# DRAFT

# How to Implement a Stand-alone Verifier for the Verificatum Mix-Net

Douglas Wikström
KTH Stockholm, Sweden
dog@csc.kth.se

January 19, 2012

**Abstract**

Verificatum, http://www.verificatum.org, is a free and open source implementation of an El Gamal based mix-net which optionally uses the Fiat-Shamir heuristic to produce a universally verifiable proof of correctness during the execution of the protocol. This document gives a detailed description of this proof targeting implementers of stand-alone verifiers.

## Protocol Version 0.1

This document covers version 0.1 of the protocol. The Verificatum package version is independent of the protocol version.

## Help Us Improve This Document

This document is a draft. The most recent version of this document can be found at: http://www.verificatum.org/verificatum/vmnv-0.1.pdf.

Please help us improve the quality of this document by reporting errors, omissions, and suggestions to dog@csc.kth.se.

# Contents

# 1 Introduction

The Verificatum mix-net [10] can optionally be executed with Fiat-Shamir proofs of correctness, i.e., non-interactive zero-knowledge proofs in the random oracle model. These proofs end up in a special *proof directory* along with all intermediate results published on the bulletin board during the execution. The proofs and the intermediate results allows anybody to verify the correctness of an execution as a whole, i.e., that the joint public key, the input ciphertexts, and the output plaintexts are related as defined by the protocol and the public parameters of the execution. The goal of this document is to give a detailed description of how to implement a stand-alone algorithm for verifying the complete contents of the proof directory.

# 2 Background

Before we delve into the details of how to implement a verifier, we recall the El Gamal cryptosystem and briefly describe the mix-net implemented in Verificatum (in the case where the Fiat-Shamir heuristic is used to construct non-interactive zero-knowledge proofs).

## 2.1 The El Gamal Cryptosystem

The El Gamal cryptosystem [2] is defined over a group $G_q$ of prime order $q$. A secret key $x \in \mathbb{Z}_q$ is sampled randomly, and a corresponding public key $(g, y)$ is defined by $y = g^x$, where $g$ is (typically) the standard generator in the underlying group $G_q$. To encrypt a plaintext $m \in G_q$, a random exponent $s \in \mathbb{Z}_q$ is chosen and the ciphertext is computed as $\mathsf{Enc}_{(g,y)}(m, s) = (g^s, y^s m)$. A plaintext can then be recovered from a ciphertext $(u, v)$ as $\mathsf{Dec}_x(u, v) = u^{-x}v = m$. To encrypt an arbitrary string of bounded length $t$ we also need an injection $\{0,1\}^t \to G_q$, which can be efficiently computed and inverted.

**Homomorphic.** The cryptosystem is homomorphic, i.e., if $(u_1, v_1) = \mathsf{Enc}_{(g,y)}(m_1, s_1)$ and $(u_2, v_2) = \mathsf{Enc}_{(g,y)}(m_2, s_2)$ are two ciphertexts, then their element-wise product $(u_1u_2, v_1v_2) = \mathsf{Enc}_{(g,y)}(m_1m_2, s_1 + s_2)$ is an encryption of $m_1m_2$. If we set $m_2 = 1$, then this feature can be used to *re-encrypt* a ciphertext without knowledge of the randomness used to form $(u_1, v_1)$. To see this, note that for every fixed $s_1$ and random $s_2$, $s_1 + s_2$ is randomly distributed in $\mathbb{Z}_q$.

**Distributed Key Generation.** The El Gamal cryptosystem also allows efficient protocols for distributed key generation and distributed decryption of ciphertexts by $k$ parties. The $l$th party generates its own secret key $x_l$ and defines a (partial) public key $y_l = g^{x_l}$. In addition to this, the parties jointly run a protocol that verifiably secret shares the secret key $x_l$ such that a threshold $\lambda$ of the parties can recover it in the event that the $l$th party fails to do its part correctly in the joint decryption of ciphertexts. The details [3, 4] of the verifiable secret sharing scheme are not important in this document. The joint public key is then defined as $(g, y)$, where $y = \prod_{l=1}^{k} y_l$. Note that the corresponding secret key is defined by $x = \sum_{l=1}^{k} x_l$.

To jointly decrypt a ciphertext $(u, v)$, the $l$th party publishes a *partial* decryption factor $f_l$ computed as $\mathsf{PDec}_{x_l}(u, v) = u^{x_l}$ and proves using a zero-knowledge proof that it computed the decryption factor correctly relative to its public key $y_l$. If the proof is rejected, then the other parties recover the secret key $x_l$ of the $l$th party and perform its part of the joint decryption in

the open. Then the decryption factors can be combined to a joint decryption factor $f = \prod_{l=1}^{k} f_l$ such that $\mathsf{PDec}_x(u, v) = f$. The ciphertext can then be trivially decrypted as $\mathsf{TDec}((u, v), f) = v/f = m$.

**A Generalization and Useful Notation.** Using a simple hybrid argument it is easy to see that a longer plaintext $m = (m_1, \ldots, m_t) \in G_q^t$ can be encrypted by encrypting each component independently, as $\big(\mathsf{Enc}_{(g,y)}(m_1, s_1), \ldots, \mathsf{Enc}_{(g,y)}(m_t, s_t)\big)$ where $s = (s_1, \ldots, s_t)$ is an element of the product ring $\mathcal{R} = \mathbb{Z}_q^t$ or randomizers.

In our setting it is more convenient to simply view the cryptosystem as defined for elements in the product group $\mathcal{M} = G_q^t$ of plaintexts directly, i.e., we define encryption as $\mathsf{Enc}_{(g,y)}(m, s) = (g^s, y^s m)$, where $s$ is an element in the product ring $\mathcal{R} = \mathbb{Z}_q^t$. Thus, the ciphertext belongs to the ciphertext space $\mathcal{C} = \mathcal{M} \times \mathcal{M}$. Here exponentiation is distributed element-wise, e.g., $g^s = (g^{s_1}, \ldots, g^{s_t})$ and multiplication is defined element-wise. Decryption and computation of decryption factors can be defined similarly.

We remark that the Verificatum mix-net is based on this generalization of the El Gamal cryptosystem. Perhaps future versions of this document will describe this general setting, but in the current version we focus on the basic case where the randomizer ring is $\mathcal{R} = \mathbb{Z}_q$, the group of plaintexts is $\mathcal{M} = G_q$, and the group of ciphertexts is $\mathcal{C} = \mathcal{M} \times \mathcal{M} = G_q \times G_q$.

## 2.2 A Mix-Net Based on the El Gamal Cryptosystem

We use the re-encryption approach of Sako and Kilian [6] combined with the pre-computation technique in Wikström [8] and the proof of a shuffle of Terelius and Wikström [7]. The choice of proof of a shuffle is mainly motivated by the fact that all other efficient proofs of shuffles are patented. We use the batching technique of Bellare et al. [1] to speed up the proofs needed during distributed decryption.

The $k$ mix-servers first run a distributed key generation protocol such that for each $1 \leqslant l \leqslant k$, the $l$th mix-server has a public key $y_l$ and a corresponding secret key $x_l \in \mathbb{Z}_q$ which is verifiably secret shared among all $k$ mix-servers such that any set of $\lambda$ parties can recover $x_l$, but no smaller subset learns anything about $x_l$. Then they define a joint public key $(g, y)$, where $y = \prod_{l=1}^{k} y_l$, to be used by senders.

There are $N$ senders. The $i$th sender encrypts its message $m_i \in \mathcal{M}$ by picking $s_i \in \mathcal{R}$ randomly and computing a ciphertext $w_{0,i} = \mathsf{Enc}_{(g,y)}(m_i, s_i)$. To preserve privacy, the sender must also prove that it knows the plaintext $m_i$ of its ciphertext. This can be ensured in different ways, but it is of no concern in this document, since we only verify the *correctness* of an execution (and not privacy).

Recall that a non-interactive proof allows a prover to convince a verifier that a given statement is true by sending a single message. The verifier then either accepts the proof as valid or rejects it as invalid. In this context a proof is said to be zero-knowledge if, loosely, it does not reveal anything about the witness of the statement known by the prover.

The mix-servers now form a list $L_0 = (w_{0,i})_{i \in [1,N]}$ of all the ciphertexts. Then the $j$th mix-server proceeds as follows for $l = 1, \ldots, \lambda$:

- If $l = j$, then it re-encrypts each ciphertext in $L_{l-1}$, permutes the result and publishes this as $L_l$. More precisely, it chooses $r_{l,i} \in \mathcal{R}$ and a permutation $\pi$ randomly and outputs

$L_l = (w_{l,i})_{i \in [1,N]}$, where

$$w_{l,i} = w_{l-1,\pi(i)} \mathsf{Enc}_{(g,y)}(1, r_{l,\pi(i)}) \ . \tag{1}$$

Then it publishes a non-interactive zero-knowledge proof of knowledge $(\tau_l, \sigma_l)$ of all the $r_{l,i} \in \mathbb{Z}_q$ and that they satisfy (1).

- If $l \neq j$, then it waits until the $l$th mix-server publishes $L_l$ and a non-interactive zero-knowledge proof of knowledge $(\tau_l, \sigma_l)$. If the proof is rejected, then $L_l$ is set equal to $L_{l-1}$.

Finally, the mix-servers jointly decrypt the ciphertexts in $L_\lambda$ as described in Section 2.1. More precisely, the $l$th mix-server computes $f_l = \mathsf{PDec}_{x_l}(L_\lambda)$ element-wise and gives a non-interactive zero-knowledge proof $(\tau_l^{dec}, \sigma_l^{dec})$ that $f_l$ was computed correctly. If the proof is rejected, then $x_l$ is recovered using the verifiable secret sharing scheme and $f_l = \mathsf{PDec}_{x_l}(L_\lambda)$ is computed in the open. Then the output of the mix-net is computed as $\mathsf{TDec}(L_\lambda, \prod_{l=1}^{k} f_l)$, where the product and $\mathsf{TDec}(\cdot, \cdot)$ are taken element-wise.

### 2.3 Outline of the Verification Algorithm

We give a brief outline of the verification algorithm that checks that intermediate results of an execution and all the zero-knowledge proofs are consistent.

1. Check that the partial public keys are consistent with the public key used by senders to encrypt their messages, i.e., check that $y = \prod_{l=1}^{k} y_l$. If not, then **reject**.
2. Check that each mix-server re-encrypted and permuted the ciphertexts in its input or was ignored in the processing, i.e., for $l = 1, \ldots, \lambda$:
   - If $(\tau_l, \sigma_l)$ is not a valid proof of knowledge of exponents $r_{l,i}$ and a permutation $\pi$ such that $w_{l,i} = w_{l-1,\pi(i)} \mathsf{Enc}_{(g,y)}(1, r_{l,\pi(i)})$, then set $L_l = L_{l-1}$.
3. Check that each party computed its decryption factors correctly or its secret key was recovered and its decryption factors computed openly, i.e., check that for $l = 1, \ldots, k$:
   - If $x_l$ was recovered such that $y_l = g^{x_l}$, then set $f_l = \mathsf{PDec}_{x_l}(L_l)$.
   - Otherwise, if $(\tau_l^{dec}, \sigma_l^{dec})$ is not a valid proof that $f_l = \mathsf{PDec}_{x_l}(L_l)$ and $y_l = g^{x_l}$, where $f_l$ are the decryption factors computed by the $l$th mix-server, then **reject**.
4. Check if the output of the mix-net is $\mathsf{TDec}(L_\lambda, \prod_{l=1}^{k} f_l)$. If not, then **reject** and otherwise **accept**.

**On the Use of Pre-computation in Verificatum.** To speed up the mixing process, the Verificatum mix-net allows most of the computations to be done before any ciphertexts have been received. To achieve this, an upper bound $N_0$ on the number of ciphertexts is established and then the ciphertexts $\mathsf{Enc}_{(g,y)}(1, r_{l,i})$ are pre-computed. This turns costly exponentiations into cheap multiplications in the mixing phase. Furthermore, the non-interactive proof of knowledge $(\tau_l, \sigma_l)$ is split into a commitment $u_l$ of a permutation $\pi$ of $N_0$ elements, a proof of a shuffle of commitments $(\tau_l^{pos}, \sigma_l^{pos})$ that this was formed correctly, and a commitment-consistent proof

of knowledge $(\tau_l^{ccpos}, \sigma_l^{ccpos})$ of the exponents $r_{l,i}$ such that (1) holds. In other words, we may think of the complete proof of a shuffle as

$$(\tau_l, \sigma_l) = \left((u_l, \tau_l^{pos}, \tau_l^{ccpos}), (\sigma_l^{pos}, \sigma_l^{ccpos})\right) \ .$$

Only the proof $(\tau_l^{ccpos}, \sigma_l^{ccpos})$ is computed (or verified by the other mix-servers) in the mixing phase, and this is very efficient compared to the first part of the proof.

If the actual number $N$ of ciphertexts is smaller than $N_0$, then the permutation commitment $u_l$ is "shrunk" before it is used. To do this, the prover simply identifies a suitable subset of size $N$ of the elements of $u_l$ (see Section 10 for details).

We remark that when $N_0$ equals the actual number of ciphertexts $N$, then the overhead cost of dividing the work in this way instead of having single proof is small compared to the cost of a single combined proof.

# 3 How to Write a Verifier

As explained in Section 2.2, an execution of the mix-net is correct if: (1) the joint public key used by senders to encrypt their messages is consistent with the partial keys of the mix-servers, (2) the joint public key was used to re-encrypt and permute the input ciphertexts, and (3) the secret keys corresponding to the partial keys were used to compute decryption factors. To turn the outline of the verification algorithm in Section 2.3 into an actual verification algorithm, we must specify: all the parameters of the execution, the representations of all arithmetic objects, the zero-knowledge proofs, and how the Fiat-Shamir heuristic is applied.

## 3.1 List of Manageable Sub-Tasks

We divide the problem into a number of more manageable sub-tasks and indicate which steps depend on previous steps to simplify the distribution of the implementation work.

1. **Byte Trees.** All of the mathematical and cryptographic objects are represented as so called *byte trees*. Section 4 describes this simple and language-independent byte-oriented format.

2. **Cryptographic Primitives.** We need concrete implementations of hash functions, pseudo-random generators, and random oracles, and we must define how these objects are represented. This is described in Section 5.

3. **Arithmetic Library.** An arithmetic library is needed to compute with algebraic objects, e.g., group elements and field elements. These objects also need to be converted to and from their representations as byte trees. Section 6 describes how this is done.

4. **Protocol Info Files.** Some of the public parameters, e.g., auxiliary security parameters, must be extracted from an XML encoded protocol info file before any verification can take place. Section 8 describes the format of this file and which parameters are extracted.

5. **Verifying Fiat-Shamir Proofs.** The tests performed during verification are quite complex. Section 9 explains how to implement these tests.

6. **Verification of a Complete Execution.** Section 10 combines all of the above steps into a single verification algorithm.

### 3.2 How to Divide the Work

Step 1 does not depend on any other step. Step 2 and Step 3 are independent from the other steps except for how objects are encoded to and from their representation as byte trees. Step 4 can be divided into the problem of parsing an XML file and then interpreting the data stored in each XML block. The first part is independent of all other steps, and the second part depends on Step 1, Step 2 and Step 3. Step 5 depends on Step 1, Step 2, and Step 3, but not on Step 4, and it may internally be divided into separate tasks. Step 6 depends on all previous steps.

## 4 Byte Trees

We use a byte-oriented format to represent objects on file and to turn them into arrays of bytes that can be input to a hash function. The goal of this format is to be as simple as possible.

### 4.1 Definition

A byte tree is either a *leaf* containing an array of bytes, or a *node* containing other byte trees. We write $\mathsf{leaf}(d)$ for a leaf with a byte array $d$ and we write $\mathsf{node}(b_1, \ldots, b_l)$ for a node with children $b_1, \ldots, b_l$. Complex byte trees are then easy to describe.

*Example* 1. The byte tree containing the data `AF`, `03E1`, and `2D52` (written in hexadecimal) in three leaves, where the first two leaves are siblings is

$$\mathsf{node}(\mathsf{node}(\mathsf{leaf}(\texttt{AF}), \mathsf{leaf}(\texttt{03E1})), \mathsf{leaf}(\texttt{2D52})) \ .$$

### 4.2 Representation as an Array of Bytes

We use $\mathsf{bytes}_k(n)$ as a short-hand to denote the two's-complement representation of $n$ in big endian byte order truncated to the $k$ least significant bytes. We also use hexadecimal notation for constants, e.g., `0A` means $\mathsf{bytes}_1(10)$.

A byte tree is represented as an array of bytes as follows.

- A leaf $\mathsf{leaf}(d)$ is represented as the concatenation of: a single byte `01` to indicate that it is a leaf, four bytes $\mathsf{bytes}_4(l)$, where $l$ is the number of bytes in $d$, and the data bytes $d$.
- A node $\mathsf{node}(b_1, \ldots, b_l)$ is represented as the concatenation of: a single byte `00` to indicate that it is a node, four bytes $\mathsf{bytes}_4(l)$ representing the number of children, and $\mathsf{bytes}(b_1) \mid \mathsf{bytes}(b_2) \mid \cdots \mid \mathsf{bytes}(b_l)$, where $\mid$ denotes concatenation and $\mathsf{bytes}(b_i)$ denotes the representation of the byte tree $b_i$ as an array of bytes.

*Example* 2 (Example 1 contd.). The byte tree is represented as the following array of bytes.

```
00 00 00 00 02
  00 00 00 00 02
    01 00 00 00 01 AF
    01 00 00 00 02 03 E1
  01 00 00 00 02 2D 52
```

It is a good idea to hand (an upper bound of) the expected recursive depth to a parser to let it fail in a controlled way if the input does not represent a valid byte tree.

ASCII strings are converted to byte trees in the natural way, i.e., a string $s$ (an array of bytes) is converted to $\mathsf{leaf}(s)$.

*Example* 3. The string `"ABCD"` is represented by $\mathsf{leaf}(\texttt{65666768})$.

Sometimes we store byte trees as the hexadecimal encoding of their representation as an array of bytes. We denote by $\mathsf{hex}(a)$ the hexadecimal encoding of an array of bytes. We abuse notation and simply write $a$ instead of $\mathsf{bytes}(a)$ when the context is clear, e.g., if $H$ is a hash function we write $H(a)$ instead of $H(\mathsf{bytes}(a))$.

### 4.3 Backus-Naur Grammar.

The above description should suffice to implement a parser for byte trees, but for completeness we give their Backus-Naur grammar. We denote the $n$-fold repetition of a symbol $\langle \mathit{rule} \rangle$ by $n\langle \mathit{rule} \rangle$. The grammar then consists of the following rules for $n = 0, \ldots, 2^{31} - 1$, where $|$ denotes choice.

$$
\begin{aligned}
\langle \mathit{bytetree} \rangle &::= \langle \mathit{leaf} \rangle \mid \langle \mathit{node} \rangle \\
\langle \mathit{leaf} \rangle &::= \texttt{01}\langle \mathit{uint}_n \rangle \langle \mathit{data}_n \rangle \\
\langle \mathit{node} \rangle &::= \texttt{00}\langle \mathit{uint}_n \rangle \langle \mathit{bytetrees}_n \rangle \\
\langle \mathit{uint}_n \rangle &::= \mathsf{bytes}_4(n) \\
\langle \mathit{data}_n \rangle &::= n\langle \mathit{byte} \rangle \\
\langle \mathit{bytetrees}_n \rangle &::= n\langle \mathit{bytetree} \rangle \\
\langle \mathit{byte} \rangle &::= \texttt{00} \mid \texttt{01} \mid \texttt{02} \mid \cdots \mid \texttt{FF}
\end{aligned}
$$

## 5 Cryptographic Primitives

For our cryptographic library we need hash functions, pseudo-random generators, and random oracles derived from these. We must also define their functionality.

### 5.1 Hash Functions

Verificatum allows an arbitrary hash function to be used, but in this document we restrict our attention to the SHA-2 family [5], i.e., SHA-256, SHA-384, and SHA-512. Before the winner of the SHA-3 competition has been announced, we see no reason to use any other cryptographic hash function to instantiate the random oracle model. We use the following notation.

- $\mathsf{Hashfunction}(s)$ – Creates a hashfunction from `"SHA-256"`, `"SHA-384"`, or `"SHA-512"`.
- $H(d)$ – Denotes the hash digest of the byte array $d$.
- $\mathsf{outlen}(H)$ – Denotes the number of bits in the output of the hash function $H$.

*Example* 4. If $H = \mathsf{Hashfunction}(\texttt{"SHA-256"})$ and $d$ is a byte tree then $H(d)$ denotes the hash digest of the array of bytes representing the byte tree as computed by SHA-256, and $\mathsf{outlen}(H)$ equals 256.

## 5.2 Pseudo-Random Generators

We need a pseudo-random generator (PRG) to expand a short challenge string into a long "random" vector to use batching techniques in the zero-knowledge proofs of Section 9. Verificatum allows any pseudo-random generator to be used, but in the random oracle model there is no need to use a provably secure PRG. We consider a simple construction based on a hash function $H$.

The PRG takes a seed $s$ of $n_H = \mathsf{outlen}(H)$ bits as input. Then it generates a sequence of bytes $r_0 \mid r_1 \mid r_2 \mid \cdots$, where $\mid$ denotes concatenation, and $r_i$ is an array of $n_H/8$ bytes defined by

$$r_i = H(s \mid \mathsf{bytes}_4(i))$$

for $i = 0, 1, \ldots, 2^{31} - 1$, i.e., in each iteration we hash the concatenation of the seed and an integer counter (four bytes). It is not hard to see that if $H(s \mid \cdot)$ is a pseudo-random function for a random choice of the seed, then this is a provably secure construction of a pseudo-random generator. We use the following notation.

- $\mathsf{PRG}(H)$ – Creates an unseeded instance $PRG$ from a hash function $H$.
- $\mathsf{seedlen}(PRG)$ – Denotes the number of seed bits needed as input by $PRG$.
- $PRG(s)$ – Denotes an array of pseudo-random bytes derived from the seed $s$. Strictly speaking this array is $2^{31}n_H$ bits long, but we simply write $(t_0, \ldots, t_l) = PRG(s)$, where $t_i$ is of a given bit length, instead of explicitly saying that we iterate the construction a suitable number of times and then truncate to the exact output length we want.

## 5.3 Random Oracles

We need a flexible random oracle that allows us to derive any number of bits. We use a construction based on a hash function $H$. To differentiate the random oracles with different output lengths, the output length is used as a prefix in the input to the hash function. The random oracle first constructs a pseudo-random generator $PRG = \mathsf{PRG}(H)$ which is used to expand the input to the requested number of bits. To evaluate the random oracle on input $d$ the random oracle then proceeds as follows, where $n_{out}$ is the output length in bits.

1. Compute $s = H(\mathsf{bytes}_4(n_{out}) \mid d)$, i.e., compress the concatenation of the output length and the actual data to produce a seed $s$.
2. Let $a$ be the $\lceil n_{out}/8 \rceil$ first bytes in the output of $PRG(s)$.
3. Set the $8 - (n_{out} \bmod 8)$ first bits of $a$ to zero, and output the result.

We remark that setting some of the first bits of the output to zero to emulate an output of arbitrary bit length is convenient in our setting, since it allows the outputs to be directly interpreted as random integers of a given (nominal) bit length.

This construction is a secure implementation of a random oracle with output length $n_{out}$ for any $n_{out} \leqslant 2^{31}\mathsf{outlen}(H)$ when $H$ is modeled as a random oracle and the PRG of Section 5.2 is used. Note that it is unlikely to be a secure implementation if a PRG is used which is provably secure under some computational assumption.

There is no need to represent random oracles as byte trees. Thus, we only need the following notation:

- $\mathsf{RandomOracle}(H, n_{out})$ – Creates a random oracle with output length $n_{out}$ from the hash function $H$.
- $RO(d)$ – Denotes the output of the random oracle $RO$ on an input byte array $d$.

# 6 Representations of Arithmetic Objects

Every arithmetic object in Verificatum is represented as a byte tree. In this section we pin down the details of these representations.

- **Integers.** A multi-precision integer $n$ is represented by $\mathsf{leaf}(\mathsf{bytes}_k(n))$ for the smallest possible $k$.

  *Example* 5. 263 is represented by `01 00 00 00 02 01 07`.

  *Example* 6. $-263$ is represented by `01 00 00 00 02 FE F9`.

- **Arrays of Booleans.** An array $(a_1, \ldots, a_l)$ of booleans is represented as $\mathsf{leaf}(b)$, where $b = (b_1, \ldots, b_l)$ is an array of bytes where $b_i$ equals `01` if $a_i$ is true and `00` otherwise.

  *Example* 7. The array $(true, false, true)$ is represented by $\mathsf{leaf}(\texttt{01 00 01})$.

  *Example* 8. The array $(true, true, false)$ is represented by $\mathsf{leaf}(\texttt{01 01 00})$.

- **Field Elements.** An element $a$ in a prime order field $\mathbb{Z}_q$ is represented by $\mathsf{leaf}(\mathsf{bytes}_k(a))$, where $a$ is identified with its integer representative in $[0, q-1]$ and $k$ is the smallest possible $k$ such that $q$ can be represented as $\mathsf{bytes}_k(q)$. In other words, field elements are represented using fixed size byte trees, where the fixed size depends on the order of the field.

  *Example* 9. $258 \in \mathbb{Z}_{263}$ is represented by `01 00 00 00 02 01 02`.

  *Example* 10. $5 \in \mathbb{Z}_{263}$ is represented by `01 00 00 00 02 00 05`.

- **Array of Field Elements.** An array $(a_1, \ldots, a_l)$ of field elements is represented by $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_l})$, where $\overline{a_i}$ is the byte tree representation of $a_i$.

  *Example* 11. The array $(1, 2, 3)$ of elements in $\mathbb{Z}_{263}$ is represented by:

```
00 00 00 00 03
  01 00 00 00 02 00 01
  01 00 00 00 02 00 02
  01 00 00 00 02 00 03
```

- **Product Ring Element.** An element $a = (a_1, \ldots, a_l)$ in a product ring is represented by $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_l})$, where $\overline{a_i}$ is the byte tree representation of the component $a_i$. Note that this representation keeps information about the order in which a product group is formed intact (see the second example below).

  *Example* 12. The element $(258, 5) \in \mathbb{Z}_{263} \times \mathbb{Z}_{263}$ is represented by:

```
00 00 00 00 02
  01 00 00 00 02 01 02
  01 00 00 00 02 00 05
```

*Example* 13. The element $((258, 6), 5) \in (\mathbb{Z}_{263} \times \mathbb{Z}_{263}) \times \mathbb{Z}_{263}$ is represented by:

```
00 00 00 00 02
  00 00 00 00 02
    01 00 00 00 02 01 02
    01 00 00 00 02 00 06
  01 00 00 00 02 00 05
```

- **Array of Product Ring Elements.** An array $(a_1, \ldots, a_l)$ of elements in a product ring, where $a_i = (a_{i,1}, \ldots, a_{i,k})$, is represented by $\mathsf{node}(\overline{b_1}, \ldots, \overline{b_k})$, where $b_i$ is the array $(a_{1,i}, \ldots, a_{l,i})$ and $\overline{b_i}$ is its representation as a byte tree.

  Thus, the structure of the representation of an array of ring elements mirrors the representation of a single ring element. This seemingly contrived representation turns out to be convenient in implementations.

  *Example* 14. The array $\big((1, 4), (2, 5), (3, 6)\big)$ of elements in $\mathbb{Z}_{263} \times \mathbb{Z}_{263}$ is represented as

```
00 00 00 00 02
  00 00 00 00 03
    01 00 00 00 02 00 01
    01 00 00 00 02 00 02
    01 00 00 00 02 00 03
  00 00 00 00 03
    01 00 00 00 02 00 04
    01 00 00 00 02 00 05
    01 00 00 00 02 00 06
```

- **Modular Group.** A subgroup $G_q$ of order $q$ of the multiplicative group $\mathbb{Z}_p^*$, where $p > 3$ is prime, with standard generator $g$ is represented by the byte tree $\mathsf{node}(\overline{p}, \overline{q}, \overline{g}, \mathsf{bytes}_4(e))$, where the integer $e \in \{0, 1, 2\}$ determines how a string is encoded into a group element. We denote by $\mathsf{ModPGroup}(b)$ the group recovered from such a byte tree $b$.

  We may only have $e = 1$ if $p = 2q + 1$, and we may only have $e = 2$ if $p = tq + 1$ and the difference between the bit lengths of $p$ and $q$ is less than $2^{10}$.

  The three encoding schemes are defined as follows.

  - If $e = 0$, then at most 3 bytes can be encoded. Let $H$ denote SHA-256.

    **Encode.** To encode a sequence $m$ of $0 \leqslant b_m \leqslant 3$ bytes into an element $a$, find an element $a \in G_q$ such that the first $b_m + 1$ bytes of $H(\overline{a})$ are of the form $\mathsf{bytes}_1(l) \mid m$ for some $l$ such that $l \bmod 4 = b_m$.

    **Decode.** To decode an element $a \in G_q$ into a message $m$, let $\mathsf{bytes}_1(l')$ be the first byte of $H(\overline{a})$ and define $l = l' \bmod 4$. Then let $m$ be the next $l$ bytes of $H(\overline{a})$.

  - If $e = 1$, then at most $b = \lfloor (n - 2)/8 \rfloor - 4$ bytes can be encoded, where $n$ is the bit length of $p$.

**Encode.** To encode a sequence $m$ of $0 \leqslant b_m \leqslant b$ bytes as an element $a \in G_q$ do as follows. If $b_m = 0$, then set $m' = \texttt{01} \mid \mathsf{bytes}_{b-1}(0)$ and otherwise set $m' = m \mid \mathsf{bytes}_{b-b_m}(0)$. Then interpret $\mathsf{bytes}_4(b_m) \mid m'$ as a positive integer $k$. Then let $a$ be $k$ or $p - k$ so that $a \in G_q$.

**Decode.** To decode an element $a \in G_q$ to a message $m$, set $k$ equal to $a$ or $p - a$ depending on if $a < p - a$ or not. Then interpret $k \bmod 2^{8(b+4)}$ as $\mathsf{bytes}_4(l) \mid m'$, where $m'$ is a sequence of $b$ bytes. If $l < 0$ or $l > b$, then set $l = 0$. Then let $m$ be the $l$ first bytes of $m'$.

– If $e = 2$, then at most $b = \lfloor n/8 \rfloor - n' - 4$ bytes can be encoded, where $n$ is the bit length of $p$ and $n' = \lceil t/8 \rceil + 1$.

**Encode.** To encode a sequence $m$ of $0 \leqslant b_m \leqslant b$ bytes as an element $a \in G_q$, interpret $\mathsf{bytes}_4(b_m) \mid m \mid \mathsf{bytes}_{b-b_m}(0)$ as an integer $k$. Then let $a$ be the smallest integer of the form $i2^{8(b+4)} + k$ in $G_q$ for some non-negative $i$.

**Decode.** To decode an element $a \in G_q$ to a message $m$, interpret $a \bmod 2^{8(b+4)}$ as $\mathsf{bytes}_4(l) \mid m'$, where $m'$ is a sequence of $b$ bytes. If $l < 0$ or $l > b$, then set $l = 0$. Then let $m$ be the $l$ first bytes of $m'$

- **Element of Modular Group.** An element $a \in G_q$, where $G_q$ is a subgroup of order $q$ of $\mathbb{Z}_p^*$ for a prime $p$ is represented by $\mathsf{leaf}(\mathsf{bytes}_k(a))$, where $a$ is identified with its integer representative in $[0, p-1]$ and $k$ is the smallest integer such that $p$ can be represented as $\mathsf{bytes}_k(p)$.

  *Example* 15. Let $G_q$ be the subgroup of order $q = 131$ in $\mathbb{Z}_{263}^*$. Then $258 \in G_q$ is represented by `01 00 00 00 02 01 02`.

- **Array of Group Elements.** An array $(a_1, \ldots, a_l)$ of group elements is represented as $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_l})$, where $\overline{a_i}$ is the byte tree representation of $a_i$.

- **Product Group Element.** An element $a = (a_1, \ldots, a_l)$ in a product group is represented by $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_l})$, where $\overline{a_i}$ is the byte tree representation of $a_i$. This is similar to the representation of product rings.

- **Array of Product Group Elements.** An array $(a_1, \ldots, a_l)$ of elements in a product group, where $a_i = (a_{i,1}, \ldots, a_{i,k})$, is represented by $\mathsf{node}(\overline{b_1}, \ldots, \overline{b_k})$, where $b_i$ is the array $(a_{1,i}, \ldots, a_{l,i})$ and $\overline{b_i}$ is its representation as a byte tree. This is similar to the representation of arrays of elements in a product ring.

**Deriving Group Elements from Random Bits.** In Section 9.2 we need to derive group elements from the output of a pseudo-random generator $PRG$. (Strictly speaking we use $PRG$ as a random oracle here, but this is secure due to how it is defined.) Exactly how this is done depends on the group and an auxiliary security parameter $n_r$. We denote this by

$$h = (h_0, \ldots, h_{N_0-1}) = G_q.\mathsf{randomArray}(N_0, PRG(s), n_r)$$

and describe how this is defined for each type of group below. The auxiliary security parameter determines the statistical distance in distribution between a randomly chosen group element and the element derived as explained below if we assume that the output of the PRG is truly random.

We stress that it must be infeasible to find a non-trivial representation of the unit of the group in terms of these generators, i.e., it should be infeasible to find $e, e_0, \ldots, e_{N_0-1}$ not all zero such that $g^e \prod_{i=0}^{N_0-1} h_i^{e_i} = 1$. In particular, it is not acceptable to derive exponents $x_0, \ldots, x_{N_0-1} \in \mathbb{Z}_q$ and then define $h_i = g^{x_i}$.

- **Modular Group.** Let $G_q$ be the subgroup of order $q$ of the multiplicative group $\mathbb{Z}_p$, where $p > 3$ is prime. Then an array $(h_0, \ldots, h_{N_0-1})$ in $G_q$ is derived as follows from a seed $s$.

  1. Let $n_p$ be the bit length of $p$.
  2. Let $(t_0, \ldots, t_{N_0-1}) = PRG(s)$, where $t_i \in \{0,1\}^{8\lceil (n_p+n_r)/8 \rceil}$ is interpreted as a *non-negative* integer.
  3. Set $t'_i = t_i \bmod 2^{n_p+n_r}$ and let $h_i = (t'_i)^{(p-1)/q} \bmod p$.

  In other words, for each group element $h_i$ we first extract the minimum number of complete bytes $\lceil (n_p + n_r)/8 \rceil$. Then we reduce the number of bits to exactly $n_p + n_r$. Finally, we map the resulting integer into $G_q$ using the canonical homomorphism.[1]

# 7 Marshalling Group Descriptions

When groups convert themselves to byte trees in Verificatum, they do not store the name of the Java class of which they are instances. Thus, to recover a group from such a representation, information about the class must be otherwise available. When we need to recover a group from file, we store not only its internal state, but also its Java class name. Then Java reflection is used to instantiate the right class with the given internal state. This gives a reasonably language independent format, since the names of classes can always be tabulated.

We use the following notation to marshal and unmarshal objects.

- $\mathsf{marshal}(a)$ – Denotes $\mathsf{node}(\mathsf{leaf}(\texttt{"JavaClassName"}), \overline{a})$, where $a$ in the execution of the Verificatum mix-net would be an instance of `JavaClassName` and $\overline{a}$ is the byte tree representation of $a$.

- $\mathsf{unmarshal}(b)$ – Denotes the instance $a$ such that in an execution of the Verificatum mix-net we would have $b = \mathsf{marshal}(a)$.

The group description stored in the protocol info file is represented by the hexadecimal encoding of its representations as an array of bytes prepended with a brief human oriented ASCII comment describing the group. The end of the comment is indicated by double colons. If $a$ is such an hexadecimal encoding, we simply write $\mathsf{unmarshal}(a)$ and assume that the comment is removed before the string is decoded into an array of bytes, which in turn is decoded to a group.

The notation used in this document for groups correspond to Java classes in Verificatum as follows.

| Notation | Java Classname in Verificatum |
|---|---|
| ModPGroup | `verificatum.arithm.ModPGroup` |

[1] This construction makes more sense if one considers an implementation. It is natural to implement a routine that derives an array of non-negative integers $t'_i$ of a given bit length $n_p + n_r$ as explained above. Group elements are then derived from such an array in the natural way by mapping the integers into $G_q$.

# 8 Protocol Info Files

The protocol info file contains all the public parameters agreed on by the operators before the key generation phase of the mix-net is executed, and some of these parameters must be extracted to verify the correctness of an execution.

## 8.1 XML Grammar

A protocol info file uses a simple XML format and contains a single `<protocol></protocol>` block. The preamble of this block contains a number of global parameters, e.g., the number $k$ of parties executing the protocol is given by a `<nopart>`$k$`</nopart>` block, and the group over which the protocol is executed is defined by a `<pgroup>123ABC</pgroup>` block, where `123ABC` is a hexadecimal encoding of a byte tree representing the group. After the global parameters follows one `<party></party>` block for each party that takes part in the protocol, and each such block contains all the public information about that party, i.e., the name of a party is given by a `<name></name>` block. The contents of the `<party></party>` blocks are important during the execution of the protocol, but they are not used to verify the correctness of an execution and can be ignored.

A parser of protocol info files must be implemented. If `protocolInfo.xml` is a protocol info file, then we denote by $p = \mathsf{ProtocolInfo}(\texttt{protocolInfo.xml})$ an object such that $p[b]$ is the data $d$ stored in a block `<`$b$`>`$d$`</`$b$`>` in the preamble of the protocol info file, i.e., preceding any `<party></party>` block. We stress that the data is stored as ASCII encoded strings.

Listing 1 gives a skeleton example of a protocol info file, where `"123ABC"` is used as a placeholder for some hexadecimal rendition of an arithmetic or cryptographic object. Listing C in Appendix C contains a complete example of a protocol info file.

```
<protocol>

   <name>Swedish Election</name>
   <nopart>3</nopart>
   <pgroup>123ABC</pgroup>
   ...

   <party>
      <name>Party1</name>
      <pubkey>123ABC</pubkey>
      ...
   </party>
   ...
</protocol>
```

Listing 1: Skeleton of a protocol info file. There are no nested blocks within a `<party></party>` block.

Listing B in Appendix B contains the formal XML schema for protocol info files, but this is not really needed to implement a parser. In fact, to keep things simple, we do not use any attributes of XML tags, i.e., all values are stored as the data between an opening tag and a closing tag. The `vmni` command can be used to output the schema in electronic form [9].

### 8.2 Extracted Values

To interpret an ASCII string $s$ as an integer we simply write $\mathsf{int}(s)$, e.g., $\mathsf{int}(\texttt{"123"}) = 123$. We let $p = \texttt{ProtocolInfo(protocolInfo.xml)}$ and define the values we later use in Section 9 and Section 10.

- $k = \mathsf{int}(p[\texttt{nopart}])$ specifies the number of parties.
- $\lambda = \mathsf{int}(p[\texttt{thres}])$ specifies the number of mix-servers that take part in the shuffling, i.e., this is the threshold number of mix-servers that must be corrupted to break the privacy of the senders.
- $N_0 = \mathsf{int}(p[\texttt{maxciph}])$ specifies the maximal number of ciphertexts (the actual number of ciphertexts is later denoted by $N$). This is zero if no pre-computation for a maximal number of ciphertexts took place before the mix-net was executed.
- $n_e = \mathsf{int}(p[\texttt{vbitlenro}])$ specifies the number of bits in each component of random vectors used for batching in proofs of shuffles and proofs of correct decryption.
- $n_r = \mathsf{int}(p[\texttt{statdist}])$ specifies the acceptable statistical error when sampling random values. The precise meaning of this parameter is hard to describe. Loosely, randomly chosen elements in the protocol are chosen with a distribution at distance at most roughly $2^{-n_r}$ from uniform.
- $n_v = \mathsf{int}(p[\texttt{cbitlenro}])$ specifies the number of bits used in the challenge of the verifier in zero-knowledge proofs, i.e., in our Fiat-Shamir proofs it is the bit length of outputs from the random oracle $RO_v$ defined in Section 5.3.
- $H = \mathsf{Hashfunction}(p[\texttt{rohash}])$ specifies the hash function used to implement the random oracles.
- $PRG = \mathsf{PRG}\big(\mathsf{Hashfunction}(p[\texttt{prg}])\big)$ specifies the pseudo-random generator used to expand challenges into arrays.
- $G_q = \mathsf{unmarshal}(p[\texttt{pgroup}])$ specifies the underlying group.

## 9 Verifying Fiat-Shamir Proofs

We use three different Fiat-Shamir proofs: a proof of a shuffle of Pedersen commitments, a commitment-consistent proof of a shuffle of ciphertexts, and a proof of correct decryption factors. We simply write $\overline{a}$ for the byte tree representation of an object $a$.

### 9.1 Random Oracles

Throughout this section we use the following two random oracles constructed from the hash function $H$, the minimum number $n_s = \mathsf{seedlen}(PRG)$ of seed bits required by the pseudo-random generator $PRG$, and the auxiliary security parameter $n_v$.

- $RO_s = \mathsf{RandomOracle}(H, n_s)$ is the random oracle used to generate seeds to $PRG$.
- $RO_v = \mathsf{RandomOracle}(H, n_v)$ is the random oracle used to generate challenges.

## 9.2 Independent Generators

The protocols in Section 9.3 and Section 9.4 also require "independent" generators and these generators must be derived using the random oracles. To do that a seed

$$s = RO_s(\rho \mid \mathsf{leaf}(\texttt{"generators"}))$$

is computed by hashing a prefix $\rho$ derived from the protocol file and a string specifying the intended use of the "independent" generators. Then the generators are defined by

$$h = (h_0, \ldots, h_{N_0-1}) = G_q.\mathsf{randomArray}(N_0, PRG(s), n_r) \ ,$$

which is defined in Section 6. The prefix $\rho$ is computed in Step 3 of the main verification routine in Section 10.2 and given as input to Algorithm 16, Algorithm 17, and Algorithm 18 below. It is essentially a hash digest of the contents of the protocol info file. In particular this means that the "independent" generators for different underlying groups are "independently" generated, since the description of the underlying group is found in the protocol info file.

## 9.3 Proof of a Shuffle of Commitments

In the Verificatum mix-net the mix-servers commit themselves in a pre-computation phase to permutations that are later used during the mixing phase. A proof of a shuffle of commitments allows a mix-server to show that it did so correctly and that it knows how to open its commitment. Below we only describe the computations performed by the verifier for a specific application of the Fiat-Shamir heuristic. For a detailed description of the complete protocol including the computations performed by the prover we refer the reader to Appendix D and Terelius and Wikström [7].

**Algorithm 16** (Verifier of Proof of a Shuffle of Commitments)**.**

| **Input** | **Description** |
|---|---|
| $\rho$ | Prefix to random oracles. |
| $N_0$ | Size of the arrays. |
| $n_e$ | Number of bits in each component of random vectors used for batching. |
| $n_r$ | Acceptable "statistical error" when deriving independent generators. |
| $n_v$ | Number of bits in challenges. |
| $PRG$ | Pseudo-random generator $PRG$ used to derive random vectors for batching. |
| $G_q$ | Group of prime order with standard generator $g$. |
| $u$ | Array $u = (u_0, \ldots, u_{N_0-1})$ of Pedersen commitments in $G_q$. |
| $\tau$ | Commitment of the Fiat-Shamir proof. |
| $\sigma$ | Reply of the Fiat-Shamir proof. |

**Program**

1. (a) Interpret $\tau$ as $\mathsf{node}(\overline{B}, \overline{A'}, \overline{B'}, \overline{C'}, \overline{D'})$, where $A', C', D' \in G_q$, and $B = (B_0, \ldots, B_{N_0-1})$ and $B' = (B'_0, \ldots, B'_{N_0-1})$ are arrays in $G_q$.

   (b) Interpret $\sigma$ as $\mathsf{node}(\overline{k_A}, \overline{k_B}, \overline{k_C}, \overline{k_D}, \overline{k_E})$, where $k_A, k_C, k_D \in \mathbb{Z}_q$, and $k_B = (k_{B,0}, \ldots, k_{B,N_0-1})$ and $k_E = (k_{E,0}, \ldots, k_{E,N_0-1})$ are arrays in $\mathbb{Z}_q$.

   Reject if this fails.

2. Compute a seed $s = RO_s\big(\rho \mid \mathsf{node}(\overline{g}, \overline{h}, \overline{u})\big)$.

3. Set $(t_0, \ldots, t_{N_0-1}) = PRG(s)$, where $t_i \in \{0,1\}^{8\lceil n_e/8 \rceil}$ is interpreted as a non-negative integer $0 \leqslant t_i < 2^{8\lceil n_e/8 \rceil}$, set $e_i = t_i \bmod 2^{n_e}$ and compute

$$A = \prod_{i=0}^{N_0-1} u_i^{e_i} \ .$$

4. Compute a challenge $v = RO_v\big(\rho \mid \mathsf{node}(\mathsf{leaf}(s), \tau)\big)$ interpreted as a non-negative integer $0 \leqslant v < 2^{n_v}$.

5. Compute

$$C = \frac{\prod_{i=0}^{N_0-1} u_i}{\prod_{i=0}^{N_0-1} h_i} \quad \text{and} \quad D = \frac{B_{N_0-1}}{h_0^{\prod_{i=0}^{N_0-1} e_i}} \ ,$$

   set $B_{-1} = h_0$, and accept if and only if:

$$A^v A' = g^{k_A} \prod_{i=0}^{N_0-1} h_i^{k_{E,i}} \qquad C^v C' = g^{k_C}$$

$$B_i^v B'_i = g^{k_{B,i}} B_{i-1}^{k_{E,i}} \ \text{ for } i = 0, \ldots, N_0 - 1 \qquad D^v D' = g^{k_D}$$

## 9.4 Commitment-Consistent Proof of a Shuffle of Ciphertexts

During the mixing in Verificatum each mix-server re-encrypts the ciphertexts in its input, permutes the resulting ciphertexts using the permutation it is committed to, and then outputs the result. Then it uses a commitment-consistent proof of a shuffle to show that it did so correctly.

We only describe a specific implementation of the verifier using the Fiat-Shamir heuristic. For a detailed description of the complete protocol including the computations performed by the prover we refer the reader to Appendix D and Wikström [8].

**Algorithm 17** (Verifier of Commitment-Consistent Proof of a Shuffle)**.**

| **Input** | **Description** |
|---|---|
| $\rho$ | Prefix to random oracles. |
| $N$ | Size of the arrays. |
| $n_e$ | Number of bits in each component of random vectors used for batching. |
| $n_r$ | Acceptable "statistical error" when deriving independent generators. |
| $n_v$ | Number of bits in challenges. |
| $PRG$ | Pseudo-random generator $PRG$ used to derive random vectors for batching. |
| $G_q$ | Group of prime order. |
| $u$ | Shrunk array $u = (u_0, \ldots, u_N)$ of Pedersen commitments in $G_q$. |
| $\mathcal{R}$ | Randomizer group. |
| $\mathcal{C}$ | Ciphertext group. |
| $pk$ | El Gamal public key. |
| $w$ | Array $w = (w_0, \ldots, w_{N-1})$ of input ciphertexts in $\mathcal{C}$. |
| $w'$ | Array $w' = (w'_0, \ldots, w'_{N-1})$ of output ciphertexts in $\mathcal{C}$. |
| $\tau$ | Commitment of the Fiat-Shamir proof. |
| $\sigma$ | Reply of the Fiat-Shamir proof. |

**Program**

1. (a) Interpret $\tau$ as $\mathsf{node}(\overline{A'}, \overline{B'})$, where $A' \in G_q$ and $B' \in \mathcal{C}$.

   (b) Interpret $\sigma$ as $\mathsf{node}(\overline{k_A}, \overline{k_B}, \overline{k_E})$, where $k_A \in \mathbb{Z}_q$, $k_B \in \mathcal{R}$, and $k_E$ is an array of $N$ elements in $\mathbb{Z}_q$.

   Reject if this fails.

2. Compute a seed $s = RO_s\big(\rho \mid \mathsf{node}(\overline{g}, \overline{h}, \overline{u}, \overline{pk}, \overline{w}, \overline{w'})\big)$.

3. Set $(t_0, \ldots, t_{N-1}) = PRG(s)$, where $t_i \in \{0,1\}^{8\lceil n_e/8 \rceil}$ is interpreted as a non-negative integer $0 \leqslant t_i < 2^{8\lceil n_e/8 \rceil}$, set $e_i = t_i \bmod 2^{n_e}$ and compute

$$A = \prod_{i=0}^{N-1} u_i^{e_i} \ .$$

4. Compute a challenge $v = RO_v\big(\rho \mid \mathsf{node}(\mathsf{leaf}(s), \tau)\big)$ interpreted as a non-negative integer $0 \leqslant v < 2^{n_v}$.

5. Compute $B = \prod_{i=0}^{N-1} w_i^{e_i}$ and accept if and only if:

$$A^v A' = g^{k_A} \prod_{i=0}^{N-1} h_i^{k_{E,i}} \qquad B^v B' = \mathsf{Enc}_{pk}(1, -k_B) \prod_{i=0}^{N-1} (w'_i)^{k_{E,i}}$$

### 9.5 Proof of Correct Decryption Factors

At the end of the mixing the parties jointly decrypt the re-encrypted and permuted list of ciphertexts. To prove that they did so correctly they use a proof of correct decryption factors. This is a standard protocol using batching for improved efficiency. The general technique originates in Bellare et al. [1].

**Algorithm 18** (Verifier of Decryption Factors)**.**

| **Input** | **Description** |
|---|---|
| $\rho$ | Prefix to random oracles. |
| $N$ | Size of the arrays. |
| $n_e$ | Number of bits in each component of random vectors used for batching. |
| $n_r$ | Acceptable "statistical error" when deriving independent generators. |
| $n_v$ | Number of bits in challenges. |
| $PRG$ | Pseudo-random generator $PRG$ used to derive random vectors for batching. |
| $G_q$ | Group of prime order. |
| $y$ | Partial public key. |
| $\mathcal{C}$ | Ciphertext group. |
| $\mathcal{M}$ | Plaintext group. |
| $w$ | Array $w = (w_0, \ldots, w_{N-1})$ of input ciphertexts in $\mathcal{C}$. |
| $f$ | Array $f = (f_0, \ldots, f_{N-1})$ of decryption factors in $G_q$. |
| $\tau$ | Commitment of the Fiat-Shamir proof. |
| $\sigma$ | Reply of the Fiat-Shamir proof. |

**Program**

1. (a) Interpret $\tau$ as $\mathsf{node}(\overline{y'}, \overline{B''})$, where $y' \in G_q$ and $B''$ is an array containing a single element $B'$ in $\mathcal{M}$.

   (b) Interpret $\sigma$ as $\overline{k_x}$, where $k_x \in \mathbb{Z}_q$.

   Reject if this fails.

2. Compute a seed $s = RO_s\big(\rho \mid \mathsf{node}(\mathsf{node}(\overline{g}, \overline{w}), \mathsf{node}(\overline{y}, \overline{f}))\big)$.

3. Set $(t_0, \ldots, t_{N-1}) = PRG(s)$, where $t_i \in \{0,1\}^{8\lceil n_e/8 \rceil}$ is interpreted as a non-negative integer $0 \leqslant t_i < 2^{8\lceil n_e/8 \rceil}$, set $e_i = t_i \bmod 2^{n_e}$ and compute

$$A = \prod_{i=0}^{N-1} w_i^{e_i} \quad \text{and} \quad B = \prod_{i=0}^{N-1} f^{e_i} \ .$$

4. Compute a challenge $v = RO_v\big(\rho \mid \mathsf{node}(\mathsf{leaf}(s), \tau)\big)$ interpreted as a non-negative integer $0 \leqslant v < 2^{n_v}$.

5. Accept if and only if $y^v y' = g^{k_x}$ and $B^v B' = \mathsf{PDec}_{k_x}(A)$.

## 10 Verification

The verification algorithm must verify that the input ciphertexts were repeatedly re-rerandomized by the mix-servers and then jointly decrypted with a secret key corresponding to the public key

used by senders to encrypt their messages. Furthermore, the parameters of the execution must match the parameters in the protocol info file.

The verification algorithm must take two parameters.

- `protocolInfo.xml` – Protocol info file containing the public parameters. Section 8 describes the format of this file and Section 8.2 introduces notation for the values we need.
- `roProof` – Proof directory containing the intermediate results and the Fiat-Shamir proofs relating the intermediate results. The contents of this directory is described below.

### 10.1 Contents of the Proof Directory

The proof directory contains not only the Fiat-Shamir proofs, but also the intermediate results. In this section we describe the formats of these files and introduce notation for their contents. Here $\langle l \rangle$ denotes a integer parameter $0 \leqslant l \leqslant k$ encoded using two decimal digits representing the index of a mix-server, but a file with suffix $l$ may not originate from the $l$th mix-server if it is corrupted and all types of files do not appear with all suffixes. For easy reference we tabulate the notation introduced below and from which file the contents is derived in Table 1.

1. FullPublicKey.bt – Full public key used to form input ciphertexts.
2. PublicKey$\langle l \rangle$.bt – Partial public key of the $l$th mix-server. The required format of this file is a byte tree $\overline{y_l}$, where $y_l \in G_q$.
3. SecretKey$\langle l \rangle$.bt – Secret key file of the $l$th party. This is only created if the $l$th mix-server is identified as a cheater. In this case its secret key is recovered and its part in the joint decryption is computed locally by the other mix-servers. The required format of this file is a byte tree $\overline{x_l}$, where $x_l \in \mathbb{Z}_q$.
4. CiphertextList$\langle l \rangle$.bt – The $l$th intermediate list of ciphertexts, i.e., normally the output of the $l$th mix-server. This file should contain a byte tree $\overline{L_l}$, where $L_l$ is an array of $N$ elements in $\mathcal{C}$ and $N \leqslant N_0$ is the number of elements in the list $L_0$ of input ciphertexts.
5. PermutationCommitment$\langle l \rangle$.bt – Commitment to a permutation. This file should contain a byte tree $\overline{u_l}$, where $u_l$ is an array of $N_0$ elements in $G_q$.
6. PoSCommitment$\langle l \rangle$.bt – "Proof commitment" of the proof of a shuffle of commitments. The required format of the byte tree $\tau_l^{pos}$ in this file is specified in Algorithm 16.
7. PoSReply$\langle l \rangle$.bt – "Proof reply" of the proof of a shuffle of commitments. The required format of the byte tree $\sigma_l^{pos}$ in this file is specified in Algorithm 16.
8. KeepList$\langle l \rangle$.bt – Keep-list used to shrink a permutation-commitment if pre-computation is used before the mix-net is executed. The file should contain a byte tree $\overline{t_l}$, where $t_l$ should be an array of $N_0$ booleans, of which exactly $N$ are true, indicating which components to keep.
9. CCPoSCommitment$\langle l \rangle$.bt – "Proof commitment" of the commitment-consistent proof of a shuffle. The required format of the byte tree $\tau_l^{ccpos}$ in this file is specified in Algorithm 17.

10. CCPoSReply$\langle l \rangle$.bt – "Proof reply" of the commitment-consistent proof of a shuffle. The required format of the byte tree $\sigma_l^{ccpos}$ in this file is specified in Algorithm 17.

11. DecryptionFactors$\langle l \rangle$.bt – Decryption factors of the $l$th mix-server used to jointly decrypt the shuffled ciphertexts. This file should contain a byte tree node$(\overline{y_l}, \overline{f_l})$, where $f_l$ is an array of $N$ elements in $G_q$.

12. DecrFactCommitment$\langle l \rangle$.bt – "Proof commitment" of the proof of correctness of the decryption factors. The required format of the byte tree $\tau_l^{dec}$ of this file is specified in Algorithm 18.

13. DecrFactReply$\langle l \rangle$.bt – "Proof reply" of the proof of correctness of the decryption factors. The required format of the byte tree $\tau_l^{dec}$ of this file is specified in Algorithm 18.

14. PlaintextElements.bt – The output plaintext elements that has not been decoded in any way. This file should contain a byte tree $\overline{m}$, where $m$ is an array of $N$ elements in $\mathcal{M}$.

| Notation | Point | File |
|---|---|---|
| $(g, y)$ | 1 | FullPublicKey.bt |
| $y_l$ | 2 | PublicKey$\langle l \rangle$.bt |
| $x_l$ | 3 | SecretKey$\langle l \rangle$.bt |
| $L_l$ | 4 | CiphertextList$\langle l \rangle$.bt |
| $u_l$ | 5 | PermutationCommitment$\langle l \rangle$.bt |
| $\tau_l^{pos}$ | 6 | PoSCommitment$\langle l \rangle$.bt |
| $\sigma_l^{pos}$ | 7 | PoSReply$\langle l \rangle$.bt |
| $t_l$ | 8 | KeepList$\langle l \rangle$.bt |
| $\tau_l^{ccpos}$ | 9 | CCPoSCommitment$\langle l \rangle$.bt |
| $\sigma_l^{ccpos}$ | 10 | CCPoSReply$\langle l \rangle$.bt |
| $f_l$ | 11 | DecryptionFactors$\langle l \rangle$.bt |
| $\tau_l^{dec}$ | 12 | DecrFactCommitment$\langle l \rangle$.bt |
| $\tau_l^{dec}$ | 13 | DecrFactReply$\langle l \rangle$.bt |
| $m$ | 14 | PlaintextElements.bt |

Table 1: Correspondence between notation and files containing the denoted objects.

## 10.2 Core Verification Algorithm

We are finally ready to describe the core verification algorithm. We stress that this algorithm only verifies that a protocol info file and the contents of the proof directory are correct. In an application the public key actually given to senders, the input ciphertexts, and the output plaintexts may be represented using an application-dependent format and it must be verified that these correspond to their counterparts in the proof directory. We discuss this in Section 10.3.

1. **Verification of Protocol Info File.** The soundness of the parameters of the execution must be verified manually by a cryptographer. If any parameter is not chosen with an acceptable security level, then **reject**. Verify that the XML of the protocol info file is compliant with the XML schema of protocol info files and **reject** otherwise.

2. **Public Parameters.** Read the public parameters from the protocol info file as described in Section 8. If this fails, then **reject**. This defines

$$k, N_0, n_r, n_v, n_e, PRG, G_q, H \ .$$

3. **Prefix to Random Oracles.** To differentiate executions on different public parameters, we let $\rho = \mathsf{leaf}(H(f))$, where $f$ is the array of bytes contained in the protocol info file, and use this as a prefix to our random oracles.

4. **Joint Public Key.** Attempt to read the joint public key $(g, y) \in G_q \times G_q$ from file as described in Point 1. If this fails, then **reject**.

5. **Individual Public Keys.** Read public keys $y_1, \ldots, y_k$ as described in Point 2. If this fails, then **reject**. Then test if $y = \prod_{l=1}^{k} y_l$, i.e., check that the keys $y_1, \ldots, y_k$ of the mix-servers are consistent with the public key $y$. If not, then **reject**.

6. **Array of Input Ciphertexts.** Read the array $L_0$ of $N \leqslant N_0$ ciphertexts as described in Point 4 for some $N$. If this fails, then **reject**.

7. **Proofs of Shuffles.** For $l = 1, \ldots, \lambda$ do:

   (a) **Verify Proof of Shuffle of Commitments.** Read a permutation commitment $u_l$, a proof commitment $\tau_l^{pos}$, and proof reply $\sigma_l^{pos}$ as described in Point 5, Point 6, and Point 7, respectively. Then execute Algorithm 16 on input $(\rho, N_0, n_e, n_r, n_v, PRG, G_q, u_l, \tau_l^{pos}, \sigma_l^{pos})$. If reading fails or if the algorithm rejects, then set $u_l = h$.

   (b) **Shrink Permutation Commitment.**

      i. Read the keep-list $t_l$ as described in Point 8. If this fails, then let $t_l$ be the array of $N_0$ booleans of which the first $N$ are true and the rest false.

      ii. Set $u_l = (u_{l,i})_{t_{l,i}=true}$ be the sub-array indicated by $t_l$.

   (c) **Array of Ciphertexts.** Read the array $L_l$ of $N$ ciphertexts in $\mathcal{C}$ as described in Point 4. If this fails, then **reject**.

   (d) **Verify Commitment-Consistent Proof of Shuffle.** Read proof commitment $\tau_l^{ccpos}$ and proof reply $\sigma_l^{ccpos}$ as described in Point 9 and Point 10, respectively. Then execute Algorithm 17 on input

   $$(\rho, N_0, N, n_e, n_r, n_v, PRG, G_q, u_l, \mathcal{R}, \mathcal{C}, L_{l-1}, L_l, pk, \tau_l^{ccpos}, \sigma_l^{ccpos}) \ .$$

   If reading fails or if the algorithm rejects, then verify that $L_l = L_{l-1}$. If this is not the case, then **reject**.

8. **Proofs of Decryption.** For $l = 1, \ldots, k$ do:

   (a) If $x_l$ can be read as described in Point 3 such that $y_l = g^{x_l}$, then set $f_l = \mathsf{PDec}_{x_l}(L_{l-1})$.

   (b) Otherwise, read $f_l$, $\tau_l$, and $\sigma_l$ of the $l$th mix-server as described in Point 11, Point 12, and Point 13, respectively. Then execute Algorithm 18 on input

   $$(\rho, N, n_e, n_r, n_v, PRG, G_q, u, \mathcal{C}, \mathcal{M}, y, L_\lambda, f_l, \tau_l, \sigma_l) \ .$$

   If it rejects, then **reject**.

9. **Verify Output.** Attempt to read the plaintext elements $m$ as described in Point 14. Then verify that $m = \mathsf{TDec}\big(L_\lambda, \prod_{l=1}^{k} f_l\big)$.

**Standard Command-Line Interface of Core Verification Algorithm.** To simplify the implementation of scripts that execute multiple verifiers that are implemented by independent parties, we require that every verifier can be used as follows, where `vmnv` denotes the verification algorithm command, `protInfo.xml` is the protocol info file, and `roProof` is the proof directory.

```
vmnv protInfo.xml roProof
```

The command should by default give no output and its exit status should be zero if the proof is accepted and non-zero otherwise. It should also have an option `-v` that activates verbose output, but it is up to the implementer to decide on the format used in the output. It is acceptable to assume that the option appears before any of the parameters.

### 10.3 Verification Algorithm

The formats used to represent the public key handed to the senders and the list of ciphertexts received from senders and how plaintext group elements are decoded into messages is application dependent. Thus, to verify the overall correctness, it must be verified that:

1. The public key actually used by senders is $(g, y)$.
2. The list of ciphertexts actually submitted by senders is $L_0$.
3. The output plaintexts are correctly decoded from the group elements in $m$.

This falls outside the scope of this document, since in general we can not anticipate the representations used by application developer or how the plaintext group elements are interpreted.

However, one natural approach is to use an existing or custom mix-net interface of Verificatum. In the user manual of the Verificatum mix-net [9] we describe how this can be done.

**Standard Command-Line Interface of Verification Algorithm.** Regardless of the approach used, we require that the verifier implements a standard command-line interface to allow easy scripting. Let `vmnv` denote the verification algorithm command, let `protInfo.xml` be the protocol info file, let `roProof` be the proof directory, let `publicKey` contain the public key, let `ciphertexts` contain the input ciphertexts, and let `plaintexts` contain the decoded plaintexts. Then we require that the following runs the verification algorithm.

```
vmnv protInfo.xml roProof pkey ciphertexts plaintexts
```

The `-v` option must activate verbose output in the same way as for the core verification command. We stress that every verifier should also implement the core verifier command-line interface.

## 11 Acknowledgments

Olivier Pereira gave valuable feedback on an initial version of this document. The suggestions of Tomer Hasid, Shahram Khazaei, Gunnar Kreitz, and Amnon Ta-Shma have greatly improved the presentation.

## References

[1] M. Bellare, J. Garay, and T. Rabin. Batch verification with applications to cryptography and checking. In *LATIN*, volume 1380 of *Lecture Notes in Computer Science*, pages 170–191. Springer Verlag, 1998.

[2] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[3] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 427–438. IEEE Computer Society Press, 1987.

[4] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptology*, 20(1):51–83, 2007.

[5] N. I. of Standards and T. (NIST). Secure hash standard. Federal Information Processing Standards Publication 180-2, 2002. `http://csrc.nist.gov/`.

[6] K. Sako and J. Kilian. Reciept-free mix-type voting scheme. In *Advances in Cryptology – Eurocrypt '95*, volume 921 of *Lecture Notes in Computer Science*, pages 393–403. Springer Verlag, 1995.

[7] B. Terelius and D. Wikström. Proofs of restricted shuffles. In *Africacrypt 2010*, volume 6055 of *Lecture Notes in Computer Science*, pages 100–113, 2010.

[8] D. Wikström. A commitment-consistent proof of a shuffle. In *ACISP*, volume 5594 of *Lecture Notes in Computer Science*, pages 407–421. Springer Verlag, 2009.

[9] D. Wikström. User manual for the Verificatum mix-net. Manuscript, 2011. Available at `http://www.verificatum.org`.

[10] D. Wikström. The Verificatum mix-net. Manuscript, 2011. In preparation.

# A Test Vectors for the Cryptographic Library

PRG(Hashfunction(leaf("SHA-256")))

Seed (32 bytes):

```
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
```

Expansion (128 bytes):

```
70f4003d52b6eb03da852e93256b5986b5d4883098bb7973bc5318cc66637a84
04a6950a06d3e3308ad7d3606ef810eb124e3943404ca746a12c51c7bf776839
0f8d842ac9cb62349779a7537a78327d545aaeb33b2d42c7d1dc3680a4b23628
627e9db8ad47bfe76dbe653d03d2c0a35999ed28a5023924150d72508668d244
```

PRG(Hashfunction(leaf("SHA-384")))

Seed (48 bytes):

```
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
202122232425262728292a2b2c2d2e2f
```

Expansion (128 bytes):

```
e45ac6c0afff343b268d4cbd773328413672a764df99ab823b53074d94152bd
27fc38bcffdb7c1dc1b6a3656b2d4819352c482da40aad3b37f333c7afa81a92
b7b54551f3009efa4bdb8937492c5afca1b141c99159b4f0f819977a4e10eb51
61edd4b1734717de4106f9c184a17a9b5ee61a4399dd755f322f5d707a581cc1
```

PRG(Hashfunction(leaf("SHA-512")))

Seed (64 bytes):

```
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
202122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f
```

Expansion (128 bytes):

```
979043771043f4f8e0a2a19b1fbfbe5a8f076c2b5ac003e0b9619e0c45faf767
47295734980602ec1d8d3cd249c165b7db62c976cb9075e35d94197c0f06e1f3
97a45017c508401d375ad0fa856da3dfed20847716755c6b03163aec2d9f43eb
c2904f6e2cf60d3b7637f656145a2d32a6029fbda96361e1b8090c9712a48938
```

RandomOracle(Hashfunction(leaf("SHA-256")), 65)

Input (32 bytes):

```
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
```

Output (9 bytes of which the last 65 bits may be non-zero):

```
001a8d6b6f65899ba5
```

RandomOracle(Hashfunction(leaf("SHA-256")), 261)
Input (32 bytes):
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
Output (33 bytes of which the last 261 bits may be non-zero):
1c04f57d5f5856824bca3af0ca466e283593bfc556ae2e9f4829c7ba8eb76db8
78

RandomOracle(Hashfunction(leaf("SHA-384")), 93)
Input (32 bytes):
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
Output (12 bytes of which the last 93 bits may be non-zero):
04713a5e22935833d436d1db

RandomOracle(Hashfunction(leaf("SHA-384")), 411)
Input (32 bytes):
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
Output (52 bytes of which the last 411 bits may be non-zero):
00dc086c320e38b92722a9c0f87f2f5de81b976400e2441da542d1c3f3f391e4
1d6bcd8297c541c2431a7272491f496b622266aa

RandomOracle(Hashfunction(leaf("SHA-512")), 111)
Input (32 bytes):
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
Output (14 bytes of which the last 111 bits may be non-zero):
28d742c34b97367eb968a3f28b6c

RandomOracle(Hashfunction(leaf("SHA-512")), 579)
Input (32 bytes):
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
Output (73 bytes of which the last 579 bits may be non-zero):
00a6f79b8450fef79af71005c0b1028c9f025f322f1485c2b245f658fe641d47
dcbb4fe829e030b52e4a81ca35466ad1ca9be6feccb451e7289af318ddc9dae0
98a5475d6119ff6fe0

# B Schema for Protocol Info Files

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
<xs:element name="protocol">
<xs:complexType>
<xs:sequence>

<xs:element name="version"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="sid"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="name"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="descr"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="nopart"
            type="xs:integer"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="thres"
            type="xs:integer"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="pgroup"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="width"
            type="xs:integer"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="inter"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="maxciph"
            type="xs:integer"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="statdist"
            type="xs:integer"
```

```
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="cbitlen"
            type="xs:integer"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="cbitlenro"
            type="xs:integer"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="vbitlen"
            type="xs:integer"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="vbitlenro"
            type="xs:integer"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="prg"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="rohash"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="corr"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>


<xs:element name="party"
            minOccurs="0"
            maxOccurs="unbounded">
<xs:complexType>
<xs:sequence>

<xs:element name="srtbyrole"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="name"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="pdescr"
```

```
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="pkey"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="http"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

<xs:element name="hint"
            type="xs:string"
            minOccurs="1"
            maxOccurs="1"/>

</xs:sequence>
</xs:complexType>
</xs:element>

</xs:sequence>
</xs:complexType>
</xs:element>

</xs:schema>
```

# C Example Protocol Info File

```
<!-- ATTENTION! WE STRONGLY ADVICE AGAINST EDITING THIS FILE!

     This is a protocol information file. It contains all the
     parameters of a protocol session as agreed by all parties.

     Each party must hold an identical copy of this file. DO NOT
     UNDER ANY CIRCUMSTANCES EDIT THIS FILE after you and the
     administrators of the other parties have agreed on its content.
     If you do, then there are no security guarantees.

     If the Fiat-Shamir heuristic is used, then DO NOT EDIT THIS
     FILE UNDER ANY CIRCUMSTANCES! Any Fiat-Shamir proof of
     correctness computed before this file was changed will be
     rejected after the change. This can not be corrected without
     access to the original version of this file! -->

<protocol>

   <!-- Protocol version for which this protocol info is intended. -->
   <version>0.1</version>

   <!-- Session identifier of this protocol execution. This must be
        globally unique for all executions of all protocols. -->
   <sid>SID</sid>
```

```
   <!-- Name of this protocol execution. This is a short descriptive
        name. -->
   <name>Swedish Election</name>

   <!-- Description of this protocol execution. This is a longer
        description than the name. -->
   <descr></descr>

   <!-- Number of parties. -->
   <nopart>3</nopart>

   <!-- Number of parties needed to violate privacy. This must at
        least be majority. -->
   <thres>2</thres>

   <!-- Group over which the protocol is executed. An instance of
        verificatum.arithm.PGroup. -->
   <pgroup>ModPGroup(safe-prime modulus=2*order+1. order bit-length = 512
)::0000000002010000001c766572696669636174756d2e61726974686d2e4d6f64504772
6f7570000000000401000000041014354c848a190b7b5fbddcd07bed36e59af5a50cc5966b
202bba0959ccc42061a2f468f87fa436451bd48d5cb333c0bb0aca763193e70c725495455
a99939276f010000004100a1aa642450c85bdafdeee683df69b72cd7ad28662cb359015dd
04ace6621030d17a347c3fd21b228dea46ae5999e05d85653b18c9f386392a4aa2ad4cc9c
93b701000000410132027413c1464af9b3ebe05f40059902857843365887f3e084e973dfd
3da198697724ac422dfce4728c2baa07760b5eae2d709bd7ff4f79d4e71fc9c2d37e26701
0000000400000001</pgroup>

   <!-- Number of ciphertexts shuffled in parallel. -->
   <width>1</width>

   <!-- Interface that defines how to communicate with the mix-net.
        Possible values are "raw", "native", "helios", and "tvs", or
        the name of a subclass of verificatum.protocol.mixnet.
        MixNetElGamalInterface. See the user manual for more
        information on interfaces. -->
   <inter>native</inter>

   <!-- Maximal number of ciphertexts for which precomputation is
        performed. If this is set to zero, then it is assumed that
        precomputation is not performed as a separate phase, i.e., it
        defaults to the number of submitted ciphertexts during mixing.
        -->
   <maxciph>10000</maxciph>

   <!-- Decides statistical error in distribution. -->
   <statdist>100</statdist>

   <!-- Bit length of challenges in interactive proofs. -->
   <cbitlen>100</cbitlen>

   <!-- Bit length of challenges in non-interactive random-oracle
        proofs. -->
   <cbitlenro>200</cbitlenro>

   <!-- Bit length of each component in random vectors used for
```

```
        batching. -->
   <vbitlen>100</vbitlen>

   <!-- Bit length of each component in random vectors used for
        batching in non-interactive random-oracle proofs. -->
   <vbitlenro>200</vbitlenro>

   <!-- Pseudo random generator used to derive random vectors from
        jointly generated seeds. This can be one of the strings "SHA-
␣␣␣␣␣␣␣␣256", "SHA-384", or "SHA-512", in which case verificatum.
        crypto.PRGHeuristic is instantiated based on this
        hashfunction, or it can be an instance of verificatum.crypto.
        PRG. -->
   <prg>SHA-256</prg>

   <!-- Hashfunction used to implement random oracles. It can be one
        of the strings "SHA-256", "SHA-384", or "SHA-512", in which
        case verificatum.crypto.HashfunctionHeuristic is is
        instantiated, or an instance of verificatum.crypto.
        Hashfunction. Random oracles with various output lengths are
        then implemented, using the given hashfunction, in verificatum.
        crypto.RandomOracle.
        WARNING! Do not change the default unless you know exactly
        what you are doing. -->
   <rohash>SHA-256</rohash>

   <!-- Determines if the proofs of correctness of an execution are
        interactive or non-interactive ("interactive" or
        "noninteractive"). -->
   <corr>noninteractive</corr>

   <party>

      <!-- Sorting attribute used to sort parties with respect to their
           roles in the protocol. This is used to assign roles in
           protocols where different parties play different roles. -->
      <srtbyrole>anyrole</srtbyrole>

      <!-- Name of party. -->
      <name>Party1</name>

      <!-- Description of party. -->
      <pdescr></pdescr>

      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048
)::00000000020100000029766572696669636174756d2e63727970746f2e5369676e6174
757265504b65794865757269737469630000000002010000000400000800010000012630 8
20122300d06092a864886f70d01010105000382010f003082010a0282010100a60c4934ff
f6b7a7dc50ec6a7f99bad0df62862b4c5f94e09b27ef71ccab1db011323e63caf0996deb7
ea4a904267e68284871a767a3b8df328b936ff69499534e14c1713b1bcbd3a142961dafc3
606f3b9b166128b2646c1085e388b830bb8d9fed72e2f9ca44c73822823e9c5104d656dcc
d975ad038050b46a388c27078f05ffdf8024d2f3332438ec712561dad9f9e4b817c39212b
dae83249ee119e5beba3726dff5bb9b83de44cf99666258a68ae300cf480867110ec29576
8d5a83618e978ae532d0eb3806a622c57cf58fee0ce1275f56a1cbf0c9eac1dbf475ec872
48f29858d7e9a008fd8d4204449bc4d39716d5797bba2ec19736c366f0bcb62d020301000
```

```
1</pkey>

      <!-- URL to our HTTP server. -->
      <http>http://mybox1.mydomain1.com:8080</http>

      <!-- Socket address given as <hostname>:<port> to our hint server.
           A hint server is a simple UDP server that reduces latency and
           traffic on the HTTP servers. -->
      <hint>mybox1.mydomain1.com:4040</hint>

   </party>

   <party>

      <!-- Sorting attribute used to sort parties with respect to their
           roles in the protocol. This is used to assign roles in
           protocols where different parties play different roles. -->
      <srtbyrole>anyrole</srtbyrole>

      <!-- Name of party. -->
      <name>Party2</name>

      <!-- Description of party. -->
      <pdescr></pdescr>

      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048
)::00000000020100000029766572696669636174756d2e63727970746f2e5369676e6174
757265504b65794865757269737469630000000002010000004000008000100000126308
20122300d06092a864886f70d01010105000382010f003082010a0282010100bc7264e6f2
e75ac5e5db3b1b4211355459e18fc5a00d07d34991735e972b7f3fb25c04d5c126cad5fef
21b9f5e9b80490e8072dac707f52b0c8de25d35bbf9e60186b1ed26fd6aaa820a6f4a22ad
2bb519300933daafa1433f3657c13ac7136c646f3feae2675ecaddb0664b7471dd620af65
530cd4d910a101e5a173360f33bb4bd2cc14240810b5047bff5eae886887a143797d6aa2c
c768764ea70ddd42799dcb42105511d758812bd56bd2948c4898abea03feb89ed5e53f581
15702c17dc33d535cb5a73a649493a1d7a52fac1122773470b9c5870d6639cf0f7ff06ca7
573775d60c208b53fec49a37e5dc98a2bc8474e6b16478165e8add68bbbf509b020301000
1</pkey>

      <!-- URL to our HTTP server. -->
      <http>http://mybox2.mydomain2.com:8080</http>

      <!-- Socket address given as <hostname>:<port> to our hint server.
           A hint server is a simple UDP server that reduces latency and
           traffic on the HTTP servers. -->
      <hint>mybox2.mydomain2.com:4040</hint>

   </party>

   <party>

      <!-- Sorting attribute used to sort parties with respect to their
           roles in the protocol. This is used to assign roles in
           protocols where different parties play different roles. -->
      <srtbyrole>anyrole</srtbyrole>
```

```
      <!-- Name of party. -->
      <name>Party3</name>

      <!-- Description of party. -->
      <pdescr></pdescr>

      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048
)::00000000020100000029766572696669636174756d2e63727970746f2e5369676e6174
757265504b65794865757269737469630000000002010000004000008000100000126308
20122300d06092a864886f70d01010105000382010f003082010a028201010086e99acd4c
48873c7cd002f5cfd5087d9ea85c1802ea058e54f66010c568411ce4e2b62da41ccbc12eb
085c2a6db04186f901acf85f9ed9ba25f8b2297d9cc25518806f71f91774f39051ce02bda
9be111c7ddda9bb124043466646d3c5a357654509b9b34190bc07660daa09d0d8373113d8
34f9a3b161e93f199be26d934228ee2abb1c1c9a77f8063d03662847f128a733902687902
69fa7b242880db10464b4be055d0192c19ccd101c7d6e83f43935a99436850ebf282a58d5
3071067bfd7d6652bf1605856143a36b6fffac5babbf423c611dd3ae7e07a1dfc6b031082
b33f50854d288ebfb82dd152155b62445d7877604c3811f0c2a015ed9bdeef29020301000
1</pkey>

      <!-- URL to our HTTP server. -->
      <http>http://mybox3.mydomain3.com:8080</http>

      <!-- Socket address given as <hostname>:<port> to our hint server.
           A hint server is a simple UDP server that reduces latency and
           traffic on the HTTP servers. -->
      <hint>mybox3.mydomain3.com:4040</hint>

   </party>

</protocol>
```

# D Protocols

**Protocol 19** (Proof of a Shuffle of Commitments)**.**
**Common Input.** Generators $g, h_0, \ldots, h_{N-1} \in G_q$ and Pedersen commitments $u_0, \ldots, u_{N-1} \in G_q$.
**Private Input.** Exponents $r = (r_0, \ldots, r_{N-1}) \in \mathbb{Z}_q^N$ and a permutation $\pi \in \mathbb{S}_N$ such that $u_i = g^{r_{\pi(i)}} h_{\pi(i)}$ for $i = 0, \ldots, N-1$.

1. $\mathcal{V}$ chooses a seed $s \in \{0,1\}^n$ randomly, defines $e \in [0, 2^{n_e} - 1]^N$ as $e = \mathsf{PRG}(s)$, hands $s$ to $\mathcal{P}$ and computes,

$$A = \prod_{i=0}^{N-1} u_i^{e_i} \ .$$

2. $\mathcal{P}$ computes the following, where $e'_i = e_{\pi^{-1}(i)}$:

   (a) *Bridging Commitments.* It chooses $b_0, \ldots, b_{N-1} \in \mathbb{Z}_q$ randomly, sets $B_{-1} = h_0$, and forms

$$B_i = g^{b_i} B_{i-1}^{e'_i} \text{ for } i = 0, \ldots, N-1 \ .$$

   (b) *Proof Commitments.* It chooses $\alpha, \beta_0, \ldots, \beta_{N-1}, \gamma, \delta \in \mathbb{Z}_q$ and $\epsilon_0, \ldots, \epsilon_{N-1} \in [0, 2^{n_e+n_v+n_r} - 1]$ randomly, sets $B_{-1} = h_0$, and forms

$$\begin{aligned} A' &= g^{\alpha} \prod_{i=0}^{N-1} h_i^{\epsilon_i} & C' &= g^{\gamma} \\ B'_i &= g^{\beta_i} B_{i-1}^{\epsilon_i} \text{ for } i = 0, \ldots, N-1 & D' &= g^{\delta} \ . \end{aligned}$$

   Then it hands $(B, A', B', C', D')$ to $\mathcal{V}$.

3. $\mathcal{V}$ chooses $v \in [0, 2^{n_v} - 1]$ randomly and hands $v$ to $\mathcal{P}$.

4. $\mathcal{P}$ computes $a = \langle r, e' \rangle$, $c = \sum_{i=0}^{N-1} r_i$. Then it sets $d_0 = b_0$ and computes $d_i = b_i + e'_i d_{i-1}$ for $i = 1, \ldots, N-1$. Finally, it sets $d = d_{N-1}$ and computes

$$\begin{aligned} k_A &= va + \alpha & k_C &= vc + \gamma \\ k_{B,i} &= vb_i + \beta_i \text{ for } i = 0, \ldots, N-1 & k_D &= vd + \delta \\ & & k_{E,i} &= ve'_i + \epsilon_i \text{ for } i = 0, \ldots, N-1 \ . \end{aligned}$$

   Then it hands $(k_A, k_B, k_C, k_D, k_E)$ to $\mathcal{V}$.

5. $\mathcal{V}$ computes

$$C = \frac{\prod_{i=0}^{N-1} u_i}{\prod_{i=0}^{N-1} h_i} \quad \text{and} \quad D = \frac{B_{N-1}}{h_0^{\prod_{i=0}^{N-1} e_i}} \ ,$$

   sets $B_{-1} = h_0$ and accepts if and only if

$$\begin{aligned} A^v A' &= g^{k_A} \prod_{i=0}^{N-1} h_i^{k_{E,i}} & C^v C' &= g^{k_C} \\ B_i^v B'_i &= g^{k_{B,i}} B_{i-1}^{k_{E,i}} \text{ for } i = 0, \ldots, N-1 & D^v D' &= g^{k_D} \end{aligned}$$

**Protocol 20** (Commitment-Consistent Proof of a Shuffle)**.**

**Common Input.** Generators $g, h_0, \ldots, h_{N-1} \in G_q$, Pedersen commitments $u_0, \ldots, u_{N-1} \in G_q$, a public key $pk$, elements $w_0, \ldots, w_{N-1} \in \mathcal{C}$ and $w'_0, \ldots, w'_{N-1} \in \mathcal{C}$.

**Private Input.** Exponents $r = (r_0, \ldots, r_{N-1}) \in \mathbb{Z}_q^N$, a permutation $\pi \in \mathbb{S}_N$, and exponents $s = (s_0, \ldots, s_{N-1}) \in \mathcal{R}^N$ such that $u_i = g^{r_{\pi(i)}} h_{\pi(i)}$ and $w'_i = \mathsf{Enc}_{pk}(1, s_{\pi^{-1}(i)}) w_{\pi^{-1}(i)}$ for $i = 0, \ldots, N-1$.

1. $\mathcal{V}$ chooses a seed $s \in \{0,1\}^n$ randomly, defines $e \in [0, 2^{n_e} - 1]^N$ as $e = \mathsf{PRG}(s)$, hands $s$ to $\mathcal{P}$ and computes

$$A = \prod_{i=0}^{N-1} u_i^{e_i} \ .$$

2. $\mathcal{P}$ chooses $\alpha \in \mathbb{Z}_q$, $\epsilon_0, \ldots, \epsilon_{N-1} \in [0, 2^{n_e+n_v+n_r} - 1]$, and $\beta \in \mathcal{R}$ randomly and computes

$$A' = g^{\alpha} \prod_{i=0}^{N-1} h_i^{\epsilon_i} \quad \text{and} \quad B' = \mathsf{Enc}_{pk}(1, -\beta) \prod_{i=0}^{N-1} (w'_i)^{\epsilon_i} \ .$$

   Then it hands $(A', B')$ to $\mathcal{V}$.

3. $\mathcal{V}$ chooses $v \in [0, 2^{n_v} - 1]$ randomly and hands $v$ to $\mathcal{P}$.

4. Let $e'_i = e_{\pi^{-1}(i)}$. $\mathcal{P}$ computes $a = \langle r, e' \rangle$, $b = \langle s, e \rangle$, and

$$k_A = va + \alpha \ , \quad k_B = vb + \beta \ , \quad \text{and} \quad k_{E,i} = ve'_i + \epsilon_i \ \text{ for } i = 0, \ldots, N-1 \ .$$

   Then it hands $(k_A, k_B, k_E)$ to $\mathcal{V}$.

5. $\mathcal{V}$ computes $B = \prod_{i=0}^{N-1} w_i^{e_i}$ and accepts if and only if

$$A^v A' = g^{k_A} \prod_{i=0}^{N-1} h_i^{k_{E,i}} \qquad B^v B' = \mathsf{Enc}_{pk}(1, -k_B) \prod_{i=0}^{N-1} (w'_i)^{k_{E,i}}$$