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

Douglas Wikström
dog@csc.kth.se

November 17, 2012

Abstract

Verificatum, `http://www.verificatum.org`, is an implementation of an El Gamal-based mix-net which 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.

DRAFT. We know that there are bugs in this document at the moment and it is not yet claimed to be correct or secure. Students of Amnon Ta-Schma at Tel Aviv University are currently running a project that will help us iron out these problems, and this document is provided solely for this purpose. Thus, if you want to attack the scheme (which we welcome), then you have to wait until a document that is claimed to be correct and secure is published at: `http://www.verificatum.org`.

Help us improve this document! The most recent version of this document can always be found at `http://www.verificatum.org`. Report errors, omissions, and suggestions to `dog@csc.kth.se`.

Contents

1 Introduction

The zero knowledge proofs in Verificatum mix-net [12] (VMN) can be made non-interactive using the Fiat-Shamir heuristic [5] and this is also the default behaviour. 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 the 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 an algorithm for verifying the complete contents of the proof directory.

VMN can be used as a blackbox to: generate a joint public key for which the secret key is verifiably secret shared among the mix-servers, re-encrypt and sort a list of ciphertexts (shuffle session), decrypt a list of ciphertexts (decryption session), or decrypt and sort a list of ciphertexts (mixing session).

Accordingly, there are three types of proofs, but the proof of a mixing session consists of a shuffle proof and a decryption proof except for how some files are named.

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 precomputation is not used and the Fiat-Shamir heuristic is applied).

2.1 The El Gamal Cryptosystem

The El Gamal cryptosystem [3] is defined over a group G_q of prime order q. The set $\mathcal{M}$ of plaintexts is defined to be the group G_q and the set of ciphertexts $\mathcal{C}$ is the product space $G_q \times G_q$. The randomness used to encrypt is sampled from $\mathcal{R} = \mathbb{Z}_q$.

A secret key $x \in \mathbb{Z}_q$ 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 G_q. To encrypt a plaintext $m \in \mathcal{M}$, a random exponent $s \in \mathcal{R}$ is chosen and the ciphertext in $\mathcal{C}$ is computed as $\mathsf{Enc}_{pk}(m, s) = (g^s, y^s m)$. 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 $\{0,1\}^t \rightarrow G_q$, which can be efficiently computed and inverted.

Homomorphic. The cryptosystem is homomorphic, i.e., if

$$(u_1, v_1) = \mathsf{Enc}_{pk}(m_1, s_1) \quad \text{and} \quad (u_2, v_2) = \mathsf{Enc}_{pk}(m_2, s_2)$$

are two ciphertexts, then their element-wise product

$$(u_1u_2, v_1v_2) = \mathsf{Enc}_{pk}(m_1m_2, s_1 + s_2)$$

is an encryption of m_1m_2. If we set $m_2 = 1$, then this feature can be used to *re-encrypt* (u_1, v_1) without knowledge of the randomness. To see this, note that for every fixed s_1 and random s_2, the sum $s_1 + s_2$ 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 lth party up to a threshold $1 \leqslant \lambda \leqslant k$ 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 among all k parties such that a threshold λ of the parties can recover it in the event that the lth party fails to do its part in the joint decryption of ciphertexts correctly. The details [4, 6] of the verifiable secret sharing scheme are not important in this document. The joint public key is then defined as $pk = (g, y)$, where $y = \prod_{l=1}^{\lambda} y_l$. Note that the corresponding secret key is defined by $x = \sum_{l=1}^{\lambda} x_l$ and that although not all parties contribute to the key, all parties receive shares of the secret keys.

To jointly decrypt a ciphertext (u, v), the lth 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 lth party and perform its part of the joint decryption in the open. Then the decryption factors can be combined to a joint decryption factor $f = \prod_{l=1}^{\lambda} f_l$ such that $\mathsf{PDec}_x(u, v) = f$. The ciphertext can then be trivially decrypted as $\mathsf{TDec}((u, v), f) = vf = m$.

A Generalization and Useful Notation. Using a simple hybrid argument it is easy to see that a longer plaintext $m = (m_1, \ldots, m_\omega) \in G_q^\omega$ can be encrypted by encrypting each component independently, as $\big(\mathsf{Enc}_{pk}(m_1, s_1), \ldots, \mathsf{Enc}_{pk}(m_\omega, s_\omega)\big)$, where $s = (s_1, \ldots, s_\omega) \in \mathbb{Z}_q^\omega$ is chosen randomly.

In our setting it is more convenient to simply view the cryptosystem as defined for elements in the product group $\mathcal{M}_\omega = G_q^\omega$ of plaintexts directly, i.e., we define encryption as $\mathsf{Enc}_{pk}(m, s) = (g^s, y^s m)$, where s is an element in the product ring $\mathcal{R}_\omega = \mathbb{Z}_q^\omega$. Thus, the ciphertext belongs to the ciphertext space $\mathcal{C}_\omega = \mathcal{M}_\omega \times \mathcal{M}_\omega$. Here multiplication and exponentiation are understood as acting element-wise, e.g.,

$$(g^s, y^s m) = \big((g^{s_1}, \ldots, g^{s_\omega}), (y^{s_1} m_1, \ldots, y^{s_\omega} m_\omega)\big) \ .$$

Decryption and computation of decryption factors can be defined similarly. We refer to ω as the *width* of the plaintexts and ciphertexts.

2.2 A Mix-Net Based on the El Gamal Cryptosystem

We use the re-encryption approach of Sako and Kilian [9] and the proof of a shuffle of Terelius and Wikström [10]. The choice of proof of a shuffle is mainly motivated by the fact that many other efficient proofs of shuffles are patented. We use the batching technique of Bellare et al. [2] to speed up the proofs needed during distributed decryption. Optionally the pre-computation technique proposed by Wikström [11] is used.

The mix-net is executed by k mix-servers.

Key Distribution. The mix-servers first run a distributed key generation protocol such that for each $1 \leqslant l \leqslant \lambda$, 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 $y = \prod_{l=1}^{\lambda} y_l$, to be used by senders.

Shuffling. We denote the number of ciphertexts by N. The ith ciphertext $w_{0,i} = \mathsf{Enc}_{pk}(m_i, s_i)$ encrypts some message $m_i \in \mathcal{M}_\omega$ using randomness $s_i \in \mathcal{R}_\omega$. To avoid Pfitzmann's attack [8] and preserve privacy, the sender of a ciphertext must prove knowledge its plaintext and all ciphertexts must be distinct. 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). (Furthermore, in some applications of mix-nets, the content of all ciphertexts may be known and the randomness replaced by a public constant.)

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 form a list $L_0 = (w_{0,0}, \ldots, w_{0,N-1})$ of all the input ciphertexts. Then the jth 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 resulting ciphertexts and publishes them as a list L_l. More precisely, it chooses $r_{l,i} \in \mathcal{R}_\omega$ and a permutation π_l randomly and outputs $L_l = (w_{l,0}, \ldots, w_{l,N-1})$, where

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

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

- If $l \neq j$, then it waits until the lth mix-server publishes L_l and a non-interactive zero-knowledge proof of knowledge ξ_l. The proof is verified and if it is rejected, then L_l is set equal to L_{l-1}.

Decryption. Finally, the mix-servers jointly decrypt the ciphertexts in L_λ as described in Section 2.1. More precisely, if $l \leqslant \lambda$, then the lth 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 $\mathsf{TDec}(L_\lambda, \prod_{l=1}^{\lambda} f_l)$, 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 the 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 $y = \prod_{l=1}^{\lambda} y_l$. 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 $(\mu_l, \tau_l^{pos}, \sigma_l^{pos})$ is not a valid proof of knowledge of exponents $r_{l,i}$ and a permutation π_l such that $w_{l,i} = w_{l-1,\pi_l(i)} \mathsf{Enc}_{pk}(1, r_{l,\pi_l(i)})$, 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 $l = 1, \ldots, \lambda$:
   - 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 lth mix-server, then **reject**.
4. Check if the output of the mix-net is $\mathsf{TDec}(L_\lambda, \prod_{l=1}^{\lambda} f_l)$. If not, then **reject** and otherwise **accept**.

2.4 Alternative Usage of the Mix-Net

The mix-net can also be used to shuffle ciphertexts without decrypting them, i.e., the ciphertexts are only re-randomized and permuted. Alternatively, it can be used to decrypt without performing any re-randomization or permutation. These modes generate proofs that correspond to one of the two phases of the mix-net.

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.

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 protocol parameters, e.g., auxiliary security parameters, must be extracted from an XML encoded protocol info file before any verification can take place. Section 7 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 8 explains how to implement these tests.
6. **Verification of a Complete Execution.** Section 9 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 of 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. 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, but the third is not, is

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

4.2 Representation as an Array of Bytes

We use $\mathsf{bytes}_k(n)$ as a short-hand to denote the $8k$-bit two's-complement representation of n in big endian byte order. We also use hexadecimal notation for constants, e.g., `0A` means $\mathsf{bytes}_1(10)$. A byte tree is represented by an array of bytes as follows.

- A leaf $\mathsf{leaf}(d)$ is represented by 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 by 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 l, 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
```

ASCII strings. ASCII strings are identified with the corresponding byte arrays. Thus, a string s can be represented as a byte tree $\mathsf{leaf}(s)$. No ending symbol is used to indicate the length of the string, since the length of the string is stored in the leaf.

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

Hexadecimal encodings. 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 denote by $\mathsf{unhex}(s)$ the reverse operation that converts an ASCII string s of an even number of digits `0-9` and `A-F` into the corresponding array of bytes.

5 Cryptographic Primitives

For our cryptographic library we need hash functions, pseudo-random generators, and random oracles derived from these.

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 [7], i.e., SHA-256, SHA-384, and SHA-512. In future versions of Verificatum it will be possible to use SHA-3 (Keccak) as well, but the standard parameters of Keccak has not yet been announced. We use the following notation.

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

Example 4. If $H = \mathsf{Hashfunction}($`"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 8. 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 based on complexity assumptions. 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 s, 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 each 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.

Appendix A contains test vectors for this pseudo-random generator.

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. If $n_{out} \bmod 8 \neq 0$, then 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 positive 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 different PRG is used. We use the following notation:

- $\mathsf{RandomOracle}(H, n_{out})$ – Creates a random oracle RO 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. We also describe how to derive group elements from an array of random bytes.

6.1 Basic Objects

Integers. A multi-precision integer n is represented by $\mathsf{leaf}(\mathsf{bytes}_k(n))$ for the smallest possible integer 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 is an array $(b_1, \ldots, b_l)$ of bytes where b_i equals `01` if a_i is true and `00` otherwise.

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

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

6.2 Prime Order Fields and Product Rings

Field Element. 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 a byte tree $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_l})$, 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 a byte tree $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_l})$, 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
```

6.3 Multiplicative Groups Modulo Primes

Group. A subgroup G_q of prime order q of the multiplicative group $\mathbb{Z}_p^*$, where $p > 3$ is prime, with standard generator g is represented by the byte tree

$$\mathsf{node}(\overline{p}, \overline{q}, \overline{g}, \mathsf{bytes}_4(e)) \enspace ,$$

where the integer e determines how a string is encoded into a group element and can be ignored for the purpose of this document.

Group Element. An element $a \in G_q$, where G_q is a subgroup of prime 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`.

Example 16. Let G_q be the subgroup of order $q = 131$ in $\mathbb{Z}_{263}^*$. Then $3 \in G_q$ is represented by `01 00 00 00 02 00 03`.

6.4 Standard Elliptic Curves over Prime Order Fields

Group. A standard elliptic curve group named FooCurve is represented by the byte tree leaf(`"FooCurve"`).

Example 17. The group P-256 from FIPS 186-3 [1] is represented by leaf(`"P-256"`).

The following curves are currently implemented by Verificatum, each defining the order of the underlying field, the curve equation, the order of the group, and the standard generator.

```
P-192  brainpoolp192r1  prime192v1  secp192k1
P-224  brainpoolp224r1  prime192v2  secp192r1
P-256  brainpoolp256r1  prime192v3  secp224k1
P-384  brainpoolp320r1  prime239v1  secp224r1
P-521  brainpoolp384r1  prime239v2  secp256k1
       brainpoolp512r1  prime239v3  secp256r1
                        prime256v1  secp384r1
                                    secp521r1
```

Group Element. Let the group be defined over a prime order field $\mathbb{Z}_q$ and let k be the smallest integer such that q can be represented as $\mathsf{bytes}_k(q)$. Then an affine point $P = (x, y)$ on the curve is represented by $\mathsf{node}(\mathsf{leaf}(\mathsf{bytes}_k(x)), \mathsf{leaf}(\mathsf{bytes}_k(y)))$ and the point at infinity is represented by $\mathsf{node}(\mathsf{leaf}(\mathsf{bytes}_k(-1)), \mathsf{leaf}(\mathsf{bytes}_k(-1)))$.

6.5 Arrays of Group Elements and Product Groups

Array of Group Elements. An array $(a_1, \ldots, a_l)$ of group elements is represented by a byte tree $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_l})$, 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 $(a_{1,i}, \ldots, a_{l,i})$ 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.

6.6 Marshalling Groups

When objects 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 an object from such a representation, information about the class must be otherwise available. Java serialization would not be portable and potentially unstable with different versions of Java. Thus, we use a simplified scheme where a group G_q represented by an instance of a Java class `PGroupClass` in Verificatum is marshalled into a byte tree

$$\mathsf{node}(\mathsf{leaf}(\texttt{"PGroupClass"}), \overline{G_q}) \ .$$

This byte tree in turn is converted into a byte array which is coded into hexadecimal and prepended with an ASCII comment. The comment and the hexadecimal coding of the byte array is separated by double colons. The resulting ASCII string is denoted by $s = \mathsf{marshal}(G_q)$ and the group G_q recovered from s by removing the comment and colons, converting the hexadecimal string to a byte array, converting the byte array into a byte tree, and converting the byte tree into a group G_q, is denoted by $G_q = \mathsf{unmarshal}(s)$.

Groups in Verificatum. Currently, there are two implementations of groups in Verificatum:

`verificatum.arithm.ModPGroup`	Multiplicative groups.
`verificatum.arithm.ECqPGroup`	Standard elliptic curve groups over prime order fields.

Example 18. The standard NIST curve P-256 [1] is marshalled into

$$\mathsf{node}(\mathsf{leaf}(\texttt{"verificatum.arithm.ECqPGroup"}), \mathsf{leaf}(\texttt{"P-256"})) \ .$$

6.7 Deriving Group Elements from Random Strings

In Section 8.2 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

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

and describe how this is defined for each type of group below. The auxiliary security parameter n_r 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 $e, e_0, \ldots, e_{N'-1}$ not all zero modulo q such that $g^e \prod_{i=0}^{N'-1} h_i^{e_i} = 1$. In particular, it is not acceptable to derive exponents $x_0, \ldots, x_{N'-1} \in \mathbb{Z}_q$ 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 $(h_0, \ldots, h_{N'-1})$ in G_q is derived as follows from a seed s.

1. Let n_p be the bit length of p.
2. Let $(t_0, \ldots, t_{N'-1}) = PRG(s)$, 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.

This construction makes 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. An array of group elements is then derived from the array of non-negative integers in the natural way by mapping the integers into G_q.

Elliptic Curves over Prime Order Fields. Let E be an elliptic curve over a prime order field $\mathbb{Z}_q$ defined by an equation $y^2 = f(x) \bmod q$. Then an array $(h_0, \ldots, h_{N'-1})$ in G_q is derived as follows from a seed s.

1. Let n_q be the bit length of q.
2. Let $(t_0, \ldots, t_a) = PRG(s)$, where $t_l \in \{0,1\}^{8\lceil (n_q+n_r)/8 \rceil}$ is interpreted as a *non-negative* integer.
3. Set $t'_l = t_l \bmod 2^{n_q+n_r}$ and let $z_l = t'_l \bmod q$.
4. Then define points $(h_0, \ldots, h_{N'-1}) = \big((x_0, y_0), \ldots, (x_{N'-1}, y_{N'-1})\big)$ as follows.
   (a) Set $l = 0$.
   (b) Let $i \geqslant l$ be the smallest integer such that $f(x_i)$ is a quadratic residue modulo q. Then set $l = i + 1$.
   (c) Let y be the square root of $f(x_i)$ that is smallest when viewed as an integer in $[0, q-1]$. (This can be computed using the ressol algorithm.) Then go to step 4b.

In other words, first we generate random positive integers in $[0, 2^{n_q+n_r} - 1]$, then these integers are reduced modulo q to get almost uniformly generated integers z_i in $[0, q-1]$. Finally, we walk through these integers until z_i is the x-coordinate of a point on the curve. On average we expect that roughly half of the values in $\mathbb{Z}_q$ are x-coordinates of points on the curve, so this procedure is efficient.

Again, this is quite natural in an implementation and allows re-use of the implementation of the functionality needed to implement the multiplicative group.

7 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.

7.1 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 in 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 either a hexadecimal encoding of a byte tree representing the group, or the ASCII name of the group in the case of a named group.

After the global parameters follows a `<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 safely be ignored when implementing a verifier.

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. All values relevant to a verifier appear in the preamble. 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 schema depends on the type of bulletin board, since different bulletin boards accept different parameters. Thus, it is wise to ignore this schema and instead use a general XML parser of well-formed documents and extract only the needed values. This works, since we do not use any attributes of XML tags, i.e., all values are stored as data between an opening tag and a closing tag.

7.2 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 $p =$ `ProtocolInfo(protocolInfo.xml)` and define the values we later use in Section 8 and Section 9.

- $\mathsf{version}_{prot} = p[\texttt{version}]$ is the version of Verificatum used during the execution that produced the proof.

- $\mathsf{sid} = p[\texttt{sid}]$ is the globally unique session identifier tied to the generation of a particular joint public key.
- $k = \mathsf{int}(p[\texttt{nopart}])$ 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 ciphertexts.
- $n_e = \mathsf{int}(p[\texttt{vbitlenro}])$ 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.
- $s_H = p[\texttt{rohash}]$ specifies the hash function H used to implement the random oracles.
- $s_{PRG} = p[\texttt{prg}]$ specifies the hash function used to implement the pseudo-random generator used to expand challenges into arrays.
- $s_{G_q} = p[\texttt{pgroup}]$ specifies the underlying group G_q.
- $\omega_{\mathit{default}} = \mathsf{int}(p[\texttt{width}])$ specifies the default width of ciphertexts and plaintexts.

8 Verifying Fiat-Shamir Proofs

We use several non-interactive zero-knowledge proofs: a proof of a shuffle, a proof of a shuffe of commitments, a commitment-consistent proof of a shuffle, and a proof of correct decryption. In a normal execution only the first and last proofs are used. If pre-computation is used, then the proof of a shuffle is essentially divided into, and replaced by, the second and third proofs.

From now on we simply write $\overline{a}$ for the byte tree representation of an object a.

8.1 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.

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

8.2 Independent Generators

The protocols in Section 8.3 and Section 8.5 also require "independent" generators in G_q and these generators must be derived using the random oracles. To do that a seed

$$s = RO_{\mathit{seed}}(\rho \mid \mathsf{leaf}(\texttt{"generators"}))$$

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

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

which is defined in Section 6.7. The prefix ρ is computed in Step 4 of the main verification routine in Section 9.3 and given as input to Algorithm 19 and Algorithm 22 below. The length N' is at least N and larger if needed for pre-computation.

8.3 Proof of a Shuffle

A proof of a shuffle is used by a mix-server to prove that it re-encrypted and permuted its input ciphertexts. 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 [10].

Protocol 19 (Proof of a Shuffle)**.**

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 used to derive random vectors for batching.
G_q	Group of prime order with standard generator g.
$\mathcal{R}_\omega$	Randomizer group.
$\mathcal{C}_\omega$	Ciphertext group.
pk	El Gamal public key.
w	Array $w = (w_0, \ldots, w_{N-1})$ of input ciphertexts in $\mathcal{C}_\omega$.
w'	Array $w' = (w'_0, \ldots, w'_{N-1})$ of output ciphertexts in $\mathcal{C}_\omega$.
μ	Permutation commitment.
τ^{pos}	Commitment of the Fiat-Shamir proof.
σ^{pos}	Reply of the Fiat-Shamir proof.

Program

1. (a) Interpret μ as an array $u = (u_0, \ldots, u_{N-1})$ of Pedersen commitments in G_q.

   (b) Interpret τ^{pos} as $\mathsf{node}(\overline{B}, \overline{A'}, \overline{B'}, \overline{C'}, \overline{D'}, \overline{F'})$, where $A', C', D' \in G_q$, $F' \in \mathcal{C}_\omega$, and B and B' are arrays of N elements in G_q.

   (c) Interpret σ^{pos} as $\mathsf{node}(\overline{k_A}, \overline{k_B}, \overline{k_C}, \overline{k_D}, \overline{k_E}, \overline{k_F})$, where $k_A, k_C, k_D, k_F \in \mathbb{Z}_q$, k_B is an array of N element in $\mathcal{R}_\omega$, and k_E is an array of N elements in $\mathbb{Z}_q$.

   Reject if this fails.

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

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

$$A = \prod_{i=0}^{N-1} u_i^{e_i} \quad \text{and} \quad F = \prod_{i=0}^{N-1} w_i^{e_i} \enspace .$$

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

5. Compute $C = \prod_{i=0}^{N-1} u_i / \prod_{i=0}^{N-1} h_i$, $D = B_{N-1}/h_0^{\prod_{i=0}^{N-1} e_i}$, set $B_{-1} = h_0$, and accept if and only if

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

8.4 Proof of a Shuffle of Commitments

A proof of a shuffle of commitments allows a mix-server to show in a pre-computation phase that it knows how to open a commitment to a permutation. 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 [10].

Algorithm 20 (Verifier of Proof of a Shuffle of Commitments)**.**

Input	**Description**
ρ	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 used to derive random vectors for batching.
G_q	Group of prime order with standard generator g.
μ	Permutation commitment.
τ^{posc}	Commitment of the Fiat-Shamir proof.
σ^{posc}	Reply of the Fiat-Shamir proof.

Program

1. (a) Interpret μ as an array $u = (u_0, \ldots, u_{N-1})$ of Pedersen commitments in G_q.

   (b) Interpret τ^{posc} 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.

   (c) Interpret σ^{posc} 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 $s = RO_{seed}\big(\rho \mid \mathsf{node}(\overline{g}, \overline{h}, \overline{u})\big)$.

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

5. Compute $C = \prod_{i=0}^{N_0-1} u_i \big/ \prod_{i=0}^{N_0-1} h_i$ and $D = B_{N_0-1} h_0^{-\prod_{i=0}^{N_0-1} e_i}$, set $B_{-1} = h_0$, and accept if and only if:

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

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

8.5 Commitment-Consistent Proof of a Shuffle

Provided pre-computation and a proof of a shuffle of commitments is used, a commitment-consistent proof of a shuffle can be used to verify the proof of a shuffle for the given permutation commitment. 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 [11].

Algorithm 21 (Verifier of Commitment-Consistent Proof of a Shuffle)**.**

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 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.
$\mathcal{R}_\omega$	Randomizer group.
$\mathcal{C}_\omega$	Ciphertext group.
pk	El Gamal public key.
w	Array $w = (w_0, \ldots, w_{N-1})$ of input ciphertexts in $\mathcal{C}_\omega$.
w'	Array $w' = (w'_0, \ldots, w'_{N-1})$ of output ciphertexts in $\mathcal{C}_\omega$.
μ	Permutation commitment.
τ^{ccpos}	Commitment of the Fiat-Shamir proof.
σ^{ccpos}	Reply of the Fiat-Shamir proof.

Program

1. (a) Interpret μ as an array $u = (u_0, \ldots, u_{N-1})$ of Pedersen commitments in G_q.
   (b) Interpret τ^{ccpos} as $\mathsf{node}(\overline{A'}, \overline{B'})$, where $A' \in G_q$ and $B' \in \mathcal{C}_\omega$.
   (c) Interpret σ^{ccpos} as $\mathsf{node}(\overline{k_A}, \overline{k_B}, \overline{k_E})$, where $k_A \in \mathbb{Z}_q$, $k_B \in \mathcal{R}_\omega$, and k_E is an array of N elements in $\mathbb{Z}_q$.

   Reject if this fails.
2. Compute a seed $s = RO_{seed}\big(\rho \mid \mathsf{node}(\overline{g}, \overline{h}, \overline{u}, \overline{pk}, \overline{w}, \overline{w'})\big)$.
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 $v = RO_{challenge}\big(\rho \mid \mathsf{node}(\mathsf{leaf}(s), \tau^{ccpos})\big)$ 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:

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

8.6 Proof of Correct Decryption

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. [2], but here we use a trick to allow combining the proofs of all parties into one before verifying.

Algorithm 22 (Verifier of Decryption Factors)**.**

Input	**Description**
j	Index of proof to verify.
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 used to derive random vectors for batching.
G_q	Group of prime order.
$y_1, \ldots, y_\lambda$	Partial public keys.
$\mathcal{C}_\omega$	Ciphertext group.
$\mathcal{M}_\omega$	Plaintext group.
w	Array $w = (w_0, \ldots, w_{N-1})$ of input ciphertexts in $\mathcal{C}_\omega$, where $w_i = (u_i, v_i)$.
$f_1, \ldots, f_\lambda$	Arrays $f_j = (f_{j,0}, \ldots, f_{j,N-1})$ of decryption factors in $\mathcal{M}_\omega$.
$\tau_1^{dec}, \ldots, \tau_\lambda^{dec}$	Commitments of the Fiat-Shamir proofs.
$\sigma_1^{dec}, \ldots, \sigma_\lambda^{dec}$	Replies of the Fiat-Shamir proofs.

Program

1. (a) Interpret τ_l^{dec} as $\mathsf{node}(\overline{y'_l}, \overline{B'_l})$, where $y'_l \in G_q$ and $B'_l \in \mathcal{M}_\omega$.

   (b) Interpret σ_l^{dec} as $\overline{k_{l,x}}$, where $k_{l,x} \in \mathbb{Z}_q$.

   Reject if this fails.

2. Compute a seed $s = RO_{seed}\big(\rho \mid \mathsf{node}(\mathsf{node}(\overline{g}, \overline{w}), \mathsf{node}(a, b))\big)$, where $a = \mathsf{node}(\overline{y_1}, \ldots, \overline{y_\lambda})$ and $b = \mathsf{node}(\overline{f_1}, \ldots, \overline{f_\lambda})$.

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

4. Compute a challenge $v = RO_{challenge}\big(\rho \mid \mathsf{node}(\mathsf{leaf}(s), \mathsf{node}(\tau_1^{dec}, \ldots, \tau_\lambda^{dec}))\big)$ interpreted as a non-negative integer $0 \leqslant v < 2^{n_v}$.

5. Compute $A = \left(\prod_{i=0}^{N-1} u_i^{e_i}, 1\right)$ and then:

   - If $j = 0$, then compute $B = \prod_{i=0}^{N-1} \left(\prod_{l=1}^{\lambda} f_{l,i}\right)^{e_i}$ and accept if and only if

   $$\left(\prod_{l=1}^{\lambda} y_l\right)^v \prod_{l=1}^{\lambda} y'_l = g^{\sum_{l=1}^{\lambda} k_{l,x}} \quad \text{and} \quad B^v \prod_{l=1}^{\lambda} B'_l = \mathsf{PDec}_{\sum_{l=1}^{\lambda} k_{l,x}}(A) \ .$$

   - If $j > 0$, then compute $B_j = \prod_{i=0}^{N-1} f_{j,i}^{e_i}$ and accept if and only if

   $$y_j^v y'_j = g^{k_{j,x}} \quad \text{and} \quad B_j^v B'_j = \mathsf{PDec}_{k_{j,x}}(A) \ .$$

9 Verification

The verification algorithm must verify that the input ciphertexts were repeatedly re-randomized 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 relevant parameters in the protocol info file.

9.1 Components of the Non-Interactive Zero-Knowledge Proof

A proof should be viewed as a capsule that relates a public key, an input, and an output in a provable way. In addition to the parameters we need from the protocol info file, we have specific parameters of a given session stored in separate files. These files are found in the root of the directory.

There is also a subdirectory proofs holding the intermediate results of the execution as well as non-interactive zero-knowledge proofs relating the intermediate results, the individual public keys, the full public key, the input, and the output.

The idea of this division of files is to emphasize that a complete verifier consists of two parts. The first part is what is discussed in this document, i.e., verifying the contents of the overall non-interactive zero-knowledge proof. The second part is to verify that the actual public key, input ciphertexts, and output plaintexts of a particular application matches those in the proof. This includes verifying the version of Verificatum, the type of proof, the auxiliary session identifier, and the width.

The first part requires a reasonable background in cryptography and programming, whereas the second part can be achieved by a very simple program. Even people with limited background in cryptography and programming can re-use verifiers implemented by knowledgeable independent parties to check the first part and then write their own simple program for verifying the second part, without peeking into the proofs directory. The latter program would also be easy to audit. The end result is a trustworthy complete verifier.

Another related reason is that there are complex tallying protocols that repeatedly uses the mix-net as a blackbox to mix, shuffle, and decrypt. Each session can be verified using verifiers written by independent parties. Then a simple verifier that relates the sessions can be implemented and audited with a limited background in cryptography.

9.1.1 Files in the Main Directory

We now give details about the files in the main directory. Recall that there are three different types of proofs corresponding to mixing, shuffling, and decryption sessions. The following files have the exact same meaning in all types of proofs.

1. version – An ASCII version string version of the Verificatum software that created the proof. This string is denoted by version.

2. type – One of the ASCII strings `mixing`, `shuffling`, or `decryption`. This string is denoted type.

3. auxsid – An auxiliary session identifier of this session as an ASCII string consisting of letters A–Z, a–z, digits 0–9, and underscore. This equals `default` unless a different auxiliary session identifier is given explicitly when executing the mix-net. This string is denoted by auxsid.

4. width – The width $\omega > 0$ of ciphertexts as a decimal number in ASCII. This may, or may not, be identical to the default width in the protocol info file.

5. FullPublicKey.bt – Full public key used to form input ciphertexts. The required format of this file is a byte tree $\overline{pk}$ where $pk \in G_q \times G_q$ (and G_q is derived from the protocol info file).

6. Ciphertexts.bt – Input ciphertexts. The required format of this file is a byte tree $\overline{L_0}$, where L_0 is an array of N elements in $\mathcal{C}_\omega$. This defines N, the number of input ciphertexts.

The file storing the output of the session comes in two forms depending on the type of session.

7. (a) Plaintexts.bt – For mixing and decryption sessions, 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 $\mathcal{M}_\omega$.
   (b) ShuffledCiphertexts.bt – For a shuffling session, the re-randomized and permuted ciphertexts. This file should contain a byte tree $\overline{L_\lambda}$, where L_λ is an array of N elements in $\mathcal{C}_\omega$.

9.1.2 Files in the Proofs Directory

The proofs directory proofs holds 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 $0 \leqslant l \leqslant k$ encoded using two decimal digits representing the index of a mix-server, but a file with suffix l may not originate from the lth mix-server if it is corrupted and all types of files do not appear with all suffixes.

Files for keys.

8. PublicKey$\langle l \rangle$.bt – Partial public key of the lth mix-server. The required format of this file is a byte tree $\overline{y_l}$, where $y_l \in G_q$.

9. SecretKey$\langle l \rangle$.bt – Secret key file of the lth party. If this file exists, then the required format is a byte tree $\overline{x_l}$, where $x_l \in \mathbb{Z}_q$.

Files for proofs of shuffles.

10. Ciphertexts$\langle l \rangle$.bt – The lth intermediate list of ciphertexts, i.e., normally 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 $\mathcal{C}_\omega$ (and N is the number of elements in the list L_0 of input ciphertexts).

11. PermutationCommitment$\langle l \rangle$.bt – Commitment to a permutation. The required format of the byte tree μ_l in this file is specified in Algorithm 19.

12. PoSCommitment$\langle l \rangle$.bt – "Proof commitment" of the proof of a shuffle. The required format of the byte tree τ_l^{pos} in this file is specified in Algorithm 19.

13. PoSReply$\langle l \rangle$.bt – "Proof reply" of the proof of a shuffle. The required format of the byte tree σ_l^{pos} in this file is specified in Algorithm 19.

Files for proofs of shuffles when pre-computation is used.

14. maxciph – The number N_0 of ciphertexts for which pre-computation was performed in ASCII decimal notation.

15. PoSCCommitment$\langle l\rangle$.bt – "Proof commitment" of the proof of a shuffle of commitments. The required format of the byte tree τ_l^{posc} in this file is specified in Algorithm 20.
16. PoSCReply$\langle l\rangle$.bt – "Proof reply" of the proof of a shuffle of commitments. The required format of the byte tree σ_l^{posc} in this file is specified in Algorithm 20.
17. 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.
18. CCPoSCommitment$\langle l\rangle$.bt – "Proof commitment" of the commitment-consistent proof of a shuffle. The required format of the byte tree τ_l^{ccpos} in this file is specified in Algorithm 21.
19. CCPoSReply$\langle l\rangle$.bt – "Proof reply" of the commitment-consistent proof of a shuffle. The required format of the byte tree σ_l^{ccpos} in this file is specified in Algorithm 21.

Files for proof of decryption.

20. 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 $\overline{f_l}$, where f_l is an array of N elements in $\mathcal{M}_\omega$.
21. DecrFactCommitment$\langle l\rangle$.bt – "Proof commitment" of the proof of correctness of the decryption factors. The required format of the byte tree τ_l^{dec} of this file is specified in Algorithm 22.
22. DecrFactReply$\langle l\rangle$.bt – "Proof reply" of the proof of correctness of the decryption factors. The required format of the byte tree τ_l^{dec} of this file is specified in Algorithm 22.
23. Plaintexts.bt – Plaintexts. The required format of this file is $\overline{m}$, where m is an array of N elements in $\mathcal{M}_\omega$.

9.1.3 Relation Between Files and Abstract Notation

For easy reference we tabulate the notation introduced below and from which file the contents is derived in the table below.

Not.	Point	File	Not.	Point	File
version	1	version	τ_l^{pos}	12	PoSCommitment$\langle l\rangle$.bt
type	2	type	σ_l^{pos}	13	PoSReply$\langle l\rangle$.bt
auxsid	3	auxsid	N_0	14	maxciph
ω	4	width	τ_l^{posc}	15	PoSCCommitment$\langle l\rangle$.bt
pk	5	FullPublicKey.bt	σ_l^{posc}	16	PoSCReply$\langle l\rangle$.bt
L_0	6	Ciphertexts.bt	t_l	17	KeepList$\langle l\rangle$.bt
m	7a	Plaintexts.bt	τ_l^{ccpos}	18	CCPoSCommitment$\langle l\rangle$.bt
L_λ	7b	ShuffledCiphertexts.bt	σ_l^{ccpos}	19	CCPoSReply$\langle l\rangle$.bt
y_l	8	PublicKey$\langle l\rangle$.bt	f_l	20	DecryptionFactors$\langle l\rangle$.bt
x_l	9	SecretKey$\langle l\rangle$.bt	τ_l^{dec}	21	DecrFactCommitment$\langle l\rangle$.bt
L_l	10	Ciphertexts$\langle l\rangle$.bt	σ_l^{dec}	22	DecrFactReply$\langle l\rangle$.bt
μ_l	11	PermutationCommitment$\langle l\rangle$.bt			

9.2 Subroutines of the Verification Algorithm

We are now ready to summarize the subroutines needed for verification in terms of the abstract notation introduced above.

Consistency of keys. We begin with a subroutine that reads the joint public key, the individual public keys, and any recovered secret keys, and then verifies that they are all consistent.

Algorithm 23 (Verifier of Keys)**.**

Input	**Description**
λ	Number of mix-servers.
G_q	Underlying group.

Program

1. **Joint public key.** Attempt to read the joint public key $pk \in G_q \times G_q$, where $pk = (g, y)$, from file as described in Point 5. If this fails, then **reject**.
2. **Public keys.** Attempt to read public keys $y_1, \ldots, y_\lambda$ as described in Point 8. If this fails, then **reject**. Then test if $y = \prod_{l=1}^{\lambda} y_l$, i.e., check that the keys $y_1, \ldots, y_\lambda$ of the mix-servers are consistent with the public key y. If not, then **reject**.
3. **Secret keys.** For $l = 1, \ldots, \lambda$:
   (a) If there is a secret key file for the lth mix-server, then **reject** if it does not contain x_l as described in Point 9. If there is no file, then set $x_l = \perp$.
   (b) If $x_l \neq \perp$ and $y_l \neq g^{x_l}$, then **reject**.
4. **Return** $\big(pk, (y_1, \ldots, y_\lambda), (x_1, \ldots, x_\lambda)\big)$.

Correctness of shuffling. Next we describe the algorithm for verifying a complete shuffling consisting of intermediate lists of ciphertexts and either proofs of shuffles, or proofs of shuffles of commitments combined with commitment-consistent proofs of shuffles.

Algorithm 24 (Verifier of Shuffling)**.**

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 used to derive random vectors for batching.
G_q	Group of prime order.
$\mathcal{R}_\omega$	Randomness group.
$\mathcal{C}_\omega$	Ciphertext group.
pk	Joint public key.
L_0	Original ciphertexts.
L_λ	Shuffled ciphertexts.
posc	Indicates that the proof of a shuffle (of commitments) should be verified.
ccpos	Indicates if the (commitment-consistent) proof of a shuffle should be verified.

Program

In case the maxciph file does not exist. For $l = 1, \ldots, \lambda$ do:

1. **Array of ciphertexts.** If $l < \lambda$, then read the array L_l of N ciphertexts in $\mathcal{C}_\omega$ as described in Point 10. If this fails, then **reject**.

2. **Verify proof of shuffle.** Read proof commitment τ_l^{pos} and proof reply σ_l^{pos} as described in Point 12 and Point 13, respectively. Then execute Algorithm 19 on input

$$(\rho, N, n_e, n_r, n_v, PRG, G_q, \mathcal{R}_\omega, \mathcal{C}_\omega, pk, L_{l-1}, L_l, \mu_l, \tau_l^{pos}, \sigma_l^{pos}) \ .$$

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

3. **Accept proof.**

Algorithm 25 (Algorithm 24 continued)**.**

In case the maxciph exists. Attempt to read N_0 as described in Point 14. If this fails, or if $N > N_0$, then **reject**. For $l = 1, \ldots, \lambda$ do:

1. **Verify proof of shuffle of commitments.** If posc $=$ $true$, then read a permutation commitment u_l, a proof commitment τ_l^{posc}, and proof reply σ_l^{posc} as described in Point 11, Point 12, and Point 13, respectively, and execute Algorithm 19 on input $(\rho, N_0, n_e, n_r, n_v, PRG, G_q, u_l, \tau_l^{posc}, \sigma_l^{posc})$. If reading fails or if the algorithm rejects, then set $u_l = h$.
2. **Potential early abort.** If ccpos $= false$, then **accept**.
3. **Shrink permutation commitment.**
   (a) Read the keep-list t_l as described in Point 17. 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.
   (b) Set $u_l = (u_{l,i})_{t_{l,i}=true}$ be the sub-array indicated by t_l.
4. **Array of ciphertexts.** If $l < \lambda$, then read the array L_l of N ciphertexts in $\mathcal{C}_\omega$ as described in Point 10. If this fails, then **reject**.
5. **Verify commitment-consistent proof of shuffle.** Read proof commitment τ_l^{ccpos} and proof reply σ_l^{ccpos} as described in Point 18 and Point 19, respectively. Then execute Algorithm 21 on input $(\rho, N, n_e, n_r, n_v, PRG, G_q, u_l, \mathcal{R}_\omega, \mathcal{C}_\omega, L_{l-1}, L_l, pk, \tau_l^{ccpos}, \sigma_l^{ccpos})$. If reading fails or if the algorithm rejects, then verify that $L_l = L_{l-1}$. If this is not the case, then **reject**.
6. **Accept proof.**

Correctness of decryption. The verifier of joint decryption is different from the literature in that it executes in parallel to allow combining the proofs of the mix-servers into a single proof before verification.

Algorithm 26 (Verifier of Decryption)**.**

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 used to derive random vectors for batching.
G_q	Group of prime order.
$\mathcal{M}_\omega$	Plaintext group.
$\mathcal{C}_\omega$	Ciphertext group.
pk	Joint public key.
$y_1, \ldots, y_\lambda$	Public keys of parties.
L	Ciphertexts.
m	Plaintexts.

Program

1. **Read proofs.** For $l = 1, \ldots, \lambda$, read f_l, τ_l^{dec}, and σ_l^{dec} of the lth mix-server as described in Point 20, Point 21, and Point 22, respectively. If this fails, then **reject**. Otherwise, set

$$f = (f_1, \ldots, f_\lambda)$$
$$\tau^{dec} = (\tau_1^{dec}, \ldots, \tau_\lambda^{dec})$$
$$\sigma^{dec} = (\sigma_1^{dec}, \ldots, \sigma_\lambda^{dec})$$

2. **Verify combined proof.** Attempt to verify all proofs at once by executing Algorithm 22 on input

$$(0, \rho, N, n_e, n_r, n_v, PRG, G_q, g, y_1, \ldots, y_\lambda, \mathcal{C}_\omega, \mathcal{M}_\omega, L, f, \tau^{dec}, \sigma^{dec}) \ .$$

   If it accepts, then go to Step 4.

3. **Verify individual proofs.** For $l = 1, \ldots, \lambda$:

   If Algorithm 22 rejects on input

$$(l, \rho, N, n_e, n_r, n_v, PRG, G_q, g, y_1, \ldots, y_\lambda, \mathcal{C}_\omega, \mathcal{M}_\omega, L, f, \tau^{dec}, \sigma^{dec})$$

   and either $x_l = \perp$ or $f_l \neq \mathsf{PDec}_{x_l}(L)$, then **reject**.

4. **Verify plaintext.** If $m \neq \mathsf{TDec}\big(L, \prod_{l=1}^{\lambda} f_l\big)$, then **reject**.

5. **Accept proof.**

9.3 Verification Algorithm

We are finally ready to describe the verification algorithm. We stress that the parameters specified in the protocol info file must be evaluated manually. If any parameter is found to be weak, then the proof can not be trusted. Furthermore, the version file contains the version of Verificatum that was used to produce the proof. The user is expected to check that its verifier is compatible.

Algorithm 27 (Verifier)**.**

Input	**Description**
protinfo	Protocol info file.
directory	Directory containing proof.
$\mathsf{type}_{expected}$	Expected type of proof.
$\mathsf{auxsid}_{expected}$	Expected auxiliary session identifier.
$\omega_{expected}$	Expected width of ciphertexts.
posc	Indicates that the proof of a shuffle (of commitments) should be verified.
ccpos	Indicates if the (commitment-consistent) proof of a shuffle should be verified.
dec	Indicates that the decryption should be verified.

Program

1. **Protocol parameters.** Verify that the XML of the protocol info file protinfo is well-formed and **reject** otherwise. Attempt to read the public parameters from protinfo as described in Section 7. If this fails, then **reject**. This defines $\mathsf{version}_{prot}$, sid, k, λ, n_e, n_r, n_v, s_H, s_{PRG}, s_{G_q}, and $\omega_{default}$.

2. **Proof parameters.** Read $\mathsf{version}_{proof}$, type, auxsid, ω, from the proof directory directory as described in Point 2 – Point 4. If this fails, then **reject**. If $\mathsf{version}_{proof} \neq \mathsf{version}_{prot}$, then **reject**. If $\mathsf{type} \neq \mathsf{type}_{expected}$, then **reject**. If $\mathsf{auxsid} \neq \mathsf{auxsid}_{expected}$, then **reject**. If $\omega_{expected} = \perp$ and $\omega \neq \omega_{default}$, then **reject**. If $\omega_{expected} \neq \perp$ and $\omega \neq \omega_{expected}$, then **reject**.

3. **Derived sets and objects.** Attempt to derive and define the underlying group $G_q = \mathsf{unmarshal}(s_{G_q})$, the set of plaintexts $\mathcal{M}_\omega = G_q^\omega$, the set of ciphertexts $\mathcal{C}_\omega = G_q^\omega \times G_q^\omega$, the set of randomness $\mathcal{R}_\omega = \mathbb{Z}_q^\omega$, the hash function $H = \mathsf{Hashfunction}(s_H)$, and the pseudo-random generator $PRG = \mathsf{PRG}(\mathsf{Hashfunction}(s_{PRG}))$. If anything fails, then **reject**.

4. **Prefix to Random Oracles.** To differentiate sessions, we compute a digest ρ of selected protocol parameters and use this as a prefix to all calls to random oracles. Then ρ is defined by

$$\rho = H\left(\begin{array}{l}\mathsf{node}\Big(\overline{\mathsf{version}_{proof}}, \overline{\mathsf{sid}|\text{"."}|\mathsf{auxsid}},\\ \quad \mathsf{bytes}_4(\omega), \mathsf{bytes}_4(n_e), \mathsf{bytes}_4(n_r), \mathsf{bytes}_4(n_v),\\ \quad \overline{s_{G_q}}, \overline{s_{PRG}}, \overline{s_H}\Big)\end{array}\right) .$$

Algorithm 28 (Algorithm 27 continued)**.**

5. **Read keys.** Execute Algorithm 23. If it fails, then **reject** and otherwise, let the output be $\big(pk, (y_1, \ldots, y_\lambda), (x_1, \ldots, x_\lambda)\big)$.

6. **Read lists.**

   (a) **Read input ciphertexts.** Read the array L_0 of N ciphertexts as described in Point 10 for some N. If this fails, then **reject**. This defines the integer N used to verify the length of other arrays.

   (b) **Read shuffled ciphertexts.** If $\mathsf{type} =$ `mixing`, then read L_λ as described in Point 10, else if $\mathsf{type} =$ `shuffling`, then read L_λ as described in Point 7b.

   (c) **Read plaintexts.** If $\mathsf{type} =$ `mixing` or $\mathsf{type} =$ `decryption`, then read m as described in Point 23.

7. **Verify relations between lists.**

   (a) **Verify shuffling.** If $\mathsf{type} =$ `mixing` or $\mathsf{type} =$ `shuffling`, and $\mathsf{posc} = \mathit{true}$ or $\mathsf{ccpos} = \mathit{true}$, then execute Algorithm 24 on input

   $$(\rho, N, n_e, n_r, n_v, PRG, G_q, \mathcal{R}_\omega, \mathcal{C}_\omega, pk, L_0, L_\lambda, \mathsf{posc}, \mathsf{ccpos}) \ .$$

   If it rejects, then **reject**.

   (b) **Verify decryption.** If $\mathsf{type} =$ `mixing` or $\mathsf{type} =$ `decryption`, and $\mathsf{dec} = \mathit{true}$, then set $L = L_\lambda$ in the former case and $L = L_0$ in the latter, and execute Algorithm 26 on input

   $$(\rho, N, n_e, n_r, n_v, PRG, G_q, \mathcal{M}_\omega, \mathcal{C}_\omega, pk, L, m) \ .$$

   If it rejects, then **reject**.

   (c) **Accept proof.**

10 Standard Command Line Interface of Verifier

To maximize interoperability and to simplify execution of multiple verifiers we require that every verifier can be invoked from the command line using the basic options described in the Verificatum mix-net user manual [13] and behave correspondingly. Below we specify how each standard command line expression is translated into a call to Algorithm 27.

All commands must be silent by default and halt with exit status 0 upon success, and otherwise halt with a non-zero exit status. The `-v` option can be used to turn on verbose output displaying progress and the verifications performed. There are no rules for what the output must look like, but it is a good idea to output something similar to what the builtin verifier `vmnv` outputs. Syntax errors or incorrect usage must print error messages even if the `-v` option is not used.

Denote by `verifier` an independently implemented command line tool. For future compatibility we require that the following command outputs a space separated list of all versions of Verificatum for which the verifier is compatible.

```
verifier -compat
```

Let `protInfo.xml` be a protocol info file and let `directory` contain the complete proof to be verified. Let verifier denote Algorithm 27. For each type of proof and proof parameters specified on the command line below, we assign values to the variables: $\mathsf{type}_{\mathit{expected}}$, $\mathsf{auxsid}_{\mathit{expected}}$, $\omega_{\mathit{expected}}$, posc, ccpos, and dec, and then require that the output of the command is given by

$$\mathsf{verifier}(\mathsf{type}_{\mathit{expected}}, \texttt{protInfo.xml}, \texttt{directory}, \mathsf{auxsid}_{\mathit{expected}}, \omega_{\mathit{expected}}, \mathsf{posc}, \mathsf{ccpos}, \mathsf{dec}) \ .$$

Proof of mixing. It must be possible to verify a proof of a mixing using the following command.

```
verifier -mix protInfo.xml directory
```

For this command we set $\mathsf{auxsid}_{\mathit{expected}} = \texttt{default}$, $\omega_{\mathit{expected}} = \perp$, $\mathsf{posc} = \mathit{true}$, $\mathsf{ccpos} = \mathit{true}$, and $\mathsf{dec} = \mathit{true}$. For each additional option the needed changes to the parameters are listed below.

Option	Changes
`-auxsid <auxsid>`	$\mathsf{auxsid}_{\mathit{expected}} = $ `<auxsid>`
`-width <width>`	$\omega_{\mathit{expected}} = $ `<width>`
`-nopos`	$\mathsf{posc} = \mathsf{ccpos} = \mathit{false}$.
`-noposc`	$\mathsf{posc} = \mathit{false}$
`-noccpos`	$\mathsf{ccpos} = \mathit{false}$
`-nodec`	$\mathsf{dec} = \mathit{false}$

Note that `<auxsid>` should be a valid auxiliary session identifier ASCII string and that `<width>` must be an ASCII coded decimal integer. The last two options in the table are only admissible if pre-computation was used during mixing, i.e., the maxciph file as described in Point 14 exists. They are also incompatible with `-nopos`.

Proof of shuffling. It must be possible to verify a proof of a shuffling using the following command.

```
verifier -shuffle protInfo.xml directory
```

For this command we set $\mathsf{auxsid}_{\mathit{expected}} = \texttt{default}$, $\omega_{\mathit{expected}} = \perp$, $\mathsf{posc} = \mathit{true}$, $\mathsf{ccpos} = \mathit{true}$, and $\mathsf{dec} = \mathit{false}$. For each additional option the needed changes to the parameters are listed below.

Option	Changes
`-auxsid <auxsid>`	$\mathsf{auxsid}_{\mathit{expected}} = $ `<auxsid>`
`-width <width>`	$\omega_{\mathit{expected}} = $ `<width>`
`-nopos`	$\mathsf{posc} = \mathsf{ccpos} = \mathit{false}$.
`-noposc`	$\mathsf{posc} = \mathit{false}$
`-noccpos`	$\mathsf{ccpos} = \mathit{false}$

The parameters must satisfy the same requirements as for proofs of mixing.

Proof of decryption. It must be possible to verify a proof of decryption using the following command.

```
verifier -decrypt protInfo.xml default
```

For this command we set $\mathsf{auxsid}_{\mathit{expected}} = \texttt{default}$, $\omega_{\mathit{expected}} = \perp$, $\mathsf{posc} = \mathit{false}$, $\mathsf{ccpos} = \mathit{false}$, and $\mathsf{dec} = \mathit{true}$. For each additional option the needed changes to the parameters are listed below.

Option	Changes
`-auxsid <auxsid>`	$\mathsf{auxsid}_{\textit{expected}} = $ `<auxsid>`
`-width <width>`	$\omega_{\textit{expected}} = $ `<width>`

The parameters must satisfy the same requirements as for proofs of mixing.

11 Additional Verifications Needed in Applications

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 may be application dependent. Thus, to verify the overall correctness in a given application, it must be verified that all parties agree on a scheme where:

1. The public key actually used by senders is an representation of pk.
2. The actual input ciphertexts is a representation of L_0.
3. In the case of verifying a shuffling without decryption, the actual output ciphertexts is a representation of L_λ.
4. In the case of verifying a mixing or a decryption the output plaintexts are correctly decoded from the group elements in m.

All of the above falls outside the scope of this document, since we can not anticipate the scheme used to represent objects or how the plaintext group elements are decoded into some other representations. However, one natural approach is to use an existing or custom representation in conjunction with the `vmnc` conversion tool documented in the Verificatum user manual [13].

12 Acknowledgments

The suggestions of Torbjörn Granlund, Tomer Hasid, Shahram Khazaei, Gunnar Kreitz, Olivier Pereira, and Amnon Ta-Shma have greatly improved the presentation.

References

[1] *Digital signature standard (DSS).* National Institute of Standards and Technology, Washington, 2000. URL: `http://csrc.nist.gov/publications/fips/`. Note: Federal Information Processing Standard 186-2.

[2] 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.

[3] 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.

[4] 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.

[5] A. Fiat and A. Shamir. How to prove yourself. practical solutions to identification and signature problems. In *Advances in Cryptology – Crypto '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–189. Springer Verlag, 1986.

[6] 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.

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

[8] B. Pfitzmann. Breaking an efficient anonymous channel. In *Advances in Cryptology – Eurocrypt '94*, volume 950 of *Lecture Notes in Computer Science*, pages 332–340. Springer Verlag, 1995.

[9] 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.

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

[11] 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.

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

[13] D. Wikström. User manual for the Verificatum mix-net. Manuscript, 2012. Available at `http://www.verificatum.org`.

A Test Vectors for Cryptographic Primitives

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

Seed (32 bytes):

```
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
```

Expansion (128 bytes):

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

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

Seed (48 bytes):

```
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
202122232425262728292a2b2c2d2e2f
```

Expansion (128 bytes):

```
e45ac6c0cafff343b268d4cbd773328413672a764df99ab823b53074d94152bd
27fc38bcffdb7c1dc1b6a3656b2d4819352c482da40aad3b37f333c7afa81a92
b7b54551f3009efa4bdb8937492c5afca1b141c99159b4f0f819977a4e10eb51
61edd4b1734717de4106f9c184a17a9b5ee61a4399dd755f322f5d707a581cc1
```

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

Seed (64 bytes):

```
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
202122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f
```

Expansion (128 bytes):

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

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

Input (32 bytes):

```
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
```

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

```
001a8d6b6f65899ba5
```

RandomOracle(Hashfunction(`"SHA-256"`), 261)

Input (32 bytes):

```
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
```

Output (33 bytes of which the last 261 bits may be non-zero):

```
1c04f57d5f5856824bca3af0ca466e283593bfc556ae2e9f4829c7ba8eb76db8
78
```

RandomOracle(Hashfunction(`"SHA-384"`), 93)

Input (32 bytes):

`000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f`

Output (12 bytes of which the last 93 bits may be non-zero):

`04713a5e22935833d436d1db`

RandomOracle(Hashfunction(`"SHA-384"`), 411)

Input (32 bytes):

`000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f`

Output (52 bytes of which the last 411 bits may be non-zero):

```
00dc086c320e38b92722a9c0f87f2f5de81b976400e2441da542d1c3f3f391e4
1d6bcd8297c541c2431a7272491f496b622266aa
```

RandomOracle(Hashfunction(`"SHA-512"`), 111)

Input (32 bytes):

`000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f`

Output (14 bytes of which the last 111 bits may be non-zero):

`28d742c34b97367eb968a3f28b6c`

RandomOracle(Hashfunction(`"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="statdist"
            type="xs:integer"
            minOccurs="1"
            maxOccurs="1"/>

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

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


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

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

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

<xs:element name="descr"
            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. WE
     RECOMMEND YOU TO NOT EDIT THIS FILE UNLESS YOU KNOW EXACTLY
     WHAT YOU ARE DOING. -->

<protocol>

   <!-- Protocol version for which this protocol info is intended. -->
   <version></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 that is NOT necessarily unique. -->
   <name>Swedish Election</name>

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

   <!-- Number of parties taking part in the protocol execution. -->
   <nopart>3</nopart>

   <!-- Statistical distance from uniform of objects sampled in
        protocols or in proofs of security. -->
   <statdist>100</statdist>

   <!-- Name of bulletin board implementation used. -->
   <bullboard>verificatum.protocol.com.BullBoardBasicHTTPW</bullboard>

   <!-- Threshold number of parties needed to violate privacy, i.e.,
        this is the number of parties needed to decrypt. -->
   <thres>2</thres>

   <!-- Group over which the protocol is executed. An instance of a
        subclass of verificatum.arithm.PGroup. -->
   <pgroup>ModPGroup(safe-prime modulus=2*order+1. order bit-length = 512
)::0000000002010000001c766572696669636174756d2e61726974686d2e4d6f64504772
6f757000000000040100000041014354c848a190b7b5fbddcd07bed36e59af5a50cc5966b
202bba0959ccc42061a2f468f87fa436451bd48d5cb333c0bb0aca763193e70c725495455
a99939276f010000004100a1aa642450c85bdafdeee683df69b72cd7ad28662cb359015dd
04ace6621030d17a347c3fd21b228dea46ae5999e05d85653b18c9f386392a4aa2ad4cc9c
93b70100000041013202741 3c1464af9b3ebe05f40059902857843365887f3e084e973dfd
3da198697724ac422dfce4728c2baa07760b5eae2d709bd7ff4f79d4e71fc9c2d37e26701
0000000400000001</pgroup>

   <!-- Bit length of challenges in interactive proofs. -->
```

```
<cbitlen>128</cbitlen>

<!-- Bit length of challenges in non-interactive random-oracle
     proofs. -->
<cbitlenro>256</cbitlenro>

<!-- Bit length of each component in random vectors used for
     batching. -->
<vbitlen>128</vbitlen>

<!-- Bit length of each component in random vectors used for
     batching in non-interactive random-oracle proofs. -->
<vbitlenro>256</vbitlenro>

<!-- Pseudo random generator used to derive random vectors from
     jointly generated seeds. This can be "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>

<!-- Default width of ciphertexts processed by the mix-net. A
     different width can still be forced for a given session by
     using the "-width" option. -->
<width>1</width>

<!-- Maximal number of ciphertexts for which precomputation is
     performed. Pre-computation can still be forced for a different
     number of ciphertexts for a given session using the "-maxciph"
     option during pre-computation. -->
<maxciph>10000</maxciph>

<party>

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

   <!-- 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>

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

```
      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048
)::0000000002010000002976657269666963617475 6d2e63727970746f2e5369676e6174
757265504b657948657572697374696300000000020100000004000008000100000126308
20122300d06092a864886f70d01010105000382010f003082010a0282010100927dfafb2e
b4b417c7404bd9b0856617226d242e7aebb2aeaad78a7477f3ab15bf488c4caf594770f7a
ff60c493943eb21d0e4b284c088884f9f327e1ea02834e19631311d079edec036224c09a5
e9d2c5959fbdec536042e4cd3223194c361991e95486b18a42db45410e7f1f2ddae9a6f9e
ca096208f892adea2699f58e42a81e7d1f2f21781899e4878777ddf148cefb77abf5ac1d6
e05492e3f107562bd32c3cfe49086b7af9a50c02fb2f9d73ed37d8d5dcff138ff49041d03
172126182797ce874c01221056d846c8227825fb4158c45808ed151e02807f4533cb3722d
b899f50271492a0a7f27d219c9223881cd7e7da0abf18182faea63133e5f06f7020301000
1</pkey>

      <!-- URL to the HTTP server of this party. -->
      <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>

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

      <!-- 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>

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

      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048
)::00000000020100000029766572696669636174756d2e63727970746f2e5369676e6174
757265504b657948657572697374696300000000020100000004000008000100000126308
20122300d06092a864886f70d01010105000382010f003082010a0282010100ad114c68c1
3c89c1e4a47880a51496481ab063b88f63976abdecd5fa60e56527a540e87028884c7bd36
c6211d41388c78ddb110f329db926e9c0b1d6f8e1fa7874b4002d92ecbccfaa3cdcd70667
1bc2d00567cfb7150a3987bb3ad82c666da96d600a0b59323831eedc7d695af1ba1aaaaf5
f26494b65601f7a87e96758ba2966dbf4e4ee678ac4746df4bc1ad6beb6087e25b57337f0
cb094937573cbfb03a12a04daeede9592cfa57879b987552580fbf65b9ecaf5a63455aa65
88bdd086820d3c10053e47abf58bc09632dc0dcc4498a4272b5805c5ca10f0afe589b9180
342b3d4c58b509287e92857ef57b2ab70d48de181d04ff93a111c262580705b1020301000
1</pkey>

      <!-- URL to the HTTP server of this party. -->
      <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>

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

      <!-- 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>

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

      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048
)::00000000020100000029766572696669636174756d2e63727970746f2e5369676e6174
757265504b657948657572697374696300000000020100000004000008000100000126308
20122300d06092a864886f70d01010105000382010f003082010a028201010091f11f4d46
29eca4ff3344d7231d63651c6106a8acf31973cafe1d1b65b325d54341947b739d3b30220
acb32429d9c369d6c5a3c3f0ffd9567cbaa6a4009e9971b01d6e4a87eb7447773c403c31a
38509a517076a4799cabf7e480905a6df4eb20edc0472f87e2fd827ec4e4e2ec26bfa7d0c
fd3473f9827fd1e0c022aada6fb5b6561923409f8fc219c7341c080117a1a339e1944a707
b69eb05de22804dffc1420dc76ff623aabec7e48cff22bf822c0737bcf8891fd341583b02
1881fee46aebd21b01d63081facd100ed2ee25ed2b44b9f93e9fe07db4e562a8671e72abb
55c5adf147716bcb74dadde2f3aa16670b497ff0e90b26d5fbaf7a6859400d43020301000
1</pkey>

      <!-- URL to the HTTP server of this party. -->
      <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 Zero-Knowledge Protocols

Protocol 29 (Proof of a Shuffle).
Common Input. Generators $g, h_0, \ldots, h_{N-1} \in G_q$ and Pedersen commitments $u_0, \ldots, u_{N-1} \in G_q$, a public key pk, elements $w_0, \ldots, w_{N-1} \in \mathcal{C}_\omega$ and $w'_0, \ldots, w'_{N-1} \in \mathcal{C}_\omega$.
Private Input. Exponents $s = (s_0, \ldots, s_{N-1}) \in \mathcal{R}_\omega^N$ and a permutation $\pi \in \mathbb{S}_N$ such that $w'_i = \mathsf{Enc}_{pk}(1, s_{\pi^{-1}(i)}) w_{\pi^{-1}(i)}$ for $i = 0, \ldots, N-1$.

1. $\mathcal{P}$ chooses $r = (r_0, \ldots, r_{N-1}) \in \mathbb{Z}_q^N$ randomly and computes $u_i = g^{r_{\pi(i)}} h_{\pi(i)}$.
2. $\mathcal{V}$ 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 $A = \prod_{i=0}^{N-1} u_i^{e_i}$ and $F = \prod_{i=0}^{N-1} w_i^{e_i}$.
3. $\mathcal{P}$ 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 $B_i = g^{b_i} B_{i-1}^{e'_i}$ 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 $\epsilon_0, \ldots, \epsilon_{N-1} \in [0, 2^{n_e+n_v+n_r} - 1]$, $\phi \in \mathcal{R}_\omega$ randomly, sets $B_{-1} = h_0$, and forms

$$
\begin{aligned}
A' &= g^\alpha \prod_{i=0}^{N-1} h_i^{\epsilon_i} & C' &= g^\gamma \\
B'_i &= g^{\beta_i} B_{i-1}^{\epsilon_i} \text{ for } i = 0, \ldots, N-1 & D' &= g^\delta \\
F' &= \mathsf{Enc}_{pk}(1, -\phi) \prod_{i=0}^{N-1} (w'_i)^{\epsilon_i} \ .
\end{aligned}
$$

   Then it hands (B, A', B', C', D', F') to $\mathcal{V}$.
4. $\mathcal{V}$ chooses $v \in [0, 2^{n_v} - 1]$ randomly and hands v to $\mathcal{P}$.
5. $\mathcal{P}$ computes $a = \langle r, e' \rangle$, $c = \sum_{i=0}^{N-1} r_i$, and $f = \langle s, e \rangle$. 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

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

   Then it hands $(k_A, k_B, k_C, k_D, k_E, k_F)$ to $\mathcal{V}$.
6. $\mathcal{V}$ computes $C = \prod_{i=0}^{N-1} u_i / \prod_{i=0}^{N-1} h_i$, $D = B_{N-1} / h_0^{\prod_{i=0}^{N-1} e_i}$, and sets $B_{-1} = h_0$ and accepts if and only if

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

Protocol 30 (Proof of a Shuffle of Commitments).
Common Input. Generators $g, h_0, \dots, h_{N-1} \in G_q$ and Pedersen commitments $u_0, \dots, u_{N-1} \in G_q$.
Private Input. Exponents $r = (r_0, \dots, r_{N-1}) \in \mathbb{Z}_q^N$ and a permutation $\pi \in \mathbb{S}_N$ such that $u_i = g^{r_{\pi(i)}} h_{\pi(i)}$ for $i = 0, \dots, N-1$.

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,
$$A = \prod_{i=0}^{N-1} u_i^{e_i} \enspace .$$
2. $\mathcal{P}$ computes the following, where $e'_i = e_{\pi^{-1}(i)}$:
   (a) *Bridging Commitments.* It chooses $b_0, \dots, b_{N-1} \in \mathbb{Z}_q$ randomly, sets $B_{-1} = h_0$, and forms
$$B_i = g^{b_i} B_{i-1}^{e'_i} \quad \text{for } i = 0, \dots, N-1 \enspace .$$
   (b) *Proof Commitments.* It chooses $\alpha, \beta_0, \dots, \beta_{N-1}, \gamma, \delta \in \mathbb{Z}_q$ and $\epsilon_0, \dots, \epsilon_{N-1} \in [0, 2^{n_e+n_v+n_r} - 1]$ randomly, sets $B_{-1} = h_0$, and forms
$$\begin{aligned} A' &= g^{\alpha} \prod_{i=0}^{N-1} h_i^{\epsilon_i} & C' &= g^{\gamma} \\ B'_i &= g^{\beta_i} B_{i-1}^{\epsilon_i} \quad \text{for } i = 0, \dots, N-1 & D' &= g^{\delta} \enspace . \end{aligned}$$
   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 $i = 1, \dots, N-1$. Finally, it sets $d = d_{N-1}$ and computes
$$\begin{aligned} k_A &= va + \alpha & k_C &= vc + \gamma \\ k_{B,i} &= vb_i + \beta_i \quad \text{for } i = 0, \dots, N-1 & k_D &= vd + \delta \\ & & k_{E,i} &= ve'_i + \epsilon_i \quad \text{for } i = 0, \dots, N-1 \enspace . \end{aligned}$$
   Then it hands $(k_A, k_B, k_C, k_D, k_E)$ to $\mathcal{V}$.
5. $\mathcal{V}$ computes
$$C = \frac{\prod_{i=0}^{N-1} u_i}{\prod_{i=0}^{N-1} h_i} \quad \text{and} \quad D = \frac{B_{N-1}}{h_0^{\prod_{i=0}^{N-1} e_i}} \enspace ,$$
   sets $B_{-1} = h_0$ and accepts if and only if
$$\begin{aligned} A^v A' &= g^{k_A} \prod_{i=0}^{N-1} h_i^{k_{E,i}} & C^v C' &= g^{k_C} \\ B_i^v B'_i &= g^{k_{B,i}} B_{i-1}^{k_{E,i}} \quad \text{for } i = 0, \dots, N-1 & D^v D' &= g^{k_D} \end{aligned}$$

Protocol 31 (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 $w_0, \ldots, w_{N-1} \in \mathcal{C}_\omega$ and $w'_0, \ldots, w'_{N-1} \in \mathcal{C}_\omega$.
Private Input. Exponents $r = (r_0, \ldots, r_{N-1}) \in \mathbb{Z}_q^N$, a permutation $\pi \in \mathbb{S}_N$, and exponents $s = (s_0, \ldots, s_{N-1}) \in \mathcal{R}_\omega^N$ 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 $A = \prod_{i=0}^{N-1} u_i^{e_i}$ and $B = \prod_{i=0}^{N-1} w_i^{e_i}$.

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 $\beta \in \mathcal{R}_\omega$ 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. $\mathcal{V}$ accepts if and only if

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