How To Implement A Stand-alone Verifier for the Verificatum Mix-Net

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

October 28, 2011

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 universally verifiable proofs of correctness during the execution of the protocol. This document gives a detailed description of these proofs targeting implementors of standalone verifiers.

Contents

1 Introduction

The Verificatum mix-net (VMN) [**?**] 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 allow 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 standalone verifier for such proofs.

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

TODO(This must be rewritten to avoid double use of w.)

The cryptosystem is defined over a group G_q of prime order q. A secret key $skx \in \mathbb{Z}_q$ is sampled randomly, and a corresponding public key $pk = (g, y)$ 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 $\mathsf{Enc}_{pk}(m, s) = (g^s, y^s m)$. A plaintext can then be recovered from a ciphertext $w = (w_0, w_1)$ as $\mathsf{Dec}_{sk}(w) = w_0^{-x} w_1 = m$. To encrypt an arbitrary string of a given length t we also need an injection $\{0,1\}^t \to G_q$, which can be efficiently computed and inverted, to convert a string into a group element before encrypting the group element.

Homomorphic. This cryptosystem is homomorphic, i.e., if $w_1 = \mathsf{Enc}_{pk}(m_1, s_1)$ and $w_2 = \mathsf{Enc}_{pk}(m_2, s_2)$ are two ciphertexts, then their element-wise product $w_1 w_2 = \mathsf{Enc}_{pk}(m_1 m_2, s_1 + s_2)$ is an encryption of $m_1 m_2$. 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 w_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. The lth party generates its own secret key x_l and defines a (partial) public key $y_l = g^{x_l}$ with a corresponding secret key $x = \sum_{l=1}^{k} x_l$. In addition to this, the parties jointly run a protocol that verifiably secret shares the secret key x_l such that a threshold λ of the parties can recover it in the event that the lth party fails to do its part correctly in the joint decryption of ciphertexts. The details of the verifiable secret sharing protocol are not important in this document. The joint public key is then defined as (g, y), where $y = \prod_{l=1}^{k} y_l$.

To jointly decrypt a ciphertext w, the lth party publishes a *partial* decryption factor v_l computed as $\mathsf{PDec}_{x_l}(w) = w_0^{x_l}$ 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 lth party and perform his part of the joint decryption. Then the decryption factors can be combined to a joint decryption factor $v = \prod_{l=1}^{k} v_l$ such that $\mathsf{PDec}_x(w) = v$. The ciphertext can then be trivially decrypted as $\mathsf{TDec}(w, v) = w_1/v = m$.

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 $\big(\mathsf{Enc}_{pk}(m_1, r_1), \ldots, \mathsf{Enc}_{pk}(m_t, r_t)\big)$ where $r = (r_1, \ldots, r_t)$ 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 $\mathsf{Enc}_{pk}(m, r) = (g^r, y^r m)$, where r 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., $g^r = (g^{r_1}, \ldots, g^{r_t})$ and multiplication is defined element-wise. Decryption and computation of decryption factors can be defined similarly.

We remark that Verificatum is based on this generalization of the El Gamal cryptosystem. Perhaps future versions of this document will cover this, 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

The mix-servers first run a distributed key generation protocol such that for each $1 \leq l \leq k$, the lth 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 λ parties can recover x_l, but no smaller subset learns anything about x_l. Then they define a joint public key $pk = (g, y)$, where $y = \prod_{l=1}^{k} y_l$, to be used by senders.

The ith sender encrypts its message $m_i \in G_q$ by picking s_i randomly and computing a ciphertext $w_{0,i} = \mathsf{Enc}_{pk}(m_i, s_i)$. 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.

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

- If $l = j$, then the jth mix-server re-encrypts each ciphertext in L_{l-1}, permutes the result and publishes this as L_l. More precisely, it chooses $r_{l,i} \in \mathbb{Z}_q$ and a permutation π randomly and outputs $L_l = (w_{l,i})_{i \in [1,N]}$, where

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

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

- If $l \neq j$, then the jth mix-server waits until the lth mix-server publishes L_l and a non-interactive zero-knowledge proof of knowledge (π_l, σ_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_λ as described in Section 2.1. More precisely, the lth mix-server computes $v_l = \mathsf{PDec}_{x_l}(L_\lambda)$ and gives a non-interactive zero-knowledge proof $(\pi_l^{dec}, \sigma_l^{dec})$ that v_l was computed correctly. If the proof is rejected, then x_l is recovered using the verifiable secret sharing scheme and $v_l = \mathsf{PDec}_{x_l}(L_\lambda)$ is computed. Then the output of the mix-net is computed as $\mathsf{TDec}(L_\lambda, \prod_{l=1}^{k} v_l)$.

Verifying the Correctness of an Execution. The goal of this document is to describe in detail how such an execution can be verified. There are many parameters to consider and the representations of all objects must be specified, but the following is an outline of the verification algorithm.

1. The partial public keys are consistent with the public key used by senders to encrypt their messages, i.e., $y = \prod_{l=1}^{k} y_l$.
2. Each mix-server re-encrypted and permuted the ciphertexts in its input, i.e., for $l = 1, \ldots, \lambda$:
   - If (π_l, σ_l) is a valid proof of knowledge of exponents $r_{l,i}$ and a permutation π such that $w_{l,i} = w_{l-1,\pi(i)}\mathsf{Enc}_{pk}(1, r_{l,\pi(i)})$.
   - Otherwise, $L_l = L_{l-1}$.
3. Each party computed its decryption factors correctly, i.e., for $l = 1, \ldots, k$:
   - If x_l was recovered such that $y_l = g^{x_l}$, then v_l is defined as $v_l = \mathsf{PDec}_{x_l}(L_l)$.
   - Otherwise, $(\pi_l^{dec}, \sigma_l^{dec})$ is a valid proof that $v_l = \mathsf{PDec}_{x_l}(L_l)$, where $y_l = g^{x_l}$.
4. The output of the mix-net is $\mathsf{TDec}(L_\lambda, \prod_{l=1}^{k} v_l)$.

The Use of Pre-computation in Verificatum. To speed up the mixing process, Verificatum 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 needed and then the ciphertexts $\mathsf{Enc}_{pk}(1, r_{l,i})$ are precomputed. Furthermore, the non-interactive proof of knowledge (π_l, σ_l) is split into a commitment u_l of a permutation π of N_0 elements, a proof $(\pi_l^{pos}, \sigma_l^{pos})$ that this was formed correctly, and a proof of knowledge $(\pi_l^{ccpos}, \sigma_l^{ccpos})$ of the exponents $r_{l,i}$ such that (1). If the actual number N of ciphertexts is smaller than N_0, then the permutation commitment u_l is "shrunk" before it is used.

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 messsages 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. We must specify the public parameters, the formats used to store objects on file, the representations of all intermediate results, 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 subtasks 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 byte oriented format.

2. **Cryptographic Primitives.** We need concrete implementations of hashfunctions, 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 and derived from sequences of bytes in a well-defined way to use the Fiat-Shamir heuristic. Section 6 describes how this is done.

4. **The Protocol Info File.** 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 compartmentalize and implement these tests.

6. **Verification of a Complete Execution.** The contents of the proof directory produced during an execution of the mix-net consists not only of the Fiat-Shamir proofs, but also of the public keys of the mix-servers and intermediate results from the execution of the mix-net. To verify the overall correctness of an execution it must be verified that these are consistent and that all individual Fiat-Shamir proofs are correct using the tests implemented in Step 5. This is detailed in Section 10.

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 from 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 contents. 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 (see Section 9 for details). 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 hashfunction. 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 data d and we write $\mathsf{node}(c_1, \ldots, c_l)$ for a node with children $c_1, \ldots, c_l$. 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 represented by

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

4.2 Representation as an Array of Bytes

A byte tree is represented as an array of bytes as follows. We use $\mathsf{bytes}_k(n)$ as a short-hand to denote the two's-complement representation of $n \bmod 2^{8k}$ as an $8k$-bit integer, i.e., $\mathsf{bytes}_k(n)$ is the representation of the integer n (with overflow) as k bytes in big endian byte order. For positive n, we drop k from our notation and simply write $\mathsf{bytes}(n)$ where k is chosen to be as small as possible. We also use hexadecimal notation for constants, e.g., 0A means $\mathsf{bytes}_1(10)$. If a is a byte tree, then we write $\mathsf{bytes}(a)$ for its representation as an array of bytes defined by the following.

- 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 $\mathsf{node}(c_1, \ldots, c_l)$ 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 $\mathsf{bytes}(c_1) \mid \mathsf{bytes}(c_2) \mid \cdots \mid \mathsf{bytes}(c_l)$, i.e., the concatenation of the representations of the children $c_1, \ldots, c_l$.

An integer n is represented by the byte tree $\mathsf{leaf}(\mathsf{bytes}(n))$. ASCII strings are converted to byte trees in the natural way, i.e., a string s (an array of bytes) is converted to $\mathsf{leaf}(s)$. Looking forward, in Section 5 and Section 6 we describe how cryptographic and other arithmetic objects are represented as byte trees.

Sometimes we store byte trees as the hexadecimal encoding of their representation as an array of bytes. We denote by hex(a) the hexadecimal encoding of an array of bytes.

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 $\langle \textit{rule} \rangle$ by $n\langle \textit{rule} \rangle$. The grammar then consists of the following rules for $n = 0, \ldots, 2^{32} - 1$.

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

5 Cryptographic Primitives

For our cryptographic library we need hashfunctions, pseudo-random generators, and random oracles derived from these. It does not suffice to simply state how these objects are represented as byte trees. We must also define their functionality.

5.1 Hashfunctions

Verificatum allows an arbitrary hashfunction to be used, but in this document we restrict our attention to 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 hashfunction in the random oracle model. We use the following notation.

- $\overline{H}$ denotes the byte tree representation[1] of an instance H. This is defined as one of the byte trees $\mathsf{leaf}(\text{"SHA-256"})$, $\mathsf{leaf}(\text{"SHA-384"})$, or $\mathsf{leaf}(\text{"SHA-512"})$ depending on the hashfunction.
- $\mathsf{Hashfunction}(\overline{H})$ – Creates an instance H from its byte tree representation $\overline{H}$.
- $H(d)$ – Denotes the hash digest of the byte array d.
- $\mathsf{outlen}(H)$ – Denotes the number of bits in the output of the hashfunction H.

For example, if $H = \mathsf{Hashfunction}(\mathsf{leaf}(\text{"SHA-256"}))$ and d is an 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 $\mathsf{outlen}(H) = 256$.

5.2 Pseudo-Random Generators

Our verifier is deterministic except that it might execute probabilistic testing of the parameters of the underlying group. However, we still 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 to reduce outside dependencies and keep this document as simple as possible we consider a single construction based on a hashfunction H.

The PRG takes a seed s of $n = \mathsf{outlen}(H)$ bits as input. Then it generates a sequence of bytes $t_0 \mid t_1 \mid t_2 \mid \cdots$, where $\mid$ denotes concatenation, and t_i is an array of $n/8$ bytes defined by

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

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.

- $\overline{PRG}$ denotes the byte tree representation of an instance PRG. This is defined as $\overline{H}$, i.e., the byte tree representation of the underlying hashfunction.

[1] It may seem overly complicated to, e.g., use a byte tree representation of the string "SHA-256" to configure the mix-net to use this hashfunction, but Verificatum can be configured with an arbitrary hashfunction, which is why an approach like this is needed.

- $\mathsf{PRG}(r)$ – Creates an unseeded instance PRG from its byte tree representation $\overline{PRG}$ or from a hashfunction 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, but strictly speaking this array is $2^{31}n$ bits long. 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 hashfunction H. To differentiate the random oracles with different output lengths, the output length is used as a prefix in the input to the hashfunction. 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 l is the output length in bits.

1. Compute $s = H(\mathsf{bytes}_4(l) \mid d)$, i.e., compress the concatenation of the output length and the actual data to produce a seed.
2. Let a be the $\lceil (n_{out} + 7)/8 \rceil$ 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.

We use the following notation:

- $\mathsf{RandomOracle}(H, l)$ – Creates a random oracle based the hashfunction H with output length l.
- $RO(d)$ – 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 2. 17 is represented by 01 00 00 00 02 01 01.

  Example 3. -17 is represented by 01 00 00 00 02 FF EF.

- **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 the integer representative of a 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 4. $18 \in \mathbb{Z}_{19}$ is represented by 01 00 00 00 02 01 02.

Example 5. $5 \in \mathbb{Z}_{19}$ 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 6. The array $(1, 2, 3)$ in $\mathbb{Z}_{19}$ 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 7. The element $(18, 6) \in \mathbb{Z}_{19} \times \mathbb{Z}_{19}$ is represented by:

  00 00 00 00 02
  01 00 00 00 02 01 02
  01 00 00 00 02 00 06

  Example 8. The element $((18, 6), 5) \in (\mathbb{Z}_{19} \times \mathbb{Z}_{19}) \times \mathbb{Z}_{19}$ 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 9. The array $\big((1,4),(2,5),(3,6)\big)$ 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
```

- **Product Group. TODO(write this with PPGroup notation)**
- **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.
- **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 $\mathsf{01}$ if a_i is true and $\mathsf{00}$ otherwise.

  Example 10. The array $(true, false, true)$ is represented by $\mathsf{leaf}(\mathsf{01\,00\,01})$.

  Example 11. The array $(true, true, false)$ is represented by $\mathsf{leaf}(\mathsf{01\,01\,00})$.

Representation of Modular Groups and Their Elements TODO(this remains...)

7 Marshalling Root Objects

When objects convert themselves to byte trees in Verificatum, they do not store any information about of which Java class they are instances. Thus, to recover an object from such a representation information about the class must be otherwise available, e.g., the byte tree representation of a field element can only be understood in the context of a given field.

For some so called "root objects" no such context exists. Thus, in Verificatum, such objects store not only their internal state, but also the their 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 translated. The details of the scheme is best explained by an example.

Example 12. A wrapper of the SHA-2 family of hashfunctions is provided by the Java class verificatum.crypto.HashfunctionHeuristic. The internal state of an instance of this class simply consists of the name of the underlying algorithm, e.g., the string "SHA-256". As explained in

Section 5.1, such an instance is converted to a byte tree leaf("SHA-256"). The complete byte tree is then

$$\mathsf{node}(\mathsf{leaf}(\text{"verificatum.crypto.HashfunctionHeuristic"}), \mathsf{leaf}(\text{"SHA-256"})) \ .$$

Fortunately, we need only parse byte trees of this more elaborate types for a few of our classes. We have the following mapping of Java class names and the notation used here.

Notation	Java Classname in Verificatum
Hashfunction	verificatum.crypto.HashfunctionHeuristic
PRG	verificatum.crypto.PRGHeuristic
ModPGroup	verificatum.arithm.ModPGroup
PPGroup	verificatum.arithm.PPGroup

To summarize we need the following notation to marshal and unmarshal objects of the above types.

- marshal(a) – Where a is an instance of ClassName with corresponding Java class name JavaClassName denotes $\mathsf{node}(\mathsf{leaf}(\text{"JavaClassName"}), \overline{a})$, where $\overline{a}$ is the byte tree representation of a.
- unmarshal(b) – Denotes the instance a such that $b = \mathsf{marshal}(a)$ if one exists.

Root objects stored in the protocol info file are represented by hexadecimal encodings of their representations as arrays of bytes prepended with a brief human oriented ASCII comment describing the root object. The end of the comment is indicated by double colons (see Listing 1 for an example). If a is such an hexadecimal encoding, we simply write 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 root object.

```
HashfunctionHeuristic(SHA-256)::00000000020100000028766572696696396174756d2e63727970746f2e4861736866756e6374696f6e486575726973746963010000000753
48412d323536
```

Listing 1: Hexadecimal encoding of HashfunctionHeuristic("SHA-256") with leading comment briefly describing the payload.

8 The Protocol Info File

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 overall 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 $a = \mathsf{ProtocolInfo}(\mathsf{protocolInfo.xml})$ an object such that $a[b]$ is the data d stored in a block d in the preamble of the protocol info file, i.e., preceeding any <party></party> block. We stress that the data is stored as ASCII encoded strings.

Listing 2 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 4 in Appendix B contains a complete example of a info file.

```
<protocol>

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

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

Listing 2: Skeleton of a protocol info file. There are no nested blocks within a <party></party> block.

Listing 3 in Appendix A 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 inbetween an opening tag and a closing tag.

8.2 Extracted Values

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

- $\mathrm{SID} = p[\text{"sid"}]$ is the globally unique session identifier.
- $k = \mathsf{int}(p[\text{"nopart"}])$ specifies the number of parties.
- $\lambda = \mathsf{int}(p[\text{"thres"}])$ 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.

- $N_0 = \mathsf{int}(p[\text{"maxciph"}])$ specifies the maximal number of ciphertexts (the actual number of ciphertexts is later denoted by N). This is zero if no precomputation for a maximal number of ciphertexts took place before the mix-net was executed.
- $n_e = \mathsf{int}(p[\text{"vbitlenro"}])$ specifies the number of bits in each component of random vectors used for batching in proofs of shuffles and proofs of correct decryption.
- $n_r = \mathsf{int}(p[\text{"statdist"}])$ specifies the acceptable statistical error when sampling random values.
- $n_v = \mathsf{int}(p[\text{"cbitlenro"}])$ 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.
- $PRG = \mathsf{unmarshal}(p[\text{"prg"}])$ specifies the pseudo-random generator used to expand challenges into arrays.
- $G_q = \mathsf{unmarshal}(p[\text{"pgroup"}])$ specifies the underlying group.
- $H = \mathsf{unmarshal}(p[\text{"rohash"}])$ specifies the hashfunction used to implement the random oracles.

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 hashfunction 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

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

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

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

The prefix ρ is computed in Step 3 of the main verification routine in Section 10.2 and given as input to Algorithm 13, Algorithm 14, and Algorithm 15 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.

Exactly how the pseudo-random bits are turned into group elements depend on the underlying group G_q. For modular groups, random integers (of suitable size) can simply be mapped into the group using the canonical homomorphism (see Section 6). We stress that it must be infeasible to find a non-trivial representation of the unit of the group in terms of these generators. In particular, it is not acceptable to generate pseudo-random exponents $x_0, \ldots, x_{N_0-1} \in \mathbb{Z}_q$ and then define $h_i = g^{x_i}$.

9.3 Proof of a Shuffle of Commitments

In Verificatum the mix-servers commit themselves in a pre-computation phase to permutations used during the mixing. 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 C and Terelius and Wikström [**?**].

Algorithm 13 (Verifier of Proof of a Shuffle of Commitments)**.**

Input	**Description**
ρ	Prefix to random oracles.
l	Index of the prover.
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.
π	Commitment of the Fiat-Shamir proof.
σ	Reply of the Fiat-Shamir proof.

Program

1. (a) Interpret π 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 σ 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 $s = RO_s\big(\rho \mid \mathsf{node}(\mathsf{leaf}(\mathsf{bytes}_4(l)), \overline{g}, \overline{h}, \overline{u})\big)$.

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 $0 \leq t_i < 2^{8\lceil n_e/8 \rceil}$, 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 $v = RO_v\big(\rho \mid \mathsf{node}(\mathsf{leaf}(s), \pi)\big)$ interpreted as a non-negative integer $0 \leq v < 2^{n_v}$.

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 C and Wikström [**?**].

Algorithm 14 (Verifier of Commitment-Consistent Proof of a Shuffle)**.**

Input	**Description**
ρ	Prefix to random oracles.
l	Index of the prover.
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	Array $u = (u_0, \ldots, u_i)$ 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 $\mathcal{C}$.
π	Commitment of the Fiat-Shamir proof.
σ	Reply of the Fiat-Shamir proof.

Program

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

   (b) Interpret σ 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 $s = RO_s\big(\rho \mid \mathsf{node}(\mathsf{leaf}(\mathsf{bytes}_4(l)), \overline{g}, \overline{h}, \overline{u}, \overline{pk}, \overline{w}, \overline{w'})\big)$.

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 $0 \leq t_i < 2^{8\lceil n_e/8 \rceil}$, set $e_i = t_i \bmod 2^{n_e}$ and compute

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

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

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. [**?**].

Algorithm 15 (Verifier of Decryption Factors)**.**

Input	**Description**
ρ	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 $\mathcal{C}$.
w'	Array $w' = (w'_0, \ldots, w'_{N-1})$ of output ciphertexts in $\mathcal{C}$.
π	Commitment of the Fiat-Shamir proof.
σ	Reply of the Fiat-Shamir proof.

Program

1. (a) Interpret π as $\mathsf{node}(\overline{y'}, \overline{B'})$, where $y' \in G_q$ and $B' \in \mathcal{M}$.

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

   Reject if this fails.

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

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 $0 \leq t_i < 2^{8\lceil n_e/8 \rceil}$, set $e_i = t_i \bmod 2^{n_e}$ and compute

$$A = \prod_{i=0}^{N-1} w_i^{e_i} \quad \text{and} \quad B = \prod_{i=0}^{N-1} (w'_i)^{e_i} \ .$$

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

5. Accept if and only if

$$y^v y' = g^{k_x} \quad \text{and} \quad B^v B' = \mathsf{PDec}_{k_x}(A) \ .$$

10 Verification of a Complete Execution

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.

10.1 Inputs to the Verification Algorithm

The verification algorithm must take the following inputs.

- protocolInfo.xml – Protocol info file containing the public parameters.
- publicKey – File containing the El Gamal public key used by senders to encrypt their input messages.
- ciphertexts – File containing the input ciphertexts.
- plaintexts – File containing the output plaintexts.
- roProof – Directory containing the intermediate results and the Fiat-Shamir proofs relating the intermediate results.

Protocol Info. The protocol info file contains the public parameters of the execution. Section 8 describes the format of this file and Section 8.2 introduces notation for the values we need.

Public Key. To allow a client to encrypt its input without parsing the protocol info file, the public key file contains not only the public key itself, but also the group over which it is defined. More precisely, it contains a hexadecimal string of the form $\mathsf{hex}(\mathsf{node}(\mathsf{marshal}(\mathcal{C}), \overline{pk}))$, **TODO(check that this is correct)** where $\mathcal{C}$ is the group of ciphertexts and $pk \in \mathcal{C}$ is the actual public key used by senders to encrypt their messages.

Ciphertexts. The ciphertext file is a newline separated list of distinct ASCII encoded strings. From these strings an array L_0 of elements in $\mathcal{C}$ are extracted. A string s results in an element $w \in \mathcal{C}$ in the array if an only if it is of the form $\mathsf{hex}(\overline{w})$. All other strings are ignored.

Plaintexts. The plaintext file is a newline separated list of ASCII encoded strings without any carriage return characters. We denote the sorted list of these strings by s.

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 an integer parameter $1 \leq l \leq k$ representing the index of a mix-server, but if a mix-server is corrupted a file is not necessarily computed by the lth mix-server. Furthermore, some files only exist for a subset of the indices.

If a file does not satisfy the required format, then we set the contents to the special symbol $\perp$ to indicate that the file can not be parsed correctly.

- PermutationCommitment$\langle l \rangle$.bt – Commitment to a permutation. This file should contain a bytetree $\overline{u_l}$, where u_l is an array of N_0 elements in G_q.
- PoSCommitment$\langle l \rangle$.bt – "Proof commitment" for of the proof of a shuffle of commitments. The required format of the byte tree π_l^{pos} in this file is specified in Algorithm 13.

- PoSReply$\langle l \rangle$.bt – "Proof reply" of the proof of a shuffle used to show that the permutation commitment is correctly formed. The required format of the byte tree σ_l^{pos} in this file is specified in Algorithm 13.
- CiphertextList$\langle l \rangle$.bt – The lth intermediate list of ciphertexts, i.e., the output of the lth 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 $N \leq N_0$ is the number of elements in L_l.
- KeepList$\langle l \rangle$.bt – Keep-list used to shrink a permutation commitment if precomputation 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.
- CCPoSCommitment$\langle l \rangle$.bt – "Proof commitment" of the commitment-consistent proof of a shuffle. The required format of the byte tree π_l^{ccpos} in this file is specified in Algorithm 14.
- CCPoSReply$\langle l \rangle$.bt – "Proof reply" of the commitment-consistent proof of a shuffle used to show that the ciphertexts are processed and then permuted according to the permutation committed to. The required format of the byte tree σ_l^{ccpos} in this file is specified in Algorithm 14.
- DecryptionFactors$\langle l \rangle$.bt – Decryption factors of the lth mix-server used to jointly decrypt the shuffled ciphertexts. This file should contain a byte tree $\overline{v_l}$, where v_l is an array of N elements in G_q.
- DecrFactCommitment$\langle l \rangle$.bt – "Proof commitment" of the proof of correctness of the decryption factors. The required format of the byte tree π_l^{dec} of this file is specified in Algorithm 15.
- DecrFactReply$\langle l \rangle$.bt – "Proof reply" of of the proof of correctness of the decryption factors. The required format of the byte tree π_l^{dec} of this file is specified in Algorithm 15.
- SecretKey$\langle l \rangle$.bt – Secret key file of the lth party. This is only created if the lth 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.

10.2 Verification Algorithm

We are finally ready to describe the algorithm used to verify the overall correctness.

1. **Manual 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**.
2. **Public Parameters.** Read the public parameters from the protocol info file as described in Section 8. This defines SID, 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.

4. **Initialize Global Prefix to Random Oracles.** Compute $d = H(a)$, where a is the contents of the file protocolInfo.xml as an array of bytes, and then set the global prefix of the random oracles by executing $\mathsf{RandomOracle.setGlobalPrefix}(d)$. This ensures that each execution of the mix-net has "independent" random oracles.

5. **Joint Public Key.** Attempt to read the joint public key $pk \in \mathcal{C}$ as described in Section 10.1. If this fails, then **reject**.

6. **Array of Input Ciphertexts.** Attempt to read the array L_0 of $N \leq N_0$ ciphertexts as described in Section 10.1. If this fails, then **reject**.

7. **Individual Public Keys.** Read public keys $y_1, \ldots, y_k$ as described in Section 10.1. Reject if this fails. 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**.

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

   (a) **Verify Proof of Shuffle of Commitments.** Execute Algorithm 13 on input $(\rho, l, N_0, n_e, n_r, n_v, PRG, G_q, u_l, \pi_l^{pos}, \sigma_l^{pos})$. If it rejects, then set $u_l = h$.

   (b) **Shrink Permutation Commitment.**

      i. Attempt to read the keep-list t_l as described in Section 10.1. 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.** Attempt to read the array L_l of N elements in $\mathcal{C}$ as described in Section 10.1. If this fails, then **reject**.

   (d) **Verify Commitment-Consistent Proof of Shuffle.** Execute Algorithm 14 on input $(\rho, l, N_0, n_e, n_r, n_v, PRG, G_q, u_l, \mathcal{R}, \mathcal{C}, L_{l-1}, L_l, pk, \pi_l^{ccpos}, \sigma_l^{ccpos})$. If it rejects, then set $L_l = L_{l-1}$.

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

   (a) If x_l can be read as described in Section 10.1 such that $y_l = g^{x_l}$, then set $v_l = \mathsf{PDec}_{x_l}(L_{l-1})$.

   (b) Otherwise, attempt to read the decryption factors v_l published by the lth mix-server as described in Section 10.1. Then execute Algorithm 15 on input $(\rho, N, n_e, n_r, n_v, PRG, G_q, u, \mathcal{C}, \mathcal{M}, y, w, w', \pi, \sigma)$. If it rejects, then **reject**.

10. **Verify Output.** Compute the array of plaintext elements $m = \mathsf{TDec}\big(L_\lambda, \prod_{l=1}^{k} v_l\big)$. Then for each i decode m_i into an array of bytes b_i. Interpret each b_i as an ASCII string s_i, where newline characters and carriage return characters are removed. Then sort the s_i lexicographically and verify that the result equals s as defined in

A 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="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="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="pubkey"
            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>
```

Listing 3: XML schema for protocol info files.

B Example Protocol Info File

```
<!-- ATTENTION! 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
     edit this file in any way after you and the administrators of
     the other parties have agreed on its content. Doing so may in
     the worst case render the execution insecure. -->

<protocol>

   <!-- Session identifier of this protocol execution. -->
   <sid>MyDemo</sid>

   <!-- Name of this protocol execution. -->
   <name>Swedish Election</name>

   <!-- Description of this protocol execution. -->
   <descr></descr>

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

   <!-- Number of parties needed to violate privacy. -->
   <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 = 51
```

```
2)::0000000002010000001c766572696669636174756d2e61726974686d2e4d6f645047
726f757000000000040100000041014354c848a190b7b5fbddcd07bed36e59af5a50cc59
66b202bba0959ccc42061a2f468f87fa436451bd48d5cb333c0bb0aca763193e70c72549
5455a99939276f010000004100a1aa642450c85bdafdeee683df69b72cd7ad28662cb359
015dd04ace6621030d17a347c3fd21b228dea46ae5999e05d85653b18c9f386392a4aa2a
d4cc9c93b70100000041001a724721f0858dcae7ec0b0624719041e348ca13a3e4e179be
dec9c54468cedb3454e6bf5b4ac2d61829c9766174f2f5afe78494637aadd5346400faf9
7d339c010000000400000001</pgroup>

   <!-- Interface that defines how to communicate with the mix-net.
        This includes not only the format, but also the underlying
        submission scheme. Possible values are "native" and "helios".
        The former is simply the byte tree format used internally in
        Verificatum. The latter is the format needed by the Helios
        system <http://www.heliosvoting.org>. -->
   <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>0</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 (instance of verificatum.crypto.PRG).
        -->
   <prg>verificatum.crypto.PRGHeuristic(SHA1 with counter)::000000000201
0000001f766572696669636174756d2e63727970746f2e505247486575726973746963301
00000000</prg>

   <!-- Hashfunction used to implement random oracles (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 use this option unless you know exactly what
        you are doing. -->
   <rohash>verificatum.crypto.HashfunctionHeuristic(SHA-256)::0000000002
```

```
0100000028766572696669636174756d2e63727970746f2e4861736866756e6374696f6e
486575726973746963010000000075348412d323536</rohash>

   <!-- Determines if the proofs of correctness of an execution are
        interactive or non-interactive ("interactive" or
        "noninteractive"). The "noninteractive"-proofs of correctness
        are incompatible with the "cramershoup" interface (option: -
        inter). -->
   <corr>noninteractive</corr>

   <party>

      <!-- Sorting attribute used to sort parties with respect to their
           roles in the protocol. -->
      <srtbyrole>a:shuffler</srtbyrole>

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

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

      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pubkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2
048)::00000000020100000029766572696669636174756d2e63727970746f2e5369676e
6174757265504b657948657572697374696300000000020100000004000080001000001
2630820122300d06092a864886f70d01010105000382010f003082010a02820101008083
2581f219293d78426a21cb4c83267a3b2eee0dbb170034258d512bdbeda379a612df4749
bd08341f07ac43607f792d181290b1e27cb4ba17f40f94fc0d0ae9ec71d4bf10712af8f2
26df19f5b4446479b23d00a8e70119e5e99800e569e9f0f1677270bd1b0873a3ac020a4b
ae93f2c1106dfe5471b6655f77ef94c60996d9dfe5baa9b5f9fd730e17d0dadfdb38e3e5
43b766e523b178c7a93e0de8ef810a1be0062b5640723da13e02af2af36d0ed3ddab77f5
d8f4a08171edb88286ed0d319e4e7d592aca06f3b37753e2d8b03c2c06f0dc1065189741
3f7935f02889c265540b529d52d327f2d8457eb9a4d68ee684a3636abe89947a384f2000
b41b0203010001</pubkey>

      <!-- URL to our HTTP server. -->
      <http>http://localhost:8081</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>localhost:4041</hint>

   </party>

   <party>

      <!-- Sorting attribute used to sort parties with respect to their
           roles in the protocol. -->
      <srtbyrole>a:shuffler</srtbyrole>

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

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

      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pubkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2
048)::00000000020100000029766572696669636174756d2e63727970746f2e5369676e
6174757265504b65794865757269737469630000000002010000000400000800010000012
2630820122300d06092a864886f70d01010105000382010f003082010a0282010100cd8c
c533068506d3242579a1c2f1c7617217d451be62e9dc8ba7e1f3fb3566968a610054a092
388ef3abe0de23e50f65197cee6900883fb8c8e3d59f6be60a78dfed417ae1f450f8b479
dbba2c5c2df08de0d8691cbd16749b1695e166c50700fc377322489a75416662408dc802
2e3c6d16d7fcdf2a0dca1d90c26321bf63dcc9035dd42717cba313c500dd5a076bb05f3f
a9669d32a380a072a11a51371efa15d26da9788f177ae464a201bd3522b48a5bc8afda20
0caf1d8e0cd36a7102c1b0b945847227bc14e6068dd0e0c1dac27c462fa41eab46da654a
b83e900d22ec500027c2bce7983b26e63ad587b4e2194c585adcb12d09a92d38b6eea541
abeb0203010001</pubkey>

      <!-- URL to our HTTP server. -->
      <http>http://localhost:8082</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>localhost:4042</hint>

   </party>

   <party>

      <!-- Sorting attribute used to sort parties with respect to their
           roles in the protocol. -->
      <srtbyrole>a:shuffler</srtbyrole>

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

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

      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pubkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2
048)::00000000020100000029766572696669636174756d2e63727970746f2e5369676e
6174757265504b65794865757269737469630000000002010000000400000800010000012
2630820122300d06092a864886f70d01010105000382010f003082010a0282010100adb7
adea5f27e71997bc69e62a08838bb1de8fb92c3c8433108c3af596e3dc62e6c61e6568af
66761de87ae53d527168dc5ce25aeed6a901467d2724ff94cc5f4259ac4abcdacb8b5200
0570d5ac62e370341adf0ce41e078d4a419c3c9ceb3ff404fc52e507ea18dcb4fd8670b2
d18dc8739730e8c1f6d63b6ac940ed38f151ce85d5166ed30db2ceee501e9a7e14cc8280
e92735bf93f9f9a019f34f4faaefa62e664f6253464924cfc3b07fd71514c00f3a5a03df
1b66587d04d67a64691def52c95c645bb360f714b39f598f26133001d32ce97788b58cb3
fc67043c962b99bb8f03e0661c49403e77c0505fbb9e44697b24599bab4af1ebfa8bc88d
7de10203010001</pubkey>
```

```
      <!-- URL to our HTTP server. -->
      <http>http://localhost:8083</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>localhost:4043</hint>

   </party>

</protocol>
```

Listing 4: Example of a complete protocol info file.

C Interactive Proofs of Shuffles

Protocol 16 (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 $\epsilon_0, \ldots, \epsilon_{N-1} \in [0, 2^{n_e+n_c+n_r} - 1]$ 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 $v \in [0, 2^{n_c} - 1]$ 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 17 (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 $\alpha \in \mathbb{Z}_q$, $\epsilon_0, \ldots, \epsilon_{N-1} \in [0, 2^{n_e+n_c+n_r} - 1]$, 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 $v \in [0, 2^{n_c} - 1]$ 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

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