Cryptographic Primitives of the Swiss Post Voting System

Pseudo-code Specification

Swiss Post

Version 0.9.7

Abstract

Cryptographic algorithms play a pivotal role in the Swiss Post Voting System: ensuring their faithful implementation is crucially important. This document provides a mathematically precise and unambiguous specification of some cryptographic primitives underpinning the Swiss Post Voting System. It focuses on the elements common to the system and its verifier, such as the verifiable mix net and non-interactive zero-knowledge proofs. We provide technical details about encoding methods between basic data types and describe each algorithm in pseudo-code format.

Disclaimer

E-Voting Community Program Material - please follow our Code of Conduct describing what you can expect from us, the Coordinated Vulnerability Disclosure Policy, and the contributing guidelines.

Revision chart

Version	Description	Author	Reviewer	Date
0.9	First version for external review	TH, OE	CK, HR, BS, KN	2021-02-05
0.9.1	Minor corrections in existing algorithms	TH, OE	CK, HR, BS, KN	2021-02-12
0.9.2	Completed mix net specification	TH, OE	CK, HR, BS, KN	2021-02-19
0.9.3	Version with reviewers' feedback for publication	TH, OE	CK, HR, BS, KN	2021-03-18
0.9.4	See change log crypto-primitives 0.8	TH, OE	CK, HR	2021-04-22
0.9.5	See change log crypto-primitives 0.9	TH, OE	CK, HR, BS	2021-06-22
0.9.6	See change log crypto-primitives 0.10	TH, OE	CK, HR	2021-07-26
0.9.7	See change log crypto-primitives 0.11	TH, OE	CK, HR	2021-09-01

Contents

Symbols

$\mathbb{A}_{Base16}$	Base16 (Hex) alphabet [15]
$\mathbb{A}_{Base32}$	Base32 alphabet [15]
$\mathbb{A}_{Base64}$	Base64 alphabet [15]
$\mathbb{A}_{UCS}$	Alphabet of the Universal Coded Character Set (UCS) according to ISO/IEC10646
$\mathcal{B}$	Set of possible values for a byte
$\mathcal{B}^*$	Set of byte arrays of arbitrary length
$\mathbb{B}^n$	Set of bit arrays of length n
g	Generator of the encryption group
$\mathbb{G}_q$	Set of quadratic residues modulo p of size q. The computational proof refers to this set as $\mathbb{Q}_p$
$\mathbb{H}_l$	Ciphertext domain ($= \underbrace{\mathbb{G}_q \times \cdots \times \mathbb{G}_q}_{l+1 \text{ times}}$)
k	Number of elements of a multi-recipient ElGamal key, $k \in \mathbb{N}^*$
l	Number of elements of a multi-recipient ElGamal message, $l \in \mathbb{N}^*$
ν	Size of a commitment key, $\nu \in \mathbb{N}^*$
$\mathbb{N}$	Set of positive integer numbers including 0
$\mathbb{N}^*$	Set of strictly positive integer numbers
$\mathbb{P}$	Set of prime numbers
p	Encryption group modulus
q	Encryption group cardinality s.t. $p = 2q + 1$
$\|x\|$	Bit length of the number x
$\mathbb{Z}_p$	Set of integers modulo p
$\mathbb{Z}_q$	Set of integers modulo q

1 Introduction

Switzerland has a longstanding tradition of direct democracy, allowing Swiss citizens to vote approximately four times a year on elections and referendums. In recent years, voter turnout hovered below 40 percent [7].

The vast majority of voters in Switzerland fill out their paper ballots at home and send them back to the municipality by postal mail, usually days or weeks ahead of the actual election date. Remote online voting (referred to as e-voting in this document) would provide voters with some advantages. First, it would guarantee the timely arrival of return envelopes at the municipality (especially for Swiss citizens living abroad). Second, it would improve accessibility for people with disabilities. Third, it would eliminate the possibility of an invalid ballot when inadvertently filling out the ballot incorrectly.

In the past, multiple cantons offered e-voting to a part of their electorate. Many voters would welcome the option to vote online - provided the e-voting system protects the integrity and privacy of their vote [8].
State-of-the-art e-voting systems alleviate the practical concerns of mail-in voting and, at the same time, provide a high level of security. Above all, they must display three properties [22]:

- Individual verifiability: allow a voter to convince herself that the system correctly registered her vote
- Universal verifiability: allow an auditor to check that the election outcome corresponds to the registered votes
- Vote secrecy: do not reveal a voter's vote to anyone

Following these principles, the Federal Chancellery defined stringent requirements for e-voting systems. The Ordinance on Electronic Voting (VEleS - Verordnung über die elektronische Stimmabgabe)[4] and its technical annex (VEleS annex)[5] describes these requirements.

Swiss democracy deserves an e-voting system with excellent security properties. In response to the release of an earlier version of the documentation and the source code of the e-voting system, multiple researchers published attacks, highlighted vulnerabilities, and suggested improvements[10, 14, 17, 23]. Swiss Post is thankful to all security researchers for their contributions and the opportunity to improve the system's security guarantees. We look forward to actively engaging with academic experts and the hacker community to maximize public scrutiny of the Swiss Post Voting System.

1.1 The Specification of Cryptographic Primitives

A vital element of a trustworthy and robust e-voting system is the description of the cryptographic algorithms in a form that leaves no room for interpretation and minimizes implementation errors [11].

Our pseudo-code description of the cryptographic algorithms—inspired by [12]—follows a consistent pattern:

- we prefix deterministic algorithms with Get* and probabilistic algorithms with Gen*;
- we designate values that do not change between runs as **Context** and variable values as **Input**;
- we ensure that each algorithm does only one thing (single responsibility principle);
- we explicit domains and ranges of input and output values;
- and we use 0-based indexing to close the representational gap between mathematics and code.

Furthermore, we believe that a specification encompassing the common elements between the Swiss Post Voting System and its verifier (an open-source software verifying the correct establishment of the election result) benefits both systems.

1.2 Validating the Cryptographic Algorithm's Correctness

We augment our specification with test values obtained from an independent implementation of the pseudo-code algorithms: our code validates against these test values to increase our confidence in the implementation's correctness. The specification embeds the test values as JSON files within the document.

2 Basic Data Types

We build upon basic data types such as bytes, integers, strings, and arrays. Moreover, we require algorithms to concatenate and truncate strings and byte arrays, to test primality and to sort arrays.

2.1 Byte Arrays

We denote a byte array B of length n as $< b_0, b_1, \ldots, b_{n-1} >$ where b_i denotes the $i+1$-th byte of the array. Byte arrays can be encoded as strings, and, conversely, decoded from strings using Base16, Base32, and Base64 encodings according to RFC4648 [15]. Table 2 shows different examples of byte arrays.

Byte Array	Byte Array (binary form)	Base64	Base32
<0xF3, 0x01, 0xA3>	11110011 00000001 10100011	"8wGj"	"6MA2G==="
<0xAC>	10101100	"rA=="	"VQ======"
<0x1F, 0x7F, 0x9D, 0x15, 0x12>	00011111 01111111 10011101 00010101 00010010	"H3+dFRI="	"D57Z2FIS"

Table 2: Example representations of different byte arrays

We indicate concatenation with the || operator.
<0xF3, 0x01, 0xA3> || <0xAC> = <0xF3, 0x01, 0xA3, 0xAC>

Algorithms 2.1, 2.2, 2.3, 2.4, 2.5 and 2.6 encode and decode byte arrays to and from Base16, Base32 and Base64 encodings. Potentially, decoding Base32 and Base64 may fail since the encoding is not bijective (only injective). For instance, one cannot decode the string "==TEOD8=" even though it is within the required alphabet.

Algorithm 2.1 Base16Encode

Input:
Byte array $B \in \mathcal{B}^*$

Operation:
1: $S \leftarrow \mathsf{Base16}(B)$

Output:
String $S \in \mathbb{A}_{Base16}$ ▷ According to RFC4648 [15]

Algorithm 2.2 Base16Decode

Input:
String $S \in \mathbb{A}_{Base16}$ ▷ According to RFC4648 [15]

Operation:
1: $B \leftarrow \mathsf{Base16}^{-1}(S)$

Output:
Byte array $B \in \mathcal{B}^*$

Algorithm 2.3 Base32Encode

Input:
Byte array $B \in \mathcal{B}^*$

Operation:
1: $S \leftarrow \mathsf{Base32}(B)$

Output:
String $S \in \mathbb{A}_{Base32}$ ▷ According to RFC4648 [15]

Algorithm 2.4 $\mathsf{Base32Decode}$

Input:
String $S \in \mathbb{A}_{Base32}$ ▷ According to RFC4648 [15]

Operation:
1: **if** S is not valid Base32 **then**
2: **return** $\perp$
3: **end if**
4: $B \leftarrow \mathsf{Base32}^{-1}(S)$

Output:
Byte array $B \in \mathcal{B}^*$

Algorithm 2.5 $\mathsf{Base64Encode}$

Input:
Byte array $B \in \mathcal{B}^*$

Operation:
1: $S \leftarrow \mathsf{Base64}(B)$

Output:
String $S \in \mathbb{A}_{Base64}$ ▷ According to RFC4648 [15]

Algorithm 2.6 $\mathsf{Base64Decode}$

Input:
String $S \in \mathbb{A}_{Base64}$

Operation:
1: **if** S is not valid Base64 **then**
2: **return** $\perp$
3: **end if**
4: $B \leftarrow \mathsf{Base64}^{-1}(S)$

Output:
Byte array $B \in \mathcal{B}^*$

2.2 Integers

When converting integers to byte array, we represent them in big-endian byte order. Since we only work with non-negative integers, we treat them as unsigned integers. Table 3 provides some example integers.

Integer	Byte Array (Hex)
3	<0x03>
128	<0x80>
23’591	<0x5C, 0x27>
23’592	<0x5C, 0x28>
4’294’967’295	<0xFF, 0xFF, 0xFF, 0xFF>
4’294’967’296	<0x01, 0x00, 0x00, 0x00, 0x00>

Table 3: Example representations of different integers

Therefore, we ignore leading zeros and define algorithm 2.7 to convert byte arrays to integers and algorithm 2.8 to convert integers to byte arrays.
$|x|$ derives the minimal bit length of an integer, e.g. $|4'294'967'295| = 32$ and $|4'294'967'296| = 33$.

Algorithm 2.7 ByteArrayToInteger

Input:

Byte array $B =< b_0, b_1, \ldots, b_{n-1} > \in \mathcal{B}^n$ of length $n \in \mathbb{N}^*$

Operation:

$x \leftarrow 0$
for $i \in [0, n)$ **do**
  $x \leftarrow 256 \cdot x + b_i$
end for

Output:

$x \in \mathbb{N}$

Algorithm 2.8 IntegerToByteArray

Input:

Positive integer $x \in \mathbb{N}$

Operation:

$n \leftarrow \lceil \frac{|x|}{8} \rceil$ ▷ Derive minimal length n of byte array
for $i \in [0, n)$ **do**
  $b_{n-i-1} \leftarrow x \mod 256$
  $x \leftarrow \lfloor \frac{x}{256} \rfloor$
end for
$B \leftarrow < b_0, b_1, \ldots, b_{n-1} >$

Output:

Byte array $B \in \mathcal{B}^n$

2.3 Strings

We encode strings in the universal coded character set (UCS) as defined in ISO/IEC10646, which is used by the encoding format UTF-8 (see RFC3629 [25]). Table 4 highlights some examples.

String	Byte Array (UCS)
"ABC"	<0x41, 0x42, 0x43>
"Ä"	<0xC3, 0x84>
"1001"	<0x31, 0x30, 0x30, 0x31>
"1A"	<0x31, 0x41>

Table 4: Example representations of different strings

Algorithms 2.9 and 2.10 convert byte arrays to strings and vice versa. Potentially, the ByteArrayToString method can fail since not every byte array maps to a valid sequence of UCS code points.

Algorithm 2.9 StringToByteArray

Input:
String $S \in \mathbb{A}_{UCS}{}^*$

Operation:
```
B ← UCS(S)        ▷ Convert S to its UCS representation
```

Output:
Byte array $B \in \mathcal{B}^*$

Algorithm 2.10 ByteArrayToString

Input:
Byte array $B =< b_0, b_1, \dots, b_{n-1} > \in \mathcal{B}^n$ of length $n \in \mathbb{N}^*$

Operation:
```
if B does not correspond to a valid sequence of UCS code points then
    return ⊥
end if
S ← UCS^{-1}(B)
```

Output:
String $S \in \mathbb{A}_{UCS}{}^*$

3 Basic Algorithms

3.1 Randomness

Several algorithms draw a value at random from a given domain and rely on a primitive providing the requested number of independent random bytes. Standard implementations for generating cryptographically secure random bytes are available in most programming languages; therefore, we omit the pseudo-code for this primitive and call it `randomBytes(length)`, where `length` is the required number of bytes.

Algorithm 3.1 `GenRandomInteger`: provide a random integer between 0 (incl.) and m (excl.)

Input:
Upper bound $m \in \mathbb{N}^*$

Operation:
```
length ← byteLength(m)
r ← ToInteger(randomBytes(length))
if r ≥ m then
    go back to step 2
end if
```

Output:
Random number $r \in [0, m)$

Algorithm 3.2 GenRandomVector: generate a random vector from ${\mathbb{Z}_q}^n$

Input:
Exclusive upperbound $q \in \mathbb{N}^*$
Length $n \in \mathbb{N}^*$

Operation:
```
for i ∈ [0, n) do
    r_i ← GenRandomInteger(q)                ▷ See algorithm 3.1
end for
```

Output:
Random vector $(r_0, \ldots, r_{n-1}) \in {\mathbb{Z}_q}^n$

Algorithms 3.3, 3.4, and 3.5 generate random strings using the Base16, Base32, and Base64 alphabet. Their typical use case is to generate random IDs of a specific alphabet; the method does not expect the output to be decodable.

Algorithm 3.3 GenRandomBase16String

Input:

Desired length of string: $l \in \mathbb{N}^*$

Operation:

1: $l_{bytes} = \lceil \frac{4*l}{8} \rceil$
2: $b = \mathsf{RandomBytes}(l_{bytes})$
3: $S = \mathsf{Truncate}(\mathsf{Base16Encode}(b), l)$ ▷ See algorithm 2.1

Output:

$S \in (\mathbb{A}_{Base16})^l$ ▷ A random string of the Base16 alphabet [15]

Algorithm 3.4 GenRandomBase32String

Input:

Desired length of string: $l \in \mathbb{N}^*$

Operation:

1: $l_{bytes} = \lceil \frac{5*l}{8} \rceil$
2: $b = \mathsf{RandomBytes}(l_{bytes})$
3: $S = \mathsf{Truncate}(\mathsf{Base32Encode}(b), l)$ ▷ See algorithm 2.3

Output:

$S \in (\mathbb{A}_{Base32})^l$ ▷ A random string of the Base32 alphabet [15]

Algorithm 3.5 GenRandomBase64String

Input:

Desired length of string: $l \in \mathbb{N}^*$

Operation:

$l_{bytes} = \lceil \frac{6*l}{8} \rceil$
$b = \mathsf{RandomBytes}(l_{bytes})$
$S = \mathsf{Truncate}(\mathsf{Base64Encode}(b), l)$ ▷ See algorithm 2.5

Output:

$S \in (\mathbb{A}_{Base64})^l$ ▷ A random string of the Base64 alphabet [15]

3.2 Recursive Hash

Our recursive hash function—inspired by CHVote[12]—ensures that different inputs (of the same type) to the hash function result in different outputs. In particular, the recursive hash function provides domain-separation: hashing ("A", "B") does not yield the same result as hashing ("AB").

There is a caveat regarding the edge case of hashing empty values; the recursive hash algorithm implies that both the empty string and the empty byte array result in the same hash-value, which is, strictly speaking, a collision. Callers of this function must therefore ensure that the domains of each element are clearly defined, thus avoiding collisions. Further, the recursive definition of the domain implies that infinite inputs are possible in theory, in which case the algorithm does not terminate. It is also the callers' responsibility to ensure only finite inputs are provided in practice.

Algorithm 3.6 RecursiveHash: Computes the hash value of multiple inputs

Context:

Pseudo-random hash function $\mathsf{Hash} : \mathcal{B}^* \mapsto \mathcal{B}^L$, $L \in \mathbb{N}^*$ ▷ Outputs a byte array of length L

Input:

Values $(v_0, \ldots, v_{k-1})$. Each value v_i is in domain $\mathcal{V}$, recursively defined as the union of:

- the set of byte arrays $\mathcal{B}^*$
- the set of valid UCS strings $\mathbb{A}_{UCS}$
- the set of non-negative integers $\mathbb{N}$
- the set of vectors $\mathcal{V}^*$

Require: $k > 0$, $L > 0$

Operation:

if k $>$ 1 **then** ▷ Avoid computing $\mathsf{Hash}(\mathsf{Hash}(v_0))$ when $k = 1$
    $\mathbf{v} \leftarrow (v_0, \ldots, v_{k-1})$
    $d \leftarrow \mathsf{RecursiveHash}(\mathbf{v})$
else
    $w \leftarrow v_0$
    if $w \in \mathcal{B}^*$ **then**
        $d \leftarrow \mathsf{Hash}(w)$
    else if $w \in \mathbb{A}_{UCS}$ **then**
        $d \leftarrow \mathsf{Hash}(\mathsf{StringToByteArray}(w))$ ▷ See algorithm 2.9
    else if $w \in \mathbb{N}$ **then**
        $d \leftarrow \mathsf{Hash}(\mathsf{IntegerToByteArray}(w))$ ▷ See algorithm 2.8
    else if $w = (w_0, \ldots, w_j)$ **then**
        if $j = 0$ **then**
            $d \leftarrow \mathsf{RecursiveHash}(w_0)$
        else
            $d \leftarrow \mathsf{Hash}(\mathsf{RecursiveHash}(w_0)||\ldots||\mathsf{RecursiveHash}(w_j))$
        end if
    else
      return $\perp$
    end if
end if

Output:

The digest $d \in \mathcal{B}^L$

Test values for algorithm 3.6 are provided in the attached recursive-hash-sha256.json file.

4 ElGamal Cryptosystem

The computational proof [20] describes the security properties of the ElGamal encryption scheme. Moreover, it explains that we can share the randomness when encrypting multiple messages (using different public keys). Optimizing the encryption scheme in this way is called multi-recipient ElGamal encryption and prevents us from repeatedly computing the left-hand side of the ciphertext.

4.1 Parameters Generation

We instantiate the ElGamal encryption scheme over the group of nonzero quadratic residues $\mathbb{G}_q \subset \mathbb{Z}_p$, defined by the following *public* parameters: modulus p, cardinality (order) q, and generator g . Since p and q are primes, quadratic residuosity implies $p = 2q + 1$. The ElGamal encryption scheme designates the default generator g as the smallest element in $\mathbb{G}_q$. Table 5 describes a *testing-only*, *default*, and an *extended* security level for the modulus p and order q. Some standards deem the *default* security level sufficient, while other standards advocate for the *extended* security level.

Security Level	Strength	Bit Length of p and q	Standards
testing-only	-	arbitrary	-
default	112 bits	$\|p\| = 2048$ bits, $\|q\| = 2047$ bits	NIST [2], ANSSI [1]
extended	128 bits	$\|p\| = 3072$ bits, $\|q\| = 3071$ bits	ECRYPT [21]

Table 5: Security levels for selecting group parameters.

We pick all the group parameters *verifiably* to demonstrate that they are devoid of hidden properties or back doors. Algorithm 4.1 details the verifiable selection of group parameters. The method takes a seed—the name of the election event—as an input. It leverages the SHAKE128 algorithm which produces a variable length digest [6]. Most implementations of SHAKE128 require as input a byte array and a length in bytes and the method returns a byte array. The initial candidate value for q, however, must be in the interval $[2^{|q|-1}, 2^{|q|})$. Therefore, we prepend the byte <0x01> to the digest's output (see section 2.2 on how we represent integers) and perform a subsequent bitwise right-shift operation.

Algorithm 4.1 GetEncryptionParameters

Context:

The security level λ defining the security strength and bit length of $|p|$ and $|q|$ according to table 5.

Input:

$seed \in \mathbb{A}_{UCS}{}^*$ ▷ The name of the election event

---

Operation:

$i \leftarrow 0$
do
  $\hat{q}_b \leftarrow \mathsf{SHAKE128}((\mathsf{StringToByteArray}(seed) \;||\; \mathsf{IntegerToByteArray}(i)), \frac{|p|}{8})$ ▷ See algorithm 2.9, 2.8, and FIPS PUB 202 [6]
  $q_b \leftarrow$ <0x01> $||\; \hat{q}_b$ ▷ Byte array concatenation
  $q \leftarrow \mathsf{ByteArrayToInteger}(q_b) \gg 2$ ▷ See algorithm 2.7 ▷ bit-wise right shift
  $q \leftarrow q + 1 - (q \bmod 2)$ ▷ Ensuring that q is odd
  $i \leftarrow i + 1$
while $(\neg\mathsf{isProbablePrime}(q, \lambda)$ **or** $\neg\mathsf{isProbablePrime}(2q + 1, \lambda))$ ▷ The certainty level of the probabilistic primality check derives from the security strength
$p \leftarrow 2q + 1$
for $i \in [2, 4]$ **do** ▷ necessarily $4 \in \mathbb{G}_q$ since every quadratic residue is a group element $(2^2 = 4)$
  if $i \in \mathbb{G}_q$ **then**
    return $g \leftarrow i$
  end if
end for

Output:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Test values for the algorithm 4.1 are provided in the attached get-encryption-parameters.json file.

4.2 Key Pair Generation

Algorithm 4.2 GenKeyPair: Generate a multi-recipient key pair

Input:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
Number of key elements $N \in \mathbb{N}^*$

Operation:

```
for i ∈ [0, N) do
    sk_i ← GenRandomInteger(q)        ▷ See algorithm 3.1
    pk_i ← g^{sk_i} mod p
end for
```

Output:

A pair of secret and public keys $\{(sk_i, pk_i)\}_{i=0}^{N-1}, sk_i \in \mathbb{Z}_q, pk_i \in \mathbb{G}_q$

In the Swiss Post Voting System, a ciphertext (i.e. an encrypted voter ballot) comprises the following elements:

- g^r, the left-hand side part of a standard ElGamal encryption,
- $pk_0^r \cdot m_0$, the encryption of the main message, under the first part of the system's public key. The message is the product of the choices of the voter,
- $pk_j^r \cdot m_j$ for all $1 \leq j < l$, where m_j is the encoding of a write-in chosen by the voter, or 1.
- When the number of write-ins a voter is eligible to vote for is smaller than the election configuration allows, the remaining keys are compressed by multiplication, so that all keys are used in every case.

The algorithms for handling the key-compression mentioned in the last point are provided below.

Algorithm 4.3 $\mathsf{CompressPublicKey}$: compress a public key to the requested length

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A multi-recipient public key $\mathbf{pk} = (\mathsf{pk}_0, \dots, \mathsf{pk}_{k-1}) \in \mathbb{G}_q^k$
The requested length l s.t. $0 < l \leq k$

Operation:

1: $\mathsf{pk}' = \prod_{i=l-1}^{k-1} \mathsf{pk}_i \mod p$

Output:

$(\mathsf{pk}_0, \dots, \mathsf{pk}_{l-2}, \mathsf{pk}') \in {\mathbb{G}_q}^l$ ▷ If $l = 1$, then the output vector contains only pk'

Algorithm 4.4 $\mathsf{CompressSecretKey}$: compress a secret key to the requested length

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A multi-recipient secret key $\mathbf{sk} = (\mathsf{sk}_0, \dots, \mathsf{sk}_{k-1}) \in \mathbb{Z}_q^k$
The requested length l s.t. $0 < l \leq k$

Operation:

1: $\mathsf{sk}' = \sum_{i=l-1}^{k-1} \mathsf{sk}_i \mod q$

Output:

$(\mathsf{sk}_0, \dots, \mathsf{sk}_{l-2}, \mathsf{sk}') \in {\mathbb{Z}_q}^l$ ▷ If $l = 1$, then the output vector contains only sk'

4.3 Encryption

A ballot can be considered as a "multi-recipient message", in which each part will be encrypted under a different key.

Algorithm 4.5 GetCiphertext: Compute a ciphertext with provided randomness

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A multi-recipient message $\mathbf{m} \in \mathbb{G}_q{}^l$
The random exponent to use $r \in \mathbb{Z}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$

Require: $0 < l \leq k$

Operation:

1: $\gamma \leftarrow g^r \mod p$
2: $(\mathsf{pk'}_0, \ldots, \mathsf{pk'}_{l-1}) \leftarrow \mathsf{CompressPublicKey}(\mathbf{pk}, l)$ ▷ See algorithm 4.3
3: **for** $i \in [0, l)$ **do**
4: $\quad \phi_i \leftarrow \mathsf{pk'}_i^r \cdot m_i \mod p$
5: **end for**

Output:

The ciphertext $(\gamma, \phi_0, \ldots, \phi_{l-1}) \in \mathbb{H}_l$

Test values for the algorithm 4.5 are provided in get-ciphertext.json.

4.4 Ciphertext Operations

Algorithm 4.6 GetCiphertextExponentiation: Exponentiate the ciphertext

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A multi-recipient ciphertext $C_a = (\gamma, \phi_0, \ldots, \phi_{l-1}) \in \mathbb{H}_l$
An exponent $a \in \mathbb{Z}_q$

Operation:

```
γ ← γ^a mod p
for i ∈ [0, l) do
    φ_i ← φ_i^a mod p
end for
```

Output:

$(\gamma, \phi_0, \ldots, \phi_{l-1}) \in \mathbb{H}_l$

Algorithm 4.7 GetCiphertextVectorExponentiation: Exponentiate a vector of ciphertexts and take the product

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A vector of ciphertexts $\vec{C} = (C_0, \ldots, C_{n-1}) \in (\mathbb{H}_l)^n$
A vector of exponents $\vec{a} = (a_0, \ldots, a_{n-1}) \in {\mathbb{Z}_q}^n$

Operation:

```
product ← GetCiphertext(1⃗, 0, pk)        ▷ Neutral element of ciphertext multiplication
for i ∈ [0, n) do
    product ← GetCiphertextProduct(product, GetCiphertextExponentiation(C_i, a_i))
                                          ▷ See algorithm 4.8 and algorithm 4.6
end for
```

Output:

The resulting product $\in \mathbb{H}_l$

Algorithm 4.8 GetCiphertextProduct: Multiply two ciphertexts

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A multi-recipient ciphertext $C_a = (\gamma_a, \phi_{a,0}, \ldots, \phi_{a,l-1}) \in \mathbb{H}_l$
Another multi-recipient ciphertext $C_b = (\gamma_b, \phi_{b,0}, \ldots, \phi_{b,l-1}) \in \mathbb{H}_l$

Operation:

1: $\gamma \leftarrow \gamma_a \cdot \gamma_b \mod p$
2: **for** $i \in [0, l)$ **do**
3: $\quad \phi_i \leftarrow \phi_{a,i} \cdot \phi_{b,i} \mod p$
4: **end for**

Output:

$(\gamma, \phi_0, \ldots, \phi_{l-1}) \in \mathbb{H}_l$

Test values for the algorithm 4.8 are provided in get-ciphertext-product.json.

4.5 Decryption

Algorithm 4.9 GetMessage: Retrieve the message from a ciphertext

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A multi-recipient ciphertext $\mathbf{c} \in \mathbb{H}_l$
A multi-recipient secret key $\mathbf{sk} \in \mathbb{Z}_q^k, 0 < l \leq k$

Operation:

1: $(\mathsf{sk}'_0, \ldots, \mathsf{sk}'_{l-1}) \leftarrow \mathsf{CompressSecretKey}(\mathbf{sk}, l)$ ▷ See algorithm 4.4
2: **for** $i \in [0, l)$ **do**
3: $\quad m_i \leftarrow \phi_i \cdot \gamma^{-\mathsf{sk}'_i} \mod p$
4: **end for**

Output:

The multi-recipient message $(m_0, \ldots, m_{l-1}) \in \mathbb{G}_q{}^l$

Since the system uses a multi-party re-encryption/decryption mixnet, in which the combined public key of the parties is used for encryption, each party actually performs a partial decryption. This entails that the actual output of the decryption phase for each party is actually still a ciphertext and the value for γ needs to be preserved to allow decryption by the following parties. This gives us the partial decryption algorithm in algorithm 4.10.

Algorithm 4.10 GetPartialDecryption: Partially decrypt a provided ciphertext

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A multi-recipient ciphertext $\mathbf{c} = (\gamma, \phi_0, \dots, \phi_{l-1}) \in \mathbb{H}_l$
A multi-recipient secret key $\mathbf{sk} \in \mathbb{Z}_q^k, 0 < l \leq k$

Operation:

1: $(m_0, \dots, m_{l-1}) \leftarrow \mathsf{GetMessage}(\mathbf{c}, \mathbf{sk})$ ▷ See algorithm 4.9
2: **return** $(\gamma, m_0, \dots, m_{l-1})$

Output:

The multi-recipient ciphertext $(\gamma, m_0, \dots, m_{l-1}) \in \mathbb{H}_l$

Algorithm 4.11 $\mathsf{GenVerifiableDecryptions}$: Provide a verifiable partial decryption of a vector of ciphertexts.

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A vector of ciphertexts $\mathbf{C} = (\mathbf{c}_0, \ldots, \mathbf{c}_{N-1}) \in (\mathbb{H}_l)^N$
A multi-recipient key pair $(\mathbf{pk}, \mathbf{sk}) \in \mathbb{G}_q^k \times \mathbb{Z}_q^k$
An array of optional additional information $\mathbf{i}_{\mathsf{aux}} \in (\mathbb{A}_{UCS}{}^*)^{\bar{*}}$

Require: $0 < l \leq k$

Operation:

```
for i ∈ [0, N) do
    c'_i ← GetPartialDecryption(c_i, sk)                    ▷ See algorithm 4.10
    (γ', φ'_0, ..., φ'_{l-1}) ← c'_i
    π_{dec,i} ← GenDecryptionProof(c_i, (pk, sk), (φ'_0, ..., φ'_{l-1}), i_aux)   ▷ See algorithm 6.2
end for
C' ← (c'_0, ..., c'_{N-1})
π_dec ← (π_{dec,0}, ..., π_{dec,N-1})
return (C', π_dec)
```

Output:

A vector of partially decrypted ciphertexts $\mathbf{C}' = (\mathbf{c}'_0, \ldots, \mathbf{c}'_{N-1}) \in (\mathbb{H}_l)^N$
A vector of decryption proofs $\boldsymbol{\pi}_{\mathsf{dec}} \leftarrow (\pi_{\mathsf{dec},0}, \ldots, \pi_{\mathsf{dec},N-1}) \in (\mathbb{Z}_q \times \mathbb{Z}_q{}^l)^N$

Algorithm 4.12 $\mathsf{VerifyDecryptions}$: Verify the decryptions of a vector of ciphertexts.

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A vector of ciphertexts $\mathbf{C} = (\mathbf{c}_0, \ldots, \mathbf{c}_{N-1}) \in (\mathbb{H}_l)^N$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A vector of partially decrypted ciphertexts $\mathbf{C}' = (\mathbf{c}'_0, \ldots, \mathbf{c}'_{N-1}) \in (\mathbb{H}_l)^N$
A vector of decryption proofs $\boldsymbol{\pi}_{\mathsf{dec}} = (\pi_{\mathsf{dec},0}, \ldots, \pi_{\mathsf{dec},N-1}) \in (\mathbb{Z}_q \times {\mathbb{Z}_q}^l)^N$
An array of optional additional information $\mathbf{i}_{\mathsf{aux}} \in ({\mathbb{A}_{UCS}}^*)^{\bar{*}}$

Require: $N > 0$
Require: $0 < l \leq k$

Operation:

for $i \in [0, N)$ **do**
  $(\gamma, \phi_0, \ldots, \phi_{l-1}) \leftarrow \mathbf{c}_i$
  $(\gamma', \phi'_0, \ldots, \phi'_{l-1}) \leftarrow \mathbf{c}'_i$
  if $\gamma \neq \gamma'$ **then**
    return $\perp$
  end if
  $\mathbf{m} \leftarrow (\phi'_0, \ldots, \phi'_{l-1})$
  $\mathsf{ok} \leftarrow \mathsf{VerifyDecryption}(\mathbf{c}_i, \mathbf{pk}, \mathbf{m}, \pi_{\mathsf{dec},i}, \mathbf{i}_{\mathsf{aux}})$ ▷ See algorithm 6.3
  if $\neg\mathsf{ok}$ **then**
    return $\perp$
  end if
end for
return $\top$ ▷ Succeed if and only if all verifications above succeeded

---

Output:

The result of the verification: $\top$ if **all** the verifications are successful, $\perp$ otherwise.

5 Mix Net

Verifiable mix nets underpin most modern e-voting schemes since they hide the relationship between encrypted votes (potentially linked to the voter's identifier) and decrypted votes [13]. A re-encryption mix net consists of a sequence of mixers, each of which shuffles and re-encrypts an input ciphertext list and returns a different ciphertext list containing the same plaintexts. Each mixer proves knowledge of the permutation and the randomness (without revealing them to the verifier). Verifying these proofs guarantees that no mixer added, deleted, or modified a vote. The most widely used verifiable mix nets are the ones from Telerius-Wikström [24] and Bayer-Groth [3]. The Swiss Post Voting System uses the Bayer-Groth mix net, which we describe in this section. The computational proof [20] discusses the security properties of the non-interactive version of the Bayer-Groth mix net. Please note that each control component in the Swiss Post Voting System combines a verifiable shuffle with a subsequent, verifiable decryption step. This section details only the verifiable shuffle.
Our implementation exposes the following two public methods:

Algorithm 5.1 `GenVerifiableShuffle`: Shuffle (including re-encryption), and provide a Bayer-Groth proof of the shuffle

---

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A vector of ciphertexts $\mathbf{C} \in (\mathbb{H}_l)^N$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$ ▷ This public key is passed as context to all sub-arguments

Require: $0 < l \leq k$
Require: $2 \leq N \leq q - 3$

Operation:

1: $(\mathbf{C}', \pi, \mathbf{r}) \leftarrow \mathsf{GenShuffle}(\mathbf{C}, \mathbf{pk})$ ▷ See algorithm 5.3
2: $(m, n) \leftarrow \mathsf{GetMatrixDimensions}(N)$ ▷ See algorithm 5.5
3: $\mathbf{ck} \leftarrow \mathsf{GetVerifiableCommitmentKey}(n)$ ▷ See algorithm 5.6 ▷ This commitment key is passed as context to all sub-arguments
4: $\mathsf{shuffleStatement} \leftarrow (\mathbf{C}, \mathbf{C}')$
5: $\mathsf{shuffleWitness} \leftarrow (\pi, \mathbf{r})$
6: $\mathsf{shuffleArgument} \leftarrow \mathsf{GetShuffleArgument}(\mathsf{shuffleStatement}, \mathsf{shuffleWitness}, m, n)$ ▷ See algorithm 5.11

Output:

$\mathbf{C}' \in (\mathbb{H}_l)^N$
$\mathsf{shuffleArgument}$

Algorithm 5.2 `VerifyShuffle`: Verify the output of a previously generated verifiable shuffle

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A vector of unshuffled ciphertexts $\mathbf{C} \in (\mathbb{H}_l)^N$
A vector of shuffled, re-encrypted ciphertexts $\mathbf{C}' \in (\mathbb{H}_l)^N$
A Bayer-Groth $\mathsf{shuffleArgument}$ ▷ See algorithm 5.11 for the domain
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$ ▷ This public key is passed as context to all sub-arguments

Require: $0 < l \leq k$
Require: $2 \leq N \leq q - 3$

Operation:

1: $(m, n) \leftarrow \mathsf{GetMatrixDimensions}(N)$ ▷ See algorithm 5.5
2: $\mathbf{ck} \leftarrow \mathsf{GetVerifiableCommitmentKey}(n)$ ▷ See algorithm 5.6 ▷ This commitment key is passed as context to all sub-arguments
3: $\mathsf{shuffleStatement} \leftarrow (\mathbf{C}, \mathbf{C}')$
4: **return** $\mathsf{VerifyShuffleArgument}(\mathsf{shuffleStatement}, \mathsf{shuffleArgument}, m, n)$ ▷ See algorithm 5.12

Output:

The result of the verification: $\top$ if the verification is successful, $\bot$ otherwise.

5.1 Pre-Requisites

5.1.1 Shuffle

Algorithm 5.3 shuffles a list of ciphertexts. We require the shuffled list of ciphertexts, the permutation, and the list of random exponents to prove the shuffle's correctness.

Algorithm 5.3 `GenShuffle`: Re-encrypting shuffle

Context:
- Group modulus $p \in \mathbb{P}$
- Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
- Group generator $g \in \mathbb{G}_q$

Input:
- A vector of ciphertexts $\mathbf{C} \in (\mathbb{H}_l)^N$
- A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$

Require: $0 < l \leq k$

Operation:
1: $(\pi_0, \ldots, \pi_{N-1}) \leftarrow \mathsf{GenPermutation}(N)$ ▷ See algorithm 5.4
2: **for** $i \in [0, N)$ **do**
3: $\quad r_i \leftarrow \mathsf{GenRandomInteger}(q)$ ▷ See algorithm 3.1
4: $\quad e \leftarrow \mathsf{GetCiphertext}(\vec{1}, r_i, \mathsf{pk})$ ▷ Ciphertext for a vector of l 1s. See algorithm 4.5
5: $\quad \mathbf{C'}_i \leftarrow e \cdot \mathbf{C}_{\pi_i}$ ▷ See algorithm 4.8 for ciphertext multiplication
6: **end for**

Output:
- $\mathbf{C'} = (\mathbf{C'}_0, \ldots, \mathbf{C'}_{N-1}) \in (\mathbb{H}_l)^N$ ▷ The result of the shuffle
- $\pi \in \Sigma_N$ ▷ The permutation used
- $\mathbf{r} = (r_0, \ldots, r_{l-1}) \in \mathbb{Z}_q{}^N$ ▷ The exponents used for re-encryption

Algorithm 5.4 provides a way to generate a random permutation of indices for a list of size N. It uses the algorithm formalized by Knuth in [16]. The pseudo-code below assumes 0-based indexing, and as such deviates from standard mathematical notation in favor of closer proximity to the implementation.

Algorithm 5.4 `GenPermutation`: Permutation of indices up to N

Input:

Permutation size $N \in \mathbb{N}^*$

Operation:

$\pi \leftarrow (0, \ldots, N-1)$
for $i \in [0, N)$ **do**
  $\mathsf{offset} \leftarrow \mathsf{GenRandomInteger}(N-i)$ ▷ See algorithm 3.1
  $\mathsf{tmp} \leftarrow \pi_i$
  $\pi_i \leftarrow \pi_{i+\mathsf{offset}}$
  $\pi_{i+\mathsf{offset}} \leftarrow \mathsf{tmp}$
end for

Output:

π ▷ A permutation of the values between 0 and $N-1$

Ensure: $\forall j \in [0, N) \rightarrow j \in \pi$

Ensure: $\pi \in \mathbb{Z}_{N-1}^{N}$ ▷ Those two elements combined ensure that $\pi \in \Sigma_N$

5.1.2 Matrix Dimensions

The Bayer-Groth mix net is memory optimal, when the ciphertexts can be arranged into matrices with an equal number of rows and columns. In the worst case, when the number of ciphertexts is prime, the resulting matrix has dimensions $1 \times N$. The below algorithm yields the optimal matrix size for a given number of ciphertexts. As an example, $N = 12$ results in $m = 3, n = 4$, $N = 18$ results in $m = 3, n = 6$, and $N = 23$ results in $m = 1, n = 23$,

---

Algorithm 5.5 `GetMatrixDimensions`: Return the optimal dimensions for the ciphertext matrix

---

Input:

Number of ciphertexts $N \in \mathbb{N}^* \setminus \{1\}$

Operation:

$m \leftarrow 1$
$n \leftarrow N$
for $i \in [\lfloor \sqrt{N} \rfloor, 1)$ **do**
  if $i \mid N$ **then**
    $m \leftarrow i$
    $n \leftarrow \frac{N}{i}$
    return m, n
  end if
end for

Output:

$m \in \mathbb{N}^*$
$n \in \mathbb{N}^* \setminus \{1\}$

5.2 Commitments

A cryptographic commitment allows a party to commit to a value (the opening), to keep the opening hidden from others, and to reveal it later [9].

We use the Pedersen commitment scheme[19] with a commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu)$ that was generated in a verifiable manner.

The Pedersen commitment scheme satisfies three properties that the Bayer-Groth mix net requires [3].

- *Perfectly hiding*: The commitment is uniformly distributed in $\mathbb{G}_q$.
- *Computationally binding*: It is computationally infeasible to find two different values producing the same commitment.
- *Homomorphic*: It holds that $\mathsf{GetCommitment}(a+b; r+s) = \mathsf{GetCommitment}(a; r)\mathsf{GetCommitment}(b; s)$ for messages a, b, a commitment key $\mathbf{ck}$ and random values r, s.

The Pedersen commitment scheme is *computationally binding* only if the commitment keys are generated independently and verifiably at random:

Algorithm 5.6 GetVerifiableCommitmentKey: Generates a verifiable commitment key

Context:

Group modulus $p \in \mathbb{P}$

Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Input:

The desired number of elements of the commitment key $\nu \in \mathbb{N}^*$

Ensure: $\nu \leq q - 3$

Operation:

```
count = 0
i = 0
v ← {}
while count ≤ ν do
    u ← ByteArrayToInteger(RecursiveHash(q, "commitmentKey", i, count))
    w ← u² mod p
    if w ∉ {0, 1, g} ∪ v then
        g_i ← w
        v ← v ∪ g_i
        count = count + 1
    end if
    i = i + 1
end while
h ← g_0                                ▷ By convention, we designate g_0 as h
```

Output:

A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Algorithm 5.7 GetCommitment: Computes a commitment to a value

Context:

Group modulus $p \in \mathbb{P}$

Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Input:

The values to commit to $\mathbf{a} \in {\mathbb{Z}_q}^l$

A random value $r \in \mathbb{Z}_q$

A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$ s.t. $\nu \geq l$

Ensure: $l > 0$

Operation:

$c \leftarrow h^r \prod_{i=1}^{l} g_i^{a_{i-1}} \mod p$

Output:

The commitment $c \in \mathbb{G}_q$

Algorithm 5.8 GetCommitmentMatrix: Computes the commitment for a matrix

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Input:
The values to be committed $A \in \mathbb{Z}_q{}^{n \times m}$ ▷ We note the columns of A as $\vec{a}_0, \ldots, \vec{a}_{m-1}$
The random values to use $\mathbf{r} \in \mathbb{Z}_q{}^m$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$ s.t. $\nu \geq n$

Ensure: $m, n > 0$

Operation:
1: **for** $i \in [0, m)$ **do**
2: $c_i \leftarrow \mathsf{GetCommitment}(\vec{a}_i, r_i, \mathbf{ck})$ ▷ See algorithm 5.7
3: **end for**

Output:
The commitments $(c_0, \ldots, c_{m-1}) \in \mathbb{G}_q{}^m$

This algorithm is consistent with the notation defined in [3], in section 2.3 Homomorphic Encryption:

> For a matrix $A \in \mathbb{Z}_q{}^{n \times m}$ with columns $\vec{a}_1, \ldots, \vec{a}_m$ we shorten notation by defining $com_{ck}(A; \vec{r}) = (com_{ck}(\vec{a}_1; r_1), \ldots, com_{ck}(\vec{a}_m; r_m))$

Algorithm 5.9 GetCommitmentVector: Compute the commitment for a transposed vector. This is only used in the algorithm 5.22, hence the specific indices

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Input:
The values to be commited $\mathbf{d} = (d_0, \ldots, d_{2m}) \in \mathbb{Z}_q{}^{2m+1}$
The random values to use $\mathbf{t} = (t_0, \ldots, t_{2m}) \in \mathbb{Z}_q{}^{2m+1}$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$ s.t. $\nu \geq 1$

Operation:
1: $(c_0, \ldots, c_{2m}) \leftarrow \mathsf{GetCommitmentMatrix}(\mathbf{d}, \mathbf{t}, \mathbf{ck})$ ▷ See algorithm 5.8, with a single row of $2m + 1$ columns
2: **return** $(c_0, \ldots, c_{2m})$

Output:
The commitments $(c_0, \ldots, c_{2m}) \in \mathbb{G}_q{}^{2m+1}$

5.3 Arguments

Conceptually, the Bayer-Groth proof of a shuffle consists of six arguments. Figure 1 highlights the hierarchy of these arguments.

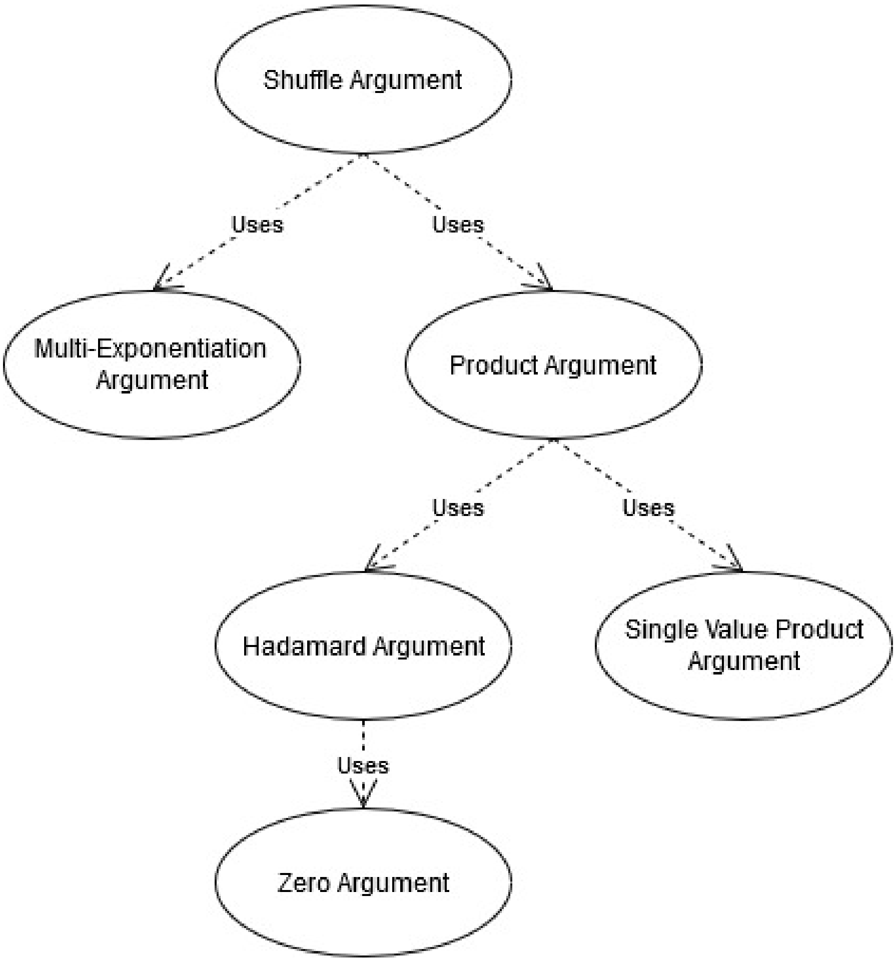

Figure 1: Bayer-Groth Argument for the Correctness of a Shuffle

The shuffle argument invokes a multi-exponentiation and a product argument. The product argument, in turn, uses a Hadamard and a single value product argument. Finally, the Hadamard argument calls a zero argument.

In all knowledge arguments, we will use the following terminology:

statement the public information for which we assert that a property holds

witness the private information we use to make arguments on the validity of the statement

argument the information we provide to a third party which allows them to verify the validity of our statement

We prove and verify the Bayer-Groth mix net in the *non-interactive* setting: the prover and the verifier recursively hash various elements to generate the argument's challenge messages. Since the mathematical group's rank q is much larger than the output domain, converting the hash function's output to an integer of byte size L is sound (byteLength(q) » L).

In some algorithms, we will use the bilinear algorithm $\star : \mathbb{Z}_q{}^n \times \mathbb{Z}_q{}^n \mapsto \mathbb{Z}_q$ defined by the value y as:

$$(a_0, \dots, a_{n-1}) \star (b_0, \dots, b_{n-1}) = \sum_{j=0}^{n-1} a_j \cdot b_j \cdot y^{j+1}$$

Where all multiplications are performed modulo q.
Let us formalize it with the following pseudocode algorithm.

Algorithm 5.10 StarMap: Defines the $\star$ bilinear map

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Input:

Value $y \in \mathbb{Z}_q$
First vector $\mathbf{a} = (a_0, \dots, a_{n-1}) \in \mathbb{Z}_q{}^n$
Second vector $\mathbf{b} = (b_0, \dots, b_{n-1}) \in \mathbb{Z}_q{}^n$

Operation:

```
s ← 0
for j ∈ [0, n) do
    s ← s + a_j · b_j · y^(j+1)        ▷ All operations are performed modulo q
end for
```

Output:

$s \in \mathbb{Z}_q$

Test values for the algorithm 5.10 are provided in the attached bilinearMap.json file.

5.3.1 Shuffle Argument

In the following pseudo-code algorithm, we will generate an argument of knowledge of a permutation $\pi \in \Sigma_N$ and randomness $\rho \in {\mathbb{Z}_q}^N$ such that for given ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$ and $\vec{C}' \in (\mathbb{H}_l)^N$ it holds that for all $i \in [0, N)$:

$$\vec{C}'_i = \mathsf{GetCiphertextProduct}(\mathsf{GetCiphertext}(\vec{1}, \rho_i, \mathbf{pk}), \vec{C}_{\pi(i)})$$

Algorithm 5.11 GetShuffleArgument: compute a cryptographic argument for the validity of the shuffle

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:
The statement composed of
- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$ s.t. $0 < l \leq k$
- The shuffled and re-encrypted list of ciphertexts $\vec{C'} \in (\mathbb{H}_l)^N$

The witness composed of
- permutation $\pi \in \Sigma_N$
- randomness $\vec{\rho} \in {\mathbb{Z}_q}^N$

The number of rows to use for ciphertext matrices $m \in \mathbb{N}^*$
The number of columns to use for ciphertext matrices $n \in \mathbb{N}^*$ s.t. $2 \leq n \leq \nu$

Ensure: $\forall i \in [0, N) : \vec{C}'_i = \mathsf{GetCiphertextProduct}(\mathsf{GetCiphertext}(\vec{1}, \rho_i, \mathbf{pk}), \vec{C}_{\pi(i)})$
Ensure: $N = mn$

Operation:
```
r ← GenRandomVector(q, m)                                         ▷ See algorithm 3.2
A ← Transpose(ToMatrix({π(i)}_{i=0}^{N−1}, m, n))   ▷ Create a n × m matrix. See algorithm 5.14 and algorithm 5.13
c_A ← GetCommitmentMatrix(A, r, ck)                               ▷ See algorithm 5.8
x ← ByteArrayToInteger(RecursiveHash(p, q, pk, ck, C⃗, C⃗', c_A))
s ← GenRandomVector(q, m)
b ← {x^{π(i)}}_{i=0}^{N−1}
B ← Transpose(ToMatrix(b, m, n))
c_B ← GetCommitmentMatrix(B, s, ck)
y ← ByteArrayToInteger(RecursiveHash(c_B, p, q, pk, ck, C⃗, C⃗', c_A))
z ← ByteArrayToInteger(RecursiveHash("1", c_B, p, q, pk, ck, C⃗, C⃗', c_A))
                                        ▷ Both C⃗ and C⃗' are passed in the vector forms here
Zneg ← Transpose(ToMatrix({−z}_{i=1}^{N}, m, n))    ▷ Vector of length N, with all values being q − z
c_{−z} ← GetCommitmentMatrix(Zneg, 0⃗, ck)          ▷ A vector of length m, with all 0 values
c_D ← c_A^y c_B                                                   ▷ Entry-wise product
D ← yA + B
t ← yr + s
b ← ∏_{i=0}^{N−1} (yi + x^i − z)
pStatement ← (c_D c_{−z}, b)
pWitness ← (D + Zneg, t)
productArgument ← GetProductArgument(pStatement, pWitness)        ▷ See algorithm 5.18
ρ ← q − (ρ⃗ · b)                             ▷ Standard inner product ∑_{i=0}^{N−1} ρ_i b_i
x⃗ ← {x^i}_{i=0}^{N−1}
C ← GetCiphertextVectorExponentiation(C⃗, x⃗)                      ▷ See algorithm 4.7
mStatement ← (ToMatrix(C⃗', m, n), C, c_B)                         ▷ See algorithm 5.13
mWitness ← (B, s, ρ)
multiExponentiationArgument ← GetMultiExponentiationArgument(mStatement, mWitness)   ▷ See
    algorithm 5.15
```

Output:
shuffleArgument $(\mathbf{c}_A, \mathbf{c}_B, \mathsf{productArgument}, \mathsf{multiExponentiationArgument}) \in {\mathbb{G}_q}^m \times {\mathbb{G}_q}^m \times \ldots \times \ldots$
▷ See algorithm 5.18 and algorithm 5.15 for their respective domains

In the following pseudo-code algorithm, we verify that a provided Shuffle argument adequately supports the corresponding statement.

Algorithm 5.12 VerifyShuffleArgument: Verify a cryptographic argument for the validity of the shuffle

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:
The statement composed of
- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_l)^N$ s.t. $0 < l \leq k$
- The shuffled and re-encrypted list of ciphertexts $\vec{C}' \in (\mathbb{H}_l)^N$
The argument composed of
- the commitment vector $\mathbf{c}_A \in {\mathbb{G}_q}^m$
- the commitment vector $\mathbf{c}_B \in {\mathbb{G}_q}^m$
- a productArgument ▷ See algorithm 5.18
- a multiExponentiationArgument ▷ See algorithm 5.15
The number of rows to use for ciphertext matrices $m \in \mathbb{N}^*$
The number of columns to use for ciphertext matrices $n \in \mathbb{N}^*$ s.t. $2 \leq n \leq \nu$

Ensure: $N = mn$

Operation:
$x \leftarrow$ ByteArrayToInteger(RecursiveHash($p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C}', \mathbf{c}_A$))
$y \leftarrow$ ByteArrayToInteger(RecursiveHash($\mathbf{c}_B, p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C}', \mathbf{c}_A$))
$z \leftarrow$ ByteArrayToInteger(RecursiveHash("1", $\mathbf{c}_B, p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C}', \mathbf{c}_A$)) ▷ Both $\vec{C}$ and $\vec{C}'$ are passed as vectors
Zneg $\leftarrow$ Transpose(ToMatrix($\{-z\}_{i=1}^N, m, n$)) ▷ Vector of length N, with all values being $q - z$, see algorithm 5.14 and algorithm 5.13
$\mathbf{c}_{-z} \leftarrow$ GetCommitmentMatrix(Zneg, $\vec{0}$, $\mathbf{ck}$) ▷ See algorithm 5.8
$\mathbf{c}_D \leftarrow \mathbf{c}_A^y \mathbf{c}_B$
$b \leftarrow \prod_{i=1}^N yi + x^i - z$
pStatement $\leftarrow (\mathbf{c}_D \mathbf{c}_{-z}, b)$
productVerif $\leftarrow$ VerifyProductArgument(pStatement, productArgument) ▷ See algorithm 5.19
$\vec{x} \leftarrow \{x^i\}_{i=0}^{N-1}$
$C \leftarrow$ GetCiphertextVectorExponentiation($\vec{C}, \vec{x}$) ▷ See algorithm 4.7
mStatement $\leftarrow$ (ToMatrix($\vec{C}', m, n$), C, $\mathbf{c}_B$) ▷ See algorithm 5.13
multiVerif $\leftarrow$ VerifyMultiExponentiationArgument(mStatement, multiExponentiationArgument) ▷ See algorithm 5.16
if productVerif $\wedge$ multiVerif **then**
  return $\top$
else
  return $\perp$
end if

Output:
The result of the verification: $\top$ if the verification is successful, $\perp$ otherwise.

Test values for the algorithm 5.12 are provided in the attached verify-shuffle-argument.json file.

One of the key features of Bayer-Groth's[3] minimal shuffle argument is the transformation of a vector of ciphertexts into a $m \times n$ matrix, by means of which a prover's computation can be optimized. Therefore, the ciphertexts, received as a vector, need to be organized into a matrix. This will be achieved by setting $M_{i,j} = \vec{v}_{ni+j}$. Similarly, the exponents in the matrices A and B need to be arranged into matrices so that the other algorithms get the expected values. However, since exponents' matrices have their dimensions transposed with respect to the ciphertexts, we obtain $M_{i,j} = \vec{v}_{i+nj}$.

For completeness, we describe below the operations of organizing the elements of a vector into a matrix and the transposition of a matrix.

Algorithm 5.13 ToMatrix: convert a vector of elements to a $m \times n$ matrix

Input:
a vector of elements $\vec{v} \in \mathbb{D}^N$
the number of requested rows $m \in \mathbb{N}^*$
the number of requested columns $n \in \mathbb{N}^*$
Ensure: $N = mn$

Operation:
```
for i ∈ [0, m) do
    for j ∈ [0, n) do
        M_{i,j} ← v⃗_{ni+j}
    end for
end for
```

Output:
The matrix $M = (M_{i,j})_{i,j=0}^{m,n} \in \mathbb{D}^{m \times n}$

Algorithm 5.14 Transpose: transpose a $m \times n$ matrix to a $n \times m$ matrix

Input:
a matrix of elements $M \in \mathbb{D}^{m \times n}$ s.t. $m, n > 0$

Operation:
```
for i ∈ [0, n) do
    for j ∈ [0, m) do
        N_{i,j} ← M_{j,i}
    end for
end for
```

Output:
The matrix $N = (N_{i,j})_{i,j=0}^{n,m} \in \mathbb{D}^{n \times m}$

5.3.2 Multi-Exponentiation Argument

Given ciphertexts $C_{0,0}, \dots, C_{m-1,n-1}$ and C, each $\in \mathbb{H}_l$, the algorithm below generates an argument of knowledge of the openings to the commitments $\vec{c}_A$ to $A = \{a_{i,j}\}_{i,j=1}^{n,m}$ such that

$$C = \mathsf{GetCiphertext}(\vec{1}, \rho, \mathbf{pk}) \cdot \prod_{i=0} \vec{C}_i^{\vec{a}_{i+1}}$$
$$\vec{c}_A = \mathsf{GetCommitmentMatrix}(A, \vec{r}, \mathbf{ck})$$

where $\vec{C}_i = (C_{i,0}, \dots, C_{i,n-1})$ and $\vec{a}_j = (a_{1,j}, \dots, a_{n,j})^T$, that is $\vec{C}_i$ refers to the i^{th} row of the matrix, whereas $\vec{a}_j$ refers to the j^{th} column. Furthermore, we use 0-based indexing for C here, which is consistent with the rest of this document, but 1-based indexing for a above, as well as $\vec{a}$ and $\mathbf{r}$ below, allowing for the generation of $\vec{a}_0$ and r_0 within the pseudo-code.

Algorithm 5.15 GetMultiExponentiationArgument: Compute a multi-exponentiation argument

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \dots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:
The statement composed of
- ciphertext matrix $(\vec{C}_0, \dots, \vec{C}_{m-1}) \in (\mathbb{H}_l)^{m \times n}$ s.t. $0 < l \leq k$ ▷ $\vec{C}_i$ refers to the i^{th} row
- ciphertext $C \in \mathbb{H}_l$
- commitment vector $\vec{c}_A \in \mathbb{G}_q{}^m$

The witness composed of
- matrix $A = (\vec{a}_1, \dots, \vec{a}_m) \in \mathbb{Z}_q{}^{n \times m}$ s.t. $n \leq \nu$ ▷ $\vec{a}_j$ refers to the j^{th} column
- exponents $\mathbf{r} = (r_1, \dots, r_m) \in \mathbb{Z}_q{}^m$
- exponent $\rho \in \mathbb{Z}_q$

Ensure: $C = \mathsf{GetCiphertext}(\vec{1}, \rho, \mathbf{pk}) \cdot \prod_{i=0}^{m-1} \vec{C}_i^{\vec{a}_{i+1}}$ ▷ Vector of 1s of length l ▷ See algorithm 4.5, algorithm 4.7, algorithm 4.8
Ensure: $\vec{c}_A = \mathsf{GetCommitmentMatrix}(A, \vec{r}, \mathbf{ck})$
Ensure: $n, m > 0$

Operation:
$\vec{a}_0 \leftarrow \mathsf{GenRandomVector}(q, n)$ ▷ See algorithm 3.2
$r_0 \leftarrow \mathsf{GenRandomInteger}(q)$ ▷ See algorithm 3.1
$(b_0, \dots, b_{2m-1}) \leftarrow \mathsf{GenRandomVector}(q, 2m)$
$(s_0, \dots, s_{2m-1}) \leftarrow \mathsf{GenRandomVector}(q, 2m)$
$(\tau_0, \dots, \tau_{2m-1}) \leftarrow \mathsf{GenRandomVector}(q, 2m)$
$b_m \leftarrow 0$
$s_m \leftarrow 0$
$\tau_m \leftarrow \rho$ ▷ Ensuring $c_{B_m} = \mathsf{GetCommitment}(0, 0, \mathbf{ck})$ and $\mathsf{GetCiphertext}(\vec{g}^{b_m}, \tau_m, \mathbf{pk}) = \mathsf{GetCiphertext}(\vec{1}, \rho, \mathbf{pk})$
$c_{A_0} \leftarrow \mathsf{GetCommitment}(\vec{a}_0, r_0, \mathbf{ck})$ ▷ See algorithm 5.7
$(D_0, \dots, D_{2m-1}) \leftarrow \mathsf{GetDiagonalProducts}((\vec{C}_0, \dots, \vec{C}_{m-1}), (\vec{a}_0, \dots, \vec{a}_m))$ ▷ See algorithm 5.17
for $k \in [0, 2m)$ **do**
  $c_{B_k} \leftarrow \mathsf{GetCommitment}((b_k), s_k, \mathbf{ck})$
  $E_k \leftarrow \mathsf{GetCiphertext}(\overrightarrow{g^{b_k}}, \tau_k, \mathbf{pk}) \cdot D_k$ ▷ See algorithm 4.5, we take a vector of messages of length l each having value g^{b_k}
end for
$x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, (\vec{C}_i)_{i=0}^{m-1}, C, \vec{c}_A, c_{A_0}, (c_{B_k})_{k=0}^{2m-1}, (E_k)_{k=0}^{2m-1}))$
▷ All operations below are performed modulo q
$\vec{a} \leftarrow \vec{a}_0 + \sum_{i=1}^{m} x^i \vec{a}_i$
$r \leftarrow r_0 + \sum_{i=1}^{m} x^i r_i$
$b \leftarrow b_0 + \sum_{k=1}^{2m-1} x^k b_k$
$s \leftarrow s_0 + \sum_{k=1}^{2m-1} x^k s_k$
$\tau \leftarrow \tau_0 + \sum_{k=1}^{2m-1} x^k \tau_k$

Output:
The argument $(c_{A_0}, (c_{B_k})_{k=0}^{2m-1}, (E_k)_{k=0}^{2m-1}, \vec{a}, r, b, s, \tau) \in \mathbb{G}_q \times \mathbb{G}_q{}^{2m} \times \mathbb{H}_l{}^{2m} \times \mathbb{Z}_q{}^n \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q$

In the following pseudo-code algorithm, we verify that a provided Multi-Exponentiation argument adequately supports the corresponding statement.

Algorithm 5.16 VerifyMultiExponentiationArgument: Verify a multi-exponentiation argument

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of
- ciphertext matrix $(\vec{C}_0, \ldots, \vec{C}_{m-1}) \in (\mathbb{H}_l)^{m \times n}$ ▷ $\vec{C}_i$ refers to the i^{th} row
- ciphertext $C \in \mathbb{H}_l$
- commitment vector $\vec{c}_A = (c_{A_1}, \ldots, c_{A_m}) \in \mathbb{G}_q{}^m$

The argument composed of
- the commitment $c_{A_0} \in \mathbb{G}_q$
- the commitment vector $\mathbf{c}_B = (c_{B_0}, \ldots, c_{B_{2m-1}}) \in \mathbb{G}_q{}^{2m}$
- the ciphertext vector $\mathbf{E} = (E_0, \ldots, E_{2m-1}) \in \mathbb{H}_l{}^{2m}$
- the vector of exponents $\vec{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q{}^n$
- the exponent $r \in \mathbb{Z}_q$
- the exponent $b \in \mathbb{Z}_q$
- the exponent $s \in \mathbb{Z}_q$
- the exponent $\tau \in \mathbb{Z}_q$

Ensure: $n, m > 0$

Operation:

```
x ← ByteArrayToInteger(RecursiveHash(p, q, pk, ck, {C⃗_i}_{i=0}^{m-1}, C, c⃗_A, c_{A_0}, {c_{B_k}}_{k=0}^{2m-1}, {E_k}_{k=0}^{2m-1}))
verifCbm ← c_{B_m} = 1
verifEm ← E_m = C

prodCa ← c_{A_0} ∏_{i=1}^{m} c_{A_i}^{x^i}
commA ← GetCommitment(a⃗, r, ck)                 ▷ See algorithm 5.7
verifA ← prodCa = commA

prodCb ← c_{B_0} ∏_{k=1}^{2m-1} c_{B_k}^{x^k}
commB ← GetCommitment((b), s, ck)
verifB ← prodCb = commB

prodE ← E_0 ∏_{k=1}^{2m-1} E_k^{x^k}
encryptedGb ← GetCiphertext(g⃗^b, τ, pk)          ▷ See algorithm 4.5, we take a vector of messages of length l each
    having value g^b
prodC ← ∏_{i=0}^{m-1} GetCiphertextVectorExponentiation(C⃗_i, x^{(m-i)-1} a⃗)   ▷ See algorithm 4.7
verifEC ← prodE = GetCiphertextProduct(encryptedGb, prodC)               ▷ See algorithm 4.8

if verifCbm ∧ verifEm ∧ verifA ∧ verifB ∧ verifEC then
    return ⊤
else
    return ⊥
end if
```

Output:

The result of the verification: ⊤ if the verification is successful, ⊥ otherwise.

Test values for the algorithm 5.16 are provided in the attached verify-multiexp-argument.json file.

Algorithm 5.17 GetDiagonalProducts: Compute the products of the diagonals of a ciphertext matrix

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$

Input:

Ciphertext matrix $C = (\vec{C}_0, \ldots, \vec{C}_{m-1}) \in (\mathbb{H}_l)^{m \times n}$ ▷ $\vec{C}_i$ refers to the i^{th} row
Exponent matrix $A = (\vec{a}_0, \ldots, \vec{a}_m) \in \mathbb{Z}_q^{n \times (m+1)}$ ▷ $\vec{a}_j$ refers to the j^{th} column

Operation:

```
for k ∈ [0, 2m) do
    d_k ← GetCiphertext(1⃗, 0, pk)    ▷ Neutral element of ciphertext multiplication
    if k < m then
        lowerbound ← m − k − 1
        upperbound ← m
    else
        lowerbound ← 0
        upperbound ← 2m − k
    end if
    for i ∈ [lowerbound, upperbound) do
        j ← k − m + i + 1
        d_k ← GetCiphertextProduct(d_k, GetCiphertextVectorExponentiation(C⃗_i, a⃗_j))    ▷ See algorithm 4.8 and algorithm 4.7
    end for
end for
```

Output:

Diagonal products $D = (d_0, \ldots, d_{2m-1}) \in \mathbb{H}_l^{2m}$

5.3.3 Product Argument

The following algorithm provides an argument that a set of committed values have a particular product.

More precisely, given commitments $\vec{c}_A = (c_{A_0}, \dots, c_{A_m})$ to $A = \{a_{i,j}\}_{i,j=0}^{n-1,m-1}$ and a value b, we want to give an argument of knowledge for $\prod_{i=0}^{n-1} \prod_{j=0}^{m-1} a_{i,j} = b$.

We will first compute a commitment c_b as follows:

$$c_b = \mathsf{GetCommitment}\left(\left(\prod_{j=0}^{m-1} a_{0,j}, \dots, \prod_{j=0}^{m-1} a_{n-1,j}\right), s, \mathbf{ck}\right)$$

We will then give an argument that c_b is correct, using a Hadamard argument (see section 5.3.4), showing that the values committed in c_b are indeed the result of the Hadamard product of the values committed in c_A. Additionally, we will show that the value b is the product of the values committed in c_b, using a Single Value Product Argument (see section 5.3.6).

If the number of ciphertexts to be shuffled is prime and they cannot be arranged into a matrix, $m = 1$ and $n = N$, the Hadamard Product is trivially equal to the first (and only) column of the matrix and we can omit the Hadamard argument, calling the Single Value Product argument directly.

Algorithm 5.18 GetProductArgument: Computes a Product Argument

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \dots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:
The statement composed of
- commitments $\vec{c}_A = (c_{A_1}, \dots, c_{A_m}) \in {\mathbb{G}_q}^m$
- the product $b \in \mathbb{Z}_q$
The witness composed of
- the matrix $A \in {\mathbb{Z}_q}^{n \times m}$
- the exponents $\vec{r} \in {\mathbb{Z}_q}^m$

Ensure: $2 \leq n \leq \nu$
Ensure: $m > 0$
Ensure: $\vec{c}_A = \mathsf{GetCommitmentMatrix}(A, \vec{r}, \mathbf{ck})$ ▷ See algorithm 5.8
Ensure: $b = \prod_{i=0}^{n-1} \prod_{j=0}^{m-1} a_{i,j} \mod q$

Operation:
if $m > 1$ **then**
  $s \leftarrow \mathsf{GenRandomInteger}(q)$
  for $i \in [0, n)$ **do**
    $b_i \leftarrow \prod_{j=0}^{m-1} a_{i,j}$
  end for
  $c_b \leftarrow \mathsf{GetCommitment}((b_0, \dots, b_{n-1}), s, \mathbf{ck})$
  $\mathsf{hStatement} \leftarrow (\vec{c}_A, c_b)$
  $\mathsf{hWitness} \leftarrow (A, (b_0, \dots, b_{n-1}), \vec{r}, s)$
  $\mathsf{hadamardArg} \leftarrow \mathsf{GetHadamardArgument}(\mathsf{hStatement}, \mathsf{hWitness})$ ▷ See algorithm 5.20
  $\mathsf{sStatement} \leftarrow (c_b, b)$
  $\mathsf{sWitness} \leftarrow ((b_0, \dots, b_{n-1}), s)$
  $\mathsf{singleValueProdArg} \leftarrow \mathsf{GetSingleValueProductArgument}(\mathsf{sStatement}, \mathsf{sWitness})$ ▷ See algorithm 5.25
else
  $\mathsf{sStatement} \leftarrow (c_{A_1}, b)$
  $\mathsf{sWitness} \leftarrow (\vec{a}_0, r_0)$
  $\mathsf{singleValueProdArg} \leftarrow \mathsf{GetSingleValueProductArgument}(\mathsf{sStatement}, \mathsf{sWitness})$ ▷ See algorithm 5.25
end if

Output:
if $m > 1$ **then**
  $(c_b, \mathsf{hadamardArg}, \mathsf{singleValueProdArg}) \in \mathbb{G}_q \times \mathsf{HadamardArgument} \times \mathsf{SingleValueProductArgument}$
  ▷ See algorithm 5.20 and algorithm 5.25 for the domains
else
  $\mathsf{singleValueProdArg}$ ▷ See algorithm 5.25 for the domain
end if

In the following pseudo-code algorithm, we verify if a provided Product argument supports the corresponding statement.

Algorithm 5.19 $\mathsf{VerifyProductArgument}$: Verify a Product argument

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of
- commitments $\vec{c}_A = (c_{A_1}, \ldots, c_{A_m}) \in \mathbb{G}_q{}^m$
- the product $b \in \mathbb{Z}_q$

The argument composed of
- the commitment $c_b \in \mathbb{G}_q$ ▷ omitted if $m = 1$
- a $\mathsf{hadamardArg} \in \mathbb{G}_q{}^{m+1} \times (\mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q{}^{2m+1} \times \mathbb{Z}_q{}^n \times \mathbb{Z}_q{}^n \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q)$ ▷ omitted if $m = 1$
- a $\mathsf{singleValueProductArg} \in \mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q \times \mathbb{Z}_q{}^n \times \mathbb{Z}_q{}^n \times \mathbb{Z}_q \times \mathbb{Z}_q$

Ensure: $n \le \nu$

Operation:

```
if m > 1 then
    hStatement ← (c⃗_A, c_b)
    sStatement ← (c_b, b)
    if VerifyHadamardArgument(hStatement, hadamardArg)∧
VerifySingleValueProductArgument(sStatement, singleValueProductArg) then
                                        ▷ See algorithm 5.21 and algorithm 5.26
        return ⊤
    else
        return ⊥
    end if
else
    sStatement ← (c_{A_1}, b)
    if VerifySingleValueProductArgument(sStatement, singleValueProductArg) then
                                                        ▷ See algorithm 5.26
        return ⊤
    else
        return ⊥
    end if
end if
```

Output:

The result of the verification: $\top$ if the verification is successful, $\perp$ otherwise.

Test values for the algorithm 5.19 are provided in the attached verify-p-argument.json file.

5.3.4 Hadamard Argument

The operations given in the algorithm below are more readable using vector notation. That is, we note $\vec{a}$ for the vector $(a_0, \ldots, a_{n-1}) \in {\mathbb{Z}_q}^n$. By extension, we denote matrix $a_{0,0}, \ldots, a_{n-1,m-1}$ as $\vec{a}_0, \ldots, \vec{a}_{m-1}$ where each vector corresponds to a column of the matrix.

In the following algorithm, we generate an argument of knowledge of the openings $\vec{a}_0, \ldots, \vec{a}_{m-1}$ and $\vec{b}$ to the commitments $\mathbf{c}_A$ and c_b, such that:

$$
\begin{aligned}
\mathbf{c}_A &= \mathsf{GetCommitmentMatrix}((\vec{a}_0, \ldots, \vec{a}_{m-1}), \mathbf{r}, \mathbf{ck}) \\
c_b &= \mathsf{GetCommitment}((b_0, \ldots, b_{n-1}), s, \mathbf{ck}) \\
b_i &= \prod_{j=0}^{m-1} a_{i,j} \text{ for } i = 0, \ldots, n-1
\end{aligned}
$$

where the product in the last line matches the entry-wise product, also known as Hadamard product.

The subsequent pseudo-code algorithm verifies if a provided Hadamard argument supports the corresponding statement.

Algorithm 5.20 GetHadamardArgument: Computes a Hadamard Argument

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \dots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:
The statement composed of
- commitment $\mathbf{c}_A = (c_{A_0}, \dots, c_{A_{m-1}}) \in {\mathbb{G}_q}^m$
- commitment $c_b \in \mathbb{G}_q$

The witness composed of
- matrix $A = (\vec{a}_0, \dots, \vec{a}_{m-1}) \in {\mathbb{Z}_q}^{n \times m}$
- vector $\mathbf{b} \in {\mathbb{Z}_q}^n$
- exponents $\mathbf{r} = (r_0, \dots, r_{m-1}) \in {\mathbb{Z}_q}^m$
- exponent $s \in \mathbb{Z}_q$

Ensure: $m \geq 2$ ▷ Hadamard product only makes sense for $m \geq 2$
Ensure: $0 < n \leq \nu$
Ensure: $\mathbf{c}_A = \mathsf{GetCommitmentMatrix}(A, \mathbf{r}, \mathbf{ck})$ ▷ See algorithm 5.8
Ensure: $c_b = \mathsf{GetCommitment}(\mathbf{b}, s, \mathbf{ck})$ ▷ See algorithm 5.7
Ensure: $\vec{b} = \prod_{j=0}^{m-1} \vec{a}_j$ ▷ Uses the Hadamard product, ie $b_i = \prod_{j=0}^{m-1} a_{i,j}$

Operation:

```
for j ∈ [0, m) do
    b⃗_j ← ∏_{i=0}^{j} a⃗_i                          ▷ Which implies that b⃗_0 = a⃗_0 and b⃗_{m−1} = b⃗
end for
s_0 ← r_0                                         ▷ Thus ensuring that GetCommitment(b⃗_0, s_0, ck) = c_{A_0}
if m > 2 then
    (s_1, ..., s_{m−2}) ← GenRandomVector(q, m − 2)
end if                                            ▷ See algorithm 3.2
s_{m−1} ← s                                       ▷ Thus ensuring that GetCommitment(b⃗_{m−1}, s_{m−1}, ck) = c_b
c_{B_0} ← c_{A_0}
for j ∈ [1, m − 1) do
    c_{B_j} ← GetCommitment(b⃗_j, s_j, ck)
end for
c_{B_{m−1}} ← c_b
x ← ByteArrayToInteger(RecursiveHash(p, q, pk, ck, c_A, c_b, (c_{B_0}, ..., c_{B_{m−1}})))
y ← ByteArrayToInteger(RecursiveHash("1", p, q, pk, ck, c_A, c_b, (c_{B_0}, ..., c_{B_{m−1}})))
                                                  ▷ Use y to define ⋆ : Z_q^n × Z_q^n ↦ Z_q, see algorithm 5.10
                                                  ▷ All exponentiations of x below are performed modulo q
for i ∈ [0, m − 1) do
    d⃗_i = x^{i+1} b⃗_i
    c_{D_i} = c_{B_i}^{x^{i+1}}
    t_i = x^{i+1} s_i
end for
d⃗ ← ∑_{i=1}^{m−1} x^i b⃗_i
c_D ← ∏_{i=1}^{m−1} c_{B_i}^{x^i}
t ← ∑_{i=1}^{m−1} x^i s_i
−1⃗ ← (q − 1, ..., q − 1) ∈ Z_q^n
c_{−1} ← GetCommitment(−1⃗, 0, ck)
statement ← ((c_{A_1}, ..., c_{A_{m−1}}, c_{−1}), (c_{D_0}, ..., c_{D_{m−2}}, c_D), y)
witness ← ((a⃗_1, ..., a⃗_{m−1}, −1⃗), (d⃗_0, ..., d⃗_{m−2}, d⃗), (r_1, ..., r_{m−1}, 0), (t_0, ..., t_{m−2}, t))
zeroArg ← GetZeroArgument(statement, witness)     ▷ See algorithm 5.22
                                                  ▷ Provide an argument that ∑_{i=0}^{m−2} a⃗_{i+1} ⋆ d⃗_i − 1⃗ ⋆ d⃗ = 0
```

Output:
$((c_{B_0}, \dots, c_{B_{m-1}}), \mathsf{zeroArg}) \in {\mathbb{G}_q}^m \times (\mathbb{G}_q \times \mathbb{G}_q \times {\mathbb{G}_q}^{2m+1} \times {\mathbb{Z}_q}^n \times {\mathbb{Z}_q}^n \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q)$

Algorithm 5.21 VerifyHadamardArgument: Verifies a Hadamard Argument

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of
- commitment $\mathbf{c}_A = (c_{A_0}, \ldots, c_{A_{m-1}}) \in {\mathbb{G}_q}^m$
- commitment $c_b \in \mathbb{G}_q$

The argument composed of
- commitment vector $\mathbf{c}_B = (c_{B_0}, \ldots, c_{B_{m-1}}) \in {\mathbb{G}_q}^m$
- a zero argument, composed of
  - commitment $c_{A_0} \in \mathbb{G}_q$
  - commitment $c_{B_m} \in \mathbb{G}_q$
  - commitment vector $\mathbf{c}_d \in {\mathbb{G}_q}^{2m+1}$
  - exponent vector $\mathbf{a}' \in {\mathbb{Z}_q}^n$
  - exponent vector $\mathbf{b}' \in {\mathbb{Z}_q}^n$
  - exponent $r' \in \mathbb{Z}_q$
  - exponent $s' \in \mathbb{Z}_q$
  - exponent $t' \in \mathbb{Z}_q$

Ensure: $n > 0$

Operation:

```
1: x ← ByteArrayToInteger(RecursiveHash(p, q, pk, ck, c_A, c_b, (c_{B_0}, ..., c_{B_{m-1}})))
2: y ← ByteArrayToInteger(RecursiveHash("1", p, q, pk, ck, c_A, c_b, (c_{B_0}, ..., c_{B_{m-1}})))
3: for i ∈ [0, m − 1) do
4:     c_{D_i} ← c_{B_i}^{x^{i+1}}
5: end for
6: c_D ← ∏_{i=1}^{m−1} c_{B_i}^{x^i}
7: −1⃗ ← (q − 1, ..., q − 1) ∈ Z_q^n
8: c_{−1} ← GetCommitment(−1⃗, 0, ck)
9: zeroStatement ← ((c_{A_1}, ..., c_{A_{m−1}}, c_{−1}), (c_{D_0}, ..., c_{D_{m−2}}, c_D), y)
10: zeroArgument ← (c_{A_0}, c_{B_m}, c_d, a', b', r', s', t')
11: if c_{B_0} = c_{A_0} ∧ c_{B_{m−1}} = c_b ∧ VerifyZeroArgument(zeroStatement, zeroArgument) then
                                                              ▷ See algorithm 5.23
        return ⊤
12: else
        return ⊥
13: end if
```

Output:

The result of the verification: ⊤ if the verification is successful, ⊥ otherwise.

Test values for the algorithm 5.21 are provided in the attached verify-h-argument.json file.

5.3.5 Zero Argument

In the following algorithm, we generate an argument of knowledge of the values $\mathbf{a}_1, \mathbf{b}_0, \ldots, \mathbf{a}_m, \mathbf{b}_{m-1}$ such that $\sum_{i=1}^{m} \mathbf{a}_i \star \mathbf{b}_{i-1} = 0$.

Algorithm 5.22 GetZeroArgument: Computes a Zero Argument

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \ldots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of
- commitments $\mathbf{c}_A \in {\mathbb{G}_q}^m$
- commitments $\mathbf{c}_B \in {\mathbb{G}_q}^m$
- the value $y \in \mathbb{Z}_q$ defining the bilinear mapping $\star$ ▷ See algorithm 5.10

The witness composed of
- matrix $A = (\vec{a}_1, \ldots, \vec{a}_m) \in {\mathbb{Z}_q}^{n \times m}$ ▷ The $\vec{a}_i$ values correspond to the columns of A
- matrix $B = (\vec{b}_0, \ldots, \vec{b}_{m-1}) \in {\mathbb{Z}_q}^{n \times m}$ ▷ The $\vec{b}_i$ values correspond to the columns of B
- vector of exponents $\mathbf{r} = (r_1, \ldots, r_m) \in {\mathbb{Z}_q}^m$
- vector of exponents $\mathbf{s} = (s_0, \ldots, s_{m-1}) \in {\mathbb{Z}_q}^m$

Ensure: $\mathbf{c}_A = \mathsf{GetCommitmentMatrix}(A, \mathbf{r}, \mathbf{ck})$ ▷ See algorithm 5.8
Ensure: $\mathbf{c}_B = \mathsf{GetCommitmentMatrix}(B, \mathbf{s}, \mathbf{ck})$ ▷ See algorithm 5.8
Ensure: $\sum_{i=1}^{m} \mathbf{a}_i \star \mathbf{b}_{i-1} = 0$
Ensure: $n, m > 0$

Operation:

$\vec{a}_0 \leftarrow \mathsf{GenRandomVector}(q, n)$ ▷ See algorithm 3.2
$\vec{b}_m \leftarrow \mathsf{GenRandomVector}(q, n)$
$r_0 \leftarrow \mathsf{GenRandomInteger}(q)$
$s_m \leftarrow \mathsf{GenRandomInteger}(q)$
$c_{A_0} \leftarrow \mathsf{GetCommitment}(\mathbf{a}_0, r_0, \mathbf{ck})$
$c_{B_m} \leftarrow \mathsf{GetCommitment}(\mathbf{b}_m, s_m, \mathbf{ck})$
$\mathbf{d} = (d_0, \ldots, d_{2m}) \leftarrow \mathsf{ComputeDVector}((\vec{a}_0, \ldots, \vec{a}_m), (\vec{b}_0, \ldots, \vec{b}_m), y)$
▷ See algorithm 5.24
$\mathbf{t} \leftarrow \mathsf{GenRandomVector}(q, 2m + 1)$
$t_{m+1} \leftarrow 0$
$\mathbf{c_d} \leftarrow \mathsf{GetCommitmentVector}((d_0, \ldots, d_{2m}), \mathbf{t}, \mathbf{ck})$
▷ See algorithm 5.9
$x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, c_{A_0}, c_{B_m}, \mathbf{c_d}, \mathbf{c}_B, \mathbf{c}_A))$
▷ See algorithm 2.7 and algorithm 3.6
▷ Below this point, all operations are performed modulo q
for $j \in [0, n)$ **do**
  $\mathbf{a'}_j \leftarrow \sum_{i=0}^{m} x^i \cdot \vec{a}_{j,i}$
  $\mathbf{b'}_j \leftarrow \sum_{i=0}^{m} x^{m-i} \cdot \vec{b}_{j,i}$
end for
$r' \leftarrow \sum_{i=0}^{m} x^i \cdot r_i$
$s' \leftarrow \sum_{i=0}^{m} x^{m-i} \cdot s_i$
$t' \leftarrow \sum_{i=0}^{2m} x^i \cdot t_i$

Output:

The argument $(c_{A_0}, c_{B_m}, \mathbf{c_d}, \mathbf{a'}, \mathbf{b'}, r', s', t') \in \mathbb{G}_q \times \mathbb{G}_q \times {\mathbb{G}_q}^{2m+1} \times {\mathbb{Z}_q}^n \times {\mathbb{Z}_q}^n \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q$

In the following algorithm, we verify if a provided zero argument supports the corresponding statement. We conform to the convention of using the symbol $\top$ for true and $\perp$ for false.

Algorithm 5.23 VerifyZeroArgument: Verifies a Zero Argument

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \dots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of
- commitments $\mathbf{c}_A = (c_{A_1}, \dots, c_{A_m}) \in {\mathbb{G}_q}^m$
- commitments $\mathbf{c}_B = (c_{B_0}, \dots, c_{B_{m-1}}) \in {\mathbb{G}_q}^m$
- the value $y \in \mathbb{Z}_q$ defining the bilinear mapping $\star$ ▷ See algorithm 5.10

The argument composed of
- the commitment $c_{A_0} \in \mathbb{G}_q$
- the commitment $c_{B_m} \in \mathbb{G}_q$
- the commitment vector $\mathbf{c}_d = (c_{d_0}, \dots, c_{d_{2m}}) \in {\mathbb{G}_q}^{2m+1}$
- the exponent vector $\mathbf{a}' \in {\mathbb{Z}_q}^n$
- the exponent vector $\mathbf{b}' \in {\mathbb{Z}_q}^n$
- the exponent $r' \in \mathbb{Z}_q$
- the exponent $s' \in \mathbb{Z}_q$
- the exponent $t' \in \mathbb{Z}_q$

Operation:

1: $x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, c_{A_0}, c_{B_m}, \mathbf{c_d}, \mathbf{c}_B, \mathbf{c}_A))$
2: $\mathsf{verifCd} \leftarrow c_{d_{m+1}} = 1$
3: $\mathsf{prodCa} \leftarrow \prod_{i=0}^{m} c_{A_i}^{x^i}$ ▷ The exponentiations of x are computed modulo q, whereas the product and the exponentiations of commitments are computed modulo p
4: $\mathsf{commA} \leftarrow \mathsf{GetCommitment}(\mathbf{a}', r', \mathbf{ck})$
5: $\mathsf{verifA} \leftarrow \mathsf{prodCa} = \mathsf{commA}$
6: $\mathsf{prodCb} \leftarrow \prod_{i=0}^{m} c_{B_{m-i}}^{x^i}$ ▷ The exponentiations of x are computed modulo q, whereas the product and the exponentiations of commitments are computed modulo p
7: $\mathsf{commB} \leftarrow \mathsf{GetCommitment}(\mathbf{b}', s', \mathbf{ck})$
8: $\mathsf{verifB} \leftarrow \mathsf{prodCb} = \mathsf{commB}$
9: $\mathsf{prodCd} \leftarrow \prod_{i=0}^{2m} c_{d_i}^{x^i}$ ▷ The exponentiations of x are computed modulo q, whereas the product and the exponentiations of commitments are computed modulo p
10: $\mathsf{prod} \leftarrow \mathbf{a}' \star \mathbf{b}'$ ▷ Using algorithm 5.10 with value y
11: $\mathsf{commD} \leftarrow \mathsf{GetCommitment}((\mathsf{prod}), t', \mathbf{ck})$
12: $\mathsf{verifD} \leftarrow \mathsf{prodCd} = \mathsf{commD}$
13: **if** $\mathsf{verifCd} \land \mathsf{verifA} \land \mathsf{verifB} \land \mathsf{verifD}$ **then**
    return $\top$
14: **else**
    return $\bot$
15: **end if**

Output:

The result of the verification: $\top$ if the verification is successful, $\bot$ otherwise.

Test values for the algorithm 5.23 are provided in the attached verify-za-argument.json file.

Algorithm 5.24 ComputeDVector: Compute the vector $\mathbf{d}$ for the algorithm 5.22

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

First matrix $A = (\vec{a}_0, \ldots, \vec{a}_m) \in {\mathbb{Z}_q}^{n \times (m+1)}$
Second matrix $B = (\vec{b}_0, \ldots, \vec{b}_m) \in {\mathbb{Z}_q}^{n \times (m+1)}$
Value $y \in \mathbb{Z}_q$

Operation:

```
for k ∈ [0, 2m] do
    d_k ← 0
    for i ∈ [max(0, k − m), m] do
        j ← (m − k) + i
        if j > m then
            break from loop and proceed with next k
        end if
        d_k ← d_k + a⃗_i ⋆ b⃗_j          ▷ See algorithm 5.10, addition is modulo q
    end for
end for
```

Output:

$\mathbf{d} = (d_0, \ldots, d_{2m}) \in {\mathbb{Z}_q}^{2m+1}$

5.3.6 Single Value Product Argument

In the following algorithm we generate an argument of knowledge of the opening $(\mathbf{a}, r)$ where $\mathbf{a} = (a_0, \dots, a_{n-1})$ s.t. $c_a = \mathsf{GenCommitment}(\mathbf{a}, r)$ and $b = \prod_{i=0}^{n-1} a_i \mod q$.

Algorithm 5.25 GetSingleValueProductArgument: Computes a Single Value Product Argument

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \dots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:
The statement composed of
- commitment $c_a \in \mathbb{G}_q$
- the product $b \in \mathbb{Z}_q$
The witness composed of
- vector $\mathbf{a} = (a_0, \dots, a_{n-1}) \in {\mathbb{Z}_q}^n$
- the randomness $r \in \mathbb{Z}_q$

Ensure: $n \geq 2$
Ensure: $c_a = \mathsf{GetCommitment}(\mathbf{a}, r, \mathbf{ck})$ ▷ See algorithm 5.7
Ensure: $b = \prod_{i=0}^{n-1} a_i \mod q$

Operation:
for $k \in [0, n)$ **do**
  $b_k \leftarrow \prod_{i=0}^{k} a_i \mod q$
end for
$(d_0, \dots, d_{n-1}) \leftarrow \mathsf{GenRandomVector}(q, n)$ ▷ See algorithm 3.2
$r_d \leftarrow \mathsf{GenRandomInteger}(q)$
$\delta_0 \leftarrow d_0$
if $n > 2$ **then**
  $(\delta_1, \dots, \delta_{n-2}) \leftarrow \mathsf{GenRandomVector}(q, n-2)$
end if
$\delta_{n-1} \leftarrow 0$
$s_0 \leftarrow \mathsf{GenRandomInteger}(q)$
$s_x \leftarrow \mathsf{GenRandomInteger}(q)$
for $k \in [0, n-1)$ **do**
  $\delta'_k \leftarrow -\delta_k d_{k+1} \mod q$
  $\Delta_k \leftarrow \delta_{k+1} - a_{k+1}\delta_k - b_k d_{k+1} \mod q$
end for
$c_d \leftarrow \mathsf{GetCommitment}((d_0, \dots, d_{n-1}), r_d, \mathbf{ck})$ ▷ See algorithm 5.7
$c_\delta \leftarrow \mathsf{GetCommitment}((\delta'_0, \dots, \delta'_{n-2}), s_0, \mathbf{ck})$
$c_\Delta \leftarrow \mathsf{GetCommitment}((\Delta_0, \dots, \Delta_{n-2}), s_x, \mathbf{ck})$
$x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, c_\Delta, c_\delta, c_d, b, c_a))$
for $k \in [0, n)$ **do**
  $\tilde{a}_k \leftarrow x a_k + d_k \mod q$
  $\tilde{b}_k \leftarrow x b_k + \delta_k \mod q$
end for
$\tilde{r} \leftarrow x r + r_d \mod q$
$\tilde{s} \leftarrow x s_x + s_0 \mod q$

Output:
$(c_d, c_\delta, c_\Delta, (\tilde{a}_0, \dots, \tilde{a}_{n-1}), (\tilde{b}_0, \dots, \tilde{b}_{n-1}), \tilde{r}, \tilde{s}) \in \mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q \times {\mathbb{Z}_q}^n \times {\mathbb{Z}_q}^n \times \mathbb{Z}_q \times \mathbb{Z}_q$

In the following pseudo-code algorithm, we verify if a provided Single Value Product argument supports the corresponding statement.

Algorithm 5.26 VerifySingleValueProductArgument: Verifies a Single Value Product Argument

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, \dots, g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of
- commitment $c_a \in \mathbb{G}_q$
- the product $b \in \mathbb{Z}_q$

The argument composed of
- the commitment $c_d \in \mathbb{G}_q$
- the commitment $c_\delta \in \mathbb{G}_q$
- the commitment $c_\Delta \in \mathbb{G}_q$
- the exponent vector $\tilde{a} = (\tilde{a}_0, \dots, \tilde{a}_{n-1}) \in \mathbb{Z}_q{}^n, n \geq 2$
- the exponent vector $\tilde{b} = (\tilde{b}_0, \dots, \tilde{b}_{n-1}) \in \mathbb{Z}_q{}^n$
- the exponent $\tilde{r} \in \mathbb{Z}_q$
- the exponent $\tilde{s} \in \mathbb{Z}_q$

Operation:

```
x ← ByteArrayToInteger(RecursiveHash(p, q, pk, ck, c_Δ, c_δ, c_d, b, c_a))
prodCa ← c_a^x c_d
commA ← GetCommitment(ã, r̃, ck)
verifA ← prodCa = commA
prodDelta ← c_Δ^x c_δ
for i ∈ [0, n − 1) do
    e_i ← x b̃_{i+1} − b̃_i ã_{i+1}
end for
commDelta ← GetCommitment((e_0, ..., e_{n−2}), s̃, ck)
verifDelta ← prodDelta = commDelta
verifB ← b̃_0 = ã_0 ∧ b̃_{n−1} = xb
if verifA ∧ verifDelta ∧ verifB then
    return ⊤
else
    return ⊥
end if
```

Output:

The result of the verification: $\top$ if the verification is successful, $\bot$ otherwise.

Test values for the algorithm 5.26 are provided in the attached verify-svp-argument.json file.

6 Zero-Knowledge Proofs

6.1 Introduction

This section introduces various Zero-Knowledge Proofs of Knowledge, based on the work by Maurer[18]. We extensively document and formalize the zero-knowledge proof system's security—including the non-interactive case—in the computational proof [20]. In each case, the idea is to make a statement, consisting of an homomorphism $\phi : \mathbb{G}_1 \mapsto \mathbb{G}_2$ and an image y and provide a Zero-Knowledge Proof of the Pre-image w such that $y = \phi(w)$. We name that pre-image the witness.

While such proofs are usually interactive, we rely on the Fiat-Shamir transform to turn them non-interactive. We use the hash function described in algorithm 3.6. The proof consists of the following steps:

- draw $b \in \mathbb{G}_1$ at random
- compute $c = \phi(b)$
- compute $e = \mathsf{RecursiveHash}(\phi, y, c, \mathsf{auxiliaryData})$
- compute $z = b \star w^e$ (where $\star$ is the group operation for $\mathbb{G}_1$, and exponentiation is the repetition of that operation)
- output $\pi = (e, z)$

The verification can be summarized as:

- compute $x = \phi(z)$
- compute $c' = x \otimes y^{-e}$ (where $\otimes$ is the group operation for $\mathbb{G}_2$, and exponentiation is the repetition of that operation)
- if and only if $\mathsf{RecursiveHash}(\phi, y, c', \mathsf{auxiliaryData}) = e$, the proof is valid

Each type of proof is a specialization of the generic prove and verify algorithm.

6.2 Decryption Proof

We prove that a decryption matches the message encrypted under the advertised public key. In this case, the phi-function maps our witness—the private key —to the public key and the decryption of the ciphertext. Hence, we define the phi-function as shown in algorithm 6.1.

Algorithm 6.1 ComputePhiDecryption: Compute the ϕ-function for decryption

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

Preimage $(x_0, \dots, x_{l-1}) \in {\mathbb{Z}_q}^l$
Base $\gamma \in \mathbb{G}_q$

Operation:

```
1: for i ∈ [0, l) do
2:     y_i ← g^{x_i}                    ▷ y_i = pk_i when x_i = sk_i
3:     y_{l+i} ← γ^{x_i}                ▷ y_{l+i} = g^{sk_i·r} = φ_i/m_i when γ = g^r and x_i = sk_i
4: end for
                                         ▷ All symbols used in the comments above are aligned with algorithms 4.2 and 4.5
```

Output:

The image $(y_0, \dots, y_{2l-1}) \in {\mathbb{G}_q}^{2l}$

This algorithm implies that for the multi-recipient ElGamal key pair $(\mathbf{pk}, \mathbf{sk})$ and the valid decryption $m = (m_0, \dots, m_{l-1})$ of the ciphertext $(\gamma, \phi_0, \dots, \phi_{l-1})$, the computation of the $\mathsf{ComputePhiDecryption}(\mathbf{sk}, \gamma)$ would yield $(\mathsf{pk}_0, \dots, \mathsf{pk}_{l-1}, \frac{\phi_0}{m_0}, \dots, \frac{\phi_{l-1}}{m_{l-1}})$.

Generating and verifying decryption proofs The algorithms below are the adaptations of the general case presented in section 6.1, with explicit domains and operations. Our ϕ-function defined in algorithm 6.1 has domain $(\mathbb{Z}_q{}^l, +)$ and co-domain $(\mathbb{G}_q{}^{2l}, \times)$. Therefore the operations given as $\star$ will be replaced with addition (modulo q), and the "exponentiation" used in the computation of z is actually a multiplication; whereas the operation denoted by $\otimes$ is multiplication (modulo p) and the exponentiation used in the computation of c' is a modular exponentiation in $\mathbb{G}_q$.

Algorithm 6.2 $\mathsf{GenDecryptionProof}$: Generate a proof of validity for the provided decryption

---

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A multi-recipient ciphertext $\mathbf{C} = (\gamma, \phi_0, \ldots, \phi_{l-1}) \in \mathbb{H}_l$
A multi-recipient key pair $(\mathbf{pk}, \mathbf{sk}) \in \mathbb{G}_q^k \times \mathbb{Z}_q^k$
A multi-recipient message $\mathbf{m} = (m_0, \ldots, m_{l-1}) \in \mathbb{G}_q{}^l$ s.t. $\mathbf{m} = \mathsf{GetMessage}(\mathbf{C}, \mathbf{sk})$
An array of optional additional information $\mathbf{i}_{\mathsf{aux}} \in (\mathbb{A}_{UCS}{}^*)^{\bar{*}}$

Require: $0 < l \leq k$

Operation:

$\mathbf{b} \leftarrow \mathsf{GenRandomVector}(q, l)$ ▷ See algorithm 3.2
$\mathbf{c} \leftarrow \mathsf{ComputePhiDecryption}(\mathbf{b}, \gamma)$ ▷ See algorithm 6.1
$\mathbf{f} \leftarrow (p, q, g, \gamma)$
$(\mathsf{pk}'_0, \ldots, \mathsf{pk}'_{l-1}) \leftarrow \mathsf{CompressPublicKey}(\mathbf{pk}, l)$ ▷ See algorithm 4.3
for $i \in [0, l)$ **do**
  $y_i \leftarrow \mathsf{pk}'_i$
  $y_{l+i} \leftarrow \frac{\phi_i}{m_i}$
end for
$\mathbf{h}_{\mathsf{aux}} \leftarrow (\text{"}\mathtt{DecryptionProof}\text{"}, (\phi_0, \ldots, \phi_{l-1}), \mathbf{m}, \mathbf{i}_{\mathsf{aux}})$ ▷ If $\mathbf{i}_{\mathsf{aux}}$ is empty, we omit it
$e \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{f}, (y_0, \ldots, y_{2l-1}), \mathbf{c}, \mathbf{h}_{\mathsf{aux}}))$ ▷ See algorithms 2.7 and 3.6
$\mathbf{sk}' \leftarrow \mathsf{CompressSecretKey}(\mathbf{sk}, l)$ ▷ See algorithm 4.4
$\mathbf{z} \leftarrow \mathbf{b} + e \cdot \mathbf{sk}'$

Output:

Proof $(e, \mathbf{z}) \in \mathbb{Z}_q \times \mathbb{Z}_q{}^l$

Algorithm 6.3 VerifyDecryption: Verifies the validity of a decryption proof

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A multi-recipient ciphertext $C = (\gamma, \phi_0, \dots, \phi_{l-1}) \in \mathbb{H}_l$
A multi-recipient public key $\mathbf{pk} = (\mathsf{pk}_0, \dots, \mathsf{pk}_{k-1}) \in \mathbb{G}_q^k$
A multi-recipient message $\mathbf{m} = (m_0, \dots, m_{l-1}) \in \mathbb{G}_q{}^l$ ▷ We expect $\mathbf{m} = \mathsf{GetMessage}(\mathbf{C}, \mathbf{sk})$
The proof $(e, \mathbf{z}) \in \mathbb{Z}_q \times \mathbb{Z}_q{}^l$
An array of optional additional information $\mathbf{i}_{\mathsf{aux}} \in (\mathbb{A}_{UCS}{}^*)^{\bar{*}}$

Require: $0 < l \leq k$

Operation:

```
x ← ComputePhiDecryption(z, γ)                        ▷ See algorithm 6.1
f ← (p, q, g, γ)
(pk'_0, ..., pk'_{l-1}) ← CompressPublicKey(pk, l)   ▷ See algorithm 4.3
for i ∈ [0, l) do
    y_i ← pk'_i
    y_{l+i} ← φ_i / m_i
end for
for i ∈ [0, 2l) do
    c'_i ← x_i y_i^{-e}
end for
h_aux ← ("DecryptionProof", (φ_0, ..., φ_{l-1}), m, i_aux)   ▷ If i_aux is empty, we omit it
h ← RecursiveHash(f, (y_0, ..., y_{2l-1}), (c'_0, ..., c'_{2l-1}), h_aux)   ▷ See algorithm 3.6
e' ← ByteArrayToInteger(h)                            ▷ See algorithm 2.7
if e = e' then
    return ⊤
else
    return ⊥
end if
```

Output:

The result of the verification: ⊤ if the verification is successful, ⊥ otherwise.

Test values for the algorithm 6.3 are provided in the attached verify-decryption.json file.

6.3 Exponentiation proof

We prove that the same secret exponent is used for a vector of exponentiations. In this case, the phi-function maps our witness—the secret exponent—to the exponentiation of a given vector of bases. We define the phi-function as shown in algorithm 6.4.

Algorithm 6.4 ComputePhiExponentiation: compute the ϕ-function for exponentiation

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Input:

Preimage $x \in \mathbb{Z}_q$
Bases $(g_0, \dots, g_{n-1}) \in \mathbb{G}_q{}^n$ s.t. $n \in \mathbb{N}^*$

Operation:

```
for i ∈ [0, n) do
    y_i ← g_i^x mod p
end for
return (y_0, ..., y_{n-1})
```

Output:

$\mathbf{y} = (y_0, \dots, y_{n-1}) \in \mathbb{G}_q{}^n$

Generating and verifying exponentiation proofs The algorithms below are the adaptations of the general case presented in section 6.1, with explicit domains and operations. Our ϕ-function defined in algorithm 6.4 has domain $(\mathbb{Z}_q, +)$ and co-domain $(\mathbb{G}_q{}^n, \times)$. Therefore the operations given as $\star$ will be replaced with addition (modulo q), and the "exponentiation" used in the computation of z is a multiplication; whereas the operation denoted by $\otimes$ is multiplication (modulo p) and the exponentiation used in the computation of c' is a modular exponentiation in $\mathbb{G}_q$.

Algorithm 6.5 $\mathsf{GenExponentiationProof}$: Generate a proof of validity for the provided exponentiation

---

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Input:

A vector of bases $\mathbf{g} = (g_0, \dots, g_{n-1}) \in \mathbb{G}_q{}^n$ s.t. $n \in \mathbb{N}^*$
The witness – a secret exponent $x \in \mathbb{Z}_q$
The statement – a vector of exponentiations $\mathbf{y} = (y_0, \dots, y_{n-1}) \in \mathbb{G}_q{}^n$ s.t. $y_i = g_i^x$
An array of optional additional information $\mathbf{i}_{\mathsf{aux}} \in (\mathbb{A}_{UCS}{}^*)^{\bar{*}}$

Operation:

1: $b \leftarrow \mathsf{GenRandomInteger}(q)$ ▷ See algorithm 3.1
2: $\mathbf{c} \leftarrow \mathsf{ComputePhiExponentiation}(b, \mathbf{g})$ ▷ See algorithm 6.4
3: $\mathbf{f} \leftarrow (p, q, \mathbf{g})$
4: $\mathbf{h}_{\mathsf{aux}} \leftarrow (\text{"}\mathtt{ExponentiationProof}\text{"}, \mathbf{i}_{\mathsf{aux}})$ ▷ If $\mathbf{i}_{\mathsf{aux}}$ is empty, we omit it
5: $e \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{f}, \mathbf{y}, \mathbf{c}, \mathbf{h}_{\mathsf{aux}}))$ ▷ See algorithms 2.7 and 3.6
6: $z \leftarrow b + ex$

Output:

Proof $(e, z) \in \mathbb{Z}_q \times \mathbb{Z}_q$

Algorithm 6.6 VerifyExponentiation: Verifies the validity of an exponentiation proof

Context:

Group modulus $p \in \mathbb{P}$

Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$

Input:

A vector of bases $\mathbf{g} = (g_0, \dots, g_{n-1}) \in \mathbb{G}_q{}^n$ s.t. $n \in \mathbb{N}^*$

The statement – a vector of exponentiations $\mathbf{y} = (y_0, \dots, y_{n-1}) \in \mathbb{G}_q{}^n$

The proof $(e, z) \in \mathbb{Z}_q \times \mathbb{Z}_q$

An array of optional additional information $\mathbf{i}_{\mathsf{aux}} \in (\mathbb{A}_{UCS}{}^*)^{\bar{*}}$

Operation:

$\mathbf{x} \leftarrow \mathsf{ComputePhiExponentiation}(z, \mathbf{g})$ ▷ See algorithm 6.4
$\mathbf{f} \leftarrow (p, q, \mathbf{g})$
for $i \in [0, n)$ **do**
  $c'_i \leftarrow x_i y_i^{-e}$
end for
$\mathbf{h}_{\mathsf{aux}} \leftarrow (\text{"}\texttt{ExponentiationProof}\text{"}, \mathbf{i}_{\mathsf{aux}})$ ▷ If $\mathbf{i}_{\mathsf{aux}}$ is empty, we omit it
$h \leftarrow \mathsf{RecursiveHash}(\mathbf{f}, \mathbf{y}, (c'_0, \dots, c'_{n-1}), \mathbf{h}_{\mathsf{aux}})$ ▷ See algorithm 3.6
$e' \leftarrow \mathsf{ByteArrayToInteger}(h)$ ▷ See algorithm 2.7
if $e = e'$ **then**
  return $\top$
else
  return $\perp$
end if

Output:

The result of the verification: $\top$ if the verification is successful, $\perp$ otherwise.

Test values for the algorithm 6.6 are provided in the attached verify-exponentiation.json file.

6.4 Plaintext equality proof

We prove that two encryptions under different keys correspond to the same plaintext. The ciphertexts are written as $\mathbf{c} = (c_0, c_1) = (g^r, h^r m)$ and $\mathbf{c}' = (c'_0, c'_1) = (g^{r'}, h'^{r'} m)$, where g is the generator , h and h' are the public keys, and m is the same message in both cases. In this case, the phi-function is defined by the primes p and q, defining $\mathbb{G}_q$, as well as the generator g and the public keys h and h', as follows:

$$\phi_{\mathsf{PlaintextEquality}} : {\mathbb{Z}_q}^2 \mapsto {\mathbb{G}_q}^3$$
$$\phi_{\mathsf{PlaintextEquality}}(x, x') = (g^x, g^{x'}, \frac{h^x}{h'^{x'}})$$

This implies that $\phi_{\mathsf{PlaintextEquality}}(r, r') = (c_0, c'_0, \frac{c_1}{c'_1})$, if and only if the message is the same in both encryptions.

Algorithm 6.7 ComputePhiPlaintextEquality: Compute the ϕ-function for plaintext equality

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

Preimage $(x, x') \in {\mathbb{Z}_q}^2$
First public key $h \in \mathbb{G}_q$
Second public key $h' \in \mathbb{G}_q$

Operation:

1: **return** $(g^x, g^{x'}, \frac{h^x}{h'^{x'}})$ ▷ All exponentiations performed modulo p

Output:

The image $(g^x, g^{x'}, \frac{h^x}{h'^{x'}}) \in {\mathbb{G}_q}^3$

Generating and verifying plaintext equality proofs The algorithms below are the adaptations of the general case presented in section 6.1, with explicit domains and operations. Our ϕ-function defined in algorithm 6.7 has domain $({\mathbb{Z}_q}^2, +)$ and co-domain $({\mathbb{G}_q}^3, \times)$. Therefore the operations given as $\star$ will be replaced with addition (modulo q), and the "exponentiation" used in the computation of z is a multiplication; whereas the operation denoted by $\otimes$ is multiplication (modulo p) and the exponentiation used in the computation of c' is a modular exponentiation in $\mathbb{G}_q$.

Algorithm 6.8 $\mathsf{GenPlaintextEqualityProof}$: Generate a proof of equality of the plaintext corresponding to the two provided encryptions

---

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

The first ciphertext $\mathbf{C} = (c_0, c_1) \in {\mathbb{G}_q}^2$
The second ciphertext $\mathbf{C}' = (c'_0, c'_1) \in {\mathbb{G}_q}^2$
The first public key $h \in \mathbb{G}_q$
The second public key $h' \in \mathbb{G}_q$
The witness—the randomness used in the encryptions— $(r, r') \in {\mathbb{Z}_q}^2$
An array of optional additional information $\mathbf{i}_{\mathsf{aux}} \in ({\mathbb{A}_{UCS}}^*)^{\bar{*}}$

Require: $0 < l \leq k$

Operation:

1: $\mathbf{b} \leftarrow \mathsf{GenRandomVector}(q, 2)$ ▷ See algorithm 3.2
2: $\mathbf{c} \leftarrow \mathsf{ComputePhiPlaintextEquality}(\mathbf{b}, h, h')$ ▷ See algorithm 6.7
3: $\mathbf{f} \leftarrow (p, q, g, h, h')$
4: $\mathbf{y} \leftarrow (c_0, c'_0, \frac{c_1}{c'_1})$
5: $\mathbf{h}_{\mathsf{aux}} \leftarrow (\text{"}\mathtt{PlaintextEqualityProof}\text{"}, c_1, c'_1, \mathbf{i}_{\mathsf{aux}})$ ▷ If $\mathbf{i}_{\mathsf{aux}}$ is empty, we omit it
6: $e \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{f}, \mathbf{y}, \mathbf{c}, \mathbf{h}_{\mathsf{aux}}))$ ▷ See algorithms 2.7 and 3.6
7: $\mathbf{z} \leftarrow \mathbf{b} + e \cdot (r, r')$

Output:

Proof $(e, \mathbf{z}) \in \mathbb{Z}_q \times {\mathbb{Z}_q}^2$

Algorithm 6.9 VerifyPlaintextEquality: Verifies the validity of a plaintext equality proof

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

The first ciphertext $\mathbf{C} = (c_0, c_1) \in {\mathbb{G}_q}^2$
The second ciphertext $\mathbf{C}' = (c'_0, c'_1) \in {\mathbb{G}_q}^2$
The first public key $h \in \mathbb{G}_q$
The second public key $h' \in \mathbb{G}_q$
The proof $(e, \mathbf{z}) \in \mathbb{Z}_q \times {\mathbb{Z}_q}^2$
An array of optional additional information $\mathbf{i}_{\mathsf{aux}} \in ({\mathbb{A}_{UCS}}^*)^{\bar{*}}$

Operation:

```
x ← ComputePhiPlaintextEquality(z, h, h')          ▷ See algorithm 6.7
f ← (p, q, g, h, h')
y ← (c_0, c'_0, c_1/c'_1)
c' ← x · y^{-e}
h_aux ← ("PlaintextEqualityProof", c_1, c'_1, i_aux)   ▷ If i_aux is empty, we omit it
e ← ByteArrayToInteger(RecursiveHash(f, y, c', h_aux)) ▷ See algorithms 2.7 and 3.6
if e = e' then
    return ⊤
else
    return ⊥
end if
```

Output:

The result of the verification: $\top$ if the verification is successful, $\bot$ otherwise.

Test values for the algorithm 6.9 are provided in the attached verify-plaintext-equality.json file.

Acknowledgements

We want to thank the BFH E-Voting Group for their review of an earlier version of the document and Pascal Junod for many valuable remarks submitted on our Gitlab platform.

References

[1] ANSSI (Agence nationale de la sécurité des systèmes d'informations). Référentiel Général de Sécurité, version 2.0, Annexe B1, Mécanismes cryptographiques, 2014.

[2] Elaine Barker. NIST Special Publication 800-57 Part 1, Revision 5, Recommendation for Key Management. Special Publication 800-57, National Institute of Standards and Technology, U.S. Department of Commerce, 2020.

[3] Stephanie Bayer and Jens Groth. Efficient Zero-Knowledge Argument for Correctness of a Shuffle. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 263–280, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[4] Swiss Federal Chancellery. Federal Chancellery Ordinance on Electronic Voting (VEleS). `https://www.admin.ch/opc/en/classified-compilation/20132343/index.html`, July 2018.

[5] Swiss Federal Chancellery. Technical and administrative requirements for electronic vote casting. Annex of the Federal Chancellery Ordinance on Electronic Voting 2.0. `https://www.admin.ch/opc/en/classified-compilation/20132343/index.html#app1`, July 2018.

[6] Morris Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions, 2015-08-04 2015.

[7] Eidgenössisches Departement für auswärtige Angelegenheiten EDA. Swiss Political System - Direct Democracy. `https://www.eda.admin.ch/aboutswitzerland/en/home/politik/uebersicht/direkte-demokratie.html/`. Retrieved on 2020-07-15.

[8] gfs.bern. Vorsichtige Offenheit im Bereich digitale Partizipation - Schlussbericht, 3 2020.

[9] Oded Goldreich. *Foundations of cryptography: volume 1, basic tools.* Cambridge University Press, 2007.

[10] Rolf Haenni. Swiss Post Public Intrusion Test: Undetectable attack against vote integrity and secrecy, 2019.

[11] Rolf Haenni, Eric Dubuis, Reto E Koenig, and Philipp Locher. CHVote: Sixteen Best Practices and Lessons Learned. In *International Joint Conference on Electronic Voting*, pages 95–111. Springer, 2020.

[12] Rolf Haenni, Reto E. Koenig, Philipp Locher, and Eric Dubuis. CHVote Protocol Specification, Version 3.2. Cryptology ePrint Archive, Report 2017/325, 2020. `https://eprint.iacr.org/2017/325`.

[13] Thomas Haines, Rageev Goré, and Bhavesh Sharma. Did you mix me? Formally Verifying Verifiable Mix Nets in Electronic Voting.

[14] Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 644–660. IEEE, 2020.

[15] Simon Josefsson et al. The Base16, Base32, and Base64 Data Encodings. Technical report, RFC 4648, October, 2006.

[16] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[17] Philipp Locher, Rolf Haenni, and Reto E Koenig. Analysis of the cryptographic implementation of the Swiss Post voting protocol, 2019.

[18] Ueli Maurer. Unifying zero-knowledge proofs of knowledge. In Bart Preneel, editor, *Progress in Cryptology – AFRICACRYPT 2009*, pages 272–286, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[19] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, pages 129–140, 1991.

[20] Swiss Post. Protocol of the Swiss Post Voting System. Computational Proof of Complete Verifiability and Privacy. Version 0.9.10. `https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/tree/master/Protocol`, Jul 2021.

[21] Nigel P Smart et al. Algorithms, key size and protocols report (2018), 2018.

[22] Ben Smyth. A foundation for secret, verifiable elections. *IACR Cryptology ePrint Archive*, 2018:225, 2018.

[23] Vanessa Teague and Olivier Pereira. Report on the SwissPost-Scytl e-voting system, trusted-server version, 2019.

[24] Björn Terelius and Douglas Wikström. Proofs of restricted shuffles. In *International Conference on Cryptology in Africa*, pages 100–113. Springer, 2010.

[25] Francois Yergeau. RFC3629: UTF-8, a transformation format of ISO 10646, 2003.