How to Implement a Stand-alone Verifier for the Verificatum Mix-Net when Used Only for Shuffling

Douglas Wikström
dog@csc.kth.se

August 27, 2013

Abstract

The Verificatum Mix-Net 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 implementors of stand-alone verifiers. For more information, see http://www.verificatum.org.

This is an abridged version of a longer document. Only parts needed to verify the correctness of a proof of a shuffle are included.

Contents

1 Introduction

The zero knowledge proofs in Verificatum mix-net (VMN) can be made non-interactive using the Fiat-Shamir heuristic [2] 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 allows anybody to verify the correctness of the execution as a whole, i.e., that the joint public key, the input ciphertexts, and the output ciphertexts 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.

2 Background

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

2.1 The El Gamal Cryptosystem

The El Gamal cryptosystem [1] is defined over a group G_q of prime order q. The set M of plaintexts is defined to be the group G_q and the set of ciphertexts C is the product space $G_q \times G_q$. The randomness used to encrypt is sampled from $\mathsf{R} = \mathbb{Z}_q$.

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 $m \in \mathsf{M}$, a random exponent $s \in \mathsf{R}$ is chosen and the ciphertext in 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 $\{0,1\}^t \to G_q$, 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

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

is an encryption of $m_1 m_2$. If we set $m_2 = 1$, then this feature can be used to *re-encrypt* (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 $\mathbb{Z}_q$.

Encrypting longer messages with multiple keys. The El Gamal cryptosystem can be generalized in several ways to encrypt longer messages. One way is to simply use multiple public keys. More precisely, suppose that $pk = (pk_1, \ldots, pk_\kappa)$ is a list of public keys with corresponding secret keys $sk = (sk_1, \ldots, sk_\kappa)$, where $pk_i \in G_q \times G_q$ and $sk_i \in \mathbb{Z}_q$. Then a message $m = (m_1, \ldots, m_\kappa) \in G_q^\kappa$ can be encrypted as

$$\mathsf{Enc}_{pk}(m, s) = \left(\mathsf{Enc}_{pk_1}(m_1, s_1), \ldots, \mathsf{Enc}_{pk_\kappa}(m_\kappa, s_\kappa)\right) \ ,$$

where $s = (s_1, \ldots, s_\kappa) \in \mathbb{Z}_q^\kappa$. We view this as the natural generalization of El Gamal to product groups. All we need to do is change the order of the group elements. We say that κ is the *key*

width and define the message space to be $\mathsf{M}_\kappa = G_q^\kappa$, the randomness space to be $\mathsf{R}_\kappa = \mathbb{Z}_q^\kappa$, and the ciphertext space to be $\mathsf{C}_\kappa = \mathsf{M}_\kappa \times \mathsf{M}_\kappa$. Then we define

$$(g_1, \ldots, g_\kappa)^{(s_1,\ldots,s_\kappa)} = (g_1^{s_1}, \ldots, g_\kappa^{s_\kappa}) \ ,$$

i.e., exponentiation is interpreted component wise. Let $g_i \in G_q$ be a generator for $i = 1, \ldots, \kappa$. Then the element $g = (g_1, g_2, \ldots, g_\kappa)$ generates M_κ in the sense that for each element $u \in \mathsf{M}_\kappa$ there is a unique vector $s \in \mathsf{R}_\kappa$ such that $u = g^s$.

A secret key for El Gamal with key width κ is a randomly chosen element $sk \in \mathsf{R}_\kappa$ and the corresponding public key pk is defined as (g, y), where $y = g^{sk}$. To encrypt a message $m \in \mathsf{M}_\kappa$ an element $s \in \mathsf{R}_\kappa$ is sampled randomly and then the ciphertext is computed as $\mathsf{Enc}_{pk}(m, s) = (g^s, y^s m)$, where $y^s m$ is interpreted as componentwise multiplication. Decryption and computation of decryption factors can be defined similarly.

Encrypting longer messages with multiple ciphertexts. Using a simple hybrid argument it is easy to see that a longer plaintext $m = (m_1, \ldots, m_\omega) \in \mathsf{M}_\kappa^\omega$ can be encrypted using a public key $pk \in \mathsf{C}_\kappa$ by encrypting each component independently, as $\big(\mathsf{Enc}_{pk}(m_1, s_1), \ldots, \mathsf{Enc}_{pk}(m_\omega, s_\omega)\big)$, where $s = (s_1, \ldots, s_\omega) \in \mathsf{R}_\kappa^\omega$ is chosen randomly.

It is convenient to generalize our notation similarly to the generalization used for multiple keys above. Thus, we let $\mathsf{M}_{\kappa,\omega} = \mathsf{M}_\kappa^\omega$ be the plaintext space, we let $\mathsf{R}_{\kappa,\omega} = \mathsf{R}_\kappa^\omega$ be the randomness space, and let $\mathsf{C}_{\kappa,\omega} = \mathsf{M}_{\kappa,\omega} \times \mathsf{M}_{\kappa,\omega}$ be the ciphertext space. With this notation encryption of a message $m \in \mathsf{M}_{\kappa,\omega}$ using randomness $s \in \mathsf{R}_{\kappa,\omega}$ is simply denoted $\mathsf{Enc}_{pk}(m, s) = (g^s, y^s m)$, where g is understood to be a generator of M_κ, $sk \in \mathsf{R}_\kappa$, and $y = g^{sk}$. Please note that with this generalization $pk \in \mathsf{C}_\kappa$ and not in $\mathsf{C}_{\kappa,\omega}$ which might have been expected by the reader. Decryption and computation of decryption factors can be defined in the natural way.

2.2 A Mix-Net Based on the El Gamal Cryptosystem

We use the re-encryption approach of Sako and Kilian [4] and the proof of a shuffle of Terelius and Wikström [5]. The choice of proof of a shuffle is mainly motivated by the fact that many other efficient proofs of shuffles are patented.

The mix-net is executed by k mix-servers with key width κ and width of ciphertexts ω.

Distributed key generation. A joint public key $pk = (g, y) \in \mathsf{C}_\kappa$ is provided by an external key generation algorithm. How this is done is not relevant for this document.

Shuffling. We denote the number of ciphertexts by N. The ith ciphertext $w_{0,i} = \mathsf{Enc}_{pk}(m_i, s_i)$ from the set $\mathsf{C}_{\kappa,\omega}$ encrypts some message $m_i \in \mathsf{M}_{\kappa,\omega}$ using randomness $s_i \in \mathsf{R}_{\kappa,\omega}$.

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 $r_{l,i} \in \mathsf{R}_{\kappa,\omega}$ and a permutation π_l randomly and outputs $L_l = (w_{l,0}, \ldots, w_{l,N-1})$, where
>
> $$w_{l,i} = w_{l-1,\pi_l(i)}\mathsf{Enc}_{pk}(\overline{1}, r_{l,\pi_l(i)}) \ . \tag{1}$$

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}.

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.

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 ξ_l is not a valid proof of knowledge of exponents $r_{l,i}$ and a permutation π_l such that $w_{l,i} = w_{l-1,\pi_l(i)}\mathsf{Enc}_{pk}(\overline{1}, r_{l,\pi_l(i)})$, then set $L_l = L_{l-1}$.

3 How to Write a Verifier

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

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

4.2 Representation as an Array of Bytes

We use $\mathsf{bytes}_k(n)$ as a short-hand to denote the $8k$-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 $\mathsf{leaf}(\texttt{65666768})$.

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

5.2 Pseudo-random Generators

We need a pseudo-random generator (PRG) to expand a short challenge string into a long "random" vector to use batching techniques in the zero-knowledge proofs of Section 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 $(true, false, true)$ is represented by $\mathsf{leaf}(\texttt{01 00 01})$.

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

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

6.5 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 $G_q = \mathsf{unmarshal}(s)$.

Multiplicative groups are implemented by the class `verificatum.arithm.ModPGroup` in Verificatum.

6.6 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.

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 $p = \texttt{ProtocolInfo(protocolInfo.xml)}$ 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.

$\kappa = \mathsf{int}(p[\texttt{keywidth}])$ specifies the key width.

$\omega_{default} = \mathsf{int}(p[\texttt{width}])$ specifies the default width of ciphertexts and plaintexts.

8 Verifying Fiat-Shamir 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.

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

$RO_{challenge} = \mathsf{RandomOracle}(H, n_v)$ is the random oracle used to generate challenges.

8.2 Independent Generators

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

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

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 6.6. The prefix ρ is computed in Step 4 of the main verification routine in Section 9.3 and given as input to Algorithm 17 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 [5].

Protocol 17 (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.
$\mathsf{R}_{\kappa,\omega}$	Randomizer group.
$\mathsf{C}_{\kappa,\omega}$	Ciphertext group.
pk	El Gamal public key.
w	Array $w = (w_0, \ldots, w_{N-1})$ of input ciphertexts in $\mathsf{C}_{\kappa,\omega}$.
w'	Array $w' = (w'_0, \ldots, w'_{N-1})$ of output ciphertexts in $\mathsf{C}_{\kappa,\omega}$.
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 $A', C', D' \in G_q$, $F' \in \mathsf{C}_{\kappa,\omega}$, 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 $k_A, k_C, k_D \in \mathbb{Z}_q$, $k_F \in \mathsf{R}_{\kappa,\omega}$, and k_B and k_E are arrays 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

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

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

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

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.

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

2. 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.

3. 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.

4. FullPublicKey.bt – Full public key used to form input ciphertexts. The required format of this file is a byte tree $\overline{pk}$, where $pk \in M_\kappa \times M_\kappa$. Here the key width κ, the basic message space $M_\kappa = G_q^\kappa$, and the underlying prime order group G_q are derived from the protocol info file.

5. 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 $C_{\kappa,\omega}$. This defines N, the number of input ciphertexts.

6. 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 $\mathsf{C}_{\kappa,\omega}$.

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 proofs of shuffles.

7. 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 $\mathsf{C}_{\kappa,\omega}$ (and N is the number of elements in the list L_0 of input ciphertexts).

8. 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 17.

9. 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 17.

10. 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 17.

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
version	1	version
auxsid	2	auxsid
ω	3	width
pk	4	FullPublicKey.bt
L_0	5	Ciphertexts.bt
L_λ	6	ShuffledCiphertexts.bt
L_l	7	Ciphertexts$\langle l \rangle$.bt
μ_l	8	PermutationCommitment$\langle l \rangle$.bt
τ_l^{pos}	9	PoSCommitment$\langle l \rangle$.bt
σ_l^{pos}	10	PoSReply$\langle l \rangle$.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.

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 18 (Verifier of Shuffling)**.**

Input	Description
ρ	Prefix to random oracles.
λ	Number of mix-servers.
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.
$\mathsf{R}_{\kappa,\omega}$	Randomness group.
$\mathsf{C}_{\kappa,\omega}$	Ciphertext group.
pk	Joint public key.
L_0	Original ciphertexts.
L_λ	Shuffled ciphertexts.

Program

For $l = 1, \ldots, \lambda$ do:

1. **Array of ciphertexts.** If $l < \lambda$, then read the array L_l of N ciphertexts in $\mathsf{C}_{\kappa,\omega}$ as described in Point 7. 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 9 and Point 10, respectively. Then execute Algorithm 17 on input

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

   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.**

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 19 (Verifier)**.**

Input	Description
protinfo	Protocol info file.
directory	Directory containing proof.
$\mathsf{auxsid}_{expected}$	Expected auxiliary session identifier.
$\omega_{expected}$	Expected width of ciphertexts.

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}, s_H, κ, and $\omega_{default}$.

2. **Proof parameters.** Read $\mathsf{version}_{proof}$, auxsid, ω, from the proof directory directory as described in Point 1 – Point 3. If this fails, then **reject**. If $\mathsf{version}_{proof} \neq \mathsf{version}_{prot}$, then **reject**. If 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 plaintexts $M_\kappa = G_q^\kappa$, randomness $R_\kappa = \mathbb{Z}_q^\kappa$, and ciphertexts $C_\kappa = M_\kappa \times M_\kappa$ defined for a given key width κ. Then define the generalized set of plaintexts $M_{\kappa,\omega} = M_\kappa^\omega$, the set of randomness $R_{\kappa,\omega} = R_\kappa^\omega$, and the set of ciphertexts $C_{\kappa,\omega} = M_{\kappa,\omega} \times M_{\kappa,\omega}$. Finally, define the hash function $H = \mathsf{Hashfunction}(s_H)$, and the pseudo-random generator $PRG = \mathsf{PRG}\big(\mathsf{Hashfunction}(s_{PRG})\big)$. 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

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

Algorithm 20 (Algorithm 19 continued)**.**

5. **Read key.** Attempt to read the joint public key pk, where $pk = (g, y)$ is contained in $\mathsf{C}_\kappa\omega$, from file as described in Point 4. If this fails, then **reject**.

6. **Read lists.**

   (a) **Read input ciphertexts.** Read the array L_0 of N ciphertexts as described in Point 7 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.** Read L_λ as described in Point 6.

7. **Verify relations between lists.**

   (a) **Verify shuffling.** Execute Algorithm 18 on input

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

   If it rejects, then **reject**.

   (b) **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 [6] and behave correspondingly. Below we specify how each standard command line expression is translated into a call to Algorithm 19.

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 19. For each type of proof and proof parameters specified on the command line below, we assign values to the variables: $\mathsf{auxsid}_{expected}$, $\omega_{expected}$, and then require that the output of the command is given by

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

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 $\mathsf{auxsid}_{expected}$ = `default`, $\omega_{expected} = \perp$ For each additional option the needed changes to the parameters are listed below.

Option	Changes
`-auxsid <auxsid>`	$\mathsf{auxsid}_{expected}$ = `<auxsid>`
`-width <width>`	$\omega_{expected}$ = `<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 the output list of ciphertexts 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. The actual output ciphertexts is a representation of L_λ.

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.

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] 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.

[2] 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.

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

[4] 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.

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

[6] 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="keywidth"
            type="xs:integer"
            minOccurs="1"
            maxOccurs="1"/>

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

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

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

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

<xs:element name="prg"
            type="xs:string"
```

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

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

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

<xs:element name="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>1.1.7</version>

   <!-- Session identifier of this protocol execution. This must be
        globally unique. -->
   <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 = 511
)::0000000002010000001c766572696669636174756d2e61726974686d2e4d6f64504772
6f7570000000000401000000041009a91c3b704e382e0c772fa7cf0e5d6363edc53d156e84
1555702c5b6f906574204bf49a551b695bed292e0218337c0861ee649d2fe4039174514fe
2c23c10f6701000000404d48e1db8271c17063b97d3e7872eb1b1f6e29e8ab7420aaab816
2db7c832ba1025fa4d2a8db4adf69497010c19be0430f7324e97f201c8ba28a7f1611e087
```

```
b3010000004100300763b0150525252e4989f51e33c4e6462091152ef2291e45699374a3a
a8acea714ff30260338bddbb48fc7446b273aaada90e3ee8326f388b582ea8a0735020100
00000400000001</pgroup>

   <!-- Width of El Gamal keys. If equal to one the standard El Gamal
        cryptosystem is used, but if it is greater than one, then the
        natural generalization over a product group of the given width
        is used. This corresponds to letting each party holding
        multiple standard public keys. -->
   <keywidth>1</keywidth>

   <!-- 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 party. -->
      <descr></descr>

      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048
)::00000000201000000297665726966696361747 56d2e63727970746f2e5369676e6174
757265504b65794865757269737469630000000002010000000400000800010000012630 8
20122300d06092a864886f70d01010105000382010f003082010a02820101009ede9fabab
1c967d60687424833be932b97c9b901de53edc2cae81130aa0a911c79421f320a77d5b018
d487d9536c9803a35a209a6843ab1865b8d80de374360f3e69ee0e19a36d57da64362d22e
3b25e4c4ea5efbea312ffa269d7c7ae90b850e91756431b63b04ea10570851a5cf32d1817
0507889aca76db264daa452e713d25ed7ed8536c1a266fc4762e34138f2550bdd84abed07
68b88fba6a10e41754bc8308a474d3cf373b6b293f3e320b9c37c972eaf9a736c09aff6de
98dcdab5ce998f25339c3ff89dbd6ac73192ce3ddfb612034973a01009b9c79a493bc478b
fe27af3f227d75eccd973785e6e1904b7c494709d048bfb6f224d7a8c461ff33020301000
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 party. -->
      <descr></descr>

      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048
)::00000000201000000297665726966696361747 56d2e63727970746f2e5369676e6174
757265504b65794865757269737469630000000002010000000400000800010000012630 8
20122300d06092a864886f70d01010105000382010f003082010a0282010100b5df4fd1f1
73bd8989426a15081a06a30f865719282bd24b18a833bca37ceac4ff97e598b2ea6b7d4ce
e58f02ea320bbf96bd0dfda0e840c07050d5b5b5e78c05106eaa226ce1ca305a7873f6287
e8c8ea79678b7ad916750659e466e828fc4154057e926a546153fc967ac1767e5ed286ead
46404a4aa949f43ffe48b1976ce2667bd3287ca7fdb8e0b89f9c58e770bede067fef17a56
446549d441786b0cdb5d955541686074db628ca85623357cc0c1bc8b44301b656db20c01c
```

```
691843c9977cff05e7d53d71987bf139ebd2deb768d4917c6bb6175bc74462419f02a92c1
6e8a9384e16d5d13c23f6f944befdc8591ebc7e661e6a6c2fb12bf2f7de292ab020301000
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 party. -->
      <descr></descr>

      <!-- Public signature key (instance of crypto.SignaturePKey). -->
      <pkey>verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048
)::00000000020100000029766572696669636174756d2e63727970746f2e5369676e6174
757265504b65794865757269737469630000000002010000000400000800010000012630 8
20122300d06092a864886f70d01010105000382010f003082010a02820101009815a090d5
44fde17f594c63509c6a0f29217b8652857501e28fb94845eb0851cc00d343e321e103fe7
c1d05053e7ffcc8ed4f5f1db0d63f048a8eac5a2bcdedd12ba4f23dc44ea358acfa09c2c2
7783fd879e790c3fe04f97fdf781be52f27dd374c35583faa5754e3f08987da0009ab62e0
42729c12a270ed374c939d4bed4a460c756fc273abab082e45e252b11c447fd31e6fb6f4c
20ce4d363742790251bc5dc6de01283c1314ef6fe504a6fed27c95c3c1462b0c50c12c7d0
e1bd481944eb251169d7f18084b5897ec5820c5a19cec05d2016a459047d7cff6d2af0a11
2bcbbefb5199657ba8df302eff45eea7277b0e1a10041061189fbd39986481b3020301000
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 21 (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 $w_0, \ldots, w_{N-1} \in \mathsf{C}_\omega$ and $w'_0, \ldots, w'_{N-1} \in \mathsf{C}_\omega$.
Private Input. Exponents $s = (s_0, \ldots, s_{N-1}) \in \mathsf{R}_\omega^N$ 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. 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. 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 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. 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 $\epsilon_0, \ldots, \epsilon_{N-1} \in [0, 2^{n_e + n_v + n_r} - 1]$, $\phi \in \mathsf{R}_\omega$ 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 V.
4. V chooses $v \in [0, 2^{n_v} - 1]$ randomly and hands v to P.
5. 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 V.
6. 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

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