Verificatum

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

VMN Version 3.0.4
2018-02-14

The Verificatum Mix-Net is an implementation of an El Gamal-based mix-net which uses the Fiat-Shamir heuristic to produce a non-interactive universally verifiable zero-knowledge 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 please visit `http://www.verificatum.org`.

Contents

1 Introduction

The zero knowledge proofs in Verificatum mix-net (VMN) can be made non-interactive using the Fiat-Shamir heuristic [5] and this is also the default behavior. 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 plaintexts/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.

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 permute a list of ciphertexts (shuffle session), decrypt a list of ciphertexts (decryption session), or re-encrypt, decrypt and permute a list of ciphertexts (mixing session).

Accordingly, there are three types of proofs, but the proof of a mixing session essentially 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 VMN (in the case where the Fiat-Shamir heuristic is applied).

2.1 El Gamal Cryptosystem

The El Gamal cryptosystem [3] is defined over a group G_q of prime order q with standard generator g. 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 $\mathcal{M} \times \mathcal{M}$. The randomness used to encrypt is sampled from $\mathcal{R} = \mathbb{Z}_q$.

A secret key $x \in \mathcal{R}$ is sampled randomly, and a corresponding public key $pk = (g, y)$ is defined by $y = g^x$. 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 such 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 $\{0,1\}^t \to \mathcal{M}$, which can be efficiently computed and inverted.

Homomorphic. The cryptosystem is homomorphic, i.e., for every public key pk and every ciphertexts

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

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 $\mathcal{R}$.

Distributed El Gamal Cryptosystem. The El Gamal cryptosystem also allows efficient protocols for distributed key generation and distributed decryption of ciphertexts by k parties with threshold λ, i.e., λ parties are needed to decrypt.

The distributed key generation protocol generates a standard Shamir secret sharing polynomial $p(z) = \sum_{s=0}^{\lambda-1} \gamma_s z^s$ of degree $(\lambda - 1)$ defined over $\mathcal{R}$. The joint output of the protocol is then a *polynomial in the exponent* $(\Gamma_0, \ldots, \Gamma_{\lambda-1})$ defined by $\Gamma_s = g^{\gamma_s}$ and the secret output x_l to the lth party is defined as the secret share $x_l = p(l)$. The joint public key is then $y = \Gamma_0 = g^x$, where $x = p(0)$, and the public key y_l of the lth party can be derived as $y_l = \prod_{s=0}^{\lambda-1} \Gamma_s^{l^s} = g^{x_l}$. The details [4, 6] of the verifiable secret sharing scheme are not important in this document.

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. Let Δ be a set of size λ of indices l such that the proofs are correct. Then Lagrange coefficients

$$c_l = \prod_{i \in \Delta \setminus \{l\}} \frac{i}{i - l}$$

such that $\sum_{l \in \Delta} c_l x_l = x$ can be computed. Instead of recovering x in the open we can perform similar operations "in the exponent". More precisely, decryption factors can be combined to a joint decryption factor

$$f = \prod_{l \in \Delta} f_l^{c_l} = \mathsf{PDec}_x(u, v) \enspace .$$

The ciphertext can then be trivially decrypted as $\mathsf{TDec}((u, v), f) = vf = m$. If no set Δ of size λ exists, then no knowledge about the plaintext is leaked and the ciphertexts can not be decrypted.

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 $pk_i \in \mathcal{C}$ and $sk_i \in \mathcal{R}$. Then a message $m = (m_1, \ldots, m_\kappa) \in \mathcal{M}^\kappa$ can be encrypted as

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

where $s \in \mathcal{R}^\kappa$. We view this as the natural generalization of El Gamal to product groups. We say that κ is the *key width* and define the message space to be $\mathcal{M}_\kappa = \mathcal{M}^\kappa$, the randomness space to be $\mathcal{R}_\kappa = \mathcal{R}^\kappa$, and the ciphertext space to be $\mathcal{C}_\kappa = \mathcal{M}_\kappa \times \mathcal{M}_\kappa$.

Let $g_{\mathcal{M}} \in \mathcal{M}$ be a generator. Then the element $g = (g_{\mathcal{M}}, g_{\mathcal{M}}, \ldots, g_{\mathcal{M}})$ generates $\mathcal{M}_\kappa$ in the sense that for each element $u \in \mathcal{M}_\kappa$ there is a unique vector $s \in \mathcal{R}_\kappa$ such that $u = (g_{\mathcal{M}}^{s_1}, \ldots, g_{\mathcal{M}}^{s_\kappa})$. We use the following notation

$$g^s = g^{(s_1, \ldots, s_\kappa)} = (g_{\mathcal{M}}^{s_1}, \ldots, g_{\mathcal{M}}^{s_\kappa}) \enspace ,$$

i.e., exponentiation is interpreted component wise.

With this notation, a secret key sk for El Gamal with key width κ is a randomly chosen element $x \in \mathcal{R}_\kappa$ and the corresponding public key pk is defined as (g, y), where $y = g^x$. To encrypt a message $m \in \mathcal{M}_\kappa$ an element $s \in \mathcal{R}_\kappa$ 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 component-wise multiplication. Decryption and computation of decryption factors is defined similarly.

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

It is convenient to generalize our notation similarly to the generalization used for multiple keys above. Thus, we let $\mathcal{M}_{\kappa,\omega} = \mathcal{M}_\kappa^\omega$ be the plaintext space, we let $\mathcal{R}_{\kappa,\omega} = \mathcal{R}_\kappa^\omega$ be the randomness space, and let $\mathcal{C}_{\kappa,\omega} = \mathcal{M}_{\kappa,\omega} \times \mathcal{M}_{\kappa,\omega}$ be the ciphertext space. With this notation, encryption of a message $m \in \mathcal{M}_{\kappa,\omega}$ using randomness $s \in \mathcal{R}_{\kappa,\omega}$ is simply denoted $\mathsf{Enc}_{pk}(m, s) = (g^s, y^s m)$, where g is understood to be a generator of $\mathcal{M}_\kappa$.

Please note that with this generalization $pk \in \mathcal{C}_\kappa$ and not in $\mathcal{C}_{\kappa,\omega}$ which might have been expected by the reader, but this is natural since the ω is variable even for a fixed public key. Decryption and computation of decryption factors can be defined in the natural way.

2.2 Mix-Net Based on the El Gamal Cryptosystem

We use the re-encryption approach of Sako and Kilian [10] and the proof of a shuffle of Terelius and Wikström [11]. The choice of proof of a shuffle is mainly motivated by the fact that many other efficient proofs of shuffles are harder to understand and implement. Some of them are also patented. We use the batching technique of Bellare et al. [1] with a twist to speed up the proofs needed during distributed decryption. Optionally the pre-computation technique proposed by Wikström [12] is used. The mix-net is executed by k mix-servers with key width κ and width of ciphertexts ω.

Distributed key generation. Let g be a generator of $\mathcal{M}_\kappa$. The mix-servers first run a distributed key generation protocol for a key width of κ. This generates a Shamir secret sharing polynomial $p(z) = \sum_{s=0}^{\lambda-1} \gamma_s z^s$ where $\gamma_s \in \mathcal{R}_\kappa$. Then a polynomial in the exponent $(\Gamma_0, \ldots, \Gamma_{\lambda-1})$ is defined by $\Gamma_s = g^{\gamma_s}$. Note that these elements belong to $\mathcal{M}_\kappa$. From these values a secret key $x = p(0)$ in $\mathcal{R}_\kappa$ (that is never recovered) and a (partial) public key $y = g^x = \Gamma_0$ are defined. The full public key is then the pair $pk = (g, y)$. Finally, each party receives a secret share $x_l = p(l)$ in $\mathcal{R}_\kappa$ for which a corresponding public key y_l can be derived as $y_l = \prod_{s=0}^{\lambda-1} \Gamma_s^{l^s}$.

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

To avoid Pfitzmann's attack [9] and preserve privacy, the sender of a ciphertext must prove knowledge of its plaintext. This can be ensured in different ways, but it is of no concern in this document. Furthermore, in some applications of mix-nets, the content of all input ciphertexts may be known and the randomness replaced by a public constant. However, with proper inputs the proof of correctness does not only guarantee the *correctness* of an execution, it also guarantees *privacy* regarding the correspondence between inputs and outputs, under the assumption that at most $\lambda - 1$ mix-servers are corrupted.

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.

VMN allows deactivating some mix-servers, i.e., they are treated as if they output a fixed message for any request. To ensure that at least λ active mix-servers are involved in the shuffling, we need λ_a mix-servers to be involved in the shuffling, where λ_a is the smallest index such that there are at least λ active mix-servers with index smaller or equal to λ_a. This guarantees that during processing at least one honest mix-server is involved in the shuffling, which in turn implies that privacy is preserved.

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 $l = 1, \ldots, \lambda_a$:

- 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 $r_{l,i} \in \mathcal{R}_{\kappa,\omega}$ 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 $r_{l,i} \in \mathcal{R}_{\kappa,\omega}$ 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}. (Note that this happens for every deactivated mix-server.)

Decryption. Finally, the mix-servers jointly decrypt the ciphertexts in L_{λ_a} as described in Section 2.1, but with a twist. More precisely, the lth mix-server computes $f_l = \mathsf{PDec}_{x_l/\alpha}(L_{\lambda_a})$, where $\alpha = \big(\mathrm{lcm}(1, \ldots, k)\big)^2$ is the square of the least common multiple of the integers $1, 2, \ldots, k$. Then it proves that the secret key x_l it used is given by $y_l = g^{x_l}$. Let Δ be a set of size λ of indices l such that the proofs are correct. Then the decryption factor can be computed as

$$f = \prod_{l \in \Delta} f_l^{\alpha c_l} = \mathsf{PDec}_x(L_{\lambda_a}) \ ,$$

where

$$c_l = \prod_{i \in \Delta \setminus \{l\}} \frac{i}{i - l}$$

is the lth Lagrange coefficient and the output of the mix-net is computed as $\mathsf{TDec}(L_{\lambda_a}, f)$. Note that we introduce α when computing decryption factors of the parties and then cancel it when combining the decryption factors. There is virtually no additional cost in computing α and x_l/α before computing the decryption factors of the lth party. The reason we introduce α is that αc_l is a relatively *small integer* if k is not too large. This reduces the complexity of computing f drastically, since no full exponentiations are needed to combine the decryption factors for practical values of k.

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. **Shuffling.** For $l = 1, \ldots, \lambda_a$: if ξ_l is not a valid proof of knowledge of exponents $r_{l,i}$ and a permutation π_l such that $w_{l,i} = w_{l-1,\pi_l(i)} \mathsf{Enc}_{pk}(1, r_{l,\pi_l(i)})$, then set $L_l = L_{l-1}$. If less than λ proofs are valid, then **reject**.

2. **Joint decryption.** A set Δ of size λ is read from the proof. Check that $f = \prod_{l \in \Delta} f_l^{\alpha c_l} = \mathsf{PDec}_x(L_{\lambda_a})$ by combining and checking the individual proofs of the mix-servers. If this does not give a joint valid proof, then **reject**.

3. **Check output.** Check if the output of the mix-net equals $\mathsf{TDec}(L_{\lambda_a}, f)$ if both are sorted lexicographically. 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 a subset of the phases described above.

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 their dependencies.

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 must be extracted.
5. **Verifying Fiat-Shamir Proofs.** Section 8 explains in detail 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.

3.3 Level of Independence of Verifiers

VMN is implemented in Java with some native code written in C based on the GMP [7] library and shell script wrappers. Thus, the code is compiled using Java and C compilers and executed using a Java virtual machine, native libraries from GMP, and shell scripts. Furthermore, all of it is executed within an operating system on a given hardware.

It may seem paranoid, but at each level it is conceivable that an attacker can covertly modify the code of VMN such that the built-in verifier in VMN accepts an invalid proof for a mix-net session that produces incorrect output.

Thus, the goal of an implementor of an independent verifier should be to minimize the common dependencies with VMN, i.e., the verifier should preferably be implemented using other languages, libraries, operating system, and hardware.

To see why this is important, consider, e.g., that at the core of any verifier it is checked that large integers are equal. It is conceivable that an adversary could inject code that alters the behavior of both VMN and all verifiers (built-in and external) in a correlated way such that the output of the mix-net is incorrect, the proof is incorrect, but still the verifiers accept the proof. This could possibly be done by introducing special random patterns to signal to the injected code snippets to be activated.

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

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

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, where n is given in decimal notation. 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
```

4.3 ASCII Strings

ASCII strings are identified with the corresponding byte arrays. 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 $\mathsf{leaf}(\texttt{65666768})$.

4.4 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 a. 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 and pseudo-random generators, and random oracles derived from these.

5.1 Hash Functions

VMN allows an arbitrary hash function to be used, but in this document we restrict our attention to the SHA-2 family [8], i.e., SHA-256, SHA-384, and SHA-512. Future versions of VMN will allow usage of SHA-3 (Keccak) as well. We use the following notation.

- $\mathsf{Hashfunction}(s)$ – Creates a hash function where s is 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. We sometimes abuse notation and simply write $H(t)$ for a byte tree t meaning $H(b)$, where b is the representation of t as an array of bytes.
- $\mathsf{outlen}(H)$ – Denotes the number of bits in the output of the hash function H. This is assumed to be a multiple of 8.

Example 4. If $H = \mathsf{Hashfunction}(\texttt{"SHA-256"})$ and t is a byte tree then $H(t)$ 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. VMN 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 a positive 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 abuse notation and 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 $n_{out} < 2^{31}\mathsf{outlen}(H)$ 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 VMN is represented as a byte tree. In this section we pin down the details of these representations. We describe how to represent elements in product rings and product groups, as well as arrays of such elements. These products are basically lists of elements and operations are applied element wise.

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

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 only 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
```

6.3 Product Rings

Product ring element. An element $a = (a_1, \ldots, a_k)$ in a product ring is represented by a byte tree $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_k})$, 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
```

6.4 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

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

where the integer e encoded as four bytes determines how a string is encoded into a group element and can be ignored for the purpose of this document. We stress that $\overline{g}$ is the byte tree representation of g viewed as a group element as defined below.

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`.

6.5 Standard Elliptic Curves over Prime Order Fields

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

Example 17. The group P-256 from FIPS 186-3 [2] is represented by $\mathsf{leaf}($`"P-256"`$)$.

The following curves are currently implemented by VMN, 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 curve be defined over a prime order field $\mathbb{Z}_p$ and let k be the smallest integer such that p can be represented as $\mathsf{bytes}_k(p)$. 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 $\mathsf{node}(\mathsf{leaf}(\mathsf{bytes}_k(-1)), \mathsf{leaf}(\mathsf{bytes}_k(-1)))$. Note that a fixed-size representation of -1 is used.

6.6 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 $a = (a_1, \ldots, a_k)$ in a product group is represented by $\mathsf{node}(\overline{a_1}, \ldots, \overline{a_k})$, where $\overline{a_i}$ is the byte tree representation of a_i.

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

6.7 Marshalling Groups

When objects are converted to byte trees in VMN, they do not store the name of the Java class of which they are instances. Thus, to recover an object from such a representation additional information must be available. Java serialization would not be language independent. Furthermore, only a few objects must be converted, so we use a simplified scheme where a group G_q represented by an instance of a Java class `PGroupClass` in VMN 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 are 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. This is denoted by $G_q = \mathsf{unmarshal}(s)$.

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

Implementation	**Description**
`com.verificatum.arithm.ModPGroup`	Multiplicative groups.
`com.verificatum.arithm.ECqPGroup`	Standard elliptic curve groups over prime order fields.

Example 18. The standard NIST curve P-256 [2] is marshalled into

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

6.8 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 $h_i = g^{x_i}$.)

Multiplicative group. Let G_q be the subgroup of prime order q of the multiplicative group $\mathbb{Z}_p$, where $p > 3$ is prime and $q \nmid (p-1)/q$. 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 $\mathbb{Z} \to G_q$.

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 nominal 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 G_q be an elliptic curve of order q over a prime order field $\mathbb{Z}_p$ defined by an equation $y^2 = f(x) \bmod p$. 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 $(t_0, \ldots, t_a) = PRG(s)$, where $t_l \in \{0,1\}^{8\lceil (n_p+n_r)/8 \rceil}$ is interpreted as a *non-negative* integer.
3. Set $t'_l = t_l \bmod 2^{n_p+n_r}$ and let $z_l = t'_l \bmod p$.
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 $j = 0$ and $l = 0$.

(b) Let $i \geqslant j$ be the smallest integer such that $f(z_i)$ is a quadratic residue modulo p. Then set $j = i + 1$, $x_l = z_i$, and $l = l + 1$.

(c) Let y be the square root of $f(x_l)$ that is smallest when viewed as an integer in $[0, p-1]$. Then go to step 4b.

In other words, first we generate random positive integers in $[0, 2^{n_p+n_r} - 1]$. Then these integers are reduced modulo p to get almost uniformly generated integers z_i in $[0, p - 1]$. 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 $\mathbb{Z}_p$ 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.

Note that the non-random choice of y-coordinate for a given x-coordinate does not make it easier to find a non-trivial representation of the unit element, since the two roots correspond to inverses as group elements.

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 UTF-8 encoding and a simple XML format and contains a single block of the form `<protocol></protocol>`. 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, where k is represented in decimal, 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 of that party, e.g., 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 `protInfo.xml` is a protocol info file, then we denote by $D = \mathsf{ProtocolInfo}($`protInfo.xml`$)$ an object such that $D[b]$ 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, but a complete example is given in Listing C in Appendix C.

Listing B in Appendix B contains the formal XML schema for protocol info files, but this schema depends on the type of bulletin board used, 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 a decimal ASCII string s as an integer we simply write $\mathsf{int}(s)$, e.g., $\mathsf{int}($`"123"`$) = 123$. We let $D =$ `ProtocolInfo(protInfo.xml)` and define the values we later use in Section 8

```
<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.

and Section 9.

- $\mathsf{version}_{prot} = D[\texttt{version}]$ is the version of VMN used during the execution which produced the proof.
- $\mathsf{sid} = D[\texttt{sid}]$ is the globally unique session identifier tied to the generation of a particular joint public key by the mix-net.
- $k = \mathsf{int}(D[\texttt{nopart}])$ specifies the number of parties.
- $\lambda = \mathsf{int}(D[\texttt{thres}])$ specifies the threshold number of mix-servers needed to decrypt ciphertexts.
- $n_e = \mathsf{int}(D[\texttt{ebitlenro}])$ specifies the number of bits in each component of random vectors used for batching in proofs of shuffles and proofs of correct decryption.
- $n_r = \mathsf{int}(D[\texttt{statdist}])$ 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.
- $n_v = \mathsf{int}(D[\texttt{vbitlenro}])$ specifies the number of bits used in the challenge of the verifier in zero-knowledge proofs.
- $s_H = D[\texttt{rohash}]$ specifies the hash function $H = \mathsf{Hashfunction}(s_H)$ used to implement the random oracles.
- $s_{PRG} = D[\texttt{prg}]$ specifies the hash function used to implement the pseudo-random generator used to expand challenges into arrays. Let $PRG = \mathsf{PRG}(\mathsf{Hashfunction}(s_{PRG}))$
- $s_{G_q} = D[\texttt{pgroup}]$ specifies the underlying group $G_q = \mathsf{unmarshal}(s_{G_q})$.
- $\kappa = \mathsf{int}(D[\texttt{keywidth}])$ specifies the key width. This also defines the plaintext space $\mathcal{M}_\kappa = G_q^\kappa$, the randomness space $\mathcal{R}_\kappa = \mathbb{Z}_q^\kappa$, and the ciphertext space $\mathcal{C}_\kappa = \mathcal{M}_\kappa \times \mathcal{M}_\kappa$.
- $\omega_{default} = \mathsf{int}(D[\texttt{width}])$ 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 shuffle 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 the PRG, which in turn is used to generate arrays of integers used for batching.
- $RO_{\mathit{challenge}} = \mathsf{RandomOracle}(H, n_v)$ 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, the auxiliary session identifier, and a string specifying the intended use of the "independent" generators. Then the generators are defined by

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

which is defined in Section 6.8. The prefix ρ is computed in Step 4 of the main verification routine in Section 9.3 and given as input to Algorithm 19, Algorithm 20, Algorithm 21, and Algorithm 22 below. The length N' is at least N and larger if needed for pre-computation, where N is the number of input ciphertexts.

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. We only describe the computations performed by the verifier for a specific application of Wikström [11] for more details.

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

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

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

$$F^v F' = \mathsf{Enc}_{pk}(1, -k_F) \prod_{i=0}^{N-1} (w_i')^{k_{E,i}} \ .$$

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. We only describe the computations performed by the verifier for a specific application of the Fiat-Shamir heuristic. See Appendix D and Terelius and Wikström [11] for more details.

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 $u = (u_0, \ldots, u_{N_0-1})$ 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 and k_E are arrays of N_0 elements in $\mathbb{Z}_q$.

   Reject if this fails.
2. Compute a seed $s = RO_{seed}\big(\rho \mid \mathsf{node}(\overline{g}, \overline{h}, \overline{u})\big)$.
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 $C = \prod_{i=0}^{N_0-1} u_i / \prod_{i=0}^{N_0-1} h_i$ 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. See Appendix D and Wikström [12] for more details.

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 with standard generator g.
$\mathcal{C}_\kappa$	Public key group.
$\mathcal{R}_{\kappa,\omega}$	Randomizer group.
$\mathcal{C}_{\kappa,\omega}$	Ciphertext group.
pk	El Gamal public key in $\mathcal{C}_\kappa$.
w	Array $w = (w_0, \ldots, w_{N-1})$ of input ciphertexts in $\mathcal{C}_{\kappa,\omega}$.
w'	Array $w' = (w'_0, \ldots, w'_{N-1})$ of output ciphertexts in $\mathcal{C}_{\kappa,\omega}$.
u	Shrunk array $u = (u_0, \ldots, u_{N-1})$ of Pedersen commitments in G_q.
τ^{ccpos}	Commitment of the Fiat-Shamir proof.
σ^{ccpos}	Reply of the Fiat-Shamir proof.

Program

1. (a) Interpret τ^{ccpos} as $\mathsf{node}(\overline{A'}, \overline{B'})$, where $A' \in G_q$ and $B' \in \mathcal{C}_{\kappa,\omega}$.
   (b) Interpret σ^{ccpos} as $\mathsf{node}(\overline{k_A}, \overline{k_B}, \overline{k_E})$, where $k_A \in \mathbb{Z}_q$, $k_B \in \mathcal{R}_{\kappa,\omega}$, 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. [1], but here we combine the proofs of all parties into one before verifying.

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

Input	**Description**
ρ	Prefix to random oracles.
N	Size of the arrays.
n_e	Number of bits in each component of random vectors used for batching.
n_r	Acceptable "statistical error" when deriving independent generators.
n_v	Number of bits in challenges.
PRG	Pseudo-random generator used to derive random vectors for batching.
G_q	Group of prime order with standard generator g.
$\mathcal{M}_\kappa$	Basic plaintext group.
$\mathcal{R}_\kappa$	Basic randomness group.
$\mathcal{C}_{\kappa,\omega}$	Ciphertext group.
$\mathcal{M}_{\kappa,\omega}$	Plaintext group.
Γ	Polynomial in the exponent where $\Gamma = (\Gamma_0, \ldots, \Gamma_{\lambda-1})$ and $\Gamma_l \in \mathcal{M}_\kappa$.
w	Array $w = (w_0, \ldots, w_{N-1})$ of input ciphertexts in $\mathcal{C}_{\kappa,\omega}$, where $w_i = (u_i, v_i)$.
$f_1, \ldots, f_k$	Arrays $f_j = (f_{j,0}, \ldots, f_{j,N-1})$ of decryption factors in $\mathcal{M}_{\kappa,\omega}$.
$\tau_1^{dec}, \ldots, \tau_k^{dec}$	Commitments of the Fiat-Shamir proofs.
$\sigma_1^{dec}, \ldots, \sigma_k^{dec}$	Replies of the Fiat-Shamir proofs.
Δ	Set of exactly λ candidate indices in $[1, k]$ to form a single valid proof.

Program

1. (a) Interpret τ_l^{dec} as $\mathsf{node}(\overline{y_l'}, \overline{B_l'})$, where $y_l' \in \mathcal{M}_\kappa$ and $B_l' \in \mathcal{M}_{\kappa,\omega}$.
   (b) Interpret σ_l^{dec} as $\overline{k_{x,l}}$, where $k_{x,l} \in \mathcal{R}_\kappa$.

   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 $a = \mathsf{node}(\overline{\Gamma_0}, \ldots, \overline{\Gamma_{\lambda-1}})$ and $b = \mathsf{node}(\overline{f_1}, \ldots, \overline{f_k})$.

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

5. Compute Lagrange coefficients $c_l = \prod_{i \in \Delta \setminus \{l\}} \frac{i}{i-l}$ for all $l \in \Delta$. Then compute joint decryption factors $f = \big(\prod_{l\in\Delta} f_{l,i}^{\alpha c_l}\big)_{i\in[1,N]}$, where $\alpha = \mathrm{lcm}(1, \ldots, k)^2$, joint proof commitments $y' = \prod_{l\in\Delta}(y_l')^{c_l}$ and $B' = \prod_{l\in\Delta}(B_l')^{c_l}$, and a reply for a joint proof $k_x = \sum_{l\in\Delta} c_l k_{x,l}$. Then compute batched elements

$$A = \left(\prod_{i=0}^{N-1} u_i^{e_i}, 1\right) \quad , \quad \text{and} \quad B = \prod_{i=0}^{N-1} f_i^{e_i}$$

   and **accept** if and only if

$$\Gamma_0^{-v} y' = g^{k_x} \quad \text{and} \quad B^v B' = \mathsf{PDec}_{k_x}(A) \ .$$

9 Verifying a Complete Proof

A proof should be viewed as a capsule that relates a public key, an input, and an output in a provable way according to given protocol and session parameters. The verification algorithm must verify that the capsule has this property.

9.1 Components of the Non-Interactive Zero-Knowledge Proofs

In addition to the parameters we need from the protocol info file, we have specific parameters of a given session stored in separate files in the root of the proof 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 verification 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 given in *external* formats of a particular application matches those in the proof. This includes verifying the version of VMN, the type of proof, the auxiliary session identifier, the key width, and the width. We stress that the description of the external formats can not be part of this document, since they are application dependent.

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 code of 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 in separate sessions. Each session can then be verified using verifiers written by independent parties. Then a simple verifier of the total tallying protocol that relates the sessions can be implemented and audited with a limited background in cryptography.

Files in the Main Directory. We now give details about the files in the main directory. Each file either contains a binary byte tree or it uses UTF-8 encoding. 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 denoted version of the VMN software that created the proof. This string is denoted by version below. The flag `-version` can be used with all commands of the VMN software to determine the installed version.

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

3. auxsid – An auxiliary session identifier of this session as an ASCII string consisting of characters from the set $\{\texttt{A}, \ldots, \texttt{Z}, \texttt{a}, \ldots, \texttt{z}, \texttt{0}, \ldots, \texttt{9}, \_\}$. This equals the string `"default"` unless a different auxiliary session identifier is given explicitly when executing the mix-net. This string is denoted by auxsid below.

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 $pk \in \mathcal{C}_\kappa$. Here the key width κ and the underlying prime order group G_q are 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 $\mathcal{C}_{\kappa,\omega}$. **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 have not been decoded in any way. This file should contain a byte tree $\overline{m}$, where m is an array of N elements in $\mathcal{M}_{\kappa,\omega}$.
   (b) ShuffledCiphertexts.bt – For a shuffling session, the re-randomized and permuted ciphertexts. This file should contain a byte tree $\overline{L_{\lambda_a}}$, where L_{λ_a} is an array of N elements in $\mathcal{C}_{\kappa,\omega}$. The meaning of the index λ_a is explained in Point 8 below.

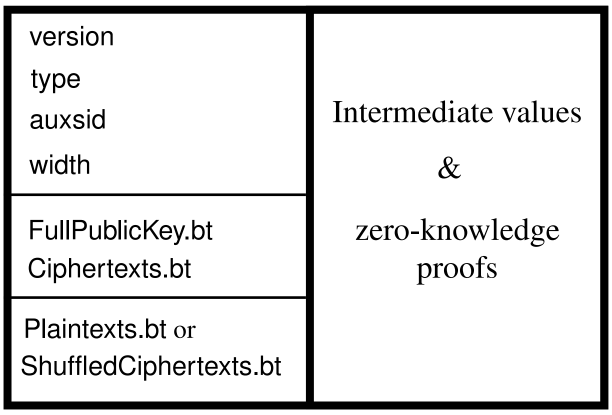

Figure 1: A "proof capsule" contains the version, the type, the auxiliary session identifier, and the width, in separate files. Combined with the protocol info file this defines the format of the remaining contents of the proof. The input consists of a full public key and a list of ciphertexts. The output is either a file Plaintexts.bt containing plaintexts, or a file ShuffledCiphertexts.bt containing shuffled ciphertexts, depending on the type of session that generated the proof. A subdirectory named proof contains the intermediate results and zero-knowledge proofs that relate the inputs and outputs.

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 usually representing the index of a mix-server, but if the lth mix-server is corrupted, then a file with suffix l does not necessarily originate from the lth mix-server. Furthermore, all types of files do not appear with all suffixes.

Active threshold.

8. activethreshold – Threshold λ_a of mix-servers claimed by at least λ mix-servers to guarantee that at least λ mix-servers are active for some set of active servers, given as a decimal number in ASCII.

   Example 23. For example if $k = 7$ and $\lambda = 4$, and the 2nd mix-server was deactivated during the session to be verified, then $\lambda_a = 5$.

Files for keys.

9. PolynomialInExponent.bt – Polynomial in the exponent corresponding to the Shamir secret sharing polynomial used to share the joint secret key. The required format of this file is a byte tree $\overline{\Gamma}$, where Γ is an array $(\Gamma_0, \ldots, \Gamma_{\lambda-1})$ of elements in $\mathcal{M}_{\kappa}$. The joint public key and public keys of the mix-servers are derived from these elements in Algorithm 24 below.

Files for proofs of shuffles.

10. Ciphertexts$\langle l \rangle$.bt – The lth intermediate list of ciphertexts. This file should contain a byte tree $\overline{L_l}$, where L_l is an array of N elements in $\mathcal{C}_{\kappa,\omega}$, and N is the number of elements in the list L_0 of input ciphertexts defined in Point 6. The list L_l is normally the output of the lth mix-server, but if it is deactivated or corrupted it is set equal to L_{l-1}.

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, given 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 of μ_l to keep.

18. CCPoSCommitment$\langle l \rangle$.bt – "Proof commitment" of a 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 a 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. CorrectIndices.bt – Boolean array indicating exactly λ parties that provided individual proofs of correct decryption factors that can be combined into a single valid proof. More precisely, a byte tree $\overline{d}$ such that d is a boolean array of length $k+1$ (the first component is ignored). Denote the set of the indicated integers by $\Delta = \{l \in [1,k] : d_l = true\}$.

21. DecryptionFactors$\langle l \rangle$.bt – Decryption factors of the lth mix-server combined 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 $\mathcal{M}_{\kappa,\omega}$.

22. DecrFactCommitment$\langle l \rangle$.bt – "Proof commitment" of a proof of correctness of the decryption factors computed by the lth mix-server. The required format of the byte tree τ_l^{dec} of this file is specified in Algorithm 22.

23. DecrFactReply$\langle l \rangle$.bt – "Proof reply" of a proof of correctness of the decryption factors computed by the lth mix-server. The required format of the byte tree σ_l^{dec} of this file is specified in Algorithm 22.

Relation Between Files and Abstract Notation. For easy reference we tabulate the notation introduced above and from which files the contents are derived.

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_a}	7b	ShuffledCiphertexts.bt	σ_l^{ccpos}	19	CCPoSReply$\langle l \rangle$.bt
λ_a	8	activethres	Δ	20	CorrectIndices.bt
Γ	9	PolynomialInExponent.bt	f_l	21	DecryptionFactors$\langle l \rangle$.bt
L_l	10	Ciphertexts$\langle l \rangle$.bt	τ_l^{dec}	22	DecrFactCommitment$\langle l \rangle$.bt
μ_l	11	PermutationCommitment$\langle l \rangle$.bt	σ_l^{dec}	23	DecrFactReply$\langle l \rangle$.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.

Reading and verifying keys. We begin with a subroutine that reads the joint public key and the polynomial in the exponent used for secret sharing and checks that they are consistent.

Algorithm 24 (Reading and Verifying Keys)**.**

Input	Description
λ	Threshold number of mix-servers needed to decrypt.
$\mathcal{M}_\kappa$	Underlying group.

Program

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

2. **Polynomial in the exponent.** Attempt to read the polynomial in the exponent $\Gamma = (\Gamma_0, \ldots, \Gamma_{\lambda-1})$, where $\Gamma_i \in \mathcal{M}_\kappa$ as described in Point 9. If this fails or if $\Gamma_0 \neq y$, then **reject**.

3. **Return** (pk, Γ).

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 25 (Verifier of Shuffling)**.**

Input	**Description**
ρ	Prefix to random oracles.
λ	Number of mix-servers needed to decrypt.
λ_a	Number of mix-servers such that at least λ are active.
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.
$\mathcal{C}_\kappa$	Public key group.
$\mathcal{R}_{\kappa,\omega}$	Randomness group.
$\mathcal{C}_{\kappa,\omega}$	Ciphertext group.
pk	Joint public key in $\mathcal{C}_\kappa$.
L_0	Original array of N ciphertexts in $\mathcal{C}_{\kappa,\omega}$.
L_{λ_a}	Array of N shuffled ciphertexts in $\mathcal{C}_{\kappa,\omega}$.
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 as defined in Point 14 does not exist, then no pre-computation took place and plain shuffling was used. The following is then executed.

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

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

2. **Verify proof of shuffle.** Read permutation commitment μ_l, proof commitment τ_l^{pos}, and proof reply σ_l^{pos} as described in Point 11, Point 12, and Point 13, respectively. Then execute Algorithm 19 on input

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

   If reading fails, or if the algorithm rejects and $L_l \neq L_{l-1}$, then **reject**.

If at least one proof is deemed valid by Algorithm 19 above, then **accept** and otherwise **reject**.

Algorithm 26 (Algorithm 25 continued)**.**

In case the maxciph file as defined in Point 14 does exist, then pre-computation took place. The following is then executed.

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

1. **Permutation commitment.** Read the byte tree representation μ_l of a permutation commitment u_l of the lth mix-server as described in Point 11.
2. If posc $= true$, then we verify the permutation commitment of the lth mix-server.
   (a) **Verify proof of a shuffle of commitments.** Read a proof commitment τ_l^{posc} and a proof reply σ_l^{posc} as described in Point 12 and Point 13, respectively, and execute Algorithm 20 on input $(\rho, N_0, n_e, n_r, n_v, PRG, G_q, \mu_l, \tau_l^{posc}, \sigma_l^{posc})$. If reading fails or if the algorithm rejects, then set u_l equal to the list of independent generators h.
   (b) **Shrink permutation commitment.** 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, and set $u_l = (u_{l,i})_{t_{l,i}=true}$ be the sub-array indicated by t_l.
3. If ccpos $= true$, then we verify the commitment-consistent proof of a shuffle.
   (a) **Read intermediate list of ciphertexts.** If $l < \lambda_a$, then read the array L_l of N ciphertexts in $\mathcal{C}_{\kappa,\omega}$ as described in Point 10. If this fails, then **reject**.
   (b) **Verify commitment-consistent proof of shuffle.** Read a proof commitment τ_l^{ccpos} and a proof reply σ_l^{ccpos} as described in Point 18 and Point 19, respectively. If reading fails, then **reject**. Then execute Algorithm 21 on input $(\rho, N, n_e, n_r, n_v, PRG, G_q, \mathcal{C}_\kappa, \mathcal{R}_{\kappa,\omega}, \mathcal{C}_{\kappa,\omega}, pk, L_{l-1}, L_l, u_l, \tau_l^{ccpos}, \sigma_l^{ccpos})$. If the algorithm rejects and $L_l \neq L_{l-1}$, then **reject**.

If the verified proofs of at least one party is deemed valid by Algorithm 21 above, then **accept** and otherwise **reject**.

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 and uses a simple trick to reduce the size of exponents when combining.

Algorithm 27 (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 with standard generator g.
$\mathcal{M}_{\kappa,\omega}$	Plaintext group.
$\mathcal{C}_{\kappa,\omega}$	Ciphertext group.
Γ	Polynomial in the exponent $(\Gamma_0, \ldots, \Gamma_{\lambda-1})$ where $\Gamma_l \in \mathcal{M}_\kappa$.
L	Array of N ciphertexts in $\mathcal{C}_{\kappa,\omega}$.
m	Array of N plaintexts in $\mathcal{M}_{\kappa,\omega}$.
Δ	Set of exactly λ candidate indices in $[1, k]$ to form a single valid proof.

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 21, Point 22, and Point 23, respectively. If this fails, then **reject**. Otherwise, set

$$f = (f_1, \ldots, f_k)$$
$$\tau^{dec} = (\tau_1^{dec}, \ldots, \tau_k^{dec})$$
$$\sigma^{dec} = (\sigma_1^{dec}, \ldots, \sigma_k^{dec})$$

2. **Verify combined proof.** Attempt to verify a single joint proof by executing Algorithm 22 on input

$$(\rho, N, n_e, n_r, n_v, PRG, G_q, \mathcal{M}_\kappa, \mathcal{R}_\kappa, \mathcal{C}_{\kappa,\omega}, \mathcal{M}_{\kappa,\omega}, \Gamma, L, f, \tau^{dec}, \sigma^{dec}, \Delta) \ .$$

   If it rejects, then **reject**.

3. **Verify plaintext.** If $m \neq \mathsf{TDec}\big(L, \prod_{l \in \Delta} f_l^{\alpha c_l}\big)$, where $\alpha = \big(\mathrm{lcm}(1, \ldots, k)\big)^2$, then **reject** and otherwise **accept**.

9.3 Complete Verification Algorithm

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

Algorithm 28 (Verifier)**.**

Input	**Description**
$\mathsf{type}_{\mathit{expected}}$	Expected type of proof.
protinfo	Protocol info file.
directory	Directory containing proof.
$\mathsf{auxsid}_{\mathit{expected}}$	Expected auxiliary session identifier.
$\omega_{\mathit{expected}}$	Expected width of ciphertexts ($\perp$ indicates that the default width is used).
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 $\mathsf{version}_{\mathit{prot}}$, sid, k, λ, n_e, n_r, n_v, s_H, s_{PRG}, s_{G_q}, s_H, κ, and $\omega_{\mathit{default}}$.

2. **Proof parameters.**

   (a) Read version, type, auxsid, ω, from the proof directory directory as described in Point 1 – Point 4. If this fails, then **reject**.

   (b) If version $\neq$ $\mathsf{version}_{\mathit{prot}}$ or version $\neq 3.0.4$, then **reject**.
   If type $\neq$ $\mathsf{type}_{\mathit{expected}}$, then **reject**.
   If auxsid $\neq$ $\mathsf{auxsid}_{\mathit{expected}}$, then **reject**.
   If $\omega_{\mathit{expected}} = \perp$ and $\omega \neq \omega_{\mathit{default}}$, then **reject**.
   If $\omega_{\mathit{expected}} \neq \perp$ and $\omega \neq \omega_{\mathit{expected}}$, then **reject**.

3. **Derived sets and objects.** Attempt to derive and define the underlying group $G_q = \mathsf{unmarshal}(s_{G_q})$, the spaces of plaintexts $\mathcal{M}_\kappa = G_q^\kappa$, randomness $\mathcal{R}_\kappa = \mathbb{Z}_q^\kappa$, and ciphertexts $\mathcal{C}_\kappa = \mathcal{M}_\kappa \times \mathcal{M}_\kappa$ defined for a given key width κ. Then define the generalized set of plaintexts $\mathcal{M}_{\kappa,\omega} = \mathcal{M}_\kappa^\omega$, the set of randomness $\mathcal{R}_{\kappa,\omega} = \mathcal{R}_\kappa^\omega$, and the set of ciphertexts $\mathcal{C}_{\kappa,\omega} = \mathcal{M}_{\kappa,\omega} \times \mathcal{M}_{\kappa,\omega}$. Finally, define 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. The digest ρ is defined by

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

Algorithm 29 (Algorithm 28 continued)**.**

5. **Read keys.** Execute Algorithm 24. If it fails, then **reject** and otherwise, let the output be (pk, Γ).

6. **Read lists.**

   (a) **Read input ciphertexts.** Read the array L_0 of N ciphertexts as described in Point 6 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 type $=$ `mixing`, then read L_{λ_a} as described in Point 6 and if type $=$ `shuffling`, then read L_{λ_a} as described in Point 7b. If this fails, then **reject**.

   (c) **Read output plaintexts.** If type $\in$ {`mixing`, `decryption`}, and dec $=$ *true*, then read the array m of plaintexts as described in Point 7a. If this fails, then **reject**. Otherwise there are no plaintexts to read.

7. **Verify relations between lists.**

   (a) **Verify shuffling.** If type $\in$ {`mixing`, `shuffling`}, and posc $=$ *true* or ccpos $=$ *true*, then execute Algorithm 25 on input

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

   and if it rejects, then **reject**.

   (b) **Verify decryption.** If dec $=$ *true* then do the following.

   If type $=$ `mixing`, then set $L = L_{\lambda_a}$,
   else if type $=$ `decryption`, then set $L = L_0$,
   and otherwise **reject**.

   Read the set of correct indices Δ as described in Point 20. If this fails, then **reject**. Then execute Algorithm 27 on input

   $$(\rho, N, n_e, n_r, n_v, PRG, G_q, \mathcal{M}_{\kappa,\omega}, \mathcal{C}_{\kappa,\omega}, \Gamma, L, m, \Delta) \enspace ,$$

   and if it rejects, then **reject**.

8. **Accept proof.**

Note that if no pre-computation was used, the inputs posc and ccpos should either both be *true* or both be *false* when invoking Algorithm 28.

10 Standard Command Line Interface of Verifier

To ensure interoperability between different independently implemented verifiers it is tempting to require that every verifier implements the above algorithm as well as providing the same command line interface as `vmnv`. However, not all implementors may be motivated to implement the full functionality and identical implementation specific options.

Thus, we define a standard command line interface with a well-defined semantic in terms of Algorithm 28 and provide a mechanism to limit the functionality. Implementors may of course add their own implementation specific options. This opens the door to implementations that eliminate all standard options and provide their own, but the requirements below makes this clear to the user.

10.1 Basic Functionality

We require that the basic command-line interface of every independent verifier `iv` is defined as follows, where we have truncated the descriptions of the options. This is basic in the sense that the implementor is free to add additional usage forms as well as additional options to existing usage forms, but it must be possible to execute `iv` in the following ways (except that some usage forms and options may be eliminated as explained in the next section).

```
Usage:
 iv -h
 iv -c
 iv -mix
    [-auxsid <value>] [-noccpos] [-nodec] [-nopos] [-noposc] [-width <value>]
    <protInfo> <nizkp>
 iv -shuffle
    [-auxsid <value>] [-noccpos] [-noposc] [-width <value>]
    <protInfo> <nizkp>
 iv -decrypt
    [-auxsid <value>] [-width <value>]
    <protInfo> <nizkp>
 iv -version
```

We require that `iv -h` prints the complete usage information of the command, and we require that `iv -version` prints the version of the verifier, but there are no restrictions on the formats of the outputs. Indeed, we expect different implementations to use different libraries for parsing and printing usage forms and options.

The output of `iv -c` is described below. The other usage forms are decoded into executions of Algorithm 28. We first define its parameters as follows in terms of command line parameters.

Parameter	**Decoding**
$\mathsf{type}_{expected}$	Assigned `mixing`, `shuffle`, or `decryption` depending on if the `-mix`, `-shuffle`, or `-decrypt` option is used respectively.
$\mathsf{protinfo}$	Assigned the value of the first parameter, i.e., the parameter denoted `<protInfo>`. This must be a path to a file.
$\mathsf{directory}$	Assigned the value of the second parameter, i.e., the parameter denoted `<nizkp>`. This must be a path to a directory.
$\mathsf{auxsid}_{expected}$	Assigned the value of the `-auxsid` option if present and `"default"` otherwise. This must be a string of characters from the set $\{\mathtt{A},\ldots,\mathtt{Z},\mathtt{a},\ldots,\mathtt{z},\mathtt{0},\ldots,\mathtt{9},\_\}$
$\omega_{expected}$	Assigned the value of the `-width` option if present and $\perp$ otherwise. This must be an integer greater than zero represented in decimal on the command line.
posc	Assigned *false* if `-nopos` or `-noposc` is used and *true* otherwise.
ccpos	Assigned *false* if `-nopos` or `-noccpos` is used and *true* otherwise.
dec	Assigned *false* if `-nodec` is used and *true* otherwise.

The exit code of `iv` must be equal to 0 or -1 depending on if Algorithm 28 accepts or rejects, respectively, on input $(\mathsf{type}_{expected}, \mathsf{protinfo}, \mathsf{directory}, \mathsf{auxsid}_{expected}, \omega_{expected}, \mathsf{posc}, \mathsf{ccpos}, \mathsf{dec})$.

10.2 Encoding and Communicating Missing Functionality

We require that the functional limitations of `iv` compared to the basic functionality are captured by a triple (b, f, p) of the following form.

1. **Proof types.** The proof types not implemented by `iv` are captured by a subset b of the set
$$\left\{\texttt{-mix}, \texttt{-shuffle}, \texttt{-decrypt}\right\} .$$
2. **Option flags.** The options not implemented by `iv` are captured by a subset f of the set
$$\left\{\texttt{-width}, \texttt{-nopos}, \texttt{-nodec}, \texttt{-noccpos}, \texttt{-noposc}\right\} .$$
3. **Pre-computation.** If `iv` is able to verify proofs where the mix-servers used pre-computation, then $p = \emptyset$ and otherwise $p = \{\texttt{-nopre}\}$.

We require that `iv` exits with exit code -3 when executed on a given list of command line parameters if and only if:

1. any string from $b \cup f$ appears as a command line parameter, or
2. $p = \{\texttt{-nopre}\}$ and there exists a file directory/`proofs/maxciphs`, where directory is defined above.

There are no requirements on the output of `iv` in this case, but it is of course wise to implement an option to enable or disable the printing of an explanation.

We require that `iv -c` prints the output of `vmnv -mc iv` x verbatim on standard output without any side effects, where x is the comma separated list of all strings in $b \cup f \cup p$.

11 Additional Verifications Needed in Applications

As explained above, and illustrated Figure 1, a proof may be viewed as a capsule consisting of:

1. an outer shell containing the public key used, input ciphertexts, and the output plaintexts or shuffled ciphertexts, as well as some additional data, and
2. an inner capsule stored in a subdirectory that contains intermediate values of an execution and zero-knowledge proofs that relate them and the values from the outer shell.

Even if such a proof capsule is verified to be correct, the values from the outer shell must be consistent to the values of the given application.

The formats used to represent the values from the outer shell, e.g., the public key handed to the senders and the list of ciphertexts received from senders and the output list of ciphertexts are application dependent. Thus, to verify the overall correctness in a given application, it *must* be verified that all parties agree on an encoding scheme such that:

1. The values version, type, auxsid, and ω match the corresponding values in the application.
2. The public key actually used by senders is a representation of pk.
3. The actual input ciphertexts is a representation of L_0.
4. Depending on which type of session is verified:
   (a) the actual output plaintexts is a decoding of the plaintext group elements in m, or
   (b) the actual output ciphertexts is a representation of L_{λ_a}.

All of the above falls outside the scope of this document, since we can not anticipate the scheme used to represent these objects.

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

[2] Digital signature standard (DSS). *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.

[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] Gnu multiple precision arithmetic library (GMP). `http://www.gmplib.org`, October 2014.

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

[9] 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.

[10] 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.

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

[12] 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.

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="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:simpleType name="inetport">
  <xs:restriction base="xs:string">
    <xs:maxLength value="512"/>
    <xs:pattern value="((([a-z0-9]([-a-z0-9]*[a-z0-9])*)(\.([a-z0-9]([-a-z0-9]*[a-z0-9])*))*)|(
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="urlport">
  <xs:restriction base="xs:string">
    <xs:maxLength value="512"/>
    <xs:pattern value="http://((([a-z0-9]([-a-z0-9]*[a-z0-9])*)(\.([a-z0-9]([-a-z0-9]*[a-z0-9])
  </xs:restriction>
</xs:simpleType>
```

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

<xs:element name="version"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="[0-9]{0,3}\.[0-9]{0,3}\.[0-9]{0,3}"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<xs:element name="sid"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="[A-Za-z][A-Za-z0-9]{1,1023}"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<xs:element name="name"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="[A-Za-z][A-Za-z0-9_ ]{1,255}"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<xs:element name="descr"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="|[A-Za-z][A-Za-z0-9:;?!.()\[\] ]{0,4000}"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<xs:element name="nopart"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxExclusive value="25"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


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

```
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxExclusive value="256"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


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

<xs:element name="thres"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxExclusive value="25"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


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

<xs:element name="keywidth"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxExclusive value="2147483647"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<xs:element name="vbitlen"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxExclusive value="2048"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<xs:element name="vbitlenro"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
```

```
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxExclusive value="2048"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<xs:element name="ebitlen"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxExclusive value="2048"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<xs:element name="ebitlenro"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxExclusive value="2048"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<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"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="interactive|noninteractive"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<xs:element name="width"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxExclusive value="2147483647"/>
   </xs:restriction>
```

```
</xs:simpleType>
</xs:element>


<xs:element name="maxciph"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxExclusive value="2147483647"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>



<xs:element name="party"
            minOccurs="0"
            maxOccurs="25">
<xs:complexType>
<xs:sequence>

<xs:element name="name"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="[A-Za-z][A-Za-z0-9_ ]{1,255}"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<xs:element name="srtbyrole"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="[A-Za-z0-9][A-Za-z0-9]{1,63}"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


<xs:element name="descr"
            minOccurs="1"
            maxOccurs="1">
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:pattern value="|[A-Za-z][A-Za-z0-9:;?!.()\[\] ]{0,4000}"/>
   </xs:restriction>
</xs:simpleType>
</xs:element>


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

```
<xs:element name="http" type="urlport"/>

<xs:element name="hint" type="inetport"/>

</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.

     Many XML features are disabled and throw errors, so parsing is more
     restrictive than the schema implies. -->

<protocol>

   <!-- Version of Verificatum Software for which this info is intended. -->
   <version>3.0.4</version>

   <!-- Session identifier of this protocol execution. This must be
        globally unique and satisfy the regular expression [A-Za-z][A-Za-z0-
        9]{1,1023}. -->
   <sid>SID</sid>

   <!-- Name of this protocol execution. This is a short descriptive name
        that is NOT necessarily unique, but satisfies the regular
        expression [A-Za-z][A-Za-z0-9_ ]{1,255}. -->
   <name>Swedish Election</name>

   <!-- Description of this protocol execution. This is merely a longer
        description than the name of the protocol execution. It must
        satisfy the regular expression |[A-Za-z][A-Za-z0-9:;?!.()\[\] ]
        {0,4000}. -->
   <descr></descr>

   <!-- Number of parties taking part in the protocol execution. This must
        be a positive integer that is at most 25. -->
   <nopart>3</nopart>

   <!-- Statistical distance from uniform of objects sampled in protocols
        or in proofs of security. This must be a non-negative integer at
        most 256. -->
   <statdist>100</statdist>

   <!-- Name of bulletin board implementation used, i.e., a subclass of com.
```

```
     verificatum.protocol.com.BullBoardBasic. WARNING! This field is not
     validated syntactically. -->
<bullboard>com.verificatum.protocol.com.BullBoardBasicHTTPW</bullboard>

<!-- Threshold number of parties needed to violate the privacy of the
     protocol, i.e., this is the number of parties needed to decrypt.
     This must be positive, but at most equal to the number of parties.
     -->
<thres>2</thres>

<!-- Group over which the protocol is executed. An instance of a
     subclass of com.verificatum.arithm.PGroup. -->
<pgroup>ModPGroup(safe-prime modulus=2*order+1. order bit-length = 511):
    :0000000002010000000020636f6d2e766572696669636174756d2e61726974686d2e4d6
    f645047726f7570000000000000040100000041009a91c3b704e382e0c772fa7cf0e5d6363ed
    c53d156e841555702c5b6f906574204bf49a551b695bed292e0218337c0861ee649d2f
    e4039174514fe2c23c10f6701000000404d48e1db8271c17063b97d3e7872eb1b1f6e29
    e8ab7420aaab8162db7c832ba1025fa4d2a8db4adf69497010c19be0430f7324e97f201
    c8ba28a7f1611e087b3010000004100300763b0150525252e4989f51e33c4e6462091152
    ef2291e45699374a3aa8acea714ff30260338bddbb48fc7446b273aaada90e3ee8326
    f388b582ea8a07350201000000000400000001</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. -->
<vbitlen>128</vbitlen>

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

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

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

<!-- Pseudo random generator used to derive random vectors for
     batchingfrom jointly generated seeds. This can be "SHA-256", "SHA-
     384", or "SHA-512", in which case com.verificatum.crypto.
     PRGHeuristic is instantiated based on this hashfunction, or it can
     be an instance of com.verificatum.crypto.PRG. WARNING! This field
     is not validated syntactically. -->
<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 com.
     verificatum.crypto.HashfunctionHeuristic is instantiated, or an
     instance of com.verificatum.crypto.Hashfunction. Random oracles
     with various output lengths are then implemented, using the given
     hashfunction, in com.verificatum.crypto.RandomOracle.
     WARNING! Do not change the default unless you know exactly what you
     are doing. This field is not validated syntactically. -->
<rohash>SHA-256</rohash>
```

```
<!-- Determines if the proofs of correctness of an execution are
     interactive or non-interactive. Legal valus are "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. This must satisfy the regular expression [A-Za-z][A-
        Za-z0-9_ ]{1,255}. -->
   <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. It must satisfy the regular expression
        |[A-Za-z][A-Za-z0-9:;?!.()\[\] ]{0,4000}. -->
   <descr></descr>

   <!-- Public signature key (instance of subclasses of com.verificatum.
        crypto.SignaturePKey). WARNING! This field is not validated
        syntactically. -->
   <pkey>com.verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048):
       :00000000020100000002d636f6d2e766572696669636174756d2e63727970746f2
       e5369676e6174757265504
       b6579486575726973746963000000000201000001263082012230 0d06092a864886
       f70d01010105000382010f003082010a0282010100a4b6dc63fe33deb0da65ee6d4a5
       cf40890805b2f909983f77bf83527e9d962fc2e283e4a730f530ac30b2ea2d1919596
       cf3fddf304497f130a8ae274e954bce48e587fcf568ae7d9f695f97777300358e4
       a521c054f68edc1dd170bbf230cdce24c9fdfec9066f2df0074e9af80aab7a13abc5
       e9ca817b25314dae9773284dede436da3a162269f33f40d58144890e7430c2209a98f
       ec55271ab8b5cae3c57884675a90bd5c169e4e7e0ba7081d55941083f74d0
       b82919104244e9361f5e1a8f9cdd14e6be9f8c844ecdc4911bc01c75a3368c75
       b4285771126695b8893a255643be81d42a1978aab31f287d0ac302e3f178a0c49439
       a935b5a1a0d6ba07cb4502030100010100000000400000800</pkey>

   <!-- URL to the HTTP server of this party. -->
   <http>http://mybox1.mydomain1.com:8080</http>

   <!-- Socket address given as <hostname>:<port> or <ip address>:<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. This must satisfy the regular expression [A-Za-z][A-
        Za-z0-9_ ]{1,255}. -->
   <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. It must satisfy the regular expression
        |[A-Za-z][A-Za-z0-9:;?!.()\[\] ]{0,4000}. -->
   <descr></descr>

   <!-- Public signature key (instance of subclasses of com.verificatum.
        crypto.SignaturePKey). WARNING! This field is not validated
        syntactically. -->
   <pkey>com.verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048):
       :0000000002010000002d636f6d2e766572696669636174756d2e63727970746f2
       e5369676e6174757265504
       b657948657572697374696300000000020100000126308201223000d06092a864886
       f70d01010105000382010f003082010a0282010100b459b153dc8c9dcf162dcaef95
       b4b4969635b3144fe29d4508fd12c20323098562c1e94e955b47defac41f655335a7f
       ec237631a906c033c8b0d57e6f4e83b6f569328c7383f80aaf05fcf39e8984756ec60
       e83504780e8d86bc3de38d55cfc876635765be70fce875b25c12f3f6395084180ccff
       c3c0a81c699233a9a235ca6fe9cfae2d91e109f4ac60101d76d6b949bff6df53a09c
       f8a0f0e44028b2e4d1ef1f65a393390a23aa75f91ae240335d8951b73d4d92053928
       e3f06e8b2d08e07d318580ef0028ceae6c51b90fd4bdeb88723b0746e3c73a98d9814
       bb1594b00daf7800e8572d649c9b2a15191c0d45efb72ff470478331de6121acf11a0
       a8ccd020301000101000000000400000800</pkey>

   <!-- URL to the HTTP server of this party. -->
   <http>http://mybox2.mydomain2.com:8080</http>

   <!-- Socket address given as <hostname>:<port> or <ip address>:<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. This must satisfy the regular expression [A-Za-z][A-
        Za-z0-9_ ]{1,255}. -->
   <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. It must satisfy the regular expression
        |[A-Za-z][A-Za-z0-9:;?!.()\[\] ]{0,4000}. -->
   <descr></descr>

   <!-- Public signature key (instance of subclasses of com.verificatum.
        crypto.SignaturePKey). WARNING! This field is not validated
        syntactically. -->
```

```
      <pkey>com.verificatum.crypto.SignaturePKeyHeuristic(RSA, bitlength=2048):
          :00000000020100000002d636f6d2e766572696669636174756d2e63727970746f2
          e5369676e6174757265504
          b65794865757269737469630000000002010000012630820122300d06092a864886
          f70d01010105000382010f003082010a0282010100a141e6d91483a9c8f09d1d5fa2e
          c849564833b3d5cf7a890dbe39bb46a3e0b569c7e1aad07b3785d95b23bea01af4ec
          b03e98b67050a4a4fb6c6e54495c520121e9166f454c5df0610829f45196cffbcb091
          a4474dc0ed3d56bae9bec0b31ab29b0311469637ff6782936e8f5ff538afec257da
          c3907a44d0b682a943b401b46a45de18fc90dd88e6277b39213362c7f5a8d49b9c69
          d9217444616933fc1e6ebc98dbe49677fc7ff7fbbd5ec1cbc00894b33d6147a92e3
          c152b5dbe37cfa7acad34f3ee5becb38a906bf393f66f46b8d3d0344aee35c5f3e3d7
          fc96c7cc186ed5398b37a79062f3ccb956c60510688d54b3bd9e8a68913d45cd0875
          c3c9d138b0203010001010000000400000800</pkey>

      <!-- URL to the HTTP server of this party. -->
      <http>http://mybox3.mydomain3.com:8080</http>

      <!-- Socket address given as <hostname>:<port> or <ip address>:<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

In the following protocols we use notation introduced above, i.e., κ and ω are the key width and width of ciphertexts, and n_s, n_e, n_v, and n_r, are auxiliary security parameters for PRG seeds, batching, challenges, and statistical sampling errors.

Protocol 30 (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 $pk \in \mathcal{C}_\kappa$, elements $w_0, \ldots, w_{N-1} \in \mathcal{C}_{\kappa,\omega}$ and $w'_0, \ldots, w'_{N-1} \in \mathcal{C}_{\kappa,\omega}$.
Private Input. Exponents $s = (s_0, \ldots, s_{N-1}) \in \mathcal{R}^N_{\kappa,\omega}$ 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 $s \in \{0,1\}^{n_s}$ 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 $\epsilon_0, \ldots, \epsilon_{N-1} \in [0, 2^{n_e+n_v+n_r} - 1]$, $\phi \in \mathcal{R}_{\kappa,\omega}$ randomly, sets $B_{-1} = h_0$, and forms

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

   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

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

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

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

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

   (a) *Bridging Commitments.* It chooses $b_0, \ldots, b_{N-1} \in \mathbb{Z}_q$ randomly, sets $B_{-1} = h_0$, and forms
   $$B_i = g^{b_i} B_{i-1}^{e'_i} \ \text{ for } i = 0, \ldots, N-1 \ .$$

   (b) *Proof Commitments.* It chooses $\alpha, \beta_0, \ldots, \beta_{N-1}, \gamma, \delta \in \mathbb{Z}_q$ and $\epsilon_0, \ldots, \epsilon_{N-1} \in [0, 2^{n_e+n_v+n_r} - 1]$ randomly, sets $B_{-1} = h_0$, and forms
   $$\begin{aligned} A' &= g^{\alpha} \prod_{i=0}^{N-1} h_i^{\epsilon_i} & C' &= g^{\gamma} \\ B'_i &= g^{\beta_i} B_{i-1}^{\epsilon_i} \ \text{ for } i = 0, \ldots, N-1 & D' &= g^{\delta} \ . \end{aligned}$$

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

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

4. $\mathcal{P}$ computes $a = \langle r, e' \rangle$, $c = \sum_{i=0}^{N-1} r_i$. Then it sets $d_0 = b_0$ and computes $d_i = b_i + e'_i d_{i-1}$ for $i = 1, \ldots, N-1$. Finally, it sets $d = d_{N-1}$ and computes
$$\begin{aligned} k_A &= va + \alpha & k_C &= vc + \gamma \\ k_{B,i} &= vb_i + \beta_i \ \text{ for } i = 0, \ldots, N-1 & k_D &= vd + \delta \\ & & k_{E,i} &= ve'_i + \epsilon_i \ \text{ for } i = 0, \ldots, N-1 \ . \end{aligned}$$

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

5. $\mathcal{V}$ computes $C = \prod_{i=0}^{N-1} u_i / \prod_{i=0}^{N-1} h_i$ and $D = B_{N-1} / h_0^{\prod_{i=0}^{N-1} e_i}$, and sets $B_{-1} = h_0$ and accepts if and only if
$$\begin{aligned} A^v A' &= g^{k_A} \prod_{i=0}^{N-1} h_i^{k_{E,i}} & C^v C' &= g^{k_C} \\ B_i^v B'_i &= g^{k_{B,i}} B_{i-1}^{k_{E,i}} \ \text{ for } i = 0, \ldots, N-1 & D^v D' &= g^{k_D} \end{aligned}$$

Protocol 32 (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 $pk \in \mathcal{C}_\kappa$, elements $w_0, \ldots, w_{N-1} \in \mathcal{C}_{\kappa,\omega}$ and $w'_0, \ldots, w'_{N-1} \in \mathcal{C}_{\kappa,\omega}$.
Private Input. Exponents $r = (r_0, \ldots, r_{N-1}) \in \mathbb{Z}_q^N$, a permutation $\pi \in \mathbb{S}_N$, and exponents $s = (s_0, \ldots, s_{N-1}) \in \mathcal{R}_{\kappa,\omega}^N$ 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 $s \in \{0,1\}^{n_s}$ 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 $\beta \in \mathcal{R}_{\kappa,\omega}$ 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
$$A^v A' = g^{k_A} \prod_{i=0}^{N-1} h_i^{k_{E,i}} \qquad B^v B' = \mathsf{Enc}_{pk}(1, -k_B) \prod_{i=0}^{N-1} (w'_i)^{k_{E,i}}$$

E Output of `vmnv -c`

```
Usage:
 vmnv -h
 vmnv -c
 vmnv -mix
      [-auxsid <value>] [-noccpos] [-nodec] [-nopos] [-noposc]
      [-width <value>]
      <protInfo> <nizkp>
 vmnv -shuffle
      [-auxsid <value>] [-noccpos] [-noposc] [-width <value>]
      <protInfo> <nizkp>
 vmnv -decrypt
      [-auxsid <value>] [-width <value>]
      <protInfo> <nizkp>
 vmnv -version


Description:

Verifies the overall correctness of an execution using the intermediate
results and the zero-knowledge proofs of correctness using the Fiat-Shamir
heuristic in the given proof directory. The verification of certain parts
can be turned off to simplify a limited form of online verification and
simplify debugging of other verifiers.

Parameters:
   <nizkp>          - Directory containing the non-interactive zero-knowledge
                      proof of correctness using the Fiat-Shamir heuristic.
   <protInfo>       - Protocol info file.

Options:
   -auxsid <value>  - Verify that the given auxiliary session identifier
                      matches that in the proof. This is required when the
                      auxiliary session identifier in the proof is not
                      "default".
        -c          - Print compatibility usage information.
  -decrypt          - Check proof of decryption.
        -h          - Print usage information.
      -mix          - Check proof of mixing.
  -noccpos          - Turn off verification of commitment-consistent proofs of
                      shuffles. This is only possible if pre-computation was
                      used during execution.
    -nodec          - Turn off verification of proof of decryption.
    -nopos          - Turn off verification of proofs of shuffles. If pre-
                      computation is used, this turns off verification of both
                      proofs of shuffles of commitments and commitment-
                      consistent proofs of shuffles.
   -noposc          - Turn off verification of proofs of shuffles of
                      commitments. This is only possible if pre-computation was
                      used during execution.
  -shuffle          - Check proof of shuffle.
  -version          - Print the package version.
    -width <value>  - Verify that the given width matches that in the proof.
                      This is required when the width in the proof is different
                      from the width in the protocol info file.
```

F Usage Information for `vmnv`

```
Usage:
 vmnv -h
 vmnv -c
 vmnv -th
 vmnv -mix
      [-a <value>] [-auxsid <value>] [-e] [-noccpos] [-nodec] [-nopos]
      [-noposc] [-t <names>] [-v] [-wd <dir>] [-width <value>]
      <protInfo> <nizkp>
 vmnv -shuffle
      [-a <value>] [-auxsid <value>] [-e] [-noccpos] [-noposc] [-t <names>]
      [-v] [-wd <dir>] [-width <value>]
      <protInfo> <nizkp>
 vmnv -decrypt
      [-a <value>] [-auxsid <value>] [-e] [-t <names>] [-v] [-wd <dir>]
      [-width <value>]
      <protInfo> <nizkp>
 vmnv -sloppy
      [-a <value>] [-e] [-t <names>] [-v] [-wd <dir>]
      <protInfo> <nizkp>
 vmnv -mc
      <command>
      [<flags>]
 vmnv -version


Description:

Verifies the overall correctness of an execution using the intermediate
results and the zero-knowledge proofs of correctness using the Fiat-Shamir
heuristic in the given proof directory. The verification of certain parts
can be turned off to simplify a limited form of online verification and
simplify debugging of other verifiers.

WARNING!
Using this in a real election gives SOME assurance, but it does NOT
eliminate the need for an independently implemented verifier according to
the human-readable description of the universally verifiable proof resulting
from an execution of the mix-net. This document is available at https://www.
verificatum.org.

The main motivations of this tool are to:
(a) debug the description of the universally verifiable proof,
(b) benchmark the running time of verifiers,
(c) serve as a reference implementation to implementors of
    their own verifiers, and
(d) check the compatibility of independent verifiers with
    the requirements of the description of the universally
    verifiable proof.

For this purpose it provides a feature-rich way to print test vectors of
intermediate results and express compatibility.

Parameters:
   <command>        - Command name of independent verifier. The name may not
                      contain any "-" characters.
   <flags>          - A comma-separated list of option flags to be removed
                      from the compatibility usage information. The following
                      flags are available:
```

```
                      -nopre -mix -shuffle -decrypt -width
                      -nopos -nodec -noposc -noccpos
   <nizkp>          - Directory containing the non-interactive zero-knowledge
                      proof of correctness using the Fiat-Shamir heuristic.
   <protInfo>       - Protocol info file.

Options:
       -a <value>   - Determines if file based arrays are used or not. Legal
                      values are "file" or "ram" and the default is "file".
   -auxsid <value>  - Verify that the given auxiliary session identifier
                      matches that in the proof. This is required when the
                      auxiliary session identifier in the proof is not
                      "default".
       -c           - Print compatibility usage information.
  -decrypt          - Check proof of decryption.
       -e           - Show stack trace of an exception.
       -h           - Print usage information.
      -mc           - Print modified compatibility usage information. This can
                      be used by others to print the usage information that
                      their own verifiers must provide. Partial implementations
                      can remove certain functionality using flags.
     -mix           - Check proof of mixing.
  -noccpos          - Turn off verification of commitment-consistent proofs of
                      shuffles. This is only possible if pre-computation was
                      used during execution.
    -nodec          - Turn off verification of proof of decryption.
    -nopos          - Turn off verification of proofs of shuffles. If pre-
                      computation is used, this turns off verification of both
                      proofs of shuffles of commitments and commitment-
                      consistent proofs of shuffles.
   -noposc          - Turn off verification of proofs of shuffles of
                      commitments. This is only possible if pre-computation was
                      used during execution.
  -shuffle          - Check proof of shuffle.
   -sloppy          - Check proof of mixing/shuffle/decryption depending on
                      what is specified in the proof itself using the auxiliary
                      session identifier and width specified in the proof
                      itself. WARNING! If these values are not verified using
                      other means, then this does not constitute a complete
                      verification.
       -t <names>   - Print the given comma-separated test vectors. The "-th"
                      option can be used to list the available test vectors.
      -th           - List the available test vectors. The names are chosen to
                      be easily related to the notation used in the document
                      that describes the non-interactive zero-knowledge proof
                      of correctness. In particular for programmers that are
                      familiar with LaTeX.
       -v           - Verbose output, i.e., turn on output.
  -version          - Print the package version.
      -wd <dir>     - Directory for temporary files (default is a unique
                      subdirectory of /tmp/com.verificatum). This directory is
                      deleted on exit.
    -width <value>  - Verify that the given width matches that in the proof.
                      This is required when the width in the proof is different
                      from the width in the protocol info file.
```