]$
$1_q \leftarrow [Xp \bmod N]$
$x \leftarrow [(x_p \cdot 1_p + x_q \cdot 1_q) \bmod N]$
return x

When the factorization of N is known, in order to map $x \bmod N$ to the mod p and mod q representation, the element x relates to $([x \bmod p], [x \bmod q])$

3.5 Extended Euclidean algorithm

Algorithm 9: Compute the Extended Euclidean algorithm (EEA) [**?**, Section 4.2].

Input: integer a and odd integer b, $a >= b >= 0$
Output: integers d, s, t such that $d = gcd(a, b)$ and $as + bt = d$.

$r \leftarrow a$
$r' \leftarrow b$
$s \leftarrow 1$
$s' \leftarrow 0$
$t \leftarrow 0$
$t' \leftarrow 1$
while $r'! = 0$ **do**
  $q \leftarrow \lfloor r/r' \rfloor$
  $r'' \leftarrow r \bmod r'$
  $(r, s, t, r', s', t') \leftarrow (r', s', t', r'', s - s'q, t - t'q)$
d ← r
return (d, s, t)

3.6 Quadratic Residues Under Composite Modules QR_N

3.6.1 The Jacobi Symbol $(a|N)$

We will use the Jacobi symbol to establish whether a group element is part of QR_N. We adapt the definition from Shoup [**?**, Section 12.2]:

Definition 3.1 (Jacobi Symbol)**.** *Let a, N be integers, where N is positive and odd, so that $N = q_1, \ldots, q_k$, where the q_i's are odd primes, not necessarily distinct. Then the* Jacobi symbol $(a|N)$ *is defined as $(a|N) := (a|q_1) \cdots (a|q_k)$, where $(a|q_i)$ is the Legendre symbol (cf. [**?**, Section 12.1]).*

When it comes to computing the Jacobi symbol, this can be done using Euler's criterion computing:

$$a^{(q_i-1)/2} \pmod{q_i}$$

for each prime factor q_i of N. However, this approach has an asymptotic complexity of $O(r \cdot \mathsf{len}(q_i)^3)$ for a composite of r odd primes q_i: $N = q_1 \cdots q_r$.

Shoup [**?**, Section 12.3] specified an efficient algorithm similar to the Euclidian Algorithm with asymptotic complexity $O(\mathsf{len}(a)\mathsf{len}(N))$. We will use said Algorithm **??** to compute the Legendre symbol $(a|q_i)$.

Algorithm 10: Compute the Jacobi symbol $(A|N)$ [**?**, Section 12.3].

Input: candidate integer a, positive odd integer N.
Output: Jacobi symbol $(a|N)$.
Invariant: N is odd and positive.
Dependencies: splitPowerRemainder()

```
1  σ ← 1
2  repeat
3      // Loop invariant:  N is odd and positive.
4
5      a ← a mod N
6      if a = 0 then
7          if N = 1 then return σ else return 0
8      compute a', h such that a = 2^h a' and a' is odd
9      if h ≢ 0 mod 2 and N ≢ ±1 mod 8 then σ ← −σ
10     if a' ≢ 1 mod 4 and N ≢ 1 mod 4 then σ ← −σ
11     (a, N) ← (N, a')
12 forever
```

3.6.2 Testing Membership of QR_N

It is intractable to determine the membership in QR_N without knowledge of the factorization of N.

Given the factorization of $N = q_1 \cdots q_r$, we can determine whether $a \in \mathbb{Z}_N^*$ is a quadratic residue in QR_N by checking that

$$(a|q_i) = 1 \text{ for all } q_i \in \{q_1, \ldots, q_r\}.$$

Consequently, for the setup of the graph signature library with a special RSA modulus of two distinct primes $N = pq$, we require

$$(a|p) = 1 \wedge (a|q) = 1.$$

3.6.3 Generating a Generator of QR_N

We adapting the following definition from Shoup [**?**, Section 2.7].

Definition 3.2 (Primitive Root)**.** *For a given positive integer N, we say that $a \in \mathbb{Z}$ with $\mathsf{gcd}(a, N) = 1$ is a* primitive root *modulo N, if the multiplicative order of a modulo N is equal to $\phi(N)$.*

Algorithm 11: elementOfQR(): Determines if integer a is an element of QR_N.

Input: candidate integer a, prime factors of positive, odd integer N: $q_1, \ldots, q_r$.
Output: true if $a \in \mathrm{QR}_N$, false if $a \notin \mathrm{QR}_N$.
Dependencies: jacobiSymbol()

```
1 o ← true
2 for all q_i:
3       if (a|q_i) ≠ 1 then o ← false
4 end
5 return o
```

Generating an element of QR_N, in general, is simply achieved by squaring uniformly-chosen random element of $\mathbb{Z}_N^*$. An integer a is a group element of $\mathbb{Z}_N^*$ if and only if $\mathsf{gcd}(N, a) = 1$.

We need to ensure that the created element is not a generator of the sub-group of size 2. That is, the resulting quadratic residue must not equal 1.

Algorithm 12: createElementOfZNS(): Generate S' number.

Input: Special RSA modulus N.
Output: random number S' of QR_N.
Dependencies: isElementOfZNS()

```
1 do
2 |  Choose at random S' ∈_R {2, N − 1}
3 while not isElementOfZNS(S')          /* check gcd(S', N) = 1 */
4 return S'
```

Algorithm 13: isElementOfZNS(): Check if number is member of $\mathbb{Z}_N^*$.

Input: number a, modulus N.
Output: boolean: true or false.
Dependencies: isElementOfZNS()

```
1 if gcd(N, a) = 1 then
2 |  return true
3 else
4 |  return false
```

3.6.4 Number Representation.

splitPowerRemainder() presented in Algorithm **??** computes the greatest power of base 2 contained in an odd integer a and its remainder a'. We note that in Java BigInteger the

Algorithm 14: verifySGeneratorOfQRN(): evaluate generator S properties.

Input: generator S, p', q'.
Output: boolean: true or false

```
1 if S ≠ 1 (mod N) then
2 |   if S^p' ≠ 1 (mod N) ∧ S^q' ≠ 1 (mod N) then
3 |   |   return true
4 else
5 |   return false
```

Algorithm 15: verifySGeneratorOfQRN(): evaluate generator S properties (alternative implementation) [**?**].

Input: generator S, modulus N.
Output: boolean: true or false

```
1 if gcd(S − 1, N) = 1 then
2 |   return true
3 else
4 |   return false
```

Algorithm 16: createQRNGenerator(): Generate generator of QR_N.

Input: Special RSA modulus N, p', q'.
Output: generator S of QR_N.
Dependencies: createElementOfZNS(), verifySGenerator()

```
1 do
2 |   S' ← createElementOfZNS(N)
3 |   S ← S'^2 (mod N)
  |   /* all p'q' elements of QR_N apart from p' + q' elements are
  |      generators                                                  */
4 while not verifySGeneratorOfQRN(S)   /* check properties of generator S
   */
5 return S
```

most significant bit of a positive integer can be computed with getBitlength().

Algorithm 17: splitPowerRemainder(): Compute the $2^h a'$ representation of integer a.

Input: odd integer a.
Output: Integers h and a' such that $a = 2^h a'$.
Post-conditions: $a = 2^h a'$ and a' is odd

1 $h \leftarrow \mathsf{mostSignificantBit}(a)$
2 $a' \leftarrow a - 2^h$
3 return (h, a')

3.7 Camenisch-Lysyanskaya Signatures

Algorithm 18: generateCLSignature(): Generate Camenisch-Lysyanskaya signature

Input: message m
Output: signature σ

1 $e \leftarrow \mathsf{createRandomPrime}(l_\rho, gs\_params)$
2 $v \leftarrow \mathsf{createRandomNumber}(l_v, gs\_params)$
3 $A = \left(\frac{Z}{R_0^m S^v}\right)^{1/e} \bmod N$
4 $\sigma \leftarrow (e, A, v)$
5 **return** σ

4 Graph Signature Library

The graph signature library implements the corresponding signature scheme (GRS) specified by Groß [**?**].

The library realizes the interactions between a signer and a recipient, meant to create a signature on a hidden committed graph, and the interactions between a prover and a verifier, meant to prove properties of a graph signature in zero-knowledge.

Graph signatures can be formed by combining a committed (hidden) sub-graph from the recipient and a issuer-known sub-graph from the signer. For the sake of the PRISMACLOUD project, it is sufficient to realize issuer-known graphs, as the graphs will be known by the signer (auditor).

4.1 Signer S

The signer is responsible to generate an appropriate key setup, to certify an encoding scheme, and to sign graphs. In the HiddenSign() protocol the signer accepts a graph commitment from the recipient, adds an issuer-known sub-graph and completes the signature with his secret key sk_S. The signer outputs a partial graph signature, subsequently completed by the recipient.

4.2 Recipient R

The recipient initializes the HiddenSign() protocol by creating a graph commitment and retaining randomness R, possibly only containing his master secret key, but no sub-graph. In this case, it is assumed that the signer knows the graph to be signed. Once the signer sends his partial signature, the recipient completes the signature with his randomness R.

4.3 Prover P

The prover role computes zero-knowledge proofs of knowledge with a policy predicate $\mathfrak{P}$ on graph signatures. These proofs can either be interactive or non-interactive.

4.4 Verifier V

The verifier role interacts with a prover to verify a policy predicate $\mathfrak{P}$. The verifier initializes the interaction, sending the policy predicate $\mathfrak{P}$ as well as a nonce that binds the session context.

4.5 Proof Context

The different prover and verifier algorithms co-create, amend and draw upon a joint proof context. The proof context is specific for a session of a zero-knowledge proof. It contains the entire proof state, that is,

- Integer and graph commitments,
- witness commitments,
- challenge,
- responses.

The prover's proof context contains additional secrets:

- The randomness of integer and graph commitments,
- the randomness corresponding to the secrets of the ZKPoK, and
- the secrets themselves (especially the actual graph and its encoding).

4.6 Abstract Description

Parameters. We offer the description of the parameters used for the graph signature scheme in Table **??**. We use the same notation as the Identity Mixer credential system, the standard realization of the Camenisch-Lysyanskaya signature scheme [**?**].

Core Interface. The graph signature library draws upon an interface with multiple operations. We first specify the abstract interface itself in Definition **??** and then discuss the inputs and outputs subsequently.

Definition 4.1 (Graph Signature Scheme)**.** *The graph signature scheme consists of the following algorithms:*

$((pk_{\mathsf{S}}, sk_{\mathsf{S}}), \sigma_{\mathsf{S,kg}}) \leftarrow$ **Keygen**$(1^{\lambda}, gs\_params)$ *A probabilistic polynomial-time algorithm which computes the key setup of the graph signature scheme and corresponding commitment scheme.*

$((pk_{\mathsf{S,E}}, sk_{\mathsf{S,E}}), \sigma_{\mathsf{S,es}}) \leftarrow$ **GraphEncodingSetup**$((pk_{\mathsf{S}}, sk_{\mathsf{S}}), \sigma_{\mathsf{S,kg}}, enc\_params)$ *A probabilistic polynomial-time algorithm which computes the setup of the graph encoding, especially, a reserved certified set of bases which are meant to hold the vertex and edge messages.*

$C \leftarrow$ **Commit**$(\mathcal{G}; R)$ *A probabilistic polynomial-time algorithm computing an Integer commitment on a graph.*

$(\sigma; \epsilon) \leftarrow$ **HiddenSign**$(pk_{\mathsf{S,E}}, C, V_{\mathsf{R}}, V_{\mathsf{S}})$ *An interactive probabilistic polynomial-time algorithm between a recipient and a signer which signs a committed graph. We note that both parties can have common inputs, namely a commitment $C = \mathsf{Commit}(\mathcal{G}_{\mathsf{R}}; R)$ encoding the recipient's graph and disclosed connections points $V_{\mathsf{R}}, V_{\mathsf{S}}$, and the Signer's extended public key $pk_{\mathsf{S,E}}$. Hence, the signer and the recipient can contribute sub-graphs to be combined.* Private inputs: *Recipient* R*:* $\mathcal{G}_{\mathsf{R}}$*, commitment randomness* R*; Signer* S*:* $sk_{\mathsf{S,E}}$*,* $\mathcal{G}_{\mathsf{S}}$*.*

0 or 1 $\leftarrow$ **Verify**$(pk_{\mathsf{S}}, C, R', \sigma)$ *A verification algorithm on graph commitment* C *and signature* σ*.*

Algorithm Input/Output Specification. We define the inputs and outputs for the abstract interface as follows.

Definition 4.2 $(((pk_{\mathsf{S}}, sk_{\mathsf{S}}), \sigma_{\mathsf{S,kg}}) \leftarrow \mathsf{Keygen}(1^{\lambda}, gs\_params))$**.** *A probabilistic polynomial-time algorithm which computes the key setup of the graph signature scheme and corresponding commitment scheme.*

> **Thomas' note**:
>
> *I would keep generation of public and secret key together. At the same time, I'd separate out the group setups and the generation of the actual keys given the setups. In turn, we'd have separate generation of the commitment group from the generation of main group* QR_N*. Hence, we'd have a main* ***keygen()*** *algorithm with four sub-routines: 1) Generate* p, q, N *as well as* QR_N*, 2) Generate generator for* QR_N*, 3) generate master secret key base* R_0*, 4) generate* Γ*,.*

Inputs:

- general security parameter (1^λ)
- key generation parameters of the graph signature scheme (*gs_params*) described in Table **??**.

Outputs:

- secret key (sk_{S}) :
  - factorization of a special RSA group with modulus bit length ℓ_n
  - group setup of the special RSA group
  - group setup of the commitment group Γ
  - foundational generator S for the Quadratic Residues under the given Special RSA modulus QR_N.
  - dedicated base for the Recipient's master key R_0.
- public key pk_{S}:
  - group setup of the special RSA group
  - group setup of the commitment group Γ
- digitally sign the given public outputs and make the signature $\sigma_{\mathsf{S,kg}}$ public
- the parameters specified in *gs_params* are stored for this instantiation of the Signer S.

Group Setup:

Algorithm 19: commitmentGroupSetup(): Group setup for commitment group

Input: size of the prime order subgroup of Γ (l_ρ), size of the commitment group modulus (l_Γ), *gs_params*

Output: order of the subgroup of the commitment group ρ, commitment group modulus Γ, generators g and h

1 $\rho \leftarrow$ generateRandomPrime(l_ρ, gs_params)
2 $\Gamma \leftarrow$ generateGroupModulus(ρ, gs_params)
3 $g \leftarrow$ createGenerator(ρ, Γ, gs_params)
4 $h \leftarrow g^r$
5 return (ρ, Γ, g, h)

Key Generation:

Algorithm 20: KeyGen(): Main key generation algorithm for the graph signature scheme and commitment scheme

Input: size of RSA modulus (l_n), gs_params
Pre-conditions: l_n must be at least 2048.
Output: public key pk, secret key sk, signature Σ

$(N, p, p', q, q') \leftarrow$ generateSpecialRSAModulus(l_n, l_{pt})
$S \leftarrow$ createQRGenerator(N)
$interval \leftarrow [2 \dots p'q' - 1]$
$x_Z \leftarrow$ createRandomNumber$(interval, gs_params)$ /* $x_Z \in_R [2 \dots p'q' - 1]$ */
$Z \leftarrow S^{x_Z} \ mod \ N$
$x_{R_0} \leftarrow$ createRandomNumber$(interval, gs_params)$ /* $x_{R_0} \in_R [2 \dots p'q' - 1]$ */
$R_0 \leftarrow S^{x_{R_0}} \ mod \ N$
$(\rho, \Gamma, g, h) \leftarrow$ commitmentGroupSetup$(l_\rho, l_\Gamma, gs_params)$
$sk \leftarrow (p', q', x_{R_0}, x_Z)$
$pk \leftarrow (N, R_0, S, Z)$
return (sk,pk)

Signature Proof of Knowledge on the correct base generation In order to guarantee that the elements of the public key lie in the correct subgroups, we construct a proof of representation on the base generation using a non-interactive proof, which is called a "Signature Proof of Knowledge". The Prover constructs a proof using Algorithm **??** and publishes the proof for the Verifier, who can verify the generated proof using Algorithm **??**.

$$SPK\{(\alpha_0, \alpha_Z) : R_0 \equiv \pm S^{\alpha_0} \pmod{N} \wedge Z \equiv \pm S^{\alpha_Z} \pmod{N}\}(Z, R_0, S, N) \quad (1)$$

Definition 4.3 $(((pk_{\mathsf{S,E}}, sk_{\mathsf{S,E}}), \sigma_{\mathsf{S,es}}) \leftarrow$ GraphEncodingSetup$((pk_{\mathsf{S}}, sk_{\mathsf{S}}), \sigma_{\mathsf{S,kg}}, enc_params))$**.** *A probabilistic polynomial-time algorithm which computes the setup of the graph encoding, especially, a reserved certified set of bases which are meant to hold the vertex and edge messages.*

Inputs:

- Signer S's secret key (sk_{S})
- Signer S's public key (pk_{S})
- encoding setup parameters (enc_params)

Outputs:

- Public Output: bases reserved to hold vertex and edge encodings, $R_i | i \in \{1, \dots, \ell_{\mathcal{V}}\}$ for vertices and $R_j | j \in \{1, \dots, \ell_{\mathcal{E}}\}$ for edges.

Algorithm 21: SPKKeyProve(): Proving correct base generation for the key generation

Input: special RSA modulus N, bases: S, Z, R_0, bases and exponents for vertices: $R_{\pi(i)}^{e_i \Pi_{k \in f_{\mathcal{V}}(i)} e_k}$, bases and exponents for edges: $R_{\pi(i,j)}^{e_i e_j \Pi_{k \in f_{\mathcal{E}}(i,j)} e_k}$

Output: Challenge: c, responses: $s_{\alpha_0}, s_{\alpha_Z}, \{s_i\}_{i \in \ell_{\mathcal{V}}}, \{s_j\}_{j \in \ell_{\mathcal{E}}}$

$r_{\alpha_0} \in_R \{0,1\}^{\ell_\varnothing}$
$r_{\alpha_Z} \in_R \{0,1\}^{\ell_\varnothing}$
for $1 \dots i$ **do**
  $r_i \in_R \{0,1\}^{\ell_\varnothing}$
  $C_i \leftarrow S^{r_i} \bmod N$
for $1 \dots i$ **do**
  for $1 \dots j$ **do**
    $r_{i,j} \in_R \{0,1\}^{\ell_\varnothing}$
    $C_{i,j} \leftarrow S^{r_{i,j}} \bmod N$
$C_{R_0} \leftarrow S^{r_{\alpha_0}} \bmod N$
$C_Z \leftarrow S^{r_{\alpha_Z}} \bmod N$
$c = \mathcal{H}(N, R_0, Z, S, C_{R_0}, C_Z, \{C_i\}_{i \in \ell_{\mathcal{V}}}, \{C_{(i,j)}\}_{(i,j) \in \ell_{\mathcal{E}}})$
$s_{\alpha_0} = r_{\alpha_0} - c \cdot \alpha_0$
$s_{\alpha_Z} = r_{\alpha_Z} - c \cdot \alpha_Z$
for $1 \dots i$ **do**
  $s_i \leftarrow r_i - c \cdot x_i$
for $1 \dots j$ **do**
  $s_j \leftarrow r_j - c \cdot x_j$
return $(c, s_{\alpha_0}, s_{\alpha_Z}, \{s_i\}_{\ell_{\mathcal{V}}}, \{s_j\}_{\ell_{\mathcal{E}}})$

Algorithm 22: SPKKeyVerify(): Verifying correct base generation for the key generation

Input: special RSA modulus N, bases: $S, Z, R_0, R_i | i \in \{1, \dots, \ell_{\mathcal{V}}\}, R_j | j \in \{1 \dots, \ell_{\mathcal{E}}\}, c, s_{\alpha_0}, s_{\alpha_Z}, \{s_i\}_{\ell_{\mathcal{V}}}, \{s_j\}_{\ell_{\mathcal{E}}})$

Output: `True` if the *Verifier* accepts challenge and `False` if not

$\widehat{C}_{R_0} = R_0^c S^{s_{\alpha_0}} \bmod N$
$\widehat{C}_Z = Z^c S^{s_{\alpha_Z}} \bmod N$
for $1 \dots i$ **do**
  $\widehat{C}_{R_i} = R_i^c S^{s_i} \bmod N$
for $1 \dots j$ **do**
  $\widehat{C}_{R_j} = R_j^c S^{s_j} \bmod N$
$\widehat{c} = \mathcal{H}(N, R_0, Z, S, \widehat{C}_{R_0}, \widehat{C}_Z, \{\widehat{C}_i\}_{i \in \ell_{\mathcal{V}}}, \{\widehat{C_j}\}_{j \in \ell_{\mathcal{E}}})$
if $\widehat{c} = c$ **then**
  return True
else
  return False

- Sign the generators, proving knowledge of their representation and binding them to public key pk_{S}.
- Signer's *extended public key* $pk_{\mathsf{S,E}}$ certified combination of original public key pk_{S} and the vertex and edge encoding bases R_i, R_j
- Signer's *extended secret key* $sk_{\mathsf{S,E}}$ combination of original secret key sk_{S} and the discrete logarithms $\mathsf{log}_S(R_k)$
- Signature $\sigma_{\mathsf{S,es}}$
- Private Output: discrete logarithms of all produced bases with respect to generator S, $\mathsf{log}_S(R_k)$.

Definition 4.4 ($C \leftarrow \mathsf{Commit}(\mathcal{G}; R)$)**.** *A probabilistic polynomial-time algorithm computing an Integer commitment on a graph.*

Inputs:

- graph $\mathcal{G}$
- randomness R

Computations: It commits to the graph in an appropriate encoding, that is, holding vertex and edge representations in different bases. As specified in the graph signature definition [**?**], the algorithm will establish a commitment as follows:

$$C = \underbrace{\cdots R_{\pi(i)}^{e_i \Pi_{k \in f_{\mathcal{V}}(i)} e_k} \cdots}_{\forall \text{ vertices } i} \underbrace{\cdots R_{\pi(i,j)}^{e_i e_j \Pi_{k \in f_{\mathcal{E}}(i,j)} e_k} \cdots}_{\forall \text{ edges } (i,j)} S^r \bmod N,$$

where e_i and e_j are vertex representatives. The label representatives e_k are obtained with the vertex mappings $f_{\mathcal{V}}(i)$ and edge mappings $f_{\mathcal{E}}(i,j)$.

Outputs:

- commitment C
- Committer retains the randomness R for future commitment opening or proofs of representation

Definition 4.5 ($(\epsilon; \sigma) \leftarrow \mathsf{HiddenSign}(pk_{\mathsf{S,E}}, C, V_{\mathsf{R}}, V_{\mathsf{S}})$)**.** *An interactive probabilistic polynomial-time algorithm between a recipient and a signer which signs a committed graph. We note that both parties can have common inputs, namely a commitment* $C = \mathsf{Commit}(\mathcal{G}_{\mathsf{R}}; R)$ *encoding the recipient's graph and disclosed connections points* $V_{\mathsf{R}}, V_{\mathsf{S}}$*, and the Signer's extended public key* $pk_{\mathsf{S,E}}$*. Hence, the signer and the recipient can contribute sub-graphs to be combined.* Private inputs: *Recipient* R*:* $\mathcal{G}_{\mathsf{R}}$*, commitment randomness* R*; Signer* S*:* $sk_{\mathsf{S,E}}$*,* $\mathcal{G}_{\mathsf{S}}$*.*

The abstract interface specification for the interactive algorithm decomposes into two interfaces for Signer S and Recipient R.

$$\mathsf{Signer.HiddenSign}(pk_{\mathsf{S,E}}, C, V_{\mathsf{R}}, V_{\mathsf{S}}; sk_{\mathsf{S,E}}, \mathcal{G}_{\mathsf{S}}), \text{and}$$

$$\mathsf{Recipient.HiddenSign}(pk_{\mathsf{S,E}}, C, V_{\mathsf{R}}, V_{\mathsf{S}}; \mathcal{G}_{\mathsf{R}}, R).$$

Let us first discuss public and private inputs. While the Integer commitment C is publicly known, it is usually computed by the Recipient R for the the given $\mathsf{HiddenSign}()$ operation with the corresponding randomness R with $\mathsf{Commit}()$. The Recipient will be required to offer a proof of representation of the commitment as part of the interactive protocol.

Note that the key-pair inputs $(pk_{\mathsf{S,E}}, sk_{\mathsf{S,E}})$ refer to extended keys, that is, public and private keys that contain the information on encoding bases of $\mathsf{GraphEncodingSetup}()$.

While the $\mathsf{HiddenSign}()$ algorithm allows for either both private input graphs $\mathcal{G}_{\mathsf{R}}$ and $\mathcal{G}_{\mathsf{S}}$ to be present or absent, the standard case realized in TOPOCERT is that we have a signer-known graph $\mathcal{G}_{\mathsf{S}}$, but no hidden/committed graph of the Recipient R.

In most cases the connection points V_{R} and V_{S} will be equal, but that is not necessary.[1]

As output on the Recipient side, R obtains a graph signature $\sigma_{\mathsf{S},\mathcal{G}}$ (or short σ) on the combined graph $\mathcal{G} = \mathcal{G}_{\mathsf{R}} \cup \mathcal{G}_{\mathsf{S}}$ valid with respect to $pk_{\mathsf{S,E}}$.

The Signer S does not produce an output.

Definition 4.6 (0 or $1 \leftarrow \mathsf{Verify}(pk_{\mathsf{S,E}}, C, R', \sigma)$)**.** *A verification algorithm on graph commitment C and signature σ.*

The algorithm $\mathsf{Verify}()$ takes as inputs the Signer's extended public key $pk_{\mathsf{S,E}}$, a signed graph commitment C and its randomness R' and the graph signature σ.

The algorithm outputs either 0 or 1, signifying that σ is either invalid or valid as graph signature on C.

We note that usually graph signatures, such as σ are used in proof of possessions and further zero-knowledge proofs of knowledge to show certain properties. In such cases, we can use $\mathsf{Verify}()$ to signify a proof of representation that the secrets encoded in commitment C are equal to the secret messages of the graph signature σ.

Then we have a zero-knowledge proof of knowledge defined as follows:

$$\begin{aligned}
&PK\{(e_i, e_j, e_k, e, v, r) : \\
&\quad Z \equiv \pm \underbrace{\cdots R_{\pi(i)}^{e_i \Pi_{k \in f_{\mathcal{V}}(i)} e_k} \cdots}_{\forall \text{ vertices } i} \underbrace{\cdots R_{\pi(i,j)}^{e_i e_j \Pi_{k \in f_{\mathcal{E}}(i,j)} e_k} \cdots}_{\forall \text{ edges } (i,j)} A^e S^v \bmod N \\
&\quad C \equiv \pm \underbrace{\cdots R_{\pi(i)}^{e_i \Pi_{k \in f_{\mathcal{V}}(i)} e_k} \cdots}_{\forall \text{ vertices } i} \underbrace{\cdots R_{\pi(i,j)}^{e_i e_j \Pi_{k \in f_{\mathcal{E}}(i,j)} e_k} \cdots}_{\forall \text{ edges } (i,j)} S^r \bmod N \\
&\}.
\end{aligned}$$

Here the first equation proves the representation of the graph signature σ and the second equation proves the representation of the commitment C, yielding equality over the secrets e_i, e_j and e_k.

[1] The first graph signature [?] proposal referred to the connection points as $\mathcal{V}_{\mathsf{R}}$, $\mathcal{V}_{\mathsf{S}}$, which would mean the set of all vertices and not a subset.

We note that the proofs of knowledge on graph properties between prover and verifier are specified as standard Σ-proofs.

4.7 Issuing Protocol

- **Round 0**

  1. *Signer* chooses a random nonce $n_1 \in_R \{0,1\}^{\ell_\varnothing}$
  2. *Signer* $\xrightarrow{n_1}$ *Recipient*

- **Round 1**

  1. *Recipient* chooses a random integer $v' \in_R \{0,1\}^{l_n+\ell_\varnothing}$
  2. *Recipient* commits to the with an appropriate encoding following the $\mathsf{Commit}()$ algorithm:

$$U = \underbrace{\cdots R_{\pi(i)}^{e_i \Pi_{k \in f_{\mathcal{V}}(i)} e_k} \cdots}_{\forall \text{ vertices } i} \underbrace{\cdots R_{\pi(i,j)}^{e_i e_j \Pi_{k \in f_{\mathcal{E}}(i,j)} e_k} \cdots}_{\forall \text{ edges } (i,j)} S^{v'} \bmod N \tag{2}$$

  3. *Recipient* computes an non-interactive proof that the commitment is correctly computed (proof of representation) using a Signature Proof of Knowledge (SPoK):

$$\begin{aligned} &SPK\{(e_i, e_j, e_k, v') : \\ &\quad U \equiv \pm \underbrace{\cdots R_{\pi(i)}^{e_i \Pi_{k \in f_{\mathcal{V}}(i)} e_k} \cdots}_{\forall \text{ vertices } i} \underbrace{\cdots R_{\pi(i,j)}^{e_i e_j \Pi_{k \in f_{\mathcal{E}}(i,j)} e_k} \cdots}_{\forall \text{ edges } (i,j)} S^{v'} \bmod N\}(n_1) \end{aligned} \tag{3}$$

  3.1 compute randomness:
  $\tilde{v'} \in_R \{0,1\}^{l_n+2\ell_\varnothing+\ell_H}$,
  $\tilde{e_i} \in_R \{0,1\}^{l_m+l_n+\ell_\varnothing+\ell_H+1}$,
  $\tilde{e_j} \in_R \{0,1\}^{l_m+l_n+\ell_\varnothing+\ell_H+1}$,
  $\tilde{e_k} \in_R \{0,1\}^{l_m+l_n+\ell_\varnothing+\ell_H+1}$,

  3.2 compute witnesses:
  $T = \underbrace{\cdots R_{\pi(i)}^{\tilde{e_i} \Pi_{k \in f_{\mathcal{V}}(i)} \tilde{e_k}} \cdots}_{\forall \text{ vertices } i} \underbrace{\cdots R_{\pi(i,j)}^{\tilde{e_i}\tilde{e_j} \Pi_{k \in f_{\mathcal{E}}(i,j)} \tilde{e_k}} \cdots}_{\forall \text{ edges } (i,j)} S^{\tilde{v'}} \bmod N$

  3.3 compute challenge:
  $c = \mathcal{H}(\texttt{context}, N, R_{\pi(i)}, R_{\pi(i,j)}, Z, S, U, T, n_1)$

  3.4 compute responses:
  $\hat{v'} = \tilde{v'} + cv'$
  for $\forall \hat{e_i}$ compute $\hat{e_i} = \tilde{e_i} + ce_i$,
  for $\forall \hat{e_j}$ compute $\hat{e_j} = \tilde{e_j} + ce_j$,
  for $\forall \hat{e_k}$ compute $\hat{e_k} = \tilde{e_k} + ce_k$,

  3.5 output proof signature: $P_1 = (c, \hat{v'}, \hat{e_i}, \hat{e_j}, \hat{e_k})$

4. *Recipient* $\xrightarrow{U,\ P_1,\ n_2 \in_R \{0,1\}^{\ell_\varnothing}}$ *Signer*: commitment U, proof signature P_1, create nonce $n_2 \in_R \{0,1\}^{\ell_\varnothing}$
5. *Recipient* persists the following structures: `context`, randomness v'

- **Round 2**: Graph Signature generation

1. *Signer* verifies proof signature P_1:
   1.1 Compute:
   $\hat{U} = U^{-c}(S^{\hat{v}'}) \underbrace{\cdots R_{\pi(i)}^{\hat{e}_i \Pi_{k \in f_{\mathcal{V}}(i)} \hat{e}_k} \cdots}_{\forall \text{ vertices } i} \underbrace{\cdots R_{\pi(i,j)}^{\hat{e}_i \hat{e}_j \Pi_{k \in f_{\mathcal{E}}(i,j)} \hat{e}_k} \cdots}_{\forall \text{ edges } (i,j)}$
   1.2 Verify challenge:
   $\hat{c} = \mathcal{H}(\texttt{context}, N, R_{\pi(i)}, R_{\pi(i,j)}, Z, S, U, T, \hat{U}, \hat{T}, n_1)$
   1.3 if $\hat{c} \neq c$ then verification fails and the issuing protocol is aborted.
   1.4 Check lenghts:
   $\hat{v'} \in_R \{0,1\}^{l_n + 2\ell_\varnothing + \ell_H}$,
   $\hat{e_i} \in_R \{0,1\}^{l_m + l_n + \ell_\varnothing + \ell_H + 1}$,
   $\hat{e_j} \in_R \{0,1\}^{l_m + l_n + \ell_\varnothing + \ell_H + 1}$,
   $\hat{e_k} \in_R \{0,1\}^{l_m + l_n + \ell_\varnothing + \ell_H + 1}$,
   1.5 If any length check fails then the issuing protocol is aborted.
2. *Signer* generates partial graph signature to be completed by the *Recipient*.
   2.1 Choose a random prime:
   $e \in_R [2^{l_e - 1}, 2^{l_e - 1} + 2^{l'_e - 1}]$
   2.2 Choose a random integer $\tilde{v} \in_R \{0,1\}^{l_v - 1}$
   2.3 Compute: $v'' = 2^{l_v - 1} + \tilde{v}$
   2.4 Compute:
   $Q = \frac{Z}{US^{v''} \cdots R_{\pi(i)}^{e_i \Pi_{k \in f_{\mathcal{V}}(i) e_k}} \cdots R_{\pi(i,j)}^{e_i e_j \Pi_{k \in f_{\mathcal{E}}(i,j) e_k}}} \bmod N$
   2.5 Compute:
   $A = Q^{e^{-1} \bmod p'q'} \bmod N$
   2.6 *Signer* sends (A, e, v'') to the *Recipient*
   2.7 *Signer* stores: $Q, v'', \texttt{context}$
3. *Signer* creates a proof of correctness:
   $SPK\{(e^{-1}) : A \equiv \pm Q^{e^{-1}} \pmod N\}(n_2)$
   3.1 Choose $r \in_R Z_{p'q'}$
   3.2 Compute: $\tilde{A} = Q^r \pmod N$
   3.3 Compute: $c' = \mathcal{H}(\texttt{context}, Q, A, n_2, \tilde{A})$
   3.4 Compute: $s_e = r - c'e^{-1} \pmod{p'q'}$
   3.5 The proof consists of $P_2 = (s_e, c')$
4. *Signer* sends $(A, e, v''), P_2, messages$ to the *Recipient*

- Round 3

1. Compute $v = v'' + v'$
2. Recipient verifies (A, e, v)
   2.1 Check that e is prime and $e \in [2^{l_e-1}, 2^{l_e-1} + 2^{l'_e-1}]$
   2.2 Compute $Q = \frac{Z}{S^v \cdots R_{\pi(i)}^{e_i \Pi_{k \in f_{\mathcal{V}}(i)} e_k} \cdots R_{\pi(i,j)}^{e_i e_j \Pi_{k \in f_{\mathcal{E}}(i,j)} e_k}} \bmod N$
   2.3 Compute $\hat{Q} = A^e \pmod N$
   2.4 if $\hat{Q} \neq Q \bmod N$, abort the issuing protocol.
3. *Recipient* verifies P_2
   3.1 Compute $\hat{A} = A^{c'+s_e \cdot e} \pmod N = A^{c'} Q^{s_e} \pmod N$
   3.2 Compute $\hat{c} = \mathcal{H}(\texttt{context}, Q, A, n_2, \hat{A})$
   3.3 If $\hat{c} \neq c'$ abort the issuing protocol
4. store credential $messages, (A, e, v)$

4.8 Proving Protocol

- Round 0
- Round 1
- Round 2

4.9 Recommendations

To securely use the graph signature scheme and the TOPOCERT tool, it is necessary to follow key size requirements specified for the Identity Mixer cryptographic library [**?**]. Here we note that the TOPOCERT tool is meant to operate in an implementation with a 2048-bits key strength and appropriately selected parameters, which implies parameters as defined in Table **??**. For a detailed specification of the parameter selection for the underlying Camenisch-Lysyanskaya signature scheme, we refer to Tables 2 and 3 of the Specification of the Identity Mixer Cryptographic Library, Version 2.3.40, on p. 43 [**?**].

Remark 1 (Security Parameter). The security parameters, especially bit length for the group setups, flow from the specification of the bit length of the special RSA modulus ℓ_n and the message space ℓ_m. The constraints placed on the respective bit lengths are crucial to maintain the soundness of the security proof of the underlying Camenisch-Lysyanskaya signature scheme (cf. Table 3 of the Specification of the Identity Mixer Cryptographic Library, Version 2.3.40, on p. 43 [**?**]).

Remark 2 (Encoding Parameters). We consider the choices made for the graph encoding scheme.

Encoding Defaults The bit length parameters for the prime encoding $\ell'_{\mathcal{V}}$ and $\ell'_{\mathcal{L}}$ follow from the available message bit length, assuming that the labels are encoded as the lowest prime representatives. However, the given defaults for number of vertices, edges, labels to be encoded $\ell_{\mathcal{V}}$, $\ell_{\mathcal{E}}$, and $\ell_{\mathcal{L}}$ are not the theoretical maxima.

Table 1: Parameters of the graph signature scheme *gs_params* and encoding setup *enc_params*. Parameters for the underlying Camenisch-Lysyanskaya signature scheme are largely adapted from the Identity Mixer Specification [**?**]. In the implementation, this table is referred to as `table:params`.

(a) Parameters of Keygen()

Parameter	Description	Bit-length
ℓ_n	Bit length of the special RSA modulus	2048
ℓ_Γ	Bit length of the commitment group	1632
ℓ_ρ	Bit length of the prime order of the subgroup of Γ	256
ℓ_m	Maximal bit length of messages encoding vertices and edges	256
ℓ_{res}	Number of reserved messages	1†
ℓ_e	Bit length of the certificate component e	597
ℓ'_e	Bit length of the interval the e values are taken from	120
ℓ_v	Bit length of the certificate component v	2724
$\ell_\varnothing$	Security parameter for statistical zero-knowledge	80
ℓ_H	Bit length of the cryptographic hash function used for the Fiat-Shamir Heuristic	256
ℓ_r	Security parameter for the security proof of the CL-scheme	80
ℓ_{pt}	The prime number generation to have an error probability to return a composite of $1 - 1/2^{\ell_{pt}}$	80†

Note: † refers to numbers that are integers, not bit lengths.

(b) Parameters of GraphEncodingSetup()

$\ell_\mathcal{V}$	Maximal number of vertices to be encoded	1000†‡
$\ell'_\mathcal{V}$	Reserved bit length for vertex encoding (bit length of the largest encodable prime representative)	120
$\ell_\mathcal{E}$	Maximal number of edges to be encoded	50.000†‡
$\ell_\mathcal{L}$	Maximal number of labels to be encoded	256†‡
$\ell'_\mathcal{L}$	Reserved bit length for label encoding	16

Note: † refers to numbers that are integers, not bit lengths; ‡ refers to the default parameter, not the theoretical maximum.

Maximal Number of Labels For a single-labeled graph with $\ell'_{\mathcal{L}} = 16$, the maximal encodable number of labels is 6542. The restrictions of the number of labels is in place to allow for multi-labeled graphs, in which case the product of the label identifiers occupies the reserved space.

Maximal Number of Vertices The maximal number of vertices for the reserved bit length $\ell'_{\mathcal{V}} = 120$ is 1.59810^{34}. The limiting factor for the number of encoded vertices, however, is *not* the reserved bit length of the message space, but the space required to store the corresponding based dedicated vertex and edge encoding. For each possibly encodable vertex and edge the graph signature scheme needs to reserve a group element with an bit length of $\ell_n = 2048$. A encoding for fully connected graphs with $\ell_{\mathcal{V}} = 1000$ and $\ell_{\mathcal{E}} = \ell_{\mathcal{V}}(\ell_{\mathcal{V}} - 1) = 999000$ would consume 244.28 kBytes for vertices and 243.89 MBytes for the edges.

Remark 3 (Signature Size). A signature of the graph signature scheme consists of one group element and two exponents $A, e, v)$. A single signature has the following bit length for the default parameters in Table **??**:

$$|(A, e, v)|_2 = \ell_n + \ell_v + \ell_e = 5369 \text{ bits.}$$

Remark 4 (Base Randomization). We note here that the graph signature scheme proposed by Groß [**?**] requires a base randomization for multi-use confidentiality of graph elements. This is because the bases referenced in the ZKPoK are public knowledge and each proof reveals which exponents are harbored by which base.

The base randomization asks that random permutations $\pi_{\mathcal{V}}$ and $\pi_{\mathcal{E}}$ be applied to the vertex and edge bases respectively. A space-efficient solution for that requirement could use keyed pseudorandom permutations.

Let an appropriate family of pseudorandom permutations $\mathcal{F}$ on group elements in QR_N be given, where pseudorandom permutations (PRPs) are defined as by Katz and Lindell [**?**]. Theoretical work on constructions of pseudorandom permutations from pseudorandom functions was spawned by the seminal work of Luby and Rackoff [**?**]. As an alternative approach, we also refer to constructions of Verifiable Secret Shuffles, such as Neff [**?**], which allow a Prover in a honest-verifier zero-knowledge proof scenario to convince a Verifier that a secret shuffled was computed correctly.

1. During the Signer's round of $\mathsf{HiddenSign}()$, S chooses a uniformly random permutation key k with appropriate bit length.

2. S applies the pseudorandom permutations $(\pi_{\mathcal{V}}, \pi_{\mathcal{E}})$ with common key k to the certified base sets, obtaining permuted base sets.

3. Signer S then encodes graph $\mathcal{G}$ on the derived base sets.

4. Signer S shares permutation key k with the Recipient together with the corresponding signature $\sigma_k = (A, e, v)_k$ along with a proof of representation that σ_k indeed fulfills the CL-equation on the derived bases.

Hence, the Signer will issue multiple signatures, one for each permutation. Each signature has a size of one group element, two exponents, and one permutation key.

5 TOPOCERT Library