Cryptographic Primitives of the Swiss Post Voting System

Pseudo-code Specification

Swiss Post

Version 1.0.1

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
0.9.8	See change log crypto-primitives 0.12	TH, OE	CK, HR, BS	2021-10-15
0.9.9	See change log crypto-primitives 0.13	TH, HR, OE	CK, BS	2022-02-17
0.9.10	See change log crypto-primitives 0.14	TH, HR, OE	CK, BS	2022-04-18
1.0.0	See change log crypto-primitives 0.15	TH, HR, OE	CK, BS	2022-06-24
1.0.1	See change log crypto-primitives 1.0	TH, HR, OE	CK, BS	2022-10-03

Contents

Symbols

$\mathbb{A}_{10}$	Alphabet of decimal numbers
$\mathbb{A}_{Base16}$	Base16 (Hex) alphabet [20]
$\mathbb{A}_{Base32}$	Base32 alphabet, including the padding character = [20]
$\mathbb{A}_{Base64}$	Base64 alphabet, including the padding character = [20]
$\mathbb{A}_{UCS}$	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
$\mathbb{N}$	Set of positive integer numbers including 0
$\mathbb{N}^+$	Set of strictly positive integer numbers
$\mathbb{P}$	Set of prime numbers
$\mathbb{Z}_p$	Set of integers modulo p
$\mathbb{Z}_q$	Set of integers modulo q
$\mathbb{G}_q$	Set of quadratic residues modulo p, which forms a group of order q
$\mathbb{H}_\ell$	Ciphertext domain ($= \underbrace{\mathbb{G}_q \times \cdots \times \mathbb{G}_q}_{\ell+1 \text{ times}}$)
p	Encryption group modulus, a large safe prime (exact bitlength defined in the relevant section)
q	Encryption group cardinality s.t. $p = 2 \cdot q + 1$. A large prime (exact bit length defined in the relevant section)
g	Generator of the encryption group
$\|x\|$	Bit length of the number x
$\top$	Truth value true or successful termination
$\bot$	Truth value false or unsuccessful termination

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 [12].

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 [13].
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 [35]:

- 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) and its technical annex (VEleS annex)[9] describes these requirements.

Swiss democracy deserves an e-voting system with excellent security properties. 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 [15].

Our pseudo-code description of the cryptographic algorithms—inspired by [16]—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. We assume that the implementation ensures the correct domain of the input and context elements. E.g. this means that when an input has the expected form $\mathbf{x} = (x_0, \dots, x_{n-1}) \in (\mathbb{G}_q)^n$, the implementation checks that the input elements have the correct form: That $\mathbf{x}$ has exactly n elements and each one is a group member, for instance by calculating that the Jacobi Symbol of each element of $\mathbf{x}$ equals 1.
- We use the range notation for loops such as $i \in [0, n)$. We include the lower bound but exclude the upper bound, i.e. $0 \leq i < n$;
- we use 0-based indexing to close the representational gap between mathematics and code;
- we use **Require** for preconditions and **Ensure** for post-conditions;
- we use **return** to indicate a potentially early termination of the algorithm with the succeeding variable as the returned value; we use **Output** to describe the values that the algorithm produces;

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 Security Level

Table 2 describes a *testing-only*, *legacy*, and an *extended* security level and the associated security parameter selection. The *extended* security level is in line with common cryptographic standards (see [34]) while ensuring an acceptable performance. By default, we use the *extended* security level. The *legacy* security level should not be used anymore and will be removed in the future. The *testing-only* security level can be used in unit tests to speed up their execution but must not be used in a productive environment.

Security Level Name	*testing-only*	*legacy*	*extended*
Security Level	-	112 bits	128 bits
Group Parameters	$\|p\| = 8n, n \in \mathbb{N}^{+*}$ $\|q\| = \|p\| - 1$ $r = 1$	$\|p\| = 2048$ bits $\|q\| = 2047$ bits	$\|p\| = 3072$ bits $\|q\| = 3071$ bits
Recursive Hash Hash Function	SHA3-256	SHA3-256	SHA3-256
RecursiveHashToZq Extendable Output Function	SHAKE-256 $\ell^* = 512$	SHAKE-256 $\ell^* = 512$	SHAKE-256 $\ell^* = 512$
KDF Hash Function	SHA-256	SHA-256	SHA-256
Symmetric Algorithm	AES-GCM-256 nonce size 12 bytes	AES-GCM-256 nonce size 12 bytes	AES-GCM-256 nonce size 12 bytes
Signature key size	3072 bits	3072 bits	3072 bits

Table 2: Security levels.

For algorithm 4.10 we rely on SHAKE-256[10], which provides 256-bits of security if and only if at least 64 bytes of output are used. Therefore the algorithm fails if fewer than 512 bits are requested.

All security levels use `RSASSA-PSS`[28] as a signature algorithm, with 3072-bit keys and `SHA-256` used as the underlying hash function **and** hash for the mask generation function. The mask generation function used for PSS is MGF1, defined in appendix B.2 of RFC8017. The length of the salt is set to the length of the underlying hash function (*i.e.* 32 bytes). The trailer field number is 1, which represents the trailer field with value `0xbc`, in accordance with the same RFC.

3 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.

3.1 Byte Arrays

We denote a byte array B of length n as $< b_0, b_1, \dots, b_{n-1} >$ 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 [20]. Table 3 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 3: Example representations of different byte arrays

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

In some cases, we need to cut a byte array to a given bit length, for instance to limit the number of iterations for algorithms such as algorithm 4.1. This is achieved in algorithm 3.1 by taking the low bytes of the given byte array, and applying a bitmask to the first byte taken so that the necessary leading bits are zeroed.

Algorithm 3.1 CutToBitLength: Cuts the given byte array to the requested bit length

Input:

Byte array $B \in \mathcal{B}^N$ s.t. $N \in \mathbb{N}^+$

Requested length in bits $n \in \mathbb{N}^+$

Require: $n \leq N \cdot 8$ ▷ This should only be used to cut leading bits

Operation:

1: $\mathsf{length} \leftarrow \lceil \frac{n}{8} \rceil$
2: $\mathsf{offset} \leftarrow N - \mathsf{length}$
3: **if** $n \mod 8 \neq 0$ **then**
4: $\quad \mathsf{B'}_0 \leftarrow \mathsf{B}_{\mathsf{offset}} \wedge (2^{(n \mod 8)} - 1)$ ▷ Apply the bitwise-AND operator to mask out excess bits in the first byte
5: **else**
6: $\quad \mathsf{B'}_0 \leftarrow \mathsf{B}_{\mathsf{offset}}$
7: **end if**
8: **for** $i \in [1, \mathsf{length})$ **do**
9: $\quad \mathsf{B'}_i \leftarrow \mathsf{B}_{\mathsf{offset}+i}$
10: **end for**
11: **return** $(\mathsf{B'}_0, \dots, \mathsf{B'}_{\mathsf{length}-1})$

Output:

$(\mathsf{B'}_0, \dots, \mathsf{B'}_{\mathsf{length}-1}) \in \mathcal{B}^*$

Test values for algorithm 3.1 are provided in the attached cut-to-bit-length.json file.

Algorithms 3.2, 3.3, 3.4, 3.5, 3.6 and 3.7 encode and decode byte arrays to and from Base16, Base32 and Base64 encodings. We refer to "standard" Base32 and Base64 encoding; we do *not* use Base64 with URL and filename safe alphabet and Base32 with extended hex alphabet. 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 3.2 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 [20]

Algorithm 3.3 Base16Decode

Input:

String $S \in \mathbb{A}_{Base16}{}^*$ ▷ According to RFC4648 [20]

Operation:

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

Output:

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

Algorithm 3.4 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 [20]

Algorithm 3.5 Base32Decode

Input:

String $S \in \mathbb{A}_{Base32}{}^*$ ▷ According to RFC4648 [20]

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 $B \in \mathcal{B}^*$ or $\perp$ if S is not valid Base32

Algorithm 3.6 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 [20]

Algorithm 3.7 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 $B \in \mathcal{B}^*$ or $\perp$ if S is not valid Base64

3.2 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 4 provides some example integers.

Integer	Byte Array (Hex)
0	<0x00>
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 4: Example representations of different integers. We use spaces to separate thousands groups.

Therefore, we ignore leading zeros (with an exception for the value 0) and define al-

gorithm 3.8 to convert byte arrays to integers and algorithm 3.9 to convert integers to byte arrays. We avoid the empty byte array <> and represent 0 as <0x00>.
$|x|$ derives the minimal bit length of an integer, e.g. $|4\,294\,967\,295| = 32$ and $|4\,294\,967\,296| = 33$.

Algorithm 3.8 ByteArrayToInteger

Input:

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

Operation:

1: $x \leftarrow 0$
2: **for** $i \in [0, n)$ **do**
3: $\quad x \leftarrow 256 \cdot x + b_i$
4: **end for**

Output:

$x \in \mathbb{N}$

Algorithm 3.9 IntegerToByteArray

Input:

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

Operation:

1: $n \leftarrow \mathsf{ByteLength}(x)$ ▷ Derive minimal length n of byte array; See algorithm 3.10
2: $n \leftarrow \max(n, 1)$ ▷ Ensure that 0 does not result in the empty byte array
3: **for** $i \in [0, n)$ **do**
4: $\quad b_{n-i-1} \leftarrow x \mod 256$
5: $\quad x \leftarrow \lfloor \frac{x}{256} \rfloor$
6: **end for**
7: $B \leftarrow < b_0, b_1, ..., b_{n-1} >$

Output:

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

We define algorithm 3.10 to compute the byte length of an integer.

Algorithm 3.10 `ByteLength`: calculate the byte length of an integer

Input:

Integer $n \in \mathbb{N}$

Operation:

1: $b \leftarrow \left\lceil \frac{|n|}{8} \right\rceil$

Output:

Byte length $b \in \mathbb{N}^+$

3.3 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 [39]). Table 5 highlights some examples.

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

Table 5: Example representations of different strings

Algorithms 3.11 and 3.12 convert byte arrays to strings and vice versa. Potentially, the ByteArrayToString method can fail since not every byte array is a valid UTF-8 encoding.

Algorithm 3.11 StringToByteArray

Input:

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

Operation:

1: $B \leftarrow \mathsf{UTF\text{-}8}(S)$ ▷ Encode S in UTF-8

Output:

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

Algorithm 3.12 ByteArrayToString

Input:

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

Operation:

1: **if** B does not correspond to a valid UTF-8 encoding **then**
2: **return** $\perp$
3: **end if**
4: $S \leftarrow \mathsf{UTF}-8^{-1}(B)$

Output:

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

Moreover, we specify a method StringToInteger that translates a decimal String representation to an integer. Beware that the method StringToInteger(String) yields a different result than the conversion ByteArrayToInteger(StringToByteArray(String)).
Table 6 highlights some examples.

String	Integer
"0"	0
"1"	1
"1001"	1001
"0021"	21
"1A"	$\perp$

Table 6: Example Conversions of Strings to Integers

Algorithm 3.13 StringToInteger

Input:

String $S \in \mathbb{A}_{10}{}^*$

Operation:

1: **if** S is not valid decimal representation **then**
2: **return** $\perp$
3: **end if**
4: $x \leftarrow \mathsf{Decimal}(S)$ $\triangleright$ Convert the String into its decimal representation (radix = 10)

Output:

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

Conversely, the algorithm 3.14 converts integers to Strings.

Algorithm 3.14 IntegerToString

Input:

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

Operation:

1: $S \leftarrow \mathsf{Decimal}^{-1}(x)$ ▷ Convert the integers' decimal representation (radix = 10) into a String

Output:

String $S \in {\mathbb{A}_{10}}^*$

Algorithm 3.15 LeftPad

Input:

String $S \in {\mathbb{A}_{UCS}}^k$, $k \in \mathbb{N}^+$
Desired string length $l \in \mathbb{N}^+$
Padding character $c \in \mathbb{A}_{UCS}$

Require: $k <= l$

Operation:

$\mathbf{p} \leftarrow (c, ..., c)$ ▷ A vector of $l - k$ times the character c
$S' \leftarrow p||S$ ▷ Vector concatenation

Output:

String $S' \in {\mathbb{A}_{UCS}}^l$

4 Basic Algorithms

4.1 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[1] are available in most programming languages; therefore, we omit the pseudo-code for this primitive and call it `RandomBytes(length)`, where `length` $\in \mathbb{N}$ is the required number of bytes, and the output is in $\mathcal{B}^{length}$.

Algorithm 4.1 `GenRandomInteger`: provide a random integer between 0 (incl.) and m (excl.)

Input:

Upper bound $m \in \mathbb{N}^+$

Operation:

1: $\mathsf{length} \leftarrow \mathsf{ByteLength}(m-1)$ ▷ See algorithm 3.10
2: $\mathsf{bitLength} \leftarrow |m-1|$
3: $\mathsf{rBytes} \leftarrow \mathsf{CutToBitLength}(\mathsf{RandomBytes}(\mathsf{length}), \mathsf{bitLength})$ ▷ See algorithm 3.1
4: $\mathsf{r} \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{rBytes})$ ▷ See algorithm 3.8
5: **if** $\mathsf{r} \geq m$ **then**
6: go back to step 3
7: **end if**

Output:

Random integer $r \in [0, m)$

Algorithm 4.2 GenRandomVector: generate a random vector from ${\mathbb{Z}_q}^n$

Input:

Exclusive upper bound $q \in \mathbb{N}^+$
Length $n \in \mathbb{N}^+$

Operation:

1: **for** $i \in [0, n)$ **do**
2: $r_i \leftarrow \mathsf{GenRandomInteger}(q)$ ▷ See algorithm 4.1
3: **end for**

Output:

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

[1] A cryptographically secure random bytes generator has the following characteristics: it is designed for cryptographic use, generates independent, unbiased (i.e. uniform) bytes and relies on a high-quality entropy source[21]

Algorithms 4.3, 4.4, and 4.5 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 4.3 GenRandomBase16String

Input:

Desired length of string: $\ell \in \mathbb{N}^+$

Operation:

1: $\ell_{bytes} = \lceil \frac{4 \cdot \ell}{8} \rceil$
2: $b = \mathsf{RandomBytes}(\ell_{bytes})$
3: $S = \mathsf{Truncate}(\mathsf{Base16Encode}(b), \ell)$ ▷ See algorithms 3.2 and 4.6

Output:

$S \in (\mathbb{A}_{Base16} \setminus \{=\})^{\ell}$ ▷ A random string of the Base16 alphabet [20] without padding character

Algorithm 4.4 GenRandomBase32String

Input:

Desired length of string: $\ell \in \mathbb{N}^+$

Operation:

1: $\ell_{bytes} = \lceil \frac{5 \cdot \ell}{8} \rceil$
2: $b = \mathsf{RandomBytes}(\ell_{bytes})$
3: $S = \mathsf{Truncate}(\mathsf{Base32Encode}(b), \ell)$ ▷ See algorithms 3.4 and 4.6

Output:

$S \in (\mathbb{A}_{Base32} \setminus \{=\})^{\ell}$ ▷ A random string of the Base32 alphabet [20] without padding character

Algorithm 4.5 GenRandomBase64String

Input:

Desired length of string: $\ell \in \mathbb{N}^+$

Operation:

$\ell_{bytes} = \lceil \frac{6 \cdot \ell}{8} \rceil$
$b = \mathsf{RandomBytes}(\ell_{bytes})$
$S = \mathsf{Truncate}(\mathsf{Base64Encode}(b), \ell)$ ▷ See algorithms 3.6 and 4.6

Output:

$S \in (\mathbb{A}_{Base64} \setminus \{=\})^{\ell}$ ▷ A random string of the Base64 alphabet [20] without padding character

Algorithm 4.6 Truncate

Input:

Character array $S \in \mathbb{A}_x^u, u \in \mathbb{N}^+$
Desired length of string: $\ell \in \mathbb{N}^+$

Require: $\ell \leq u$

Operation:

1: **for** $i \in [0, \ell)$ **do**
2: $\quad S_i^{'} \leftarrow S_i$
3: **end for**

Output:

The truncated string $S' \in \mathbb{A}_x^{\ell}$

Algorithm 4.7 GenUniqueDecimalStrings

Input:

Desired length of each code: $l \in \mathbb{N}^+$

Number of unique codes: $n \in \mathbb{N}^+$

Require: $n \leq 10^l$

Operation:

1: $\texttt{codes} \leftarrow ()$
2: $m \leftarrow 10^l$
3: **while** $|\texttt{codes}| < n$ **do**
4: $x \leftarrow \mathsf{GenRandomInteger}(m)$ ▷ See algorithm 4.1
5: $c = \mathsf{LeftPad}(\mathsf{IntegerToString}(x), l, \text{“}0\text{”})$ ▷ See algorithm 3.15
6: **if** $c \notin \texttt{codes}$ **then**
7: $\texttt{codes} \leftarrow \texttt{codes} \cup \{c\}$
8: **end if**
9: **end while**

Output:

$\texttt{codes} \in (\mathbb{A}_{10})^{l \times n}$

4.2 Recursive Hash

Our recursive hash function—inspired by CHVote[16]—ensures that different inputs 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").

To prevent collisions across the different possible input domains, we prepend a single byte to the scalar input values, according to their type. This implies that RecursiveHash will give a different result for the input string "A" than for byte array <0x41>.

The recursive definition of the domain implies that infinite inputs are possible in theory (such as self-referencing inputs), in which case the algorithm does not terminate. It is the callers' responsibility to ensure only finite inputs are provided in practice.

Algorithm 4.8 RecursiveHash: Computes the hash value of multiple inputs

Context:

Cryptographic 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**
    $d \leftarrow \mathsf{Hash}(<0x00>||w)$
  else if $w \in \mathbb{N}$ **then**
    $d \leftarrow \mathsf{Hash}(<0x01>||\mathsf{IntegerToByteArray}(w))$ ▷ See algorithm 3.9
  else if $w \in \mathbb{A}_{UCS}$ **then**
    $d \leftarrow \mathsf{Hash}(<0x02>||\mathsf{StringToByteArray}(w))$ ▷ See algorithm 3.11
  else if $w = (w_0, \ldots, w_j)$ **then**
    $d \leftarrow \mathsf{Hash}(<0x03>||\mathsf{RecursiveHash}(w_0)|| \ldots ||\mathsf{RecursiveHash}(w_j))$
  else
    return $\perp$
  end if
end if

Output:

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

Test values for algorithm 4.8 are provided in the attached recursive-hash-sha3-256.json file.

All test files provided in the current document for the algorithms relying on this algorithm assume that the hash function defined in the security level (section 2) is used.

In some case, the output of a hash needs to be uniformly distributed across $\mathbb{Z}_q$. To achieve this, we use an extendable output function, limiting the output to the number of bits of q, and drawing a new value until the result is within the target domain, in order to avoid modulo-bias. Algorithm 4.9 deterministically draws new values, relying on algorithm 4.10 to perform the actual hashing. For algorithm 4.10, without loss of generality, we assume a signature for an extendable output function that takes the requested byte length as first parameter, and the input byte array as second parameter.

Algorithm 4.9 RecursiveHashToZq: Computes the hash value of multiple inputs uniformly into $\mathbb{Z}_q$

Input:

Exclusive upper bound $q \in \mathbb{N}^+$

Values $\mathbf{v} = (v_0, \dots, v_{k-1})$. 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: $k > 0$, $|q| \geq 512$

Operation:

1: $h \leftarrow$ ByteArrayToInteger(RecursiveHashOfLength($|q|, \mathbf{v}$)) ▷ See algorithms 3.8 and 4.10
2: **while** $h \geq q$ **do**
3: $\quad h \leftarrow$ ByteArrayToInteger(RecursiveHashOfLength($|q|, h||\mathbf{v}$)) ▷ Prepend h to $\mathbf{v}$
4: **end while**
5: **return** h

Output:

$h \in \mathbb{Z}_q$

Test values for algorithm 4.9 are provided in the attached recursive-hash-to-zq.json file.

Algorithm 4.10 RecursiveHashOfLength: Computes the hash value of multiple inputs to a given bit length

Context:

Extendable output function $\mathsf{XOF} : u \in \mathbb{N}^+ \times \mathcal{B}^* \mapsto \mathcal{B}^u$

Input:

Requested bit length $\ell \in \mathbb{N}^+$

Values $(v_0, ..., v_{k-1})$. 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: $k > 0$, $\ell \geq \ell^*$ ▷ See section

Operation:

1: $L \leftarrow \lceil \ell/8 \rceil$
2: **if** k > 1 **then** ▷ Avoid computing $\mathsf{Hash}(\mathsf{Hash}(v_0))$ when $k = 1$
3: $\mathbf{v} \leftarrow (v_0, ..., v_{k-1})$
4: $d \leftarrow \mathsf{RecursiveHashOfLength}(\ell, \mathbf{v})$
5: **else**
6: $w \leftarrow v_0$
7: **if** $w \in \mathcal{B}^*$ **then**
8: $d \leftarrow \mathsf{CutToBitLength}(\mathsf{XOF}(L, <0x00>||w), \ell)$ ▷ See algorithm 3.1
9: **else if** $w \in \mathbb{N}$ **then**
10: $d \leftarrow \mathsf{CutToBitLength}(\mathsf{XOF}(L, <0x01>||\mathsf{IntegerToByteArray}(w)), \ell)$ ▷ See algorithm 3.9
11: **else if** $w \in \mathbb{A}_{UCS}$ **then**
12: $d \leftarrow \mathsf{CutToBitLength}(\mathsf{XOF}(L, <0x02>||\mathsf{StringToByteArray}(w)), \ell)$ ▷ See algorithm 3.11
13: **else if** $w = (w_0, ..., w_j)$ **then**
14: **for** $i \in [0, j]$ **do**
15: $h_i \leftarrow \mathsf{RecursiveHashOfLength}(\ell, w_i)$
16: **end for**
17: $d \leftarrow \mathsf{CutToBitLength}(\mathsf{XOF}(L, <0x03>||h_0||...||h_j), \ell)$
18: **else**
return $\perp$
19: **end if**
20: **end if**

Output:

The digest $d \in \mathcal{B}^L$ ▷ Where the $8L - \ell$ first bits are set to 0

4.3 Hash and Square

At various places in the protocol, we break the homomorphic properties of certain operations by hashing and squaring values.[2] Given our choice of group parameters, modular squaring the hash's output ensures that the resulting value is a mathematical group member.

Algorithm 4.11 HashAndSquare: Hashes a value and squares the result

Context:

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

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

Input:

$x \in \mathbb{N}$

Operation:

1: $x_h \leftarrow \mathsf{RecursiveHashToZq}(q-1, x) + 1$ ▷ See algorithm 4.9, avoiding the case $x_h = 0$
2: $y \leftarrow x_h^2 \bmod p$

Output:

$y \in \mathbb{G}_q$

[2]The exponentiation function is a pseudo-random function if the input is randomly distributed.

4.4 KDF

The algorithms below rely on RFC5869[23] to describe a HMAC-based key derivation function (`HKDF`) to produce key material from a high-entropy source. The pseudo-code algorithms below only use the `HKDF-expand` part of the RFC, because the system only uses this function with cryptographically strong keys.

We note the function specified in section 2.3 of the RFC as HKDF-Expand, using the Hash function defined in the context of the pseudo-code and giving it the following inputs, in this order:

- a pseudo-random key, of length at least equal to the block size of the Hash function;
- additional context info, a byte array of arbitrary length;
- and the required length.

Section 2 defines the hash function to use in the implementation.

Algorithm 4.12 KDF: Key derivation function using HKDF-expand

Context:

Cryptographic hash function $\mathsf{Hash} : \mathcal{B}^* \mapsto \mathcal{B}^L$, $L \in \mathbb{N}^+$ ▷ Outputs a byte array of length L, see section 2 for the concrete choice

Input:

The cryptographically strong pseudo-random key $\mathsf{PRK} \in \mathcal{B}^{l_{\mathsf{key}}}$

Additional context information $(\mathsf{info}_0, \ldots, \mathsf{info}_{n-1}) \in (\mathbb{A}_{UCS}{}^*)^n$, s.t. $n \in \mathbb{N}$

The required byte length $\ell \in \mathbb{N}^+$

Require: $l_{\mathsf{key}} \geq L$

Require: $\ell \leq 255 \cdot L$

Require: $\mathsf{length}(\mathsf{StringToByteArray}(\mathsf{info}_i)) \leq 255 \ \forall i$ ▷ The length is the number of bytes in the byte array

---

Operation:

1: $\mathsf{info} \leftarrow <>$ ▷ Start with the empty byte array
2: **for** $i \in [0, n)$ **do**
3: $\quad \mathsf{info}_{i,\mathsf{bytes}} \leftarrow \mathsf{StringToByteArray}(\mathsf{info}_i)$ ▷ See algorithm 3.11
4: $\quad \mathsf{info} \leftarrow \mathsf{info}||\mathsf{length}(\mathsf{info}_{i,\mathsf{bytes}})||\mathsf{info}_{i,\mathsf{bytes}}$ ▷ Since the length is restricted to 255 bytes, it is encoded as a single unsigned byte
5: **end for**
▷ *I.e.* the empty byte array if $n = 0$
6: $\mathsf{OKM} \leftarrow \text{HKDF-Expand}(\mathsf{PRK}, \mathsf{info}, \ell)$ ▷ As per RFC5869[23], section 2.3
7: **return** OKM

Output:

The output keying material $\mathsf{OKM} \in \mathcal{B}^\ell$

Test values for algorithm 4.12 are provided in the attached kdf.json file.

The algorithm below is used when we need the resulting key material to be in $\mathbb{Z}_q$.

Algorithm 4.13 KDFToZq: Use the KDF function to generate a value in $\mathbb{Z}_q$

Context:

Cryptographic hash function $\mathsf{Hash} : \mathcal{B}^* \mapsto \mathcal{B}^L$, $L \in \mathbb{N}^+$ ▷ Outputs a byte array of length L, see section 2 for the concrete choice

Input:

The cryptographically strong pseudo-random key $\mathsf{PRK} \in \mathcal{B}^l$
Additional context information $\mathsf{info} \in (\mathbb{A}_{UCS}{}^*)^n$, s.t. $n \in \mathbb{N}$
The requested exclusive upper bound $q \in \mathbb{N}^+$

Require: $l \geq L$

Require: $\mathsf{ByteLength}(q) \geq L$ ▷ See algorithm 3.10

Operation:

1: $\ell \leftarrow \mathsf{ByteLength}(q)$ ▷ See algorithm 3.10
2: $h \leftarrow \mathsf{KDF}(\mathsf{PRK}, \mathsf{info}, \ell)$ ▷ See algorithm 4.12
3: $u \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{CutToBitLength}(h, |q|))$ ▷ See algorithm 3.8 and algorithm 3.1
4: **while** $u \geq q$ **do**
5: $\quad h \leftarrow \mathsf{KDF}(h, \mathsf{info}, \ell)$
6: $\quad u \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{CutToBitLength}(h, |q|))$
7: **end while**
8: **return** u

Output:

The value $u \in \mathbb{Z}_q$

Test values for algorithm 4.13 are provided in the attached kdf-to-zq.json file.

4.5 Argon2

Argon2[6] is a memory-hard key derivation function and represents the state of the art for password storage. Since it can be parametrized for parallelism and memory usage as well as iteration count, it is very efficient at increasing the cost of brute-force attacks, more so than `PBKDF` based on standard hashing functions, which can be computed more efficiently by attackers with specialized hardware.

We use Argon2 on key material that needs to be entered by humans, to keep the system usable while offsetting the limited entropy by making brute-force attacks more expensive. To avoid making any assumptions on the feasibility of side-channels attacks, we use the `Argon2id` variant, designed for this purpose.

We use a fixed tag length of 32 bytes, and a salt length of 16 bytes. As for the parallelism, memory usage and iteration count parameters, these should be chosen taking in consideration the entropy of the input (and thus the need for additional cost) and the maximal acceptable delay for each use.

We distinguish the case where the salt needs to be generated (creation of the reference tag – algorithm 4.14) and the case where the salt is provided as input, to ensure the same tag is generated – algorithm 4.15.

In both cases, we assume without loss of generality the existence of a function `argon2id` : `parameters` $\times \mathcal{B}^* \mapsto \mathcal{B}^*$, where the inputs are the parameters provided to Argon2 and the low-entropy key material and the output is the resulting tag.

Algorithm 4.14 GenArgon2id: compute the Argon2 tag

Context:

Memory usage parameter $m \in [14, 24]$
Parallelism parameter $p \in [1, 16]$
Iteration count $i \in [2, 256]$

Input:

Input keying material $k \in \mathcal{B}^*$

Operation:

1: $s \leftarrow \mathsf{RandomBytes}(16)$
2: $t \leftarrow \mathsf{GetArgon2id}(k, s)$ ▷ See algorithm 4.15
3: **return** (t, s)

Output:

The tag and the salt: $(t, s) \in \mathcal{B}^{32} \times \mathcal{B}^{16}$

Test values are provided in gen-argon2id.json.

Algorithm 4.15 GetArgon2id: compute the Argon2 tag

Context:

Memory usage parameter $m \in [14, 24]$
Parallelism parameter $p \in [1, 16]$
Iteration count $i \in [2, 256]$

Input:

Input keying material $k \in \mathcal{B}^*$
The salt $s \in \mathcal{B}^{16}$

Operation:

1: $c \leftarrow \{$
   `tagLength` : 32,
   `salt` : s,
   `memory` : 2^m, ▷ Given in `KiB`, thus $m = 21$ implies a memory usage of 2`GiB`
   `parallelism` : p,
   `iterations` : i
   $\}$
2: $t \leftarrow$ `argon2id`(c, k)

Output:

The tag $t \in \mathcal{B}^{32}$

Test values are provided in get-argon2id.json.

4.6 Primality testing

The primality of a given parameter is a common prerequisite in public-key systems. A widely used probabilistic test is the Miller-Rabin[27, 32] test. However, Albrecht et al.[1] have conducted a review of various libraries and shown that when the numbers being tested are generated under adversarial conditions, composites can be misidentified as primes with probability much greater than announced. They recommend using a Baillie-PSW[3] test instead, which combines one round of Miller-Rabin with base 2 and a Lucas probable prime test. Albeit no formal bounds have been determined on the probability of said test to declare a composite number to be prime, it has no known pseudoprimes despite theoretical and experimental search, whereas both Miller-Rabin and Lucas test have known families of pseudoprimes when used individually. The Big Integer libraries in both Java and Golang use an implementation closely based on Baillie-PSW for primality testing.

Furthermore, Baillie, Fiori and Wagstaff[2] have published an enhanced version of the Baillie-PSW test, adding additional checks at a limited cost, with experimental data suggesting must less frequent pseudoprimes.

Based on those elements, we chose enhanced Baillie-PSW (`EBPSW`) for primality testing. The following pseudocode algorithms specify it as described in the article above.

In the pseudocode below, we note `TestBit(n, i)` a method that returns `true` if and only if the i^{th} bit of n is set, with index 0 denoting the least significant bit. We note `lsb(n)` the index of the lowest significant bit of n.

Algorithm 4.16 IsProbablePrime: perform a probabilistic primality test

Input:

An integer $n \in \mathbb{N}$

Operation:

1: $s_p \leftarrow 8530092$
▷ *I.e.* 0b100000100010100010101100: for all primes $p \leq 23$, $\texttt{TestBit}(s_p, p) = \texttt{true}$
2: **if** $n \leq 23$ **then** ▷ Quick check for small values, simpler and faster than testing equality on each option
3: **return** $\texttt{TestBit}(s_p, n)$
4: **end if**
5: **if** $\neg\texttt{TestBit}(n, 0)$ **then** ▷ *I.e.* n is even and, as per line 2, $n > 2$ thus composite
6: **return** $\perp$
7: **end if** ▷ From this point on, n is odd and > 23
8: **if** $\neg\texttt{PassesMillerRabin}(n, 1, \texttt{true})$ **then** ▷ See algorithm 4.17
9: **return** $\perp$ ▷ Step 1 of EBPSW - `sprp`
10: **end if**
11: $(D, P, Q) \leftarrow \texttt{GetLucasParameters}(n)$ ▷ See algorithm 4.19. If it returns $\perp$, then n is composite, and we return $\perp$ here as well
▷ Step 2 of EBPSW, using Method A*, see algorithm 4.19
12: $(u_d, (v_{d\cdot 2^0}, \ldots, v_{d\cdot 2^s}), q^{\frac{n+1}{2}}) \leftarrow \mathsf{GetLucasSequenceValues}(D, P, Q)$ ▷ See algorithm 4.18
13: **if** $u_d \neq 0 \wedge v_{d\cdot 2^r} \neq 0 \quad \forall r \in [0, s)$ **then**
14: **return** $\perp$ ▷ Step 3 of EBPSW - `slprp`
15: **end if**
16: **if** $v_{d\cdot 2^s} \not\equiv 2 \cdot Q \pmod{n}$ **then**
17: **return** $\perp$ ▷ Step 4 of EBPSW - `vprp`
18: **end if**
19: **if** $q^{\frac{n+1}{2}} \not\equiv Q \cdot \left(\frac{Q}{n}\right) \pmod{n}$ **then**
▷ Where $\left(\frac{Q}{n}\right)$ is the Jacobi symbol $\in \{-1, 0, 1\}$
20: **return** $\perp$ ▷ Step 5 of EBPSW
21: **end if**
22: **return** $\top$

Output:

$\top$ if the number is a probable prime, $\perp$ if the number can be determined to be composite.

In algorithm 4.17, we provide a pseudo-code for the Miller-Rabin [27, 32] probabilistic primality test, with the added option of forcing one of the bases to be 2, as is needed for the Baillie-PSW test. This pseudo-code is based on algorithm HAC4.24[26].

Algorithm 4.17 PassesMillerRabin: verify if the given integer passes the Miller-Rabin test

Input:

And odd integer being tested $n > 23$

▷ Values $n \leq 23$ are handled in algorithm 4.16, line 2

The requested number of tests $t \in \mathbb{N}^+$

Flag to force one base a to 2, $f \in \{\top, \bot\}$

Operation:

1: $s \leftarrow \mathsf{lsb}(n-1)$
2: $r \leftarrow (n-1) \gg s$ ▷ Bitwise right shift, $n - 1 = 2^s \cdot r$
3: **for** $i \in [1, t+1)$ **do**
4: **if** $f \wedge (i = 1)$ **then** ▷ Force the first base to be 2
5: $a \leftarrow 2$
6: **else**
7: $a \leftarrow \mathsf{GenRandomInteger}(n-3) + 2$ ▷ See algorithm 4.1, $2 \leq a \leq n-2$
8: **end if**
9: $y \leftarrow a^r \pmod{n}$
10: **if** $y \neq 1 \wedge y \neq n-1$ **then**
11: $j \leftarrow 1$
12: **while** $j \leq s-1 \wedge y \neq n-1$ **do**
13: $y \leftarrow y^2 \pmod{n}$
14: **if** $y = 1$ **then**
15: **return** $\bot$
16: **end if**
17: $j \leftarrow j+1$
18: **end while**
19: **if** $y \neq n-1$ **then**
20: **return** $\bot$
21: **end if**
22: **end if**
23: **end for**
24: **return** $\top$

Output:

$\top$ if the Miller-Rabin test declares n to be a probable prime, $\bot$ if n is determined to be composite.

Algorithm 4.18 GetLucasSequenceValues: Compute the relevant values of the Lucas sequence

Input:

Lucas parameters $(D, P, Q) \in \mathbb{Z}^3$
The integer being tested $n \in \mathbb{N}^+, n \equiv 1 \pmod 2, n > 3$

Operation:

1: $n_1 \leftarrow n + 1$
2: $s \leftarrow \texttt{lsb}(n_1)$ ▷ Lowest significant bit: lowest index for which n_1's s^{th} bit is 1
3: $d \leftarrow n_1 \gg s$ ▷ Thus $n_1 = d \cdot 2^s$
4: $(u_k, v_k, q^k) \leftarrow (1, P, Q)$ ▷ Start from $k = 1$
5: **for** $i \in [|d| - 2, 0]$ **do**
6: $(u_k, v_k, q^k) \leftarrow \texttt{LucasDoubleK}(u_k, v_k, q^k, n)$ ▷ See algorithm 4.20
7: **if** $\texttt{TestBit}(d, i)$ **then**
8: $(u_k, v_k, q^k) \leftarrow \texttt{LucasIncrementK}(u_k, v_k, q^k, P, Q, D, n)$ ▷ See algorithm 4.21
9: **end if**
10: **end for** ▷ $k = d$
11: $(u_d, v_{d \cdot 2^0}) \leftarrow (u_k, v_k)$ ▷ We write $v_{d \cdot 2^0}$ for v_d, to keep the notation parallel to the paper
12: **for** $r \in [1, s]$ **do**
13: $(u_k, v_k, q^k) \leftarrow \texttt{LucasDoubleK}(u_k, v_k, q^k, n)$ ▷ See algorithm 4.20
14: $v_{d \cdot 2^r} \leftarrow v_k$
15: **if** $r = s - 1$ **then**
16: $q^{\frac{n+1}{2}} \leftarrow q^k$
17: **end if**
18: **end for** ▷ $k = n + 1$
19: **return** $(u_d, (v_{d \cdot 2^0}, \ldots, v_{d \cdot 2^s}), q^{\frac{n+1}{2}})$ ▷ Note that $v_{d \cdot 2^0}$ is v_d, while $v_{d \cdot 2^s}$ is v_{n+1}

Output:

$(u_d, (v_{d \cdot 2^0}, \ldots, v_{d \cdot 2^s}), q^{\frac{n+1}{2}}) \in \mathbb{Z}_n \times \mathbb{Z}_n^{s+1} \times \mathbb{Z}_n$

Algorithm 4.19 GetLucasParameters: Get Lucas series parameters using method A* from [2]

Input:

The integer being tested $n > 23$ and odd

▷ Values $n \leq 23$ are handled in algorithm 4.16, line 2

Operation:

1: $D \leftarrow 5$
2: **while** $\left(\frac{D}{n}\right) = -1$ **do** ▷ Where $\left(\frac{D}{n}\right)$ is the jacobi symbol
3: **if** $D > 0$ **then**
4: $D \leftarrow -(D + 2)$
5: **else**
6: $D \leftarrow (-D) + 2$
7: **end if** ▷ $5, -7, 9, -11, 13, ...$
8: **if** $D = 45 \wedge \mathsf{IsPerfectSquare}(n)$ **then** ▷ We've encountered 20 values of D s.t. $\left(\frac{D}{n}\right) = 1$, check for perfect squares, see algorithm 4.22
9: **return** $\perp$
10: **else if** $|D| > 10000$ **then**
▷ Values of D are expected to be significantly smaller than this hard limit, similar to Nicely[29], or GoLang's implementation.
11: **return** $\perp$
12: **end if**
13: **end while**
14: **if** $\left(\frac{D}{n}\right) = 0$ **then**
15: **return** $\perp$
16: **end if**
17: **if** $D \neq 5$ **then**
18: $P \leftarrow 1$
19: $Q \leftarrow (1 - D)/4$
20: **else**
21: $P \leftarrow 5$
22: $Q \leftarrow 5$
23: **end if**
24: **return** (D, P, Q)

Output:

The parameters for the Lucas sequence $(D, P, Q) \in \mathbb{Z}^3$
Or $\perp$ if either a D is found such that $\left(\frac{D}{n}\right) = 0$, n is a perfect square, or the value of D goes beyond a reasonable value.

In the algorithms below, we use the equations given in [7], page 628, and in [38] to increase the index in the Lucas sequence. In algorithm 4.20, we use equations 4.2.6 and 4.2.7 from Williams to double the index, and square q^k to double the exponent. In

algorithm 4.21, we use equation 4.2.21 from the same book to increment the index, and multiply q_k by Q to increment the exponent.

Algorithm 4.20 LucasDoubleK: double the index k for the Lucas sequence

Input:

$u_k \in \mathbb{Z}_n$
$v_k \in \mathbb{Z}_n$
$q^k \in \mathbb{Z}_n$
$n \in \mathbb{N}^+$

Operation:

1: $u_{2k} \leftarrow u_k \cdot v_k \pmod{n}$
2: $v_{2k} \leftarrow v_k^2 - 2 \cdot q^k \pmod{n}$
3: $q^{2k} \leftarrow (q^k)^2 \pmod{n}$
4: **return** (u_{2k}, v_{2k}, q^{2k})

Output:

$(u_{2k}, v_{2k}, q^{2k}) \in \mathbb{Z}_n^3$

Algorithm 4.21 LucasIncrementK: increment the index k for the Lucas sequence

Input:

$u_k \in \mathbb{Z}_n$
$v_k \in \mathbb{Z}_n$
$q^k \in \mathbb{Z}_n$
$P \in 1, 5$
$Q \in \mathbb{Z}$
$D \in 5, -7, 9, -11, 13, ...$
$n \in \mathbb{N}^+$

Operation:

1: $u_{num} \leftarrow P \cdot u_k + v_k$
2: **if** $\texttt{TestBit}(u_{num}, 0)$ **then**
3: $\quad u_{num} \leftarrow u_{num} + n$
4: **end if**
5: $u_{k+1} \leftarrow (u_{num} \gg 1) \pmod n$
6: $v_{num} \leftarrow D \cdot u_k + P \cdot v_k$
7: **if** $\texttt{TestBit}(v_{num}, 0)$ **then**
8: $\quad v_{num} \leftarrow v_{num} + n$
9: **end if**
10: $v_{k+1} \leftarrow (v_{num} \gg 1) \pmod n$
11: $q^{k+1} \leftarrow Q \cdot q^k \pmod n$
12: **return** $(u_{k+1}, v_{k+1}, q^{k+1})$

Output:

$(u_{k+1}, v_{k+1}, q^{k+1}) \in \mathbb{Z}_n^3$

Algorithm 4.22 IsPerfectSquare: checks if the given number is a perfect square, using Newton's method

Input:

an odd integer n s.t. $n \geq 5$

Operation:

1: $a \leftarrow \lfloor \frac{n}{2} \rfloor$
2: **while** $a^2 > n$ **do**
3: $\quad a \leftarrow \lfloor \frac{a + \lfloor \frac{n}{a} \rfloor}{2} \rfloor$
4: **end while**
5: **return** $a^2 = n$

Output:

$\top$ if n is a perfect square, $\bot$ otherwise.

5 Symmetric Authenticated Encryption

We define a symmetric authenticated encryption scheme based on Authenticated Encryption with Associated Data (AEAD), as defined in RFC5116[25].

We denote the function specified in section 2.1 as AuthenticatedEncryption with the following inputs in this order:

- a secret key, a byte array of length k;
- a nonce, a byte array of length n;
- a plaintext with the data to be encrypted, a byte array of length p which may be zero;
- the associated data with the data to be authenticated but not encrypted, a byte array which may be of length zero;

It produces a single output ciphertext, a byte array at least as long as the plaintext.

We denote the function specified in section 2.2 as AuthenticatedDecryption with the following inputs (as defined above) in this order: secret key, nonce, associated data, ciphertext. It produces a single output, either plaintext or $\perp$.

For the implementation we use AES-GCM-256 as the AEAD algorithm (see section 2) with $n = 12$ bytes (as described in [25] and recommended in [19]) and $p \leq 64 \times 10^9$ bytes ([11] section 3). We use a randomized nonce as described in [11] section 8.2.2, with the given limitation that "the total number of invocations of the authenticated encryption function shall not exceed 2^{32}" for a given key. Moreover, it is critical that nonces should not be reused, otherwise the security properties break down. In particular, algorithm 5.1 should not be used in a virtualized environment, as a rollback could lead to a nonce reuse. Callers of these algorithms should make sure that the preceding properties are respected.

Algorithm 5.1 GenCiphertextSymmetric: Symmetric authenticated encryption

Context:

Authenticated encryption function $\mathsf{AuthenticatedEncryption} : (\mathcal{B}^k, \mathcal{B}^n, \mathcal{B}^p, \mathcal{B}^*) \mapsto \mathcal{B}^c$ s.t. $k \in \mathbb{N}^+, n \in \mathbb{N}^+, p \in \mathbb{N}, c \in \mathbb{N}^+$ with the constraints on k, n, p of the specific algorithm used

Input:

The encryption key $\mathsf{K} \in \mathcal{B}^k$

The plaintext $\mathsf{P} \in \mathcal{B}^p$

Associated data $(\mathsf{associated}_0, \dots, \mathsf{associated}_{d-1}) \in (\mathbb{A}_{UCS}{}^*)^d$, s.t. $d \in \mathbb{N}$

Require: $\mathsf{length}(\mathsf{StringToByteArray}(\mathsf{associated}_i)) \leq 255 \ \forall i$ ▷ The length is the number of bytes in the byte array

---

Operation:

1: $\mathsf{nonce} \leftarrow \mathsf{RandomBytes}(\mathsf{n})$
2: $\mathsf{associated} \leftarrow <>$ ▷ Start with the empty byte array
3: **for** $i \in [0, d)$ **do**
4: $\quad \mathsf{associated}_{i,\mathsf{bytes}} \leftarrow \mathsf{StringToByteArray}(\mathsf{associated}_i)$ ▷ See algorithm 3.11
5: $\quad \mathsf{associated} \leftarrow \mathsf{associated}||\mathsf{length}(\mathsf{associated}_{i,\mathsf{bytes}})||\mathsf{associated}_{i,\mathsf{bytes}}$ ▷ Since the length is restricted to 255, it is encoded as a single unsigned byte
6: **end for**
7: $\mathsf{C} \leftarrow \mathrm{AuthenticatedEncryption}(\mathsf{K}, \mathsf{nonce}, \mathsf{P}, \mathsf{associated})$ ▷ As per [25], section 2.1

Output:

The authenticated ciphertext $\mathsf{C} \in \mathcal{B}^c$

The nonce $\mathsf{nonce} \in \mathcal{B}^n$

Test values are provided in the gen-ciphertext-symmetric.json file.

Algorithm 5.2 GetPlaintextSymmetric: Symmetric authenticated decryption

Context:

Authenticated decryption function $\mathsf{AuthenticatedDecryption} : (\mathcal{B}^k, \mathcal{B}^n, \mathcal{B}^*, \mathcal{B}^c) \mapsto \mathcal{B}^p$, $k \in \mathbb{N}^+, n \in \mathbb{N}^+, p \in \mathbb{N}, c \in \mathbb{N}^+$ with the constraints on k, n of the specific algorithm used

Input:

The encryption key $\mathsf{K} \in \mathcal{B}^k$

The ciphertext $\mathsf{C} \in \mathcal{B}^c$

The nonce $\mathsf{nonce} \in \mathcal{B}^n$

Associated data $(\mathsf{associated}_0, \dots, \mathsf{associated}_{d-1}) \in (\mathbb{A}_{UCS}{}^*)^d$, s.t. $d \in \mathbb{N}$

Require: $\mathsf{length}(\mathsf{StringToByteArray}(\mathsf{associated}_i)) \leq 255 \ \forall i$ ▷ The length is the number of bytes in the byte array

---

Operation:

1: $\mathsf{associated} \leftarrow <>$ ▷ Start with the empty byte array
2: **for** $i \in [0, d)$ **do**
3: $\quad \mathsf{associated}_{i,\mathsf{bytes}} \leftarrow \mathsf{StringToByteArray}(\mathsf{associated}_i)$ ▷ See algorithm 3.11
4: $\quad \mathsf{associated} \leftarrow \mathsf{associated}||\mathsf{length}(\mathsf{associated}_{i,\mathsf{bytes}})||\mathsf{associated}_{i,\mathsf{bytes}}$ ▷ Since the length is restricted to 255, it is encoded as a single unsigned byte
5: **end for**
6: $\mathsf{P} \leftarrow \mathrm{AuthenticatedDecryption}(\mathsf{K}, \mathsf{nonce}, \mathsf{associated}, \mathsf{C})$ ▷ As per [25], section 2.2

Output:

The authenticated plaintext $\mathsf{P} \in \mathcal{B}^p$

Or $\perp$ if the ciphertext does not authenticate

Test values are provided in the get-plaintext-symmetric.json file.

6 Digital signatures

An integral part of the security of any distributed system consists of ensuring that each message did indeed originate from an authorized party.

In the context of the Swiss Post e-voting system in particular, and in systems with distribution of trust in general, this further entails requiring that each contribution comes from the expected party.

The pseudo-code algorithms provided in this section rely on established digital signature standards providing authenticity and integrity of the communications. The specific algorithms used are defined in section 2. These elements are meant to address the issues raised by Thomas Haines[17].

They are meant to be used in the following way:

- the operators of each authority generate a private key and a certificate for the matching public key,
- a well-documented process ensures that each authority loads and securely stores the other authorities' certificates, validating their authenticity,
- upon sending messages, each authority signs them with their private key,
- upon receiving messages, each authority verifies that the signature is valid for the message, using the public key contained in the certificate of the purported author,
- auditors of the system are also provided with each authority's certificate, and ensure that each certificate has indeed been sent by the operators of the corresponding authority,
- finally, the auditors verify that each message has indeed been signed correctly by the expected authority.

The pseudo-code algorithms below rely on well-established standards, with well-tested libraries available in most programming languages. The abstraction level in this section diverges slightly from the rest of the document, to put the focus more on the nature of the elements required rather than their exact form, therefore allowing flexibility to use existing libraries, rather than reinventing the wheel.

6.1 Generating a signing key and certificate

The following algorithm is used by each authority to generate a private key and a certificate containing the corresponding public key, with use restricted to signing, for a limited duration.

The period of validity for the certificate should strike a balance between limiting the risks related to prolonged use and practicality. Since each participant (4 sets of authority operators, and typically at least one auditor per electoral authority using the system) needs to verify that each certificate is provided by the expected set of operators, the process is inherently tedious.

Given the signature and verification algorithms defined in section 2, we assume there exist matching functions for: key pair generation, which we will note $\mathsf{GenKeyPair}()$; creation of a certificate for the public key, signed by the private key, as a self-signed x.509 certificate[8] encoded according to DER[33], which we will note as follows:
$\mathsf{GetCertificate}(\text{pubKey}, \text{privKey}, \text{info})$ where the third parameter defines the additional properties of the certificate, including identity information, validity, and key usage.

Algorithm 6.1 GenKeysAndCert: Generate a key pair and matching certificate

Context:

The signature algorithms, providing GenKeyPair and GetCertificate as described above

Information about the identity of the authority generating keys, including

- common name CN
- country C
- state ST
- locality L
- organisation O

Input:

Start of validity validFrom

End of validity validUntil

Require: validFrom $<$ validUntil

Operation:

1: $(\text{privKey}, \text{pubKey}) \leftarrow \mathsf{GenKeyPair}()$
2: $\text{info} \leftarrow \{\mathsf{CN}, \mathsf{C}, \mathsf{ST}, \mathsf{L}, \mathsf{O}\}$
3: $\text{info} \leftarrow \text{info} \cup \{\mathsf{validFrom}, \mathsf{validUntil}\}$
4: $\text{usage} \leftarrow (\mathsf{CertificateSign}, \mathsf{DigitalSignature})$
5: $\text{info} \leftarrow \text{info} \cup \{\text{usage}\}$
6: $\text{cert} \leftarrow \mathsf{GetCertificate}(\text{privKey}, \text{pubKey}, \text{info})$
7: **return** $(\text{privKey}, \text{cert})$

Output:

the private key privKey which the authority will keep secret and use for signing,
the certificate cert which will be shared with the other authorities, so that they can verify messages signed by this authority.

6.2 Importing a trusted certificate

We assume each authority has access to a dedicated trust store or a similar trust mechanism. These trust stores are intially empty and only properly validated certificates can be imported. The distribution of certificates must rely on an existing authenticated channel and the process for the distribution must be documented in sufficient detail. The validation of the certificate previous to the import is a human-led process, which requires (at least) the following checks:

- Does the identity claimed by the certificate match the identity of the authority? This includes validating the ASN.1 fields for country, state, locality, organisation and common name.
- Does the period of validity declared in the certificate match the expected period?

- Are the declared uses for the key exclusively restricted to 1) signing the certificate itself and 2) digital signature only ?

Only after those elements have all been verified, should a certificate be imported into the authority's trust store.

6.3 Signing a message

Given the signature algorithm defined in section 2, we note $\mathsf{Sign}(\text{privKey}, m)$ the underlying signature algorithm, returning the byte array representing the signature. Every outgoing message from the authorities MUST be signed according to the algorithm below.
We assume that the current timestamp can be retrieved with $\mathsf{GetTimestamp}()$. Further, we assume that the validity starting and ending timestamps for the certificate cert can be retrieved with $\mathsf{ValidFrom}(\text{cert})$ and $\mathsf{ValidUntil}(\text{cert})$ respectively.

Algorithm 6.2 GenSignature: generate a signature for the given message

Context:

The private key privKey
The matching certificate cert

Input:

The message to sign $m \in \mathcal{V}$ ▷ See algorithm 4.8
Additional context data $c \in \mathcal{V}$

Operation:

1: $t \leftarrow \mathsf{GetTimestamp}()$
2: **if** $\mathsf{ValidFrom}(\text{cert}) \leq t < \mathsf{ValidUntil}(\text{cert})$ **then**
3: $h \leftarrow \mathsf{RecursiveHash}((m, c))$ ▷ See algorithm 4.8
4: **return** $\mathsf{Sign}(\text{privKey}, h)$ ▷ See algorithm 3.11
5: **else**
6: **return** $\perp$
7: **end if**

Output:

The signature for the message $\in \mathcal{B}^*$
Or $\perp$ if the message is timestamped at a date the certificate cert is not valid for.

6.4 Verifying a message

Given the signature algorithm defined in section 2, we note $\mathsf{Verify}(\text{pubKey}, m, s)$ the underlying signature verification algorithm, returning $\top$ if the signature is valid, $\bot$ otherwise. Every incoming message expected to originate from a system authority MUST be verified according to the algorithm below.

As mentioned in section 6.2, we assume each authority keeps a trust store of certificates, containing only verified and validated certificates for the known authorities of the system.[3] We assume each certificate cert is identified by a unique string id, and can be retrieved with $\mathsf{FindCertificate}(\text{id})$, and the included public key can be retrieved with $\mathsf{GetPublicKey}(\text{cert})$.
As in section 6.3, we assume that the current timestamp can be retrieved with $\mathsf{GetTimestamp}()$. Further, we assume that the validity starting and ending timestamps for the certificate cert can be retrieved with $\mathsf{ValidFrom}(\text{cert})$ and $\mathsf{ValidUntil}(\text{cert})$ respectively.

Algorithm 6.3 VerifySignature: verify that a signature is valid, and from the expected authority

Context:

The signature algorithms, providing Verify as described above
The trust store, providing FindCertificate as described above

Input:

The identifier of the authority expected to have signed the message $\text{id} \in {\mathbb{A}_{UCS}}^*$
The message to sign $m \in \mathcal{V}$ ▷ See algorithm 4.8
Additional context data $c \in \mathcal{V}$
The signature $s \in \mathcal{B}^*$

Operation:

$\text{cert} \leftarrow \mathsf{FindCertificate}(\text{id})$
$t \leftarrow \mathsf{GetTimestamp}()$
if $t < \mathsf{ValidFrom}(\text{cert}) \vee t \geq \mathsf{ValidUntil}(\text{cert})$ **then** ▷ Check validity
  return $\bot$
end if
$\text{pubKey} \leftarrow \mathsf{GetPublicKey}(\text{cert})$
$h \leftarrow \mathsf{RecursiveHash}((m, c))$ ▷ See algorithm 4.8
return $\mathsf{Verify}(\text{pubKey}, h, s)$ ▷ See algorithm 3.11

Output:

$\top$ if the signature is valid and the message has a timestamp during which the certificate was valid, $\bot$ otherwise.

[3] No `web of trust`, only directly authenticated parties.

7 ElGamal Cryptosystem

The computational proof [31] 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.

7.1 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 . We pick p and q prime with $p = 2 \cdot q + 1$. Section 2 defines the bit length of p and q. We define the smallest integer $x > 1$ s.t. x is a generator of $\mathbb{G}_q$ as g.
We pick all the group parameters *verifiably* to demonstrate that they are devoid of hidden properties or back doors. Algorithm 7.1 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 [10]. 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 3.2 on how we represent integers) and perform a subsequent bitwise right-shift operation.

Algorithm 7.1 GetEncryptionParameters

Context:

The security level λ defining the bit length of $|p|$, the bit length of $|q|$ and the number of rounds r for primality testing, according to table 2.

Input:

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

Require: $|p| \mod 8 = 0$ ▷ The algorithm below assumes the bit length of p is a multiple of 8

Operation:

1: $i \leftarrow 0$
2: **do**
3: $\hat{q}_b \leftarrow \mathsf{SHAKE128}((\mathsf{StringToByteArray}(seed) \,||\, \mathsf{IntegerToByteArray}(i)), \frac{|p|}{8})$ ▷ See algorithm 3.11, 3.9, and FIPS PUB 202 [10]
4: $q_b \leftarrow$ <0x01> $||\, \hat{q}_b$ ▷ Byte array concatenation
5: $q \leftarrow \mathsf{ByteArrayToInteger}(q_b) \gg 2$ ▷ See algorithm 3.8 ▷ bit-wise right shift
6: $q \leftarrow q + 1 - (q \bmod 2)$ ▷ Ensuring that q is odd
7: $i \leftarrow i + 1$
8: **while** $(\neg\mathsf{isProbablePrime}(q, r)$ **or** $\neg\mathsf{isProbablePrime}(2 \cdot q + 1, r))$
9: $p \leftarrow 2 \cdot q + 1$
10: **for** $i \in [2, 4]$ **do** ▷ necessarily $4 \in \mathbb{G}_q$ since every quadratic residue is a group element $(2^2 = 4)$
11: **if** $i \in \mathbb{G}_q$ **then**
12: **return** $g \leftarrow i$
13: **end if**
14: **end for**

Output:

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

Test values for the algorithm 7.1 are provided in the attached get-encryption-parameters.json file.

7.2 Prime Selection

The Swiss Post Voting System encodes voting options using small primes. Given a mathematical group, we require an algorithm that returns a list of small prime numbers (excluding the generator g) of this mathematical group.

Algorithm 7.2 GetSmallPrimeGroupMembers

Input:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
Group generator $g \in \mathbb{G}_q$
Desired number of prime group members $r \in \mathbb{N}^+$

Require:

$g \in [2, 4]$
$r \leq q - 4$
$r < 10,000$ ▷ For efficiency reasons

Operation:

1: $\mathsf{current} \leftarrow 5$
2: $\mathbf{p} \leftarrow ()$
3: $\mathsf{count} \leftarrow 0$
4: **while** $\mathsf{count} < r \wedge \mathsf{current} < p \wedge \mathsf{current} < 2^{31}$ **do**
5: **if** $\mathsf{current} \in \mathbb{G}_q \wedge \mathsf{IsSmallPrime}(\mathsf{current})$ **then** ▷ See algorithm 7.3
6: $\mathbf{p}[\mathsf{count}] \leftarrow \mathsf{current}$
7: $\mathsf{count} \leftarrow \mathsf{count} + 1$
8: **end if**
9: $\mathsf{current} \leftarrow \mathsf{current} + 2$
10: **end while**
11: **if** $\mathsf{count} \neq r$ **then**
12: **return** $\perp$
13: **end if**

Output:

The small prime group members in ascending order $\mathbf{p} = (\mathrm{p}_0, \ldots, \mathrm{p}_{r-1}), \mathrm{p}_i \in (\mathbb{G}_q \cap \mathbb{P}) \setminus \{2, 3\}$
$\perp$ if r is bigger than the number of primes in the $\mathbb{G}_q$ group.

We define a deterministic primality test that is efficient for small primes.

Algorithm 7.3 IsSmallPrime

Context:

Input:

Number $n \in \mathbb{N}^+$

Require: $n < 2^{31}$ ▷ This covers up to the 105 millionth prime

Operation:

1: **if** $n = 1$ **then**
2: **return** $\perp$
3: **else if** $n = 2$ **then**
4: **return** $\top$
5: **else**
6: **for** $i \in [2, \lceil\sqrt{n}\rceil]$ **do** ▷ We use ceil to take into account floating point arithmetic limitations
7: **if** $n \mod i = 0$ **then**
8: **return** $\perp$
9: **end if**
10: **end for**
11: **return** $\top$
12: **end if**

Output:

$\top$ if n is prime, $\perp$ otherwise.

7.3 Key Pair Generation

Algorithm 7.4 describes the generation of a multi-recipient ElGamal key pair. We include 0 and 1, in the secret and public key ranges respectively, since the ElGamal encryption scheme is correct and semantically secure, even for those edge cases [5, 37]. However, one must be careful when using the key pair for purposes other than ElGamal encryption, especially when an adversary might maliciously choose the key pair. For instance, imagine if one successively exponentiates a value x by the keys $k_1, \dots, k_n$:

$$(((x^{k_1})^{k_2})\dots)^{k_n}$$

Here, a single key $k_i = 0$ would cancel the contributions of all other keys since the result is guaranteed to be 1 — potentially leading to undesired consequences. In that case, the cryptographic protocol must draw the secret keys from ${\mathbb{Z}_q}^+$ (to exclude 0) and the public keys from the generators of $\mathbb{G}_q$ (to exclude 1).

Algorithm 7.4 GenKeyPair: Generate a multi-recipient key pair

Input:

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

Operation:

1: **for** $i \in [0, N)$ **do**
2: $sk_i \leftarrow \mathsf{GenRandomInteger}(q)$ ▷ See algorithm 4.1
3: $pk_i \leftarrow g^{sk_i} \mod p$
4: **end for**

Output:

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

7.4 Encryption

We consider a "multi-recipient message", in which different public keys encrypt different messages. If there are more public keys than messages ($\ell < k$), we drop the excess public keys.

Algorithm 7.5 GetCiphertext: Compute a ciphertext with provided randomness

Context:

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

Input:

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

Require: $0 < \ell \leq k$

Operation:

1: $\gamma \leftarrow g^r \mod p$
2: **for** $i \in [0, \ell)$ **do**
3: $\quad \phi_i \leftarrow \mathsf{pk}_i^r \cdot m_i \mod p$
4: **end for**

Output:

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

Test values for the algorithm 7.5 are provided in get-ciphertext.json.

7.5 Ciphertext Operations

Algorithm 7.6 GetCiphertextExponentiation: Exponentiate each ciphertext element by an exponent a

Context:

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

Input:

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

Operation:

1: $\gamma \leftarrow \gamma^a \mod p$
2: **for** $i \in [0, \ell)$ **do**
3: $\quad \phi_i \leftarrow \phi_i^a \mod p$
4: **end for**

Output:

$(\gamma, \phi_0, \dots, \phi_{\ell-1}) \in \mathbb{H}_\ell$

Algorithm 7.7 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. $p = 2 \cdot q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

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

Operation:

1: product $\leftarrow$ GetCiphertext$(\vec{1}, 0, \mathbf{pk})$ $\quad \triangleright$ Neutral element of ciphertext multiplication
2: **for** $i \in [0, n)$ **do**
3: $\quad$ product $\leftarrow$ GetCiphertextProduct(product, GetCiphertextExponentiation(C_i, a_i))
$\quad \triangleright$ See algorithm 7.8 and algorithm 7.6
4: **end for**

Output:

The resulting product $\in \mathbb{H}_\ell$

Algorithm 7.8 GetCiphertextProduct: Multiply two ciphertexts

Context:

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

Input:

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

Operation:

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

Output:

$(\gamma, \phi_0, \dots, \phi_{\ell-1}) \in \mathbb{H}_\ell$

Test values for the algorithm 7.8 are provided in get-ciphertext-product.json.

7.6 Decryption

Algorithm 7.9 GetMessage: Retrieve the message from a ciphertext

Context:

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

Input:

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

Operation:

1: **for** $i \in [0, \ell)$ **do**
2: $\quad m_i \leftarrow \phi_i \cdot \gamma^{-\mathsf{sk}_i} \mod p$
3: **end for**

Output:

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

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 7.10.

Algorithm 7.10 GetPartialDecryption: Partially decrypt a provided ciphertext

Context:

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

Input:

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

Operation:

1: $(m_0, \dots, m_{\ell-1}) \leftarrow \mathsf{GetMessage}(\mathbf{c}, \mathbf{sk})$ ▷ See algorithm 7.9
2: **return** $(\gamma, m_0, \dots, m_{\ell-1})$

Output:

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

Algorithm 7.11 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. $p = 2 \cdot q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A vector of ciphertexts $\mathbf{C} = (\mathbf{c}_0, \dots, \mathbf{c}_{N-1}) \in (\mathbb{H}_\ell)^N$
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 $\mathbf{i}_{\mathsf{aux}} \in (\mathbb{A}_{UCS}{}^*)^*$

Require: $0 < \ell \leq k$

Operation:

1: **for** $i \in [0, N)$ **do**
2: $\quad \mathbf{c}'_i \leftarrow \mathsf{GetPartialDecryption}(\mathbf{c}_i, \mathbf{sk})$ ▷ See algorithm 7.10
3: $\quad (\gamma', \phi'_0, \dots, \phi'_{\ell-1}) \leftarrow \mathbf{c}'_i$
4: $\quad \pi_{\mathsf{dec},i} \leftarrow \mathsf{GenDecryptionProof}(\mathbf{c}_i, (\mathbf{pk}, \mathbf{sk}), (\phi'_0, \dots, \phi'_{\ell-1}), \mathbf{i}_{\mathsf{aux}})$ ▷ See algorithm 9.5
5: **end for**
6: $\mathbf{C}' \leftarrow (\mathbf{c}'_0, \dots, \mathbf{c}'_{N-1})$
7: $\pi_{\mathsf{dec}} \leftarrow (\pi_{\mathsf{dec},0}, \dots, \pi_{\mathsf{dec},N-1})$
8: **return** $(\mathbf{C}', \pi_{\mathsf{dec}})$

Output:

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

Algorithm 7.12 VerifyDecryptions: Verify the decryptions of a vector of ciphertexts.

Context:

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

Input:

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

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

Operation:

1: **for** $i \in [0, N)$ **do**
2: $(\gamma, \phi_0, \dots, \phi_{\ell-1}) \leftarrow \mathbf{c}_i$
3: $(\gamma', \phi'_0, \dots, \phi'_{\ell-1}) \leftarrow \mathbf{c}'_i$
4: **if** $\gamma \neq \gamma'$ **then**
5: **return** $\bot$
6: **end if**
7: $\mathbf{m} \leftarrow (\phi'_0, \dots, \phi'_{\ell-1})$
8: $\mathsf{ok} \leftarrow \mathsf{VerifyDecryption}(\mathbf{c}_i, \mathbf{pk}, \mathbf{m}, \pi_{\mathsf{dec},i}, \mathbf{i}_{\mathsf{aux}})$ ▷ See algorithm 9.6
9: **if** $\neg\mathsf{ok}$ **then**
10: **return** $\bot$
11: **end if**
12: **end for**
13: **return** $\top$ ▷ Succeed if and only if all verifications above succeeded

Output:

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

7.7 Combining ElGamal Multi-recipient Public Keys

In this section we provide a mechanism to combine public keys in such a way that encryption under the combined public key can be inverted by successive decryptions with the secret key components. This allows consumers to encrypt their messages in a way that requires the collaboration of all authorities to retrieve the plaintext, without those authorities ever needing to reveal their secret key. Such trust distribution systems can be subjected to rogue key attacks, where a component's public key is built using parts of the other components' public keys, leading to possible attacks on the combined key. For this reason, systems relying on this mechanism to distribute trust should consider using zero-knowledge proofs of knowledge of the corresponding secret key.

Since we are using ElGamal encryption, this can be achieved by simply taking the product of the public key components as the combined key.

By construction of the public keys, we have $\mathsf{pk}_j = g^{\mathsf{sk}_j}$. It follows that $\prod_j \mathsf{pk}_j = g^{\sum_j \mathsf{sk}_j}$. By definition of the ElGamal encryption function, we have $E_{\prod_j \mathsf{pk}_j}(m, r) = (g^r, (\prod_j \mathsf{pk}_j)^r \cdot m)$. Using the previous equality to replace the product of public keys, we get $(g^r, g^{r \cdot \sum_j \mathsf{sk}_j} \cdot m)$. From this point, each successive partial decryption by the corresponding secret key sk_j multiplies the second term by $g^{r \cdot (-\mathsf{sk}_j)}$, eventually leaving only the term m. This can be generalised for the multi-recipient ElGamal encryption scheme, by combining each set of keys independently.

Algorithm 7.13 CombinePublicKeys: combine a set of multi-recipient ElGamal keys

Context:

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

Input:

A list of multi-recipient ElGamal public keys $(\mathbf{pk}_0, \dots, \mathbf{pk}_s) \in {\mathbb{G}_q}^{N \times s}$

$\triangleright$ Where N is the number of elements in each multi-recipient key, and s is the number of keys

Operation:

1: **for** $i \in [0, N)$ **do**
2: $\quad \mathsf{pk}_{\mathsf{combined},i} \leftarrow \prod_{j=0}^{s} \mathsf{pk}_{j,i} \bmod p$
3: **end for**
4: **return** $(\mathsf{pk}_{\mathsf{combined},0}, \dots, \mathsf{pk}_{\mathsf{combined},N-1})$

Output:

$\mathbf{pk}_{\mathsf{combined}} = (\mathsf{pk}_{\mathsf{combined},0}, \dots, \mathsf{pk}_{\mathsf{combined},N-1}) \in {\mathbb{G}_q}^N$

8 Mix Net

Verifiable mix nets underpin most modern e-voting schemes with non-trivial tallying methods since they hide the relationship between encrypted votes (potentially linked to the voter's identifier) and decrypted votes [18]. 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 Terelius-Wikström [36] and Bayer-Groth [4]. The Swiss Post Voting System uses the Bayer-Groth mix net, which we describe in this section. The computational proof [31] 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 8.1 `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. $p = 2 \cdot q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

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

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

Operation:

1: $(\mathbf{C}', \pi, \mathbf{r}) \leftarrow \mathsf{GenShuffle}(\mathbf{C}, \mathbf{pk})$ ▷ See algorithm 8.3
2: $(m, n) \leftarrow \mathsf{GetMatrixDimensions}(N)$ ▷ See algorithm 8.5
3: $\mathbf{ck} \leftarrow \mathsf{GetVerifiableCommitmentKey}(n)$ ▷ See algorithm 8.6 ▷ 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 8.11

Output:

$\mathbf{C}' \in (\mathbb{H}_\ell)^N$
shuffleArgument

Algorithm 8.2 `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. $p = 2 \cdot q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

A vector of unshuffled ciphertexts $\mathbf{C} \in (\mathbb{H}_\ell)^N$
A vector of shuffled, re-encrypted ciphertexts $\mathbf{C}' \in (\mathbb{H}_\ell)^N$
A Bayer-Groth $\mathsf{shuffleArgument}$ ▷ See algorithm 8.11 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: $0 < \ell \leq k$
Require: $2 \leq N \leq q - 3$

Operation:

1: $(m, n) \leftarrow \mathsf{GetMatrixDimensions}(N)$ ▷ See algorithm 8.5
2: $\mathbf{ck} \leftarrow \mathsf{GetVerifiableCommitmentKey}(n)$ ▷ See algorithm 8.6 ▷ 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 8.12

Output:

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

8.1 Pre-Requisites

8.1.1 Shuffle

Algorithm 8.3 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 8.3 `GenShuffle`: Re-encrypting shuffle

Context:

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

Input:

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

Require: $0 < \ell \leq k$

Operation:

1: $(\pi_0, \dots, \pi_{N-1}) \leftarrow \mathsf{GenPermutation}(N)$ ▷ See algorithm 8.4
2: **for** $i \in [0, N)$ **do**
3: $\quad r_i \leftarrow \mathsf{GenRandomInteger}(q)$ ▷ See algorithm 4.1
4: $\quad e \leftarrow \mathsf{GetCiphertext}(\vec{1}, r_i, \mathsf{pk})$ ▷ Ciphertext for a vector of ℓ 1s. See algorithm 7.5
5: $\quad \mathbf{C'}_i \leftarrow e \cdot \mathbf{C}_{\pi_i}$ ▷ See algorithm 7.8 for ciphertext multiplication
6: **end for**

Output:

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

Algorithm 8.4 provides a way to generate a random permutation of indices for a list of size N. It uses the algorithm formalized by Knuth in [22]. 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 8.4 `GenPermutation`: Permutation of indices up to N

Input:

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

Operation:

1: $\pi \leftarrow (0, \dots, N-1)$
2: **for** $i \in [0, N)$ **do**
3: $\quad \mathsf{offset} \leftarrow \mathsf{GenRandomInteger}(N-i)$ ▷ See algorithm 4.1
4: $\quad \mathsf{tmp} \leftarrow \pi_i$
5: $\quad \pi_i \leftarrow \pi_{i+\mathsf{offset}}$
6: $\quad \pi_{i+\mathsf{offset}} \leftarrow \mathsf{tmp}$
7: **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 $\pi \in \Sigma_N$

8.1.2 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 8.5 `GetMatrixDimensions`: Return the optimal dimensions for the ciphertext matrix

Input:

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

Operation:

1: $m \leftarrow 1$
2: $n \leftarrow N$
3: **for** $i \in [\lfloor \sqrt{N} \rfloor, 1)$ **do**
4: **if** $i \mid N$ **then**
5: $m \leftarrow i$
6: $n \leftarrow \frac{N}{i}$
7: **return** m, n
8: **end if**
9: **end for**

Output:

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

8.2 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 [14].

We use the Pedersen commitment scheme[30] with a commitment key $\mathbf{ck} = (h, g_1, ..., g_\nu)$ that was generated in a verifiable manner.

The Pedersen commitment scheme satisfies three properties that the Bayer-Groth mix net requires [4].

- *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 8.6 GetVerifiableCommitmentKey: Generates a verifiable commitment key

Context:

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

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

Input:

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

Require: $\nu \leq q - 3$

Operation:

1: $count = 0$
2: $i = 0$
3: $v \leftarrow \{\}$
4: **while** $count \leq \nu$ **do**
5: $u \leftarrow \mathsf{RecursiveHashToZq}(q, (q, \text{"}\texttt{commitmentKey}\text{"}, i, count))$
▷ See algorithm 4.9. q is both the upper bound and the first value to hash.
6: $w \leftarrow u^2 \bmod p$
7: **if** $w \notin \{0, 1, g\} \cup v$ **then**
8: $g_i \leftarrow w$
9: $v \leftarrow v \cup g_i$
10: $count = count + 1$
11: **end if**
12: $i = i + 1$
13: **end while**
14: $h \leftarrow g_0$ ▷ By convention, we designate g_0 as h

Output:

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

Test values for the algorithm 8.6 are provided in the attached get-verifiable-commitment-key.json file.

Algorithm 8.7 GetCommitment: Computes a commitment to a value

Context:

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

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

Input:

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

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

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

Require: $\ell > 0$

Operation:

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

Output:

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

Algorithm 8.8 GetCommitmentMatrix: Computes the commitment for a matrix

Context:

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

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

Input:

The values to be committed $A \in {\mathbb{Z}_q}^{n \times m}$ ▷ We note the columns of A as $\vec{a}_0, \dots, \vec{a}_{m-1}$

The random values to use $\mathbf{r} \in {\mathbb{Z}_q}^{m}$

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

Require: $m, n > 0$

Operation:

1: **for** $i \in [0, m)$ **do**

2: $c_i \leftarrow \mathsf{GetCommitment}(\vec{a}_i, r_i, \mathbf{ck})$ ▷ See algorithm 8.7

3: **end for**

Output:

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

This algorithm is consistent with the notation defined in [4], in section 2.3 Homomorphic Encryption:

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

Algorithm 8.9 GetCommitmentVector: Compute the commitment for a transposed vector. This is only used in the algorithm 8.22, hence the specific indices

Context:

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

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

Input:

The values to be commited $\mathbf{d} = (d_0, \dots, d_{2 \cdot m}) \in {\mathbb{Z}_q}^{2 \cdot m+1}$

The random values to use $\mathbf{t} = (t_0, \dots, t_{2 \cdot m}) \in {\mathbb{Z}_q}^{2 \cdot m+1}$

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

Operation:

1: $(c_0, \dots, c_{2 \cdot m}) \leftarrow \mathsf{GetCommitmentMatrix}(\mathbf{d}, \mathbf{t}, \mathbf{ck})$ ▷ See algorithm 8.8, with a single row of $2 \cdot m + 1$ columns

2: **return** $(c_0, \dots, c_{2 \cdot m})$

Output:

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

8.3 Arguments

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

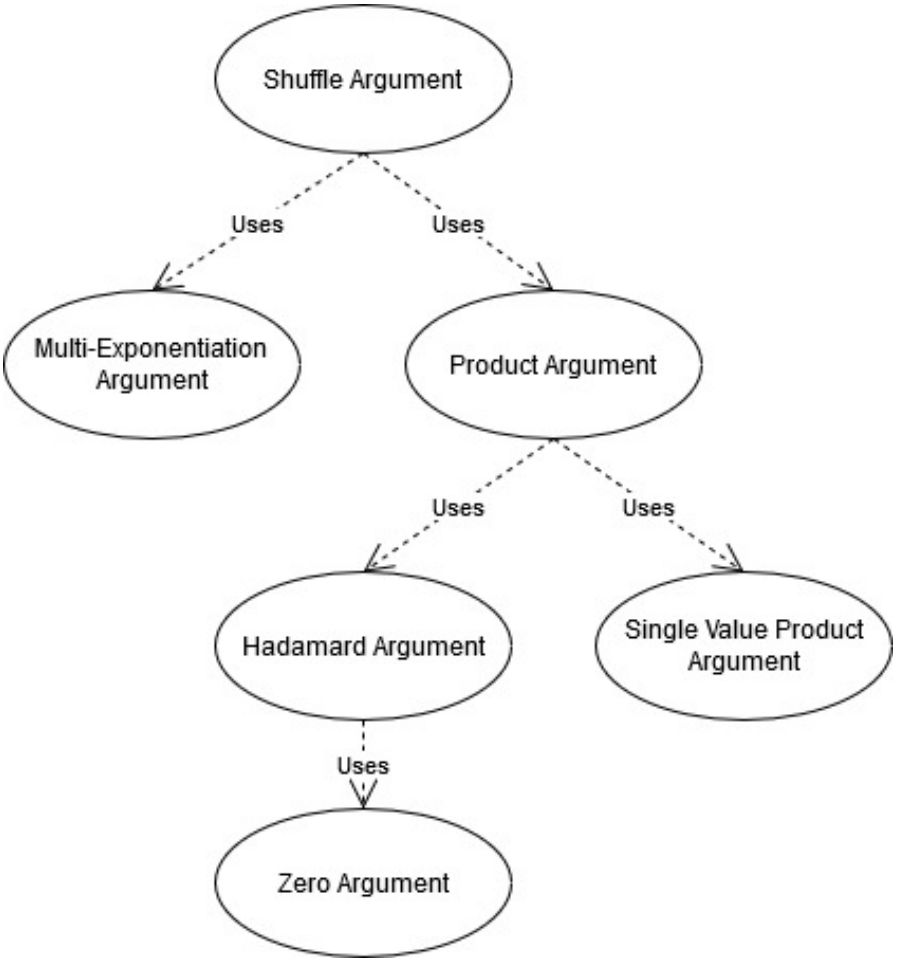

Figure 1: Bayer-Groth Argument for the Correctness of a Shuffle

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 8.10 StarMap: Defines the $\star$ bilinear map

Context:

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

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:

1: $s \leftarrow 0$
2: **for** $j \in [0, n)$ **do**
3: $\quad s \leftarrow s + a_j \cdot b_j \cdot y^{j+1}$ ▷ All operations are performed modulo q
4: **end for**

Output:

$s \in \mathbb{Z}_q$

Test values for the algorithm 8.10 are provided in the attached bilinearMap.json file.

8.3.1 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 $\vec{C} \in (\mathbb{H}_\ell)^N$ and $\vec{C'} \in (\mathbb{H}_\ell)^N$ 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 8.11 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. $p = 2 \cdot q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$
A commitment key $\mathbf{ck} = (h, g_1, ..., g_\nu) \in (\mathbb{G}_q \setminus \{1, g\})^{\nu+1}$

Input:

The statement composed of
- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_\ell)^N$ s.t. $0 < \ell \leq k$
- The shuffled and re-encrypted list of ciphertexts $\vec{C'} \in (\mathbb{H}_\ell)^N$
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. $2 \leq n \leq \nu$

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

Operation:

1: $\mathbf{r} \leftarrow \mathsf{GenRandomVector}(q, m)$ ▷ See algorithm 4.2
2: $A \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\{\pi(i)\}_{i=0}^{N-1}, m, n))$ ▷ Create a $n \times m$ matrix. See algorithm 8.14 and algorithm 8.13
3: $\mathbf{c}_A \leftarrow \mathsf{GetCommitmentMatrix}(A, \mathbf{r}, \mathbf{ck})$ ▷ See algorithm 8.8
4: $x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C'}, \mathbf{c}_A))$
5: $\mathbf{s} \leftarrow \mathsf{GenRandomVector}(q, m)$
6: $\mathbf{b} \leftarrow \{x^{\pi(i)}\}_{i=0}^{N-1}$
7: $B \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\mathbf{b}, m, n))$
8: $\mathbf{c}_B \leftarrow \mathsf{GetCommitmentMatrix}(B, \mathbf{s}, \mathbf{ck})$
9: $y \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{c}_B, p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C'}, \mathbf{c}_A))$
10: $z \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\text{“}1\text{”}, \mathbf{c}_B, p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C'}, \mathbf{c}_A))$
▷ Both $\vec{C}$ and $\vec{C'}$ are passed in the vector forms here
11: $\mathsf{Zneg} \leftarrow \mathsf{Transpose}(\mathsf{ToMatrix}(\{-z\}_{i=1}^{N}, m, n))$ ▷ Vector of length N, with all values being $q - z$
12: $\mathbf{c}_{-z} \leftarrow \mathsf{GetCommitmentMatrix}(\mathsf{Zneg}, \vec{0}, \mathbf{ck})$ ▷ A vector of length m, with all 0 values
13: $\mathbf{c}_D \leftarrow \mathbf{c}_A^y \mathbf{c}_B$ ▷ Entry-wise product
14: $D \leftarrow yA + B$
15: $\mathbf{t} \leftarrow y\mathbf{r} + \mathbf{s}$
16: $b \leftarrow \prod_{i=0}^{N-1}(yi + x^i - z)$
17: $\mathsf{pStatement} \leftarrow (\mathbf{c}_D \mathbf{c}_{-z}, b)$
18: $\mathsf{pWitness} \leftarrow (D + \mathsf{Zneg}, \mathbf{t})$
19: $\mathsf{productArgument} \leftarrow \mathsf{GetProductArgument}(\mathsf{pStatement}, \mathsf{pWitness})$ ▷ See algorithm 8.18
20: $\rho \leftarrow q - (\vec{\rho} \cdot \mathbf{b})$ ▷ Standard inner product $\sum_{i=0}^{N-1} \rho_i b_i$
21: $\vec{x} \leftarrow \{x^i\}_{i=0}^{N-1}$
22: $C \leftarrow \mathsf{GetCiphertextVectorExponentiation}(\vec{C}, \vec{x})$ ▷ See algorithm 7.7
23: $\mathsf{mStatement} \leftarrow (\mathsf{ToMatrix}(\vec{C'}, m, n), C, \mathbf{c}_B)$ ▷ See algorithm 8.13
24: $\mathsf{mWitness} \leftarrow (B, \mathbf{s}, \rho)$
25: $\mathsf{multiExponentiationArgument} \leftarrow \mathsf{GetMultiExponentiationArgument}(\mathsf{mStatement}, \mathsf{mWitness})$ ▷ See algorithm 8.15

Output:

shuffleArgument $(\mathbf{c}_A, \mathbf{c}_B, \mathsf{productArgument}, \mathsf{multiExponentiationArgument}) \in {\mathbb{G}_q}^m \times {\mathbb{G}_q}^m \times ... \times ...$
▷ See algorithm 8.18 and algorithm 8.15 for their respective domains

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

Algorithm 8.12 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. $p = 2 \cdot q + 1$
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
- The incoming list of ciphertexts $\vec{C} \in (\mathbb{H}_\ell)^N$ s.t. $0 < \ell \leq k$
- The shuffled and re-encrypted list of ciphertexts $\vec{C'} \in (\mathbb{H}_\ell)^N$
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 8.18
- a multiExponentiationArgument ▷ See algorithm 8.15
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. $2 \leq n \leq \nu$
Require: $N = mn$

Operation:
1: $x \leftarrow$ ByteArrayToInteger(RecursiveHash$(p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C'}, \mathbf{c}_A)$)
2: $y \leftarrow$ ByteArrayToInteger(RecursiveHash$(\mathbf{c}_B, p, q, \mathbf{pk}, \mathbf{ck}, \vec{C}, \vec{C'}, \mathbf{c}_A)$)
3: $z \leftarrow$ 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
4: Zneg $\leftarrow$ Transpose(ToMatrix$(\{-z\}_{i=1}^N, m, n)$) ▷ Vector of length N, with all values being $q - z$, see algorithm 8.14 and algorithm 8.13
5: $\mathbf{c}_{-_z} \leftarrow$ GetCommitmentMatrix(Zneg, $\vec{0}$, $\mathbf{ck}$) ▷ See algorithm 8.8
6: $\mathbf{c}_D \leftarrow \mathbf{c}_A^y \mathbf{c}_B$
7: $b \leftarrow \prod_{i=1}^N yi + x^i - z$
8: pStatement $\leftarrow (\mathbf{c}_D \mathbf{c}_{-_z}, b)$
9: productVerif $\leftarrow$ VerifyProductArgument(pStatement, productArgument) ▷ See algorithm 8.19
10: $\vec{x} \leftarrow \{x^i\}_{i=0}^{N-1}$
11: $C \leftarrow$ GetCiphertextVectorExponentiation$(\vec{C}, \vec{x})$ ▷ See algorithm 7.7
12: mStatement $\leftarrow$ (ToMatrix$(\vec{C'}, m, n), C, \mathbf{c}_B)$ ▷ See algorithm 8.13
13: multiVerif $\leftarrow$ VerifyMultiExponentiationArgument(mStatement, multiExponentiationArgument) ▷ See algorithm 8.16
14: **if** productVerif $\wedge$ multiVerif **then**
15: **return** $\top$
16: **else**
17: **return** $\bot$
18: **end if**

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

Test values for the algorithm 8.12 are provided in the attached verify-shuffle-argument.json file.

One of the key features of Bayer-Groth's[4] 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 8.13 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 $n \in \mathbb{N}^+$

Require: $N = m \cdot n$

Operation:

1: **for** $i \in [0, m)$ **do**
2: **for** $j \in [0, n)$ **do**
3: $M_{i,j} \leftarrow \vec{v}_{ni+j}$
4: **end for**
5: **end for**

Output:

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

Algorithm 8.14 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:

1: **for** $i \in [0, n)$ **do**
2: **for** $j \in [0, m)$ **do**
3: $N_{i,j} \leftarrow M_{j,i}$
4: **end for**
5: **end for**

Output:

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

8.3.2 Multi-Exponentiation Argument

Given ciphertexts $C_{0,0}, \dots, C_{m-1,n-1}$ and C, each $\in \mathbb{H}_\ell$, 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 8.15 GetMultiExponentiationArgument: Compute a multi-exponentiation argument

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
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 $(\vec{C}_0, \dots, \vec{C}_{m-1}) \in (\mathbb{H}_\ell)^{m \times n}$ s.t. $0 < \ell \leq k$ ▷ $\vec{C}_i$ refers to the i^{th} row
- ciphertext $C \in \mathbb{H}_\ell$
- 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 $\rho \in \mathbb{Z}_q$

Require: $C = \mathsf{GetCiphertext}(\vec{1}, \rho, \mathbf{pk}) \cdot \prod_{i=0}^{m-1} \vec{C}_i^{\vec{a}_{i+1}}$ ▷ Vector of 1s of length ℓ ▷ See algorithm 7.5, algorithm 7.7, algorithm 7.8
Require: $\vec{c}_A = \mathsf{GetCommitmentMatrix}(A, \vec{r}, \mathbf{ck})$
Require: $n, m > 0$

Operation:
1: $\vec{a}_0 \leftarrow \mathsf{GenRandomVector}(q, n)$ ▷ See algorithm 4.2
2: $r_0 \leftarrow \mathsf{GenRandomInteger}(q)$ ▷ See algorithm 4.1
3: $(b_0, \dots, b_{2 \cdot m-1}) \leftarrow \mathsf{GenRandomVector}(q, 2 \cdot m)$
4: $(s_0, \dots, s_{2 \cdot m-1}) \leftarrow \mathsf{GenRandomVector}(q, 2 \cdot m)$
5: $(\tau_0, \dots, \tau_{2 \cdot m-1}) \leftarrow \mathsf{GenRandomVector}(q, 2 \cdot m)$
6: $b_m \leftarrow 0$
7: $s_m \leftarrow 0$
8: $\tau_m \leftarrow \rho$ ▷ Ensuring $c_{B_m} = \mathsf{GetCommitment}(0, 0, \mathbf{ck})$ and $\mathsf{GetCiphertext}(\vec{g}^{b_m}, \tau_m, \mathbf{pk}) = \mathsf{GetCiphertext}(\vec{1}, \rho, \mathbf{pk})$
9: $c_{A_0} \leftarrow \mathsf{GetCommitment}(\vec{a}_0, r_0, \mathbf{ck})$ ▷ See algorithm 8.7
10: $(D_0, \dots, D_{2 \cdot m-1}) \leftarrow \mathsf{GetDiagonalProducts}((\vec{C}_0, \dots, \vec{C}_{m-1}), (\vec{a}_0, \dots, \vec{a}_m))$ ▷ See algorithm 8.17
11: **for** $k \in [0, 2 \cdot m)$ **do**
12: $\quad c_{B_k} \leftarrow \mathsf{GetCommitment}((b_k), s_k, \mathbf{ck})$
13: $\quad E_k \leftarrow \mathsf{GetCiphertext}(\overrightarrow{g^{b_k}}, \tau_k, \mathbf{pk}) \cdot D_k$ ▷ See algorithm 7.5, we take a vector of messages of length ℓ each having value g^{b_k}
14: **end for**
15: $x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, (\vec{C}_i)_{i=0}^{m-1}, C, \vec{c}_A, c_{A_0}, (c_{B_k})_{k=0}^{2 \cdot m-1}, (E_k)_{k=0}^{2 \cdot m-1}))$
▷ All operations below are performed modulo q
16: $\vec{a} \leftarrow \vec{a}_0 + \sum_{i=1}^{m} x^i \vec{a}_i$
17: $r \leftarrow r_0 + \sum_{i=1}^{m} x^i r_i$
18: $b \leftarrow b_0 + \sum_{k=1}^{2 \cdot m-1} x^k b_k$
19: $s \leftarrow s_0 + \sum_{k=1}^{2 \cdot m-1} x^k s_k$
20: $\tau \leftarrow \tau_0 + \sum_{k=1}^{2 \cdot m-1} x^k \tau_k$

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

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

Algorithm 8.16 VerifyMultiExponentiationArgument: Verify a multi-exponentiation argument

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
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 $(\vec{C}_0, \dots, \vec{C}_{m-1}) \in (\mathbb{H}_\ell)^{m \times n}$ ▷ $\vec{C}_i$ refers to the i^{th} row
- ciphertext $C \in \mathbb{H}_\ell$
- commitment vector $\vec{c}_A = (c_{A_1}, \dots, c_{A_m}) \in {\mathbb{G}_q}^m$

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

Require: $n, m > 0$

Operation:

1: $x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, \{\vec{C}_i\}_{i=0}^{m-1}, C, \vec{c}_A, c_{A_0}, \{c_{B_k}\}_{k=0}^{2 \cdot m-1}, \{E_k\}_{k=0}^{2 \cdot m-1}))$
2: $\mathsf{verifCbm} \leftarrow c_{B_m} = 1$
3: $\mathsf{verifEm} \leftarrow E_m = C$

4: $\mathsf{prodCa} \leftarrow c_{A_0} \prod_{i=1}^{m} c_{A_i}^{x^i}$
5: $\mathsf{commA} \leftarrow \mathsf{GetCommitment}(\vec{a}, r, \mathbf{ck})$ ▷ See algorithm 8.7
6: $\mathsf{verifA} \leftarrow \mathsf{prodCa} = \mathsf{commA}$

7: $\mathsf{prodCb} \leftarrow c_{B_0} \prod_{k=1}^{2 \cdot m-1} c_{B_k}^{x^k}$
8: $\mathsf{commB} \leftarrow \mathsf{GetCommitment}((b), s, \mathsf{ck})$
9: $\mathsf{verifB} \leftarrow \mathsf{prodCb} = \mathsf{commB}$

10: $\mathsf{prodE} \leftarrow E_0 \prod_{k=1}^{2 \cdot m-1} E_k^{x^k}$
11: $\mathsf{encryptedGb} \leftarrow \mathsf{GetCiphertext}(\vec{g}^b, \tau, \mathbf{pk})$ ▷ See algorithm 7.5, we take a vector of messages of length ℓ each having value g^b
12: $\mathsf{prodC} \leftarrow \prod_{i=0}^{m-1} \mathsf{GetCiphertextVectorExponentiation}(\vec{C}_i, x^{(m-i)-1}\vec{a})$ ▷ See algorithm 7.7
13: $\mathsf{verifEC} \leftarrow \mathsf{prodE} = \mathsf{GetCiphertextProduct}(\mathsf{encryptedGb}, \mathsf{prodC})$ ▷ See algorithm 7.8

14: **if** $\mathsf{verifCbm} \wedge \mathsf{verifEm} \wedge \mathsf{verifA} \wedge \mathsf{verifB} \wedge \mathsf{verifEC}$ **then**
15: **return** $\top$
16: **else**
17: **return** $\bot$
18: **end if**

Output:

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

Test values for the algorithm 8.16 are provided in the attached verify-multiexp-argument.json file.

Algorithm 8.17 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. $p = 2 \cdot q + 1$
Group generator $g \in \mathbb{G}_q$
A multi-recipient public key $\mathbf{pk} \in \mathbb{G}_q^k$

Input:

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

Operation:

1: **for** $k \in [0, 2 \cdot m)$ **do**
2: $d_k \leftarrow \mathsf{GetCiphertext}(\vec{1}, 0, \mathbf{pk})$ ▷ Neutral element of ciphertext multiplication
3: **if** $k < m$ **then**
4: $\mathsf{lowerbound} \leftarrow m - k - 1$
5: $\mathsf{upperbound} \leftarrow m$
6: **else**
7: $\mathsf{lowerbound} \leftarrow 0$
8: $\mathsf{upperbound} \leftarrow 2 \cdot m - k$
9: **end if**
10: **for** $i \in [\mathsf{lowerbound}, \mathsf{upperbound})$ **do**
11: $j \leftarrow k - m + i + 1$
12: $d_k \leftarrow \mathsf{GetCiphertextProduct}(d_k, \mathsf{GetCiphertextVectorExponentiation}(\vec{C}_i, \vec{a}_j))$ ▷ See algorithm 7.8 and algorithm 7.7
13: **end for**
14: **end for**

Output:

Diagonal products $D = (d_0, \dots, d_{2 \cdot m - 1}) \in {\mathbb{H}_\ell}^{2 \cdot m}$

8.3.3 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 8.3.4), 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 8.3.6).

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 8.18 GetProductArgument: Computes a Product Argument

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
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 $\vec{r} \in {\mathbb{Z}_q}^m$
Require: $2 \leq n \leq \nu$
Require: $m > 0$
Require: $\vec{c}_A = \mathsf{GetCommitmentMatrix}(A, \vec{r}, \mathbf{ck})$ ▷ See algorithm 8.8
Require: $b = \prod_{i=0}^{n-1} \prod_{j=0}^{m-1} a_{i,j} \mod q$

Operation:
1: **if** $m > 1$ **then**
2: $s \leftarrow \mathsf{GenRandomInteger}(q)$
3: **for** $i \in [0, n)$ **do**
4: $b_i \leftarrow \prod_{j=0}^{m-1} a_{i,j}$
5: **end for**
6: $c_b \leftarrow \mathsf{GetCommitment}((b_0, \dots, b_{n-1}), s, \mathbf{ck})$
7: $\mathsf{hStatement} \leftarrow (\vec{c}_A, c_b)$
8: $\mathsf{hWitness} \leftarrow (A, (b_0, \dots, b_{n-1}), \vec{r}, s)$
9: $\mathsf{hadamardArg} \leftarrow \mathsf{GetHadamardArgument}(\mathsf{hStatement}, \mathsf{hWitness})$ ▷ See algorithm 8.20
10: $\mathsf{sStatement} \leftarrow (c_b, b)$
11: $\mathsf{sWitness} \leftarrow ((b_0, \dots, b_{n-1}), s)$
12: $\mathsf{singleValueProdArg} \leftarrow \mathsf{GetSingleValueProductArgument}(\mathsf{sStatement}, \mathsf{sWitness})$ ▷ See algorithm 8.25
13: **else**
14: $\mathsf{sStatement} \leftarrow (c_{A_1}, b)$
15: $\mathsf{sWitness} \leftarrow (\vec{a}_0, r_0)$
16: $\mathsf{singleValueProdArg} \leftarrow \mathsf{GetSingleValueProductArgument}(\mathsf{sStatement}, \mathsf{sWitness})$ ▷ See algorithm 8.25
17: **end if**

Output:
18: **if** $m > 1$ **then**
$(c_b, \mathsf{hadamardArg}, \mathsf{singleValueProdArg}) \in \mathbb{G}_q \times \mathsf{HadamardArgument} \times \mathsf{SingleValueProductArgument}$
▷ See algorithm 8.20 and algorithm 8.25 for the domains
19: **else**
singleValueProdArg ▷ See algorithm 8.25 for the domain
20: **end if**

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

Algorithm 8.19 VerifyProductArgument: Verify a Product argument

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
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 argument composed of
- the commitment $c_b \in \mathbb{G}_q$ ▷ omitted if $m = 1$
- a hadamardArg $\in {\mathbb{G}_q}^{m+1} \times (\mathbb{G}_q \times \mathbb{G}_q \times {\mathbb{G}_q}^{2 \cdot m+1} \times {\mathbb{Z}_q}^n \times {\mathbb{Z}_q}^n \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q)$ ▷ omitted if $m = 1$
- a singleValueProductArg $\in \mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q \times {\mathbb{Z}_q}^n \times {\mathbb{Z}_q}^n \times \mathbb{Z}_q \times \mathbb{Z}_q$

Require: $n \leq \nu$

Operation:

1: **if** $m > 1$ **then**
2: hStatement $\leftarrow (\vec{c}_A, c_b)$
3: sStatement $\leftarrow (c_b, b)$
4: **if** VerifyHadamardArgument(hStatement, hadamardArg)$\wedge$
5: VerifySingleValueProductArgument(sStatement, singleValueProductArg) **then** ▷ See algorithm 8.21 and algorithm 8.26
return $\top$
6: **else**
return $\bot$
7: **end if**
8: **else**
9: sStatement $\leftarrow (c_{A_1}, b)$
10: **if** VerifySingleValueProductArgument(sStatement, singleValueProductArg) **then** ▷ See algorithm 8.26
return $\top$
11: **else**
return $\bot$
12: **end if**
13: **end if**

Output:

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

Test values for the algorithm 8.19 are provided in the attached verify-p-argument.json file.

8.3.4 Hadamard Argument

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

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

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

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 8.20 GetHadamardArgument: Computes a Hadamard Argument

Context:

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

Require: $m \geq 2$ ▷ Hadamard product only makes sense for $m \geq 2$
Require: $0 < n \leq \nu$
Require: $\mathbf{c}_A = \mathsf{GetCommitmentMatrix}(A, \mathbf{r}, \mathbf{ck})$ ▷ See algorithm 8.8
Require: $c_b = \mathsf{GetCommitment}(\mathbf{b}, s, \mathbf{ck})$ ▷ See algorithm 8.7
Require: $\vec{b} = \prod_{j=0}^{m-1} \vec{a}_j$ ▷ Uses the Hadamard product, ie $b_i = \prod_{j=0}^{m-1} a_{i,j}$

Operation:

for $j \in [0, m)$ **do**
  $\vec{b}_j \leftarrow \prod_{i=0}^{j} \vec{a}_i$ ▷ Which implies that $\vec{b}_0 = \vec{a}_0$ and $\vec{b}_{m-1} = \vec{b}$
end for
$s_0 \leftarrow r_0$ ▷ Thus ensuring that $\mathsf{GetCommitment}(\vec{b_0}, s_0, \mathbf{ck}) = c_{A_0}$
if $m > 2$ **then**
  $(s_1, \dots, s_{m-2}) \leftarrow \mathsf{GenRandomVector}(q, m-2)$
end if ▷ See algorithm 4.2
$s_{m-1} \leftarrow s$ ▷ Thus ensuring that $\mathsf{GetCommitment}(\vec{b}_{m-1}, s_{m-1}, \mathbf{ck}) = c_b$
$c_{B_0} \leftarrow c_{A_0}$
for $j \in [1, m-1)$ **do**
  $c_{B_j} \leftarrow \mathsf{GetCommitment}(\vec{b}_j, s_j, \mathbf{ck})$
end for
$c_{B_{m-1}} \leftarrow c_b$
$x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, \mathbf{c}_A, c_b, (c_{B_0}, \dots, c_{B_{m-1}})))$
$y \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\text{“}1\text{”}, p, q, \mathbf{pk}, \mathbf{ck}, \mathbf{c}_A, c_b, (c_{B_0}, \dots, c_{B_{m-1}})))$
▷ Use y to define $\star : {\mathbb{Z}_q}^n \times {\mathbb{Z}_q}^n \mapsto \mathbb{Z}_q$, see algorithm 8.10
▷ All exponentiations of x below are performed modulo q
for $i \in [0, m-1)$ **do**
  $\vec{d}_i = x^{i+1} \vec{b}_i$
  $c_{D_i} = c_{B_i}^{x^{i+1}}$
  $t_i = x^{i+1} s_i$
end for
$\vec{d} \leftarrow \sum_{i=1}^{m-1} x^i \vec{b}_i$
$c_D \leftarrow \prod_{i=1}^{m-1} c_{B_i}^{x^i}$
$t \leftarrow \sum_{i=1}^{m-1} x^i s_i$
$-\vec{1} \leftarrow (q-1, \dots, q-1) \in {\mathbb{Z}_q}^n$
$c_{-1} \leftarrow \mathsf{GetCommitment}(-\vec{1}, 0, \mathbf{ck})$
$\mathsf{statement} \leftarrow ((c_{A_1}, \dots, c_{A_{m-1}}, c_{-1}), (c_{D_0}, \dots, c_{D_{m-2}}, c_D), y)$
$\mathsf{witness} \leftarrow ((\vec{a}_1, \dots, \vec{a}_{m-1}, -\vec{1}), (\vec{d}_0, \dots, \vec{d}_{m-2}, \vec{d}), (r_1, \dots, r_{m-1}, 0), (t_0, \dots, t_{m-2}, t))$
$\mathsf{zeroArg} \leftarrow \mathsf{GetZeroArgument}(\mathsf{statement}, \mathsf{witness})$ ▷ See algorithm 8.22
▷ Provide an argument that $\sum_{i=0}^{m-2} \vec{a}_{i+1} \star \vec{d}_i - \vec{1} \star \vec{d} = 0$

Output:

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

Algorithm 8.21 VerifyHadamardArgument: Verifies a Hadamard Argument

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
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 argument composed of
- commitment vector $\mathbf{c}_B = (c_{B_0}, \dots, c_{B_{m-1}}) \in {\mathbb{G}_q}^m$
- a zero argument, composed of
  - commitment $c_{A_0} \in \mathbb{G}_q$
  - commitment $c_{B_m} \in \mathbb{G}_q$
  - commitment vector $\mathbf{c}_d \in {\mathbb{G}_q}^{2 \cdot m + 1}$
  - 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 $t' \in \mathbb{Z}_q$

Require: $n > 0$

Operation:

1: $x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, \mathbf{c}_A, c_b, (c_{B_0}, \dots, c_{B_{m-1}})))$
2: $y \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\text{“}1\text{”}, p, q, \mathbf{pk}, \mathbf{ck}, \mathbf{c}_A, c_b, (c_{B_0}, \dots, c_{B_{m-1}})))$
3: **for** $i \in [0, m-1)$ **do**
4: $\quad c_{D_i} \leftarrow c_{B_i}^{x^{i+1}}$
5: **end for**
6: $c_D \leftarrow \prod_{i=1}^{m-1} c_{B_i}^{x^i}$
7: $-\vec{1} \leftarrow (q-1, \dots, q-1) \in {\mathbb{Z}_q}^n$
8: $c_{-1} \leftarrow \mathsf{GetCommitment}(-\vec{1}, 0, \mathbf{ck})$
9: $\mathsf{zeroStatement} \leftarrow ((c_{A_1}, \dots, c_{A_{m-1}}, c_{-1}), (c_{D_0}, \dots, c_{D_{m-2}}, c_D), y)$
10: $\mathsf{zeroArgument} \leftarrow (c_{A_0}, c_{B_m}, \mathbf{c}_d, \mathbf{a}', \mathbf{b}', r', s', t')$
11: **if** $c_{B_0} = c_{A_0} \wedge c_{B_{m-1}} = c_b \wedge \mathsf{VerifyZeroArgument}(\mathsf{zeroStatement}, \mathsf{zeroArgument})$ **then** ▷ See algorithm 8.23
$\quad$ **return** $\top$
12: **else**
$\quad$ **return** $\bot$
13: **end if**

Output:

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

Test values for the algorithm 8.21 are provided in the attached verify-h-argument.json file.

8.3.5 Zero Argument

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

Algorithm 8.22 GetZeroArgument: Computes a Zero Argument

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
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 $\mathbf{c}_A \in {\mathbb{G}_q}^m$
- commitments $\mathbf{c}_B \in {\mathbb{G}_q}^m$
- the value $y \in \mathbb{Z}_q$ defining the bilinear mapping $\star$ ▷ See algorithm 8.10
The witness composed of
- matrix $A = (\vec{a}_1, \dots, \vec{a}_m) \in {\mathbb{Z}_q}^{n \times m}$ ▷ The $\vec{a}_i$ values correspond to the columns of A
- matrix $B = (\vec{b}_0, \dots, \vec{b}_{m-1}) \in {\mathbb{Z}_q}^{n \times m}$ ▷ The $\vec{b}_i$ values correspond to the columns of B
- vector of exponents $\mathbf{r} = (r_1, \dots, r_m) \in {\mathbb{Z}_q}^m$
- vector of exponents $\mathbf{s} = (s_0, \dots, s_{m-1}) \in {\mathbb{Z}_q}^m$
Require: $\mathbf{c}_A = \mathsf{GetCommitmentMatrix}(A, \mathbf{r}, \mathbf{ck})$ ▷ See algorithm 8.8
Require: $\mathbf{c}_B = \mathsf{GetCommitmentMatrix}(B, \mathbf{s}, \mathbf{ck})$ ▷ See algorithm 8.8
Require: $\sum_{i=1}^{m} \mathbf{a}_i \star \mathbf{b}_{i-1} = 0$
Require: $n, m > 0$

Operation:
1: $\vec{a}_0 \leftarrow \mathsf{GenRandomVector}(q, n)$ ▷ See algorithm 4.2
2: $\vec{b}_m \leftarrow \mathsf{GenRandomVector}(q, n)$
3: $r_0 \leftarrow \mathsf{GenRandomInteger}(q)$
4: $s_m \leftarrow \mathsf{GenRandomInteger}(q)$
5: $c_{A_0} \leftarrow \mathsf{GetCommitment}(\mathbf{a}_0, r_0, \mathbf{ck})$
6: $c_{B_m} \leftarrow \mathsf{GetCommitment}(\mathbf{b}_m, s_m, \mathbf{ck})$
7: $\mathbf{d} = (d_0, \dots, d_{2 \cdot m}) \leftarrow \mathsf{ComputeDVector}((\vec{a}_0, \dots, \vec{a}_m), (\vec{b}_0, \dots, \vec{b}_m), y)$ ▷ See algorithm 8.24
8: $\mathbf{t} \leftarrow \mathsf{GenRandomVector}(q, 2 \cdot m + 1)$
9: $t_{m+1} \leftarrow 0$
10: $\mathbf{c_d} \leftarrow \mathsf{GetCommitmentVector}((d_0, \dots, d_{2 \cdot m}), \mathbf{t}, \mathbf{ck})$ ▷ See algorithm 8.9
11: $x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, c_{A_0}, c_{B_m}, \mathbf{c_d}, \mathbf{c}_B, \mathbf{c}_A))$ ▷ See algorithm 3.8 and algorithm 4.8
▷ Below this point, all operations are performed modulo q
12: **for** $j \in [0, n)$ **do**
13: $\quad \mathbf{a'}_j \leftarrow \sum_{i=0}^{m} x^i \cdot \vec{a}_{j,i}$
14: $\quad \mathbf{b'}_j \leftarrow \sum_{i=0}^{m} x^{m-i} \cdot \vec{b}_{j,i}$
15: **end for**
16: $r' \leftarrow \sum_{i=0}^{m} x^i \cdot r_i$
17: $s' \leftarrow \sum_{i=0}^{m} x^{m-i} \cdot s_i$
18: $t' \leftarrow \sum_{i=0}^{2 \cdot m} x^i \cdot t_i$

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

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 $\bot$ for false.

Algorithm 8.23 VerifyZeroArgument: Verifies a Zero Argument

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
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 8.10

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 $\mathbf{c}_d = (c_{d_0}, \dots, c_{d_{2 \cdot m}}) \in {\mathbb{G}_q}^{2 \cdot m+1}$
- 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: $\mathsf{prodCd} \leftarrow \prod_{i=0}^{2 \cdot m} c_{d_i}^{x^i}$ ▷ 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 8.10 with value y
11: $\mathsf{commD} \leftarrow \mathsf{GetCommitment}((\mathsf{prod}), t', \mathbf{ck})$
12: $\mathsf{verifD} \leftarrow \mathsf{prodCd} = \mathsf{commD}$
13: **if** $\mathsf{verifCd} \wedge \mathsf{verifA} \wedge \mathsf{verifB} \wedge \mathsf{verifD}$ **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 8.23 are provided in the attached verify-za-argument.json file.

Algorithm 8.24 ComputeDVector: Compute the vector $\mathbf{d}$ for the algorithm 8.22

Context:

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

Input:

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

Operation:

1: **for** $k \in [0, 2 \cdot m]$ **do**
2: $d_k \leftarrow 0$
3: **for** $i \in [\max(0, k - m), m]$ **do**
4: $j \leftarrow (m - k) + i$
5: **if** $j > m$ **then**
6: break from loop and proceed with next k
7: **end if**
8: $d_k \leftarrow d_k + \vec{a}_i \star \vec{b}_j$ ▷ See algorithm 8.10, addition is modulo q
9: **end for**
10: **end for**

Output:

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

8.3.6 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 8.25 GetSingleValueProductArgument: Computes a Single Value Product Argument

Context:
Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
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 $r \in \mathbb{Z}_q$
Require: $n \geq 2$
Require: $c_a = \mathsf{GetCommitment}(\mathbf{a}, r, \mathbf{ck})$ ▷ See algorithm 8.7
Require: $b = \prod_{i=0}^{n-1} a_i \mod q$

Operation:
1: **for** $k \in [0, n)$ **do**
2: $b_k \leftarrow \prod_{i=0}^{k} a_i \mod q$
3: **end for**
4: $(d_0, \dots, d_{n-1}) \leftarrow \mathsf{GenRandomVector}(q, n)$ ▷ See algorithm 4.2
5: $r_d \leftarrow \mathsf{GenRandomInteger}(q)$
6: $\delta_0 \leftarrow d_0$
7: **if** $n > 2$ **then**
8: $(\delta_1, \dots, \delta_{n-2}) \leftarrow \mathsf{GenRandomVector}(q, n-2)$
9: **end if**
10: $\delta_{n-1} \leftarrow 0$
11: $s_0 \leftarrow \mathsf{GenRandomInteger}(q)$
12: $s_x \leftarrow \mathsf{GenRandomInteger}(q)$
13: **for** $k \in [0, n-1)$ **do**
14: $\delta'_k \leftarrow -\delta_k d_{k+1} \mod q$
15: $\Delta_k \leftarrow \delta_{k+1} - a_{k+1}\delta_k - b_k d_{k+1} \mod q$
16: **end for**
17: $c_d \leftarrow \mathsf{GetCommitment}((d_0, \dots, d_{n-1}), r_d, \mathbf{ck})$ ▷ See algorithm 8.7
18: $c_\delta \leftarrow \mathsf{GetCommitment}((\delta'_0, \dots, \delta'_{n-2}), s_0, \mathbf{ck})$
19: $c_\Delta \leftarrow \mathsf{GetCommitment}((\Delta_0, \dots, \Delta_{n-2}), s_x, \mathbf{ck})$
20: $x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, c_\Delta, c_\delta, c_d, b, c_a))$
21: **for** $k \in [0, n)$ **do**
22: $\tilde{a}_k \leftarrow x \cdot a_k + d_k \mod q$
23: $\tilde{b}_k \leftarrow x \cdot b_k + \delta_k \mod q$
24: **end for**
25: $\tilde{r} \leftarrow x \cdot r + r_d \mod q$
26: $\tilde{s} \leftarrow x \cdot s_x + s_0 \mod q$

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 8.26 VerifySingleValueProductArgument: Verifies a Single Value Product Argument

Context:

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

1: $x \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(p, q, \mathbf{pk}, \mathbf{ck}, c_\Delta, c_\delta, c_d, b, c_a))$
2: $\mathsf{prodCa} \leftarrow c_a^x \cdot c_d$
3: $\mathsf{commA} \leftarrow \mathsf{GetCommitment}(\tilde{a}, \tilde{r}, \mathbf{ck})$
4: $\mathsf{verifA} \leftarrow \mathsf{prodCa} = \mathsf{commA}$
5: $\mathsf{prodDelta} \leftarrow c_\Delta^x \cdot c_\delta$
6: **for** $i \in [0, n-1)$ **do**
7: $\quad e_i \leftarrow x \cdot \tilde{b}_{i+1} - \tilde{b}_i \cdot \tilde{a}_{i+1}$
8: **end for**
9: $\mathsf{commDelta} \leftarrow \mathsf{GetCommitment}((e_0, \dots, e_{n-2}), \tilde{s}, \mathbf{ck})$
10: $\mathsf{verifDelta} \leftarrow \mathsf{prodDelta} = \mathsf{commDelta}$
11: $\mathsf{verifB} \leftarrow \tilde{b}_0 = \tilde{a}_0 \wedge \tilde{b}_{n-1} = x \cdot b$
12: **if** $\mathsf{verifA} \wedge \mathsf{verifDelta} \wedge \mathsf{verifB}$ **then**
$\quad$ **return** $\top$
13: **else**
$\quad$ **return** $\bot$
14: **end if**

Output:

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

Test values for the algorithm 8.26 are provided in the attached verify-svp-argument.json file.

9 Zero-Knowledge Proofs

9.1 Introduction

This section introduces various Zero-Knowledge Proofs of Knowledge, based on the work by Maurer[24]. We extensively document and formalize the zero-knowledge proof system's security—including the non-interactive case—in the computational proof [31]. 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 4.8. 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.

9.2 Schnorr proof

In this section we provide a proof of knowledge of a discrete logarithm, also known as a Schnorr proof. Given the values x, y, g and p such that $x \equiv \log_g(y) \pmod{p}$, we want to prove knowledge of x without revealing its value.

In this case, the phi-function is $x \mapsto g^x \bmod p$, with domain $(\mathbb{Z}_q, +)$ and co-domain $(\mathbb{G}_q, \times)$. As such, the operations given as $\star$ consist of additions modulo q and the "exponentiation" used in the computation of z is a multiplication; whereas the operation noted as $\otimes$ is a multiplication modulo p, and the exponentiation given in the computation of c' is a modular exponentiation in $\mathbb{G}_q$.

Algorithm 9.1 ComputePhiSchnorr: compute the phi-function for a Schnorr proof

Context:

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

Input:

An exponent $x \in \mathbb{Z}_q$

Operation:

1: $y \leftarrow g^x \bmod p$
2: **return** y

Output:

The power $y \in \mathbb{G}_q$

Algorithm 9.2 GenSchnorrProof: generate a proof of knowledge of a discrete logarithm

Context:

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

Input:

The witness – a secret exponent $x \in \mathbb{Z}_q$
The statement – a power $y \in \mathbb{G}_q$ s.t. $y = g^x$
An array of optional additional information $\mathbf{i}_{\text{aux}} \in (\mathbb{A}_{UCS}{}^*)^*$

Operation:

1: $b \leftarrow \mathsf{GenRandomInteger}(q)$ ▷ See algorithm 4.1
2: $c \leftarrow \mathsf{ComputePhiSchnorr}(b)$ ▷ See algorithm 9.1
3: $\mathbf{f} \leftarrow (p, q, g)$
4: $\mathbf{h}_{\text{aux}} \leftarrow (\text{"}\texttt{SchnorrProof}\text{"}, \mathbf{i}_{\text{aux}})$ ▷ If $\mathbf{i}_{\text{aux}}$ is empty, we omit it
5: $e \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{f}, y, c, \mathbf{h}_{\text{aux}}))$ ▷ See algorithms 3.8 and 4.8
6: $z \leftarrow b + e \cdot x \bmod q$

Output:

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

Algorithm 9.3 VerifySchnorr: Verifies the validity of a Schnorr proof

Context:

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

Input:

The proof $(e, z) \in \mathbb{Z}_q \times \mathbb{Z}_q$
The statement – a power $y \in \mathbb{G}_q$ s.t. $y = g^x$
An array of optional additional information $\mathbf{i}_{\text{aux}} \in (\mathbb{A}_{UCS}{}^*)^*$

Operation:

1: $\mathbf{x} \leftarrow \mathsf{ComputePhiSchnorr}(z)$ ▷ See algorithm 9.1
2: $\mathbf{f} \leftarrow (p, q, g)$
3: $c' \leftarrow x \cdot y^{-e} \bmod p$
4: $\mathbf{h}_{\text{aux}} \leftarrow (\text{“}\texttt{SchnorrProof}\text{”}, \mathbf{i}_{\text{aux}})$ ▷ If $\mathbf{i}_{\text{aux}}$ is empty, we omit it
5: $h \leftarrow \mathsf{RecursiveHash}(\mathbf{f}, y, c', \mathbf{h}_{\text{aux}})$ ▷ See algorithm 4.8
6: $e' \leftarrow \mathsf{ByteArrayToInteger}(h)$ ▷ See algorithm 3.8
7: **if** $e = e'$ **then**
    return $\top$
8: **else**
    return $\bot$
9: **end if**

Output:

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

Test values for the algorithm 9.3 are provided in the attached verify-schnorr.json file.

9.3 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 9.4.

Algorithm 9.4 ComputePhiDecryption: Compute the phi-function for decryption

Context:

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

Input:

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

Operation:

1: **for** $i \in [0, \ell)$ **do**
2: $y_i \leftarrow g^{x_i}$ ▷ $y_i = \mathsf{pk}_i$ when $x_i = \mathsf{sk}_i$
3: $y_{\ell+i} \leftarrow \gamma^{x_i}$ ▷ $y_{\ell+i} = g^{\mathsf{sk}_i \cdot r} = \frac{\phi_i}{m_i}$ when $\gamma = g^r$ and $x_i = \mathsf{sk}_i$
4: **end for**

▷ All symbols used in the comments above are aligned with algorithms 7.4 and 7.5

Output:

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

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

Generating and verifying decryption proofs The algorithms below are the adaptations of the general case presented in section 9.1, with explicit domains and operations. Our phi-function defined in algorithm 9.4 has domain $({\mathbb{Z}_q}^{\ell}, +)$ and co-domain $({\mathbb{G}_q}^{2\ell}, \times)$. 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 9.5 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. $p = 2 \cdot q + 1$
Group generator $g \in \mathbb{G}_q$

Input:

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

Require: $0 < \ell \leq k$

Operation:

1: $\mathbf{b} \leftarrow \mathsf{GenRandomVector}(q, \ell)$ ▷ See algorithm 4.2
2: $\mathbf{c} \leftarrow \mathsf{ComputePhiDecryption}(\mathbf{b}, \gamma)$ ▷ See algorithm 9.4
3: $\mathbf{f} \leftarrow (p, q, g, \gamma)$
4: **for** $i \in [0, \ell)$ **do**
5: $\quad y_i \leftarrow \mathsf{pk}_i$
6: $\quad y_{\ell+i} \leftarrow \frac{\phi_i}{m_i}$
7: **end for**
8: $\mathbf{h}_{\mathsf{aux}} \leftarrow (\text{"}\texttt{DecryptionProof}\text{"}, (\phi_0, \dots, \phi_{\ell-1}), \mathbf{m}, \mathbf{i}_{\mathsf{aux}})$ ▷ If $\mathbf{i}_{\mathsf{aux}}$ is empty, we omit it
9: $e \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{f}, (y_0, \dots, y_{2\cdot\ell-1}), \mathbf{c}, \mathbf{h}_{\mathsf{aux}}))$ ▷ See algorithms 3.8 and 4.8
10: $\mathbf{sk}' \leftarrow (\mathsf{sk}_0, \dots, \mathsf{sk}_{\ell-1})$
11: $\mathbf{z} \leftarrow \mathbf{b} + e \cdot \mathbf{sk}'$

Output:

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

Algorithm 9.6 VerifyDecryption: Verifies the validity of a decryption proof

Context:

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

Input:

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

Require: $0 < \ell \leq k$

Operation:

1: $\mathbf{x} \leftarrow \mathsf{ComputePhiDecryption}(\mathbf{z}, \gamma)$ ▷ See algorithm 9.4
2: $\mathbf{f} \leftarrow (p, q, g, \gamma)$
3: **for** $i \in [0, \ell)$ **do**
4: $\quad y_i \leftarrow \mathsf{pk}_i$
5: $\quad y_{\ell+i} \leftarrow \frac{\phi_i}{m_i}$
6: **end for**
7: **for** $i \in [0, 2 \cdot \ell)$ **do**
8: $\quad c'_i \leftarrow x_i y_i^{-e}$
9: **end for**
10: $\mathbf{h}_{\mathsf{aux}} \leftarrow (\text{“}\texttt{DecryptionProof}\text{”}, (\phi_0, \dots, \phi_{\ell-1}), \mathbf{m}, \mathbf{i}_{\mathsf{aux}})$ ▷ If $\mathbf{i}_{\mathsf{aux}}$ is empty, we omit it
11: $h \leftarrow \mathsf{RecursiveHash}(\mathbf{f}, (y_0, \dots, y_{2\cdot\ell-1}), (c'_0, \dots, c'_{2\cdot\ell-1}), \mathbf{h}_{\mathsf{aux}})$ ▷ See algorithm 4.8
12: $e' \leftarrow \mathsf{ByteArrayToInteger}(h)$ ▷ See algorithm 3.8
13: **if** $e = e'$ **then**
$\quad$ **return** $\top$
14: **else**
$\quad$ **return** $\bot$
15: **end if**

Output:

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

Test values for the algorithm 9.6 are provided in the attached verify-decryption.json file.

9.4 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 9.7.

Algorithm 9.7 ComputePhiExponentiation: compute the phi-function for exponentiation

Context:

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

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:

1: **for** $i \in [0, n)$ **do**
2: $\quad y_i \leftarrow g_i^x \mod p$
3: **end for**
4: **return** $(y_0, \dots, y_{n-1})$

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 9.1, with explicit domains and operations. Our phi-function defined in algorithm 9.7 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 9.8 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. $p = 2 \cdot q + 1$

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 $\mathbf{i}_{\text{aux}} \in ({\mathbb{A}_{UCS}}^*)^*$

Operation:

1: $b \leftarrow \mathsf{GenRandomInteger}(q)$ ▷ See algorithm 4.1
2: $\mathbf{c} \leftarrow \mathsf{ComputePhiExponentiation}(b, \mathbf{g})$ ▷ See algorithm 9.7
3: $\mathbf{f} \leftarrow (p, q, \mathbf{g})$
4: $\mathbf{h}_{\text{aux}} \leftarrow (\text{"}\texttt{ExponentiationProof}\text{"}, \mathbf{i}_{\text{aux}})$ ▷ If $\mathbf{i}_{\text{aux}}$ is empty, we omit it
5: $e \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{f}, \mathbf{y}, \mathbf{c}, \mathbf{h}_{\text{aux}}))$ ▷ See algorithms 3.8 and 4.8
6: $z \leftarrow b + e \cdot x \mod q$

Output:

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

Algorithm 9.9 VerifyExponentiation: Verifies the validity of an exponentiation proof

Context:

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

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

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 $\mathbf{i}_{\text{aux}} \in ({\mathbb{A}_{UCS}}^*)^*$

Operation:

1: $\mathbf{x} \leftarrow \text{ComputePhiExponentiation}(z, \mathbf{g})$ ▷ See algorithm 9.7
2: $\mathbf{f} \leftarrow (p, q, \mathbf{g})$
3: **for** $i \in [0, n)$ **do**
4: $\quad c'_i \leftarrow x_i \cdot y_i^{-e}$
5: **end for**
6: $\mathbf{h}_{\text{aux}} \leftarrow (\text{``ExponentiationProof''}, \mathbf{i}_{\text{aux}})$ ▷ If $\mathbf{i}_{\text{aux}}$ is empty, we omit it
7: $h \leftarrow \text{RecursiveHash}(\mathbf{f}, \mathbf{y}, (c'_0, \dots, c'_{n-1}), \mathbf{h}_{\text{aux}})$ ▷ See algorithm 4.8
8: $e' \leftarrow \text{ByteArrayToInteger}(h)$ ▷ See algorithm 3.8
9: **if** $e = e'$ **then**
$\quad$ **return** $\top$
10: **else**
$\quad$ **return** $\bot$
11: **end if**

Output:

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

Test values for the algorithm 9.9 are provided in the attached verify-exponentiation.json file.

9.5 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 9.10 ComputePhiPlaintextEquality: Compute the phi-function for plaintext equality

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
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 9.1, with explicit domains and operations. Our phi-function defined in algorithm 9.10 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 9.11 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. $p = 2 \cdot q + 1$
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 $\mathbf{i}_{\text{aux}} \in ({\mathbb{A}_{UCS}}^*)^*$

Operation:

1: $(b_1, b_2) \leftarrow \text{GenRandomVector}(q, 2)$ ▷ See algorithm 4.2
2: $\mathbf{c} \leftarrow \text{ComputePhiPlaintextEquality}((b_1, b_2), h, h')$ ▷ See algorithm 9.10
3: $\mathbf{f} \leftarrow (p, q, g, h, h')$
4: $\mathbf{y} \leftarrow (c_0, c'_0, \frac{c_1}{c'_1})$
5: $\mathbf{h}_{\text{aux}} \leftarrow (\text{“}\texttt{PlaintextEqualityProof}\text{”}, c_1, c'_1, \mathbf{i}_{\text{aux}})$ ▷ If $\mathbf{i}_{\text{aux}}$ is empty, we omit it
6: $e \leftarrow \text{ByteArrayToInteger}(\text{RecursiveHash}(\mathbf{f}, \mathbf{y}, \mathbf{c}, \mathbf{h}_{\text{aux}}))$ ▷ See algorithms 3.8 and 4.8
7: $\mathbf{z} \leftarrow (b_1 + e \cdot r, b_2 + e \cdot r')$

Output:

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

Algorithm 9.12 VerifyPlaintextEquality: Verifies the validity of a plaintext equality proof

Context:

Group modulus $p \in \mathbb{P}$
Group cardinality $q \in \mathbb{P}$ s.t. $p = 2 \cdot q + 1$
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 $\mathbf{i}_{\text{aux}} \in (\mathbb{A}_{UCS}{}^*)^*$

Operation:

1: $\mathbf{x} \leftarrow \mathsf{ComputePhiPlaintextEquality}(\mathbf{z}, h, h')$ ▷ See algorithm 9.10
2: $\mathbf{f} \leftarrow (p, q, g, h, h')$
3: $\mathbf{y} \leftarrow (c_0, c'_0, \frac{c_1}{c'_1})$
4: $\mathbf{c}' \leftarrow \mathbf{x} \cdot \mathbf{y}^{-e}$
5: $\mathbf{h}_{\text{aux}} \leftarrow (\text{"}\texttt{PlaintextEqualityProof}\text{"}, c_1, c'_1, \mathbf{i}_{\text{aux}})$ ▷ If $\mathbf{i}_{\text{aux}}$ is empty, we omit it
6: $e' \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecursiveHash}(\mathbf{f}, \mathbf{y}, \mathbf{c}', \mathbf{h}_{\text{aux}}))$ ▷ See algorithms 3.8 and 4.8
7: **if** $e = e'$ **then**
    return $\top$
8: **else**
    return $\bot$
9: **end if**

Output:

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

Test values for the algorithm 9.12 are provided in the attached verify-plaintext-equality.json file.

Acknowledgements

Swiss Post is thankful to all security researchers for their contributions and the opportunity to improve the system's security quarantees. In particular, we want to thank the following experts for their reviews or suggestions reported on our Gitlab repository. We list them here in alphabetical order:

- Aleksander Essex (Western University Canada)
- Rolf Haenni, Reto Koenig, Philipp Locher, Eric Dubuis (Bern University of Applied Sciences)
- Thomas Edmund Haines (Australian National University)
- Sarah Jamie Lewis (Open privacy)
- Pascal Junod (modulo p SA)
- Olivier Pereira (Universtité catholique Louvain)
- Vanessa Teague (Thinking Cybersecurity)

References

[1] Martin R. Albrecht, Jake Massimo, Kenneth G. Paterson, and Juraj Somorovsky. Prime and Prejudice: Primality Testing Under Adversarial Conditions. Cryptology ePrint Archive, Report 2018/749, 2018. `https://ia.cr/2018/749`.

[2] Robert Baillie, Andrew Fiori, and Samuel Wagstaff Jr. Strengthening the Baillie-PSW primality test. *Mathematics of Computation*, 90(330):1931–1955, 2021.

[3] Robert Baillie and Samuel S Wagstaff. Lucas pseudoprimes. *Mathematics of Computation*, 35(152):1391–1417, 1980.

[4] 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.

[5] Mihir Bellare, Alexandra Boldyreva, and Jessica Staddon. Randomness Re-use in Multi-recipient Encryption Schemeas. In *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, pages 85–99, 2003.

[6] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302. IEEE, 2016.

[7] John Brillhart, Derrick H Lehmer, and John L Selfridge. New Primality Criteria and Factorizations of $2^m \pm 1$. *Mathematics of computation*, 29(130):620–647, 1975.

[8] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, W Timothy Polk, et al. Internet X. 509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. *RFC*, 5280:1–151, 2008.

[9] Die Schweizerische Bundeskanzlei (BK). Federal Chancellery Ordinance on Electronic Voting (OEV), 01 July 2022.

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

[11] Morris J Dworkin. *Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac.* National Institute of Standards & Technology, 2007.

[12] 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.

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

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

[15] 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.

[16] 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`.

[17] Thomas Haines. Finding Report: SwissPost Voting System - Signature Verification. `https://gitlab.anu.edu.au/u1113289/thomas-public-paper/-/raw/a5f1c738d7e034c360dc5fccddf49ea9555a42b1/SwissPostSigningMarch2021.pdf`, November 2021. Accessed: 2022-02-23.

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

[19] Tetsu Iwata, Keisuke Ohashi, and Kazuhiko Minematsu. Breaking and repairing gcm security proofs. In *Annual Cryptology Conference*, pages 31–49. Springer, 2012.

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

[21] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.

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

[23] Hugo Krawczyk and Pasi Eronen. HMAC-based extract-and-expand key derivation function (HKDF). Technical report, RFC 5869, May, 2010.

[24] 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.

[25] David McGrew. An Interface and Algorithms for Authenticated Encryption. RFC 5116, January 2008.

[26] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[27] Gary L Miller. Riemann's hypothesis and tests for primality. *Journal of computer and system sciences*, 13(3):300–317, 1976.

[28] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, November 2016.

[29] Thomas R. Nicely. The Baillie-PSW primality test. `https://faculty.lynchburg.edu/~nicely/misc/bpsw.html`, 2012. Accessed: 2022-02-21.

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

[31] Swiss Post. Protocol of the Swiss Post Voting System. Computational Proof of Complete Verifiability and Privacy. Version 1.0.0. `https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation/-/tree/master/Protocol`, July 2022.

[32] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of number theory*, 12(1):128–138, 1980.

[33] ITUT Recommendation. Itu-t recommendation x. 690 asn. 1 encoding rules: Specification of basic encoding rules (ber), canonical encoding rules (cer) and distinguished encoding rules (der), 2008.

[34] Nigel P Smart et al. Algorithms, key size and protocols report. *ECRYPT - CSA, H2020-ICT-2014, Project 645421*, 2018.

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

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

[37] Yiannis Tsiounis and Moti Yung. On the security of elgamal based encryption. In *Public Key Cryptography*, pages 117–134, 1998.

[38] Hugh C Williams. *Édouard Lucas and primality testing*, volume 23. John Wiley & Sons, 1998.

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