Bern University of Applied Sciences, CH-2501 Biel, Switzerland

UniVote2 System Specification

Version 0.6.10

Dubuis Eric, Haenni Rolf, Hauser Severin, Koenig Reto E.

01.09.2015

On behalf of the student unions of the University of Bern (SUB), the University of Zürich (VSUZH), and the Bern University of Applied Sciences (VSBFH).

Revision History

Revision	**Date**	**Author(s)**	**Description**
0.1	14.07.2012	Haenni Rolf	Initial Draft.
0.2	15.08.2012	Haenni Rolf	Complete revision of initial draft.
0.3	10.09.2012	Haenni Rolf	Includes now the case of late registrations.
0.4	13.03.2013	Haenni Rolf	Improved ballot acceptance and closing the urn.
0.5	04.04.2013	Haenni Rolf	Revised section on zero-knowledge proofs.
0.5.1	30.04.2013	Haenni Rolf	Improved notational consistency.
0.5.2	21.05.2013	Haenni Rolf	Two important notational corrections.
0.6	07.01.2015	Koenig Reto E.	Quality: Change of title; re-establishement of the version history.
0.6.1	15.06.2015	Koenig Reto E.	Revised section on Randomness.
0.6.2	16.06.2015	Hauser Severin	Revision of UniVote.
0.6.3	16.06.2015	Koenig Reto E.	Revision of UniVote.
0.6.4	26.06.2015	Haenni Rolf	Table in Section 7 adjusted.
0.6.5	20.07.2015	Haenni Rolf	Updated tables in Section 7; Updated late registration procedure.
0.6.6	22.07.2015	Haenni Rolf	Final minor protocol adjustments.
0.6.7	04.08.2015	Haenni Rolf	Cleanup in encoding and hashing.
0.6.8	18.08.2015	Haenni Rolf	Encoding section updated.
0.6.9	27.08.2015	Haenni Rolf	Password-Based Key Derivation.
0.6.10	01.09.2015	Dubuis Eric	Reformatted tables for actions.

Contents

1. Introduction

Part I.

Theoretical Background

2. Preliminaries

2.1. Notational Conventions

As a general rule, we use upper-case latin or greek letters for sets, tuples, and sequences of atomic elements, and lower-case latin or greek letters for their elements. For example $X = \{x_1, \ldots, x_n\}$ for a set, $Y = (x_1, \ldots, x_n) \in X_1 \times \cdots \times X_n$ for a tuple, or $Z = \langle x_1, \ldots, x_n \rangle \in X^*$ for a finite sequence. $|X|$ denotes the cardinality of X. For families of subsets, sets of tuples, or sets of sequences, we usually use calligraphic upper-case latin letters, for example $\mathcal{X} \subseteq X_1 \times \cdots \times X_n$ for a set of tuples.

The set of integers is denoted by $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and the set of natural numbers by $\mathbb{N} = \{0, 1, 2, \ldots\}$. For $n \geq 1$, we write $\mathbb{Z}_n = \{0, \ldots, n-1\}$ for the set of natural numbers and $\mathbb{P}_n = \{x \in \mathbb{Z}_n : \mathit{prime}(x) = \mathit{true}\}$ for the set of prime numbers smaller then n. For an integer $x \in \mathbb{Z}$, we write $\mathrm{abs}(x)$ for the absolute value of x and $|x| = \lfloor \log_2(\mathrm{abs}(x)) \rfloor + 1$ for the *bit length* of x (excluding a sign bit). The set of all natural numbers with a given bit length $l \geq 0$ is denoted by $\mathbb{Z}_{|x|=l} = \{x \in \mathbb{N} : |x| = l\} = \mathbb{Z}_{2^l} \setminus \mathbb{Z}_{2^{l-1}}$ and the corresponding set of prime numbers by $\mathbb{P}_{|x|=l}$.

To denote mathematical functions, we generally use one or multiple lower-case latin letters, single upper-case latin letters for word boundaries, and multiple upper-case latin letters for abbreviations, for example $f(x)$, $\mathit{function}(x)$, $\mathit{myFunction}(x)$, or $\mathit{GCD}(x)$.

For a person or party involved in cryptographic protocols, we use upper-case latin letters in sans-serif font, for example CA for a certificate authority, EA for an election administration, or V_i for voters.

2.2. Byte Arrays

Let $B = \langle b_0, \ldots, b_{n-1} \rangle$ denote an array of bytes $b_i \in \mathcal{B}$, where $\mathcal{B} = \{\mathtt{0}, \mathtt{1}\}^8$ the set of all 256 bytes. Its length is denoted by $|B| = n$. We use standard array notation $B[i] = b_i$ to select from B the byte at index $i \in \{0, \ldots, n-1\}$ and hexadecimal notation for individual bytes. For example, $B = \langle \mathtt{0A}, \mathtt{23}, \mathtt{EF} \rangle$ denotes a byte array containing three bytes $B[0] = \mathtt{0A}_{16} = \mathtt{00001010}_2$, $B[1] = \mathtt{23}_{16} = \mathtt{001000011}_2$, and $B[2] = \mathtt{EF}_{16} = \mathtt{11101111}_2$.

If B' is a second byte array of length $n' = |B'|$, then $B \,||\, B'$ denotes the concatenated byte array of length $n + n'$ with the bytes of B placed before the bytes of B'. Furthermore, we write $B \wedge B'$, $B \vee B'$, and $B \oplus B'$ for the byte array of length $\min(n, n')$ obtained from applying bit-wise the logical AND, OR, and XOR operators, respectively (the additional bytes of the longer byte array are discarded). Similarly, $\neg B$ denotes the result of applying bit-wise the NOT operator. Finally, if $x \in \mathbb{N}$ is a natural number with bit length $|x| \leq 8 \cdot |B|$, we write $B + x$ for the result of adding x to B in base two (the overflow bit is discarded).

2.2.1. Representing Integers

Let $x \in \mathbb{Z}$ be an integer. We use $bytes_k(x) \in \mathcal{B}^k$ to denote the byte array obtained from truncating the k least significant bytes from the (infinitely long) two's complement representation of x in big-endian order, where $k \geq \lceil (|x|+1)/8 \rceil \geq 1$. We use $bytes(x) = bytes_{k_{min}}(x)$ as a short-cut notation for the shortest possible such byte array representation of length $k_{min} = \lceil (|x|+1)/8 \rceil$. Note that the most significant bit of $bytes(x)[0]$ always represents the sign of x, also for $bytes(0) = \langle \mathtt{00} \rangle$, which contains a single zero byte. The empty byte array is not a valid integer representation.

The following table shows the byte array representations for different integers x and $k \leq 4$:

	$bytes_k(x)$						
x	$k=1$	$k=2$	$k=3$	$k=4$	$\cdots$	k_{min}	$bytes(x)$
0	$\langle 00 \rangle$	$\langle 00, 00 \rangle$	$\langle 00, 00, 00 \rangle$	$\langle 00, 00, 00, 00 \rangle$		1	$\langle 00 \rangle$
1	$\langle 01 \rangle$	$\langle 00, 01 \rangle$	$\langle 00, 00, 01 \rangle$	$\langle 00, 00, 00, 01 \rangle$		1	$\langle 01 \rangle$
127	$\langle 7F \rangle$	$\langle 00, 7F \rangle$	$\langle 00, 00, 7F \rangle$	$\langle 00, 00, 00, 7F \rangle$		1	$\langle 7F \rangle$
-128	$\langle 80 \rangle$	$\langle FF, 80 \rangle$	$\langle FF, FF, 80 \rangle$	$\langle FF, FF, FF, 80 \rangle$		1	$\langle 80 \rangle$
-2	$\langle FE \rangle$	$\langle FF, FE \rangle$	$\langle FF, FF, FE \rangle$	$\langle FF, FF, FF, FE \rangle$		1	$\langle FE \rangle$
-1	$\langle FF \rangle$	$\langle FF, FF \rangle$	$\langle FF, FF, FF \rangle$	$\langle FF, FF, FF, FF \rangle$		1	$\langle FF \rangle$
128	$-$	$\langle 00, 80 \rangle$	$\langle 00, 00, 80 \rangle$	$\langle 00, 00, 00, 80 \rangle$		2	$\langle 00, 80 \rangle$
255	$-$	$\langle 00, FF \rangle$	$\langle 00, 00, FF \rangle$	$\langle 00, 00, 00, FF \rangle$		2	$\langle 00, FF \rangle$
-256	$-$	$\langle FF, 00 \rangle$	$\langle FF, FF, 00 \rangle$	$\langle FF, FF, FF, 00 \rangle$		2	$\langle FF, 00 \rangle$
-129	$-$	$\langle FF, 7F \rangle$	$\langle FF, FF, 7F \rangle$	$\langle FF, FF, FF, 7F \rangle$		2	$\langle FF, 7F \rangle$
256	$-$	$\langle 01, 00 \rangle$	$\langle 00, 01, 00 \rangle$	$\langle 00, 00, 01, 00 \rangle$		2	$\langle 01, 00 \rangle$
32767	$-$	$\langle 7F, FF \rangle$	$\langle 00, 7F, FF \rangle$	$\langle 00, 00, 7F, FF \rangle$		2	$\langle 7F, FF \rangle$
-32768	$-$	$\langle 80, 00 \rangle$	$\langle FF, 80, 00 \rangle$	$\langle FF, FF, 80, 00 \rangle$		2	$\langle 80, 00 \rangle$
-257	$-$	$\langle FE, FF \rangle$	$\langle FF, FE, FF \rangle$	$\langle FF, FF, FE, FF \rangle$		2	$\langle FE, FF \rangle$
32768	$-$	$-$	$\langle 00, 80, 00 \rangle$	$\langle 00, 00, 80, 00 \rangle$		3	$\langle 00, 80, 00 \rangle$
65535	$-$	$-$	$\langle 00, FF, FF \rangle$	$\langle 00, 00, FF, FF \rangle$		3	$\langle 00, FF, FF \rangle$
-65536	$-$	$-$	$\langle FF, 00, 00 \rangle$	$\langle FF, FF, 00, 00 \rangle$		3	$\langle FF, 00, 00 \rangle$
-32769	$-$	$-$	$\langle FF, 7F, FF \rangle$	$\langle FF, FF, 7F, FF \rangle$		3	$\langle FF, 7F, FF \rangle$

The two's complement representation in big-endian byte order is the default integer representation considered in this document. It is also used to reconstruct an integer $x = integer(B) = bytes_k^{-1}(B) \in \mathbb{Z}$ from an arbitrary byte array $B \in \mathcal{B}^k$ of length $k \geq 1$, or a non-negative integer $x = integer^+(B) = bytes_{k+1}^{-1}(\mathtt{00} \,||\, B) \in \mathbb{N}$ from a byte array of length $k \geq 0$.

```
// Create the default big-endian converter
BigIntegerToByteArray converter = BigIntegerToByteArray.getInstance();

// Convert various integers
ByteArray b1 = converter.convert(1);    // returns "01"
ByteArray b2 = converter.convert(127);  // returns "7F"
ByteArray b3 = converter.convert(-257); // returns "FE|FF"

// Reconvert the byte arrays
BigInteger i1 = converter.reconvert(b1); // returns 1
```

```
BigInteger i2 = converter.reconvert(b2); // returns 127
BigInteger i3 = converter.reconvert(b3); // returns -257
```

Listing 2.1: Coding example using UniCrypt.

2.2.2. Representing Strings

Let U be the *Universal Character Set* (UCS) as defined by ISO/IEC 10646. A string of length n is a sequence $S = \langle c_1 \cdots c_n \rangle \in U^*$ of characters $c_i \in U$. U^* denotes the set of all UCS strings, including the empty string. Selecting the i-th character from S is written as $S[i] = c_i$ for $i \in \{1, \ldots, n\}$. Concrete string instances are written in the usual string notation, for example `""` (empty string), `"x"` (string consisting of a single character `'x'`), or `"Hello"`.

To encode a string $S \in U^*$ as byte array, we use the UTF-8 character encoding as defined in ISO/IEC 10646 (Annex D). Let $bytes(S)$ denote the corresponding byte array, in which characters use 1, 2, 3, or 4 bytes of space depending on the type of character. For example, $bytes(\texttt{"Hello"}) = \langle \texttt{48}, \texttt{65}, \texttt{6C}, \texttt{6C}, \texttt{6F} \rangle$ is a byte array of length 5, because it only consists of Basic Latin characters, whereas $bytes(\texttt{"Voilà"}) = \langle \texttt{56}, \texttt{6F}, \texttt{69}, \texttt{6C}, \texttt{C3}, \texttt{A0} \rangle$ contains 6 bytes due to the Latin-1 Supplement character `'à'` translating into two bytes. UTF-8 is the only character encoding used in this document.

```
// Create the default UTF-8 converter
StringToByteArray converter = StringToByteArray.getInstance();

// Convert various strings
ByteArray s1 = converter.convert("");      // returns ""
ByteArray s2 = converter.convert("Hello"); // returns "48|65|6C|6C|6F"
ByteArray s3 = converter.convert("Voilà"); // returns "56|6F|69|6C|C3|A0"

// Reconvert the byte arrays
String i1 = converter.reconvert(s1); // returns ""
String i2 = converter.reconvert(s2); // returns "Hello"
String i3 = converter.reconvert(s3); // returns "Voilà"
```

Listing 2.2: Coding example using UniCrypt.

2.3. Pairing

Let $x, y \in \mathbb{N}$ be two non-negative integers. A *pairing function* is a bijective mapping $pair : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, which maps pairs $(x, y) \in \mathbb{N} \times \mathbb{N}$ into a single paired value $pair(x, y) \in \mathbb{N}$. Let $unpair : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ denote the corresponding *unpair function*, for which $(x, y) = unpair(pair(x, y))$ holds for all $x, y \in \mathbb{N}$.

One of the simplest pairing function, called *elegant pairing* [?], is defined as

$$pair(x, y) = \begin{cases} x^2 + x + y, & \text{if } x \geq y, \\ x + y^2, & \text{otherwise.} \end{cases} \tag{2.1}$$

The following table lists the paired values for all $x, y \leq 5$.

		x						
		0	1	2	3	4	5	...
	0	0	2	6	12	20	30	
	1	1	3	7	13	21	31	
	2	4	5	8	14	22	32	
y	3	9	10	11	15	23	33	
	4	16	17	18	19	24	34	
	5	25	26	27	28	29	35	
	⋮							

Note that the bit length of a paired value $z = \mathit{pair}(x, y)$ is either $|z| = 2|m|$ or $|z| = 2|m|-1$, where $m = \max(x, y)$ denotes the maximum of the two inputs. In other words, elegant pairing doubles the length of the larger input.

In case of elegant pairing, the corresponding unpairing function is defined by

$$\mathit{unpair}(z) = \begin{cases} (t, s), & \text{if } t < s, \\ (s, t-s), & \text{otherwise,} \end{cases}$$

where $s = \lfloor \sqrt{z} \rfloor$ and $t = z - s^2$.

```
// Perform pairing and unpairing
BigInteger p = MathUtil.pair(4, 5);   // returns 29
BigInteger[] u = MathUtil.unpair(p); // returns [4,5]
```

Listing 2.3: Coding example using UniCrypt.

There are three important generalizations of a pairing functions, which can be added individually or jointly. The first one extends the domain of the pairing function from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}^k$ for $k \geq 0$. There are multiple ways of defining such a k-ary pairing function $\mathit{pair}_k : \mathbb{N}^k \to \mathbb{N}$ recursively. If we assume that all input values are equally long, then a length-optimal recursive definition is as follows:

$$\mathit{pair}_k(x_1, \ldots, x_k) = \begin{cases} \mathit{pair}_{\frac{k}{2}}(\mathit{pair}(x_1, x_2), \ldots, \mathit{pair}(x_{k-1}, x_k)), & \text{if } k > 2 \text{ is even,} \\ \mathit{pair}_{\frac{k+1}{2}}(\mathit{pair}(x_1, x_2), \ldots, \mathit{pair}(x_{k-2}, x_{k-1}), x_k), & \text{if } k > 2 \text{ is odd,} \\ \mathit{pair}(x_1, x_2), & \text{if } k = 2, \\ x_1, & \text{if } k = 1, \\ 0, & \text{if } k = 0. \end{cases}$$

The corresponding function $\mathit{unpair}_k : \mathbb{N} \to \mathbb{N}^k$ inverts the above recursion:

$$\mathit{unpair}_k(z) = \begin{cases} (\mathit{unpair}(z_1), \ldots, \mathit{unpair}(z_{\frac{k}{2}})), \text{ for } z_i \in \mathit{unpair}_{\frac{k}{2}}(z), & \text{if } k > 2 \text{ is even,} \\ (\mathit{unpair}(z_1), \ldots, \mathit{unpair}(z_{\frac{k-1}{2}}), z_{\frac{k+1}{2}}), \text{ for } z_i \in \mathit{unpair}_{\frac{k+1}{2}}(z), & \text{if } k > 2 \text{ is odd,} \\ \mathit{unpair}(z), & \text{if } k = 2, \\ (z), & \text{if } k = 1, \\ (), & \text{if } k = 0. \end{cases}$$

To compute this function, the number k of output values must be known. This problem can be solved by an additional pairing with the length k of the input list, or in an optimized version by pairing it with $k-1$ and by treating the special case $k=0$ is separately:

$$pair'(x_1,\dots,x_k) = \begin{cases} pair(pair_k(x_1,\dots,x_k), k-1)+1, & \text{if } k>0,\\ 0, & \text{if } k=0,\end{cases}$$

$$unpair'(z) = \begin{cases} unpair_{k'+1}(z'), \text{ for } (z',k') = unpair(z-1), & \text{if } z>0,\\ (), & \text{if } z=0.\end{cases}$$

```
// Perform pairing and unpairing of multiple values
BigInteger p = MathUtil.pairWithSize(12, 29, 8); // returns 530669269373
BigInteger[] u = MathUtil.unpairWithSize(p);     // returns [12,29,8]
```

Listing 2.4: Coding example using UniCrypt.

The second generalization allows the pairing of even more complex structures containing non-negative integers, for example $X = (12,(4,5),8)$. Let $\mathcal{X}$ denote the set of all such composed structures. Then we obtain a function $Pair : \mathcal{X} \rightarrow \mathbb{N}$ by applying the above generalized pairing function recursively. To be able to distinguish the general and the base case of the recursion in $Unpair : \mathbb{N} \rightarrow \mathcal{X}$, we add an additional bit $b=1$ (general case) and $b=0$ (base case) to the respective results:

$$Pair(X) = \begin{cases} 2\cdot pair'(Pair(X_1),\dots,Pair(X_k))+1, & \text{if } X=(X_1,\dots,X_k),\\ 2X, & \text{if } X \text{ is an integer.}\end{cases}$$

$$Unpair(z) = \begin{cases} (Unpair(z_1),\dots,Unpair(z_k)), \text{ for } z_i \in unpair'(\frac{z-1}{2}), & \text{if } z \text{ is odd},\\ \frac{z}{2}, & \text{if } z \text{ is even.}\end{cases}$$

The third generalization extends the domain from $\mathbb{N}\times\mathbb{N}$ to $\mathbb{Z}\times\mathbb{Z}$ by first mapping each input $x\in\mathbb{Z}$ into a non-negative integer $fold(x)\in\mathbb{N}$, where

$$fold(x) = \begin{cases} 2x, & \text{if } x\geq 0,\\ 2|x|-1, & \text{otherwise,}\end{cases}$$

denotes the *folding function*. The corresponding *unfolding function* is defined by

$$unfold(x) = \begin{cases} \frac{1}{2}x, & \text{if } x \text{ is even,}\\ -\frac{1}{2}(x+1), & \text{otherwise.}\end{cases}$$

```
// Perform folding and unfolding
BigInteger f = MathUtil.fold(27);  // returns 57
BigInteger u = MathUtil.unfold(f); // return 27
```

Listing 2.5: Coding example using UniCrypt.

3. Cryptographic Primitives

The UniVote system is based on several cryptographic building blocks. Apart from standard ElGamal encryption and decryption, we also need hash functions, random oracles, Schnorr signatures, threshold decryptions, non-interactive zero-knowledge proofs of knowledge, verifiable exponentiation and re-encryption mix-nets, an anonymous channel, and an append-only public bulletin board. These building blocks will be described below.

3.1. Hash Functions and Keyed Hash Functions

A *hash function* defines a mapping $hash_n : \{0,1\}^* \rightarrow \{0,1\}^n$ from an arbitrarily long input bit sequences to a fixed-length output bit sequence of length n. For a given input bit string $B \in \{0,1\}^*$, we call $hash_n(B) \in \{0,1\}^n$ the *hash value* of B. Whenever the output length n is clear from the context, we allow $hash(B)$ as a shortcut notation.

All hash functions relevant for practical applications restrict the length of the input and output bit sequence to a multiple of 8. In other words, practical hash functions deal with bytes rather than bits. Therefore, if $\mathcal{B} = \{\texttt{0}, \texttt{1}\}^8$ denotes the set of all possible bytes, then a practical hash function is a mapping $hash_n : \mathcal{B}^* \rightarrow \mathcal{B}^{\frac{n}{8}}$ from arbitrarily long byte arrays to fixed-length byte arrays. The only hash functions we consider in this document are the NIST standards SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512, which output byte arrays of length 20 (160 bits), 28 (224 bits), 32 (256 bits), 48 (384 bits), and 64 (512 bits), respectively (see Chapter 7).

To apply such a hash function to a simple mathematical object such as an integer or string X, we first convert X into a byte array $B = bytes(X)$ and then apply the hash function to B. This allows us to write $hash(X) = hash(bytes(X))$ for more general inputs. In case of composed mathematical objects, for example $X = (127, (\texttt{"Hello"}, 1), -257)$, we apply the hash function recursively:

$$hash(X) = \begin{cases} hash(bytes(X)), & \text{if } X \text{ is atomic,} \\ hash(hash(X_1) \mid\mid \cdots \mid\mid hash(X_k)), & \text{if } X = (X_1, \ldots, X_k). \end{cases}$$

Sometimes, if $X = (X_1, \ldots, X_k)$ is a composed object, we write $hash(X)$ as the application of a k-ary hash function $hash(X_1, \ldots, X_k)$ to multiple arguments.

The following example hash values have been computed with SHA-1:

$$
\begin{aligned}
& hash(127) = hash(\langle \texttt{7F} \rangle) \\
& \quad = \langle \texttt{23}, \texttt{83}, \texttt{34}, \texttt{62}, \texttt{F5}, \texttt{55}, \texttt{15}, \texttt{A9}, \texttt{00}, \texttt{E0}, \texttt{16}, \texttt{DB}, \texttt{2E}, \texttt{B9}, \texttt{43}, \texttt{FB}, \texttt{47}, \texttt{4C}, \texttt{19}, \texttt{F6} \rangle \\
& hash(\texttt{"Hello"}) = hash(\langle \texttt{48}, \texttt{65}, \texttt{6C}, \texttt{6C}, \texttt{6F} \rangle) \\
& \quad = \langle \texttt{F7}, \texttt{FF}, \texttt{9E}, \texttt{8B}, \texttt{7B}, \texttt{B2}, \texttt{E0}, \texttt{9B}, \texttt{70}, \texttt{93}, \texttt{5A}, \texttt{5D}, \texttt{78}, \texttt{5E}, \texttt{0C}, \texttt{C5}, \texttt{D9}, \texttt{D0}, \texttt{AB}, \texttt{F0} \rangle \\
& hash(1) = hash(\langle \texttt{01} \rangle) \\
& \quad = \langle \texttt{BF}, \texttt{8B}, \texttt{45}, \texttt{30}, \texttt{D8}, \texttt{D2}, \texttt{46}, \texttt{DD}, \texttt{74}, \texttt{AC}, \texttt{53}, \texttt{A1}, \texttt{34}, \texttt{71}, \texttt{BB}, \texttt{A1}, \texttt{79}, \texttt{41}, \texttt{DF}, \texttt{F7} \rangle \\
& hash(-257) = hash(\langle \texttt{FE}, \texttt{FF} \rangle) \\
& \quad = \langle \texttt{26}, \texttt{23}, \texttt{78}, \texttt{00}, \texttt{2C}, \texttt{95}, \texttt{AE}, \texttt{7E}, \texttt{29}, \texttt{53}, \texttt{5C}, \texttt{B9}, \texttt{F4}, \texttt{38}, \texttt{DB}, \texttt{21}, \texttt{9A}, \texttt{DF}, \texttt{98}, \texttt{F5} \rangle \\
& hash(\texttt{"Hello"}, 1) = hash(hash(\texttt{"Hello"}) \,||\, hash(1)) \\
& \quad = \langle \texttt{49}, \texttt{2F}, \texttt{6E}, \texttt{17}, \texttt{0A}, \texttt{CF}, \texttt{F2}, \texttt{0A}, \texttt{E8}, \texttt{F2}, \texttt{0D}, \texttt{09}, \texttt{A6}, \texttt{11}, \texttt{66}, \texttt{D3}, \texttt{B1}, \texttt{41}, \texttt{99}, \texttt{DB} \rangle \\
& hash(127, (\texttt{"Hello"}, 1), -257) = hash(hash(127) \,||\, hash(\texttt{"Hello"}, 1) \,||\, hash(-127)) \\
& \quad = \langle \texttt{F4}, \texttt{65}, \texttt{C7}, \texttt{A9}, \texttt{9E}, \texttt{1A}, \texttt{DB}, \texttt{28}, \texttt{6C}, \texttt{23}, \texttt{29}, \texttt{6B}, \texttt{36}, \texttt{C8}, \texttt{C1}, \texttt{A0}, \texttt{83}, \texttt{03}, \texttt{AF}, \texttt{78} \rangle
\end{aligned}
$$

A *keyed hash function* defines a mapping $hash_{k,n} : \{0,1\}^* \times \{0,1\}^k \rightarrow \{0,1\}^n$ from an arbitrarily long input bit sequence and a fixed-length key of length k to a fixed-length output bit sequence of length n. For a given input bit string $B \in \{0,1\}^*$ and a key $K \in \{0,1\}^k$, we call $hash_{k,n}(B, K) \in \{0,1\}^n$ the *keyed hash value* of B. Whenever the key length k and the output length n are clear from the context, we allow $hash(B, K)$ as a shortcut notation.

The most widely used construction of a keyed hash function from a regular hash function is called HMAC [**? ?**]. It is fully compatible with the SHA hash function family. Corresponding instantiations are called HMAC-SHA1, HMAC-SHA256, etc. We use them as a building block for generating random bit sequences (see Section 3.2.1) and for password-based key derivation (see Section 3.2.3).

3.2. Generating Random Bits, Random Numbers, and Password-Based Keys

In this section, we give an overview of how random bits, random numbers, and password-based keys are generated in UniVote. There are two fundamentally different situations for generating randomness, one in which a third party needs to be convinced about how the randomness has been generated and one in which the randomness needs to be kept secret. Both goals are achieved using the similar cryptographic tools.

3.2.1. Deterministic Random Bit Generators

A *deterministic random bit generator* (DRBG) is a function $DRBG : \{0,1\}^m \rightarrow \{0,1\}^\infty$ for generating infinitely long bit sequences whose properties approximate the properties of true random bit sequences, except for the fact they are deterministic and periodic. The m input bits $S \in \{0,1\}^m$ of a PRBG are called *seed*, and the output bits $DRBG(S) \in \{0,1\}^\infty$ are called *random bit sequence*. The seed determines the initial internal state of the DRBG, and the bit length of the internal state determines the maximal period of the resulting random bit sequence.

Consider the following basic DRBG construction, in which a one-way function $f : \{0,1\}^m \rightarrow \{0,1\}^n$ is applied repeatedly to $S+i$ for a counter $i \in \{0,1,2,\ldots\}$. The output is the random bit sequence

$$DRBG_{\mathsf{CTR}}(S) = f(S) \mid\mid f(S+1) \mid\mid f(S+2) \mid\mid \cdots,$$

which comes in chunks of n bits [?]. The internal state of the generator is the value $S+i$. We call it a *counter mode* (CTR) construction. It is typically instantiated with either a hash function, a keyed hash function, or a block cipher. In this document, we only consider instantiations with hash functions,

$$DRBG^n_{\mathsf{CTR}}(S) = hash_n(S) \mid\mid hash_n(S+1) \mid\mid hash_n(S+2) \mid\mid \cdots,$$

with a default seed length m.

3.2.2. Random Oracles and Common Reference String

In cryptography, a *random oracle* is a mapping $randomOracle : \{0,1\}^* \rightarrow \{0,1\}^\infty$, which responds to each input bit sequence $Q \in \{0,1\}^*$ (the *query*) with an infinitely long output bit sequence $randomOracle(Q) \in \{0,1\}^\infty$, in which every bit is chosen uniformly and independently. If the same query Q is repeated, the random oracle responds the same way every time. Since no function computable by a finite algorithm can implement a true random oracle, they exist only as a theoretical model. As such, they are an important mathematical abstraction used in numerous cryptographic proofs, often as a replacement for hash functions. The corresponding security model is called *random oracle model*.

The following table gives an overview of the fundamental differences between hash functions, deterministic random bit sequences, and random oracles.

	domain	co-domain	state width	max. period
Hash function	$\{0,1\}^*$	$\{0,1\}^n$	–	–
Deterministic random bit sequence	$\{0,1\}^m$	$\{0,1\}^\infty$	m	2^m
Random oracle	$\{0,1\}^*$	$\{0,1\}^\infty$	∞	∞

In practice, random oracles can at most be approximated, for example using the function $DRBG_{\mathsf{CTR}}$ defined above. For this, the arbitrarily long query Q must be mapped into a seed S of length m. In an instantiation with a hash function $hash_n$ and a default seed length $m = n$, we can use the same hash function for this purpose:

$$randomOracle(Q) = DRBG^n_{\mathsf{CTR}}(hash_n(Q)).$$

To make random oracles more flexible with respect to the type of query they accept, we allow writing $randomOracle(X)$ for an arbitrary mathematical object X. In such a case, the initial hash value $hash_n(X)$ is computed recursively from the byte representations of its components (see Section 3.1).

In UniVote, we need random oracles to generate sequences of random challenges in some zero-knowledge proofs. In other situations, we only need a single deterministic sequence of randomly looking bits, independently of any query. We can easily define such a *common reference string* by querying the random oracle with a default query, for example with the empty bit string $Q = \langle\rangle$. In UniVote, we use

$$\mathit{referenceString} = \mathit{randomOracle}(\langle\rangle)$$

to generate independent generators of cyclic groups, which are important building blocks for Pedersen commitments and shuffle proofs.

3.2.3. Password-Based Key Derivation

Another important application of deterministic random bit generators is the derivation of cryptographic keys from a secret password and a non-secret salt. The purpose of the salt is to prevent the construction of rainbow tables in exhaustive-search attacks. The password and the salt determine the output of the DRBG. To make exhaustive search attacks more expensive, the process of updating the internal DRBG state is repeated r times without producing any output. In this way, the performance of the generator is therefore artificially decreased, with the effect of increasing the cost of an attack. The number of rounds is therefore an additional parameter in the construction of the DRBG.

A standard construction of a DRBG for password-based key derivation is called PBKDF2 [**?** **?**]. Most implementations of PBKDF2 use HMAC as pseudorandom function in each round of the derivation process. The author of [**?**] recommends a salt length of at least $m = 64$ bits, but this number was increased to $m = 128$ bits in [**?**]. The recommended minimum number of iterations is $r \geq 1000$ for ordinary keys and $r \geq 10'000'000$ for very critical keys. In UniVote, the default PBKDF2 instantiation uses HMAC-SHA256 and performs $r = 10'000$ iterations. Applying this default instantiation to a password $P \in \{0,1\}^n$ and a salt $S \in \{0,1\}^m$ is denoted by $\mathit{PBKDF2}(P, S) \in \{0,1\}^\infty$. If the password is given as a string $P \in U^*$, then we first apply the UTF-8 character encoding to P and then use the bits in $\mathit{bytes}(P)$ as input for $\mathit{PBKDF2}$. The salt should be generated by a secure (non-deterministic or hybrid) randomness source.

3.2.4. Non-Deterministic and Hybrid Random Bit Generators

Hash-DRBG

HMAC-DRBG

3.2.5. Generating Random Numbers from Random Bits

Choosing an integer uniformly at random from $\mathbb{Z}_n$ can be done by generating a random bit sequence of length $l = |n - 1|$, converting it to an integer $x \in \mathbb{Z}_{2^l}$, and repeating this procedure until $x \in \mathbb{Z}_n$. This process is denoted by $x \in_R \mathbb{Z}_n$.

For an interval $[a,b] \subseteq \mathbb{N}$, we write $x \in_R [a,b]$ for the process of randomly choosing one of its elements, for example by selecting $y \in_R \mathbb{Z}_{b-a+1}$ and returning $x = y + a$. For $a = 2^{l-1}$ and $b = 2^l - 1$, this corresponds to generating random numbers $x \in_R \mathbb{Z}_{|x|=l}$ of a given bit length l. In this particular case, the whole procedure is equivalent to generating a random bit sequence of length $l-1$, pre-pending a 1-bit, and converting the result to an integer.

The process for choosing random prime numbers $p \in_R \mathbb{P}_{|x|=l}$ of a given bit length l is similar. It consists of selecting $p \in_R \mathbb{Z}_{|x|=l}$, running a (probabilistic) primality test $prime(p)$ to check whether p is prime, and repeating this procedure until $prime(p) = true$. Generating safe primes is similar but requires an additional primality test for $q = (p-1)/2$ in each iterative step.

Finally, we consider the problem of choosing a random permutation $\pi : \mathbb{Z}_n \to \mathbb{Z}_n$ uniformly from the set Π_n of all such permutations. For this, we select a random rank $x \in \mathbb{Z}_{n!}$ and input it to Myrvold and Wendy's linear-time unranking algorithm $unrank : \mathbb{Z}_{n!} \to \Pi_n$ [**?**]. We denote the whole process by $\pi \in_R \Pi_n$.

3.3. ElGamal Cryptosystem

The *ElGamal cryptosystem* is based on a multiplicative cyclic group $(G_q, \cdot, 1)$ of order q, for which the decisional Diffie-Hellman assumption (DDH) is believed to hold [**?**]. The most common choice for such a group is the subgroup of quadratic residues $G_q \subset \mathbb{Z}_p^*$ of prime order q, where $p = 2q+1$ is a *safe prime*. Typically, p is chosen to be large enough (>1024 bits) to resist index-calculus and other methods of solving the discrete logarithm problem. The public parameters of an ElGamal cryptosystem are thus p, q, and a generator g of $G_q = \langle g \rangle$. A suitable generator can be found by picking an arbitrary value $\gamma \in \mathbb{Z}_p^*$ and by checking that $g = \gamma^2$ is different from 1.

An ElGamal key pair is a tuple (x, y), where $x \in_R \mathbb{Z}_q$ is the randomly chosen private decryption key and $y = g^x \in G_q$ the corresponding public encryption key. If $m \in G_q$ denotes the plaintext to encrypt, then

$$encrypt_y(m, r) = (g^r, m \cdot y^r) \in G_q \times G_q \tag{3.1}$$

is the ElGamal encryption of m with randomization $r \in_R \mathbb{Z}_q$.[1] Note that its bit length is twice the bit length of p. For a given encryption $E = (a,b) = encrypt_y(m,r)$, m can be recovered by using the private decryption key x to compute

$$decrypt_x(E) = a^{-x} \cdot b = m. \tag{3.2}$$

Note that m can also be recovered by $m = b \cdot y^{-r}$ in case the randomization r is known.

The ElGamal encryption function is *homomorphic* with respect to multiplication, which means that the component-wise multiplication of two ciphertexts yields an encryption of the product of respective plaintexts:

$$encrypt_y(m_1, r_1) \cdot encrypt_y(m_2, r_2) = encrypt_y(m_1 \cdot m_2, r_1 + r_2). \tag{3.3}$$

[1] For improved efficiency, we can pick a randomization r with a reduced, but large enough bit length to resist birthdate attacks on discrete logarithms (160–512 bits). Furthermore, we can pre-compute both parts of an ElGamal encryption prior to knowing the plaintext m.

In a homomorphic cryptosystem like ElGamal, a given encryption $E = \mathit{encrypt}_y(m, r)$ can be *re-encrypted* by multiplying E with an encryption of the neutral element 1. The resulting re-encryption,

$$\mathit{reEncrypt}_y(E, r') = E \cdot \mathit{encrypt}_y(1, r') = \mathit{encrypt}_y(m, r + r'), \tag{3.4}$$

is clearly an encryption of m with a fresh randomization $r + r'$.

Practical applications often require the plaintext to be in $\mathbb{Z}_q$ rather than G_q. With a safe prime p, we can use the following mapping $\mathit{subGroup} : \mathbb{Z}_q \to G_q$ to encode any integer plaintext $m \in \mathbb{Z}_q$ by a group element $m' \in G_q$, which can then be encrypted as described above:

$$m' = \mathit{subGroup}(m) = \begin{cases} m + 1, & \text{if } (m+1)^q = 1, \\ p - (m + 1), & \text{otherwise.} \end{cases} \tag{3.5}$$

When we obtain $m' \in G_q$ from decrypting the ciphertext, we can reconstruct $m \in \mathbb{Z}_q$ by applying the inverse function $\mathit{subGroup}^{-1} : G_q \to \mathbb{Z}_q$ to m':

$$m = \mathit{subGroup}^{-1}(m') = \begin{cases} m' - 1, & \text{if } m' \leq q, \\ (p - m') - 1, & \text{otherwise.} \end{cases} \tag{3.6}$$

Note that by adding such an encoding to the ElGamal cryptosystem, it is no longer homomorphic with respect to plaintexts in $\mathbb{Z}_q$, but re-encryptions can still be computed in the same way as explained above.

3.4. Schnorr Signatures

The *Schnorr signature scheme* has a setting similar to the ElGamal cryptosystem. It is based on a multiplicative cyclic group $(G_q, \cdot, 1)$ of order q, for which the discrete logarithm problem (DLP) is believed to be intractable in the random oracle model [**?**]. The most common choice is a *Schnorr group*, a subgroup $G_q \subset \mathbb{Z}_p^*$ of prime order q, where $p = kq + 1$ is a prime large enough (>1024 bits) to resist methods for solving the discrete logarithm problem, while q is large enough (160–512 bits) to resist birthday attacks on discrete logarithm problems. The public parameters of a Schnorr signature scheme are thus p, q, and a generator g of $G_q = \langle g \rangle$. A suitable generator can be found by selecting an arbitrary value $\gamma \in \mathbb{Z}_p^*$ and by checking that $g = \gamma^k$ is different from 1. Furthermore, all involved parties must agree on a cryptographic hash function $\mathit{hash} : \{0,1\}^* \to \{0,1\}^n$. In this document, only SHA-256 is used for this purpose.

A Schnorr signature key pair is a tuple (sk, vk), where $sk \in_R \mathbb{Z}_q$ is the randomly chosen private signature key and $vk = g^{sk} \in G_q$ the corresponding public verification key. Let $m \in \{0,1\}^*$ denote an arbitrary message to sign. If $r \in_R \mathbb{Z}_q$ is a randomly selected value and $a = \mathit{hash}(m, g^r) \bmod q$ the integer representation of the hash value of (m, g^r) modulo q, then

$$\mathit{sign}_{sk}(m, r) = (a, r - a \cdot sk) \in \mathbb{Z}_q \times \mathbb{Z}_q \tag{3.7}$$

is the Schnorr signature of m. Note that its bit length is twice the bit length of q. Using the public verification key vk, a given signature $S = (a, b) = sign_{sk}(m, r)$ for message m can be verified by computing

$$verify_{vk}(m, S) = \begin{cases} accept, & \text{if } a = hash(m, g^b \cdot vk^a) \bmod q, \\ reject, & \text{otherwise.} \end{cases} \tag{3.8}$$

3.5. Digital Certificates

Let X be a unique identifier of the holder of a public encryption or verification key k. The purpose of a *digital certificate* Z_{X} is to bind the key k to its holder X. For this, a certificate contains the signature of a trustworthy third party who guarantees the binding. Let CA be the unique identifier of such a *certificate authority* and $(sk_{\mathsf{CA}}, vk_{\mathsf{CA}})$ its signature key pair.

In practice, a digital certificate contains much more information than X, k, and $S = sign_{sk_{\mathsf{CA}}}(\mathsf{X}, k)$. Examples of additional information stored in a certificate are the issuer's name and unique identifier, a serial number, the validity period, the key type, the signature algorithm ID, and optional extensions. The most common standard in practice is X.509, which we adopt in this document. Note that the signature contained in an X.509 certificate is based on an ASN.1 binary encoding of the relevant certificate data. In this document, we denote X.509 certificates simply by

$$Z_{\mathsf{X}} = certify_{sk_{\mathsf{CA}}}(\mathsf{X}, k), \tag{3.9}$$

thus without specifying further details about the additional information stored in Z_{X} besides X and k. Similarly, the process of validating the correctness of Z_{X} is denoted by

$$verify_{vk_{\mathsf{CA}}}(Z_{\mathsf{X}}) \in \{reject, reject\}. \tag{3.10}$$

Note the the validation of an X.509 certificate Z_{X} includes checking the whole certificate chain towards a given *root certificate authority*. This is a standardized process which we do not further specify in this document.

3.6. Zero-Knowledge Proofs of Knowledge

A *zero-knowledge proof* is a cryptographic protocol, where the *prover* P tries to convince the *verifier* V that a mathematical statement is true, but without revealing any information other than the truth of the statement. A *proof of knowledge* is a particular proof allowing P to demonstrate knowledge of a secret information involved in the mathematical statement.

3.6.1. Non-Interactive Preimage Proof

One of the most fundamental zero-knowledge proofs of knowledge is the *preimage proof*. Let $(X, +, 0)$ be an additively and $(Y, \cdot, 1)$ a multiplicatively written group of finite order, and let $\phi : X \to Y$ a one-way group homomorphism. If P knows the preimage $a \in X$ (the *private input*) of a publicly known value $b = \phi(a) \in Y$ (the *public input*), then proving knowledge of a is achieved with the following non-interactive version of the so-called Σ-*protocol*. To generate the proof, P performs the following steps:

1. Choose $\omega \in_R X$ uniformly at random.
2. Compute $t = \phi(\omega)$.
3. Compute $c = hash((b,t),\mathsf{P}) \bmod q$, for $q = |image(\phi)|$.[2]
4. Compute $s = \omega + c \cdot a$.

The triple $(t,c,s) = NIZKP\{(a) : b = \phi(a)\}$ is the resulting non-interactive preimage proof, which can be published without revealing any information about a. Note that $image(\phi) = Y$ holds in many concrete instantiations of the preimage proof, which implies $q = |Y|$. To verify a given proof $\pi = (t,c,s)$, V performs the following check:

$$verify(\pi) = \begin{cases} accept, & \text{if } c = hash(b,t,\mathsf{P}) \bmod q \text{ and } \phi(s) = t \cdot b^c, \\ reject, & \text{otherwise.} \end{cases} \tag{3.11}$$

3.6.2. Examples

Knowledge of Discrete Logarithm (Schnorr)

- Let g be a generator of G_q
- Let $c = g^m$ be a publicly known commitment of $m \in \mathbb{Z}_q$
- P proves knowledge of m using the Σ-protocol for:

$$a = m,$$
$$b = c,$$
$$\phi(x) = g^x,$$

where $\phi : \underbrace{\mathbb{Z}_q}_{X} \to \underbrace{G_q}_{Y}$

Equality of Discrete Logarithms

- Let g_1 and g_2 be generators of G_q
- Let $c_1 = g_1^m$ and $c_2 = g_2^m$ be public commitments of $m \in \mathbb{Z}_q$
- P proves knowledge of m using the Σ-protocol for:

$$a = m,$$
$$b = (c_1, c_2),$$
$$\phi(x) = (g_1^x, g_2^x),$$

where $\phi : \underbrace{\mathbb{Z}_q}_{X} \to \underbrace{G_q \times G_q}_{Y}$

[2] Making the challenge c dependent on the prover's identity P prevents attacks based on copying proofs from another prover. We do not specify the exact format of the identifier. When generating the challenge for b, t, and P, we apply *hash* recursively, first on b and t, and then on P, i.e., $hash((b,t),\mathsf{P}) = hash(hash(hash(b) \mid\mid hash(t)) \mid\mid hash(\mathsf{P}))$.

- Note that $t = (t_1, t_2)$

3.6.3. Composition of Preimage Proofs

AND Composition

- Consider n one-way group homomorphism $\phi_i : X_i \to Y_i$
- Let $b_1, \dots, b_n$ be publicly known, where $b_i = \phi_i(a_i)$
- P proves knowledge of $a_1, \dots, a_n$ using the Σ-protocol for:

$$\begin{aligned}
&a = (a_1, \dots, a_n),\\
&b = (b_1, \dots, b_n),\\
&\phi(x_1, \dots, x_n) = (\phi_1(x_1), \dots, \phi_n(x_n)),
\end{aligned}$$

  where $\phi : \underbrace{X_1 \times \cdots \times X_n}_{X} \to \underbrace{Y_1 \times \cdots \times Y_n}_{Y}$

- Note that $\omega = (\omega_1, \dots, \omega_n)$, $t = (t_1, \dots, t_n)$, $s = (s_1, \dots, s_n)$, which implies proofs of size $O(n)$

Equality Proof

- Consider n one-way group homomorphism $\phi_i : X \to Y_i$
- Let $b_1, \dots, b_n$ be publicly known, where $b_i = \phi_i(a)$
- P proves knowledge of a using the Σ-protocol for:

$$\begin{aligned}
&a,\\
&b = (b_1, \dots, b_n),\\
&\phi(x) = (\phi_1(x), \dots, \phi_n(x)),
\end{aligned}$$

  where $\phi : X \to \underbrace{Y_1 \times \cdots \times Y_n}_{Y}$

- Note that $t = (t_1, \dots, t_n)$, which implies proofs of size $O(n)$

3.7. Threshold Cryptosystem

A cryptosystem such as ElGamal is called threshold cryptosystem, if the private decryption key x is shared among n parties, and if the decryption can be performed by a threshold number of parties $t \leq n$ without explicitly reconstructing x and without disclosing any information about the individual key shares x_i. A general threshold version of the ElGamal cryptosystem results from sharing the private key x using Shamir's secret sharing scheme [? ?]. To avoid the need for a trusted third party to generate the shares of the private key, it is possible to let the n parties execute a distributed key generation protocol [?]. We do not further introduce these techniques here, but we will assume their application throughout this document, for example by saying that some parties jointly generate a private key or that they jointly decrypt a ciphertext.

A threshold cryptosystem, which is limited to the particular case of $t = n$, is called *distributed cryptosystem*. A simple distributed version of the ElGamal cryptosystem results from setting $x = \sum_i x_i$. To avoid that x gets publicly known, each of the n parties secretly selects its own key share $x_i \in_R \mathbb{Z}_q$ and publishes $y_i = g^{x_i}$ as a commitment of x_i. The product $y = \prod_i y_i = g^{\sum_i x_i} = g^x$ is then the common public encryption key. If $E = (a, b) = \mathit{encrypt}_y(m, r)$ is a given encryption, then m can be jointly recovered if each of the n parties computes $a_i = a^{-x_i}$ using its own key share x_i. The resulting product $a^{-x} = \prod_i a_i$ can then be used to derive $m = \mathit{decrypt}_x(E) = a^{-x} \cdot b$ from b.[3] Instead of performing this simple operation in parallel, it is also possible to perform essentially the same operation sequentially in form of a *partial decryption function* $\mathit{decrypt}'_{x_i}(E) = (a, a^{-x_i} \cdot b)$. Applying $\mathit{decrypt}'_{x_i}$ "removes" from E the public key share y_i by transforming it into a new encryption $E' = \mathit{decrypt}'_{x_i}(E)$ for a new public key $y \cdot y_i^{-1}$. If all public key shares are removed in this way (in an arbitrary order), we obtain a trivial encryption (a, m) from which m can be extracted.

To guarantee the correct outcome of a threshold or distributed decryption, all involved must prove that they followed the protocol properly. In the case of the above distributed version of the ElGamal cryptosystem, each party must deliver two types of non-interactive zero-knowledge proofs:

- $\mathit{NIZKP}\{(x_j) : y_j = g^{x_j}\}$, to prove knowledge of the discrete logarithm of y_j after committing to x_j,
- $\mathit{NIZKP}\{(x_j) : y_j = g^{x_j} \wedge a_j = a^{-x_j}\}$, to prove equality of the discrete logarithms of y_j and a_j^{-1} after computing a_j.

Note that the first proof seems to be subsumed by the second proof, but it is important to provide the first proof along with y_j to guarantee the correctness of y *before* using it as a public encryption key.

If $\{E_1, \ldots, E_N\}$ is a batch of encryptions $E_i = (a_i, b_i)$ to decrypt and $a_{i,j} = a_i^{-x_j}$ the corresponding partial decryptions, then it is more efficient to provide a single combined proof,

$$\mathit{NIZKP}\{(x_j) : y_j = g^{x_j} \wedge (\bigwedge_i a_{i,j} = a_i^{-x_j})\}, \tag{3.12}$$

instead of N individual proofs of the second type. As discussed in Subsection 3.6, a combined proof like this can be implemented efficiently as a batch proof.

3.8. Verifiable Mix-Nets

not yet implemented

[3]Alternatively, each party may compute $m_i = \mathit{decrypt}_{x_i}(E) = a^{-x_i} \cdot b$ by applying the normal ElGamal decryption function. The plaintext message can then be recovered by $m = b^{1-n} \cdot \prod_i m_i$.

Part II.

Formal Specification

4. UniBoard

UniBoard is a public bulletin board, which can be used for posting public messages, so that they can be read by anybody. A user who posts a message to the board is called *author*, and a user reading from the board is called *reader*. As the requirements of a bulletin board are highly depended of the application, UniBoard uses a configurable component model with a generic interface for the components. This allows UniBoard to be configured exactly as needed. The following section describes the basic operations of UniBoard for users (Section 4.1). UniBoard supports various properties, which provide different guarantees to authors and readers. We distinguish between *posting properties* (Section 4.2) and *query properties* (Section 4.3).

4.1. Basic Operations

The UniBoard interface consists of two principal operations, one for posting a new message to the board and one for reading the board's current content. Let $\mathcal{M}$ denote the *message space*, the set of possible messages the board can accept (for example the language A^* of all finite strings over an alphabet A). Furthermore, let $\alpha_i \in \mathcal{A}_i$ and $\beta_i \in \mathcal{B}_i$ be so-called *attributes*, which represent additional information written to the board along with every message. We distinguish between *user attributes* α_i provided by the author and *board attributes* β_i provided by UniBoard. The exact shape of the user and board attributes depends on the properties of UniBoard. They will be introduced in the following sections.

- $\text{Post}(m, \alpha) : \beta$

  To publish a message $m \in \mathcal{M}$ on the board, the author of m calls this operation with user attributes $\alpha = (\alpha_1, \dots, \alpha_u) \in \mathcal{A}$ for $\mathcal{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_u$. Additional board attributes $\beta = (\beta_1, \dots, \beta_v) \in \mathcal{B}$ are generated by UniBoard, where $\mathcal{B} = \mathcal{B}_1 \times \cdots \times \mathcal{B}_v$. UniBoard returns the board attributes β to the author when the message is accepted. The triple $p = (m, \alpha, \beta) \in \mathcal{M} \times \mathcal{A} \times \mathcal{B}$ is the information stored on the board. If $\mathcal{P}$ denotes the board's current set of such *posts*, then it is updated by $\mathcal{P} \leftarrow \mathcal{P} \cup \{p\}$ when p is added.

- $\text{Get}(Q) : R, \gamma$

  This operation is called to retrieve the published data from the board. For a given query $Q \subseteq \mathcal{M} \times \mathcal{A} \times \mathcal{B}$, UniBoard returns the subset of posts $R = Q \cup \mathcal{P}$ satisfying the query and some additional *result attributes* γ. The exact shape of γ depends on the properties supported by UniBoard.

For a query restricting only the i-th user attribute to a single value $\alpha_i \in \mathcal{A}_i$ or to a subset of values $A_i \subseteq \mathcal{A}_i$, we use the simplified notation $Q = \langle \alpha_i \rangle$ or $Q = \langle A_i \rangle$, respectively. Similarly, we write $\langle \alpha_i, \alpha_j \rangle = \langle \alpha_i \rangle \cap \langle \alpha_j \rangle$ or $\langle A_i, A_j \rangle = \langle A_i \rangle \cap \langle A_j \rangle$ for restrictions on multiple user attributes i and j. The same notational convention can be applied to board attributes or to mixed restrictions on user and board attributes. For a query Q, the resulting subset of posts $R \subseteq \mathcal{P}$ will also be denoted by $\mathcal{P}_Q$.
Certain properties work with a sublist of user attributes α_I where $I \subseteq \{1, \ldots, u\}$. The same can be denoted for board attributes with β_I.

4.2. Posting Properties

In the simplest case of a public bulletin board, all messages posted to the board are accepted, published, and kept forever. The problem with such a totally unrestricted board is that it will also accept any irrelevant, improper, or unauthorized message. There may be applications requiring a board with absolutely no control over its content, but filtering unwanted messages is often a desirable property for an application to run smoothly. Therefore, we assume that a board has a number of publicly known posting regulations. A post $p = (m, \alpha, \beta)$ is called *valid* if it satisfies these regulations, and *invalid* otherwise.

In this subsection, we introduce such regulations for UniBoard. Their goal is to guarantee various properties of the post operation. They are achieved by corresponding user or board attributes. In each of the following eight subsections, we discuss one such property and the necessary attributes to achieve it. In total, there will be four user and four board attributes. Based on a precise specification of these attributes, we will be able to give a detailed description of the UniBoard posting process, which is induced each time the post operation is called (Section 4.5).

Property 1: Sectioned. A public bulletin board is called *sectioned*, if it consists of multiple equally shaped sections. The goal of a sectioned bulletin board is to group related and to separate unrelated messages. Let $\mathcal{S}$ be the set of available sections. To enable the dispatching of an incoming post into the right section, the author must provide the section $s \in \mathcal{S}$ as a user attribute. A post containing an invalid section $s \notin \mathcal{S}$ is invalid and must be rejected by the board. In UniVote, the data of each election will be written to an individual section.

Property 2: Grouped. In a *grouped* bulletin board, messages are organized into groups. Typically, messages contained in the same group are similar in shape and content. Let $\mathcal{G}$ be the set of available groups. When posting a message, the author must indicate the group $g \in \mathcal{G}$ to which the message belongs as a user attribute. A post containing an invalid group $g \notin \mathcal{G}$ is invalid and must be rejected by the board. Note that groups are independent of sections, i.e., every section in a sectioned board works with the same set of groups $\mathcal{G}$. Figure 4.1 show an example of a board with three sections and three groups.

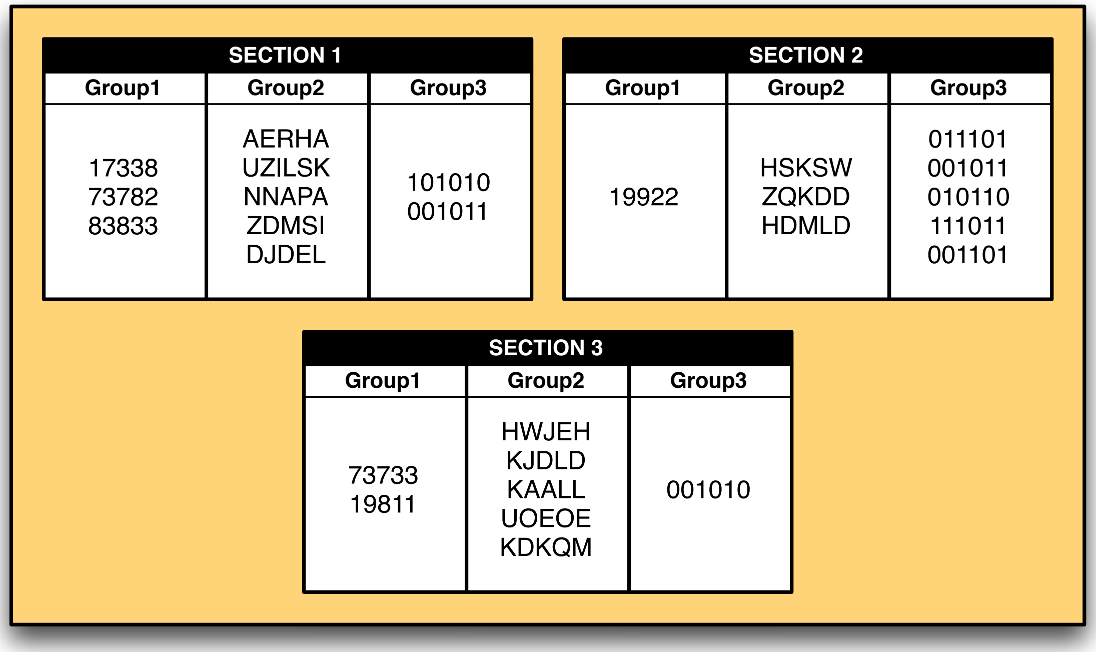

Figure 4.1.: Example of a structured bulletin board with three sections, three groups, and corresponding messages.

Property 3: Typed. A grouped bulletin board is called *typed*, if each group $g_i \in \mathcal{G}$ defines its own set $\mathcal{M}_i \subseteq \mathcal{M}$ of valid messages. $\mathcal{M}_i$ is called *type* of g_i. In a typed board, an incoming messages m for group g_i is accepted if $m \in \mathcal{M}_i$, and all other messages are rejected. The example of Figure 4.1 shows a typed board with different types of messages for each group, for example $\mathcal{M}_1 = \{0, \ldots, 9\}^5$, $\mathcal{M}_2 = \{A, \ldots, Z\}^5$, $\mathcal{M}_3 = \{0, 1\}^6$.

Property 4: Access-Controlled. In certain applications, only a well-defined set of users is authorized to post messages to the bulletin board. A bulletin board is called *access-controlled*, if it provides an access-control mechanism that identifies the author of a message and rejects the message if the user is unauthorized. To enable the board doing this check, we assume that a set $\mathcal{K}$ of public signature keys—one for each authorized user—is known to the board at every moment. This set is either *static* or *dynamic*, depending on whether the set of authorized users is fixed or can change over time. In the static case, $\mathcal{K}$ is publicly known and can not be changed, whereas in the dynamic case, $\mathcal{K} = K(\mathcal{P}, \alpha, t)$ is defined implicitly by a publicly known function K, where $\mathcal{P}$ is the current set of posts published on the board, α the list of user attributes accompanying the message, and t the current time. The arguments of K are optional, i.e., not all three of them are relevant in every case. For a message m to be accepted by the board, it must be signed by the user using a private signature key sk for a public key in $vk \in \mathcal{K}$. The signature $S_m = sign_{sk_U}(m, \alpha_I)$ and the public key vk are included in the post as user attributes. The board can then perform the checks $vk \in \mathcal{K}$ and $verify_{vk}(m, \alpha_I, S_m)$ to decide if the user of m is authorized.

Property 5: Ordered. The ordered property ensures that all published posts $\mathcal{P}_{\langle s\rangle}$ for a section have a total order. This is achieved by adding a sequence number $n \in \mathbb{N}$ to β for the post, where $n = |\mathcal{P}_{\langle s\rangle}| + 1$.

Property 6: Chronological. A chronological board adds to every incoming message a timestamp $t \in \mathcal{T}$ into β, which denotes when the message was received by the board.

Property 7: Append-Only. The append-only property ensures that no posted message can be removed from the board or be changed. So, $\mathcal{P}_{\langle t\rangle} \subseteq \mathcal{P}_{\langle t+1\rangle}$ where $\mathcal{P}_{\langle t\rangle}$ represent the content on the board at time t. The solution presented here requires that the board has the ordered property. The board creates a hash-chain H_s over $\mathcal{P}_{\langle s\rangle}$. For each p_i it creates a hash $h_i \in H_s$ which is the result of the hash function $Hash(h_{i-1}, p_i, \alpha, \beta_I)$, where h_{i-1} is the hash of the preceding post p_{i-1} and $h_0 = 0$. This hash h_i is then added to β.

Property 8: Certified Posting. With certified posting every user receives after a successful post a receipt from the board, which confirms that the message has been published. Upon posting a message m, the board creates the signature $S_p = Sign_{sk_{BB}}(m, \alpha, \beta_I)$ including the message, and the user attributes and a sub set of the board attributes β_I. This signature is added to β.

4.3. Query Properties

Property 9: Certified Reading. This property forces the board to commit to every result R it returns. This is achieved by adding a timestamp t and a signature $S_q = Sign_{sk_{BB}}(Q, R, \gamma_I)$ to γ.

4.4. Further Properties

Property 10: Notifying. The notification property of the board allows an entity e to register itself to be notified when a certain set of messages $\mathcal{P}(Q)$ is posted on the board. In order to accomplish that, two additional methods must be introduced:

- Register$(e, Q) : c$
  Where Q represents the query for the messages the entity is interested in and c a notification code, which can be used to unregister.
- Unregister$(c) : -$
  By providing his/her notification code c, one can unregister and will not receive any further notification.

4.5. Properties of UniVote and UniCert

For UniVote and UniCert a bulletin board that supports all properties described in the previous sections is used. This results in our UniBoard configuration supporting following four operations. Also note that Schnorr signatures are used for signing. Therefore every sign operation needs an additional random value r (Chapter 3.4).

4.5.1. Post

$\text{Post}(m, (s, g, S_m, vk)) : (t, n, S_p)$

1. Check that section s exists.
2. Check that group g exists.
3. Check if message m is of the type of group g.
4. Check that S_m is a correct signature of $(m, (s, g))$ and belongs to vk.
5. Check that k is authorized by checking $\mathcal{K}$.
6. Add the current time t to the message.
7. Define the total order n of the message in s .
8. Create the signature for the post $S_p = Sign_{sk_{BB}}(m, (s, g, S_m, vk), (t, n))$
9. Send notifications for this message if necessary.
10. Save the post.
11. Return (t, n, S_p) to the user.

A user can use the compact notation $\text{Post}(m, s, g)$, which represents the following three steps:

1. Create the Schnorr signature $S_m = sign_{sk_U}((m, (s, g)), r)$
2. Post the message on the board $\text{Post}(m, (s, g, S_m, vk))$
3. Validate and save the response (n, t, h, S_p).

4.5.2. Query

For the get operation the query is $Q \subseteq \mathcal{M} \times \mathcal{S} \times \mathcal{G} \times \Sigma_a \times \mathcal{K} \times \mathbb{N} \times \mathcal{T} \times \mathcal{H} \times \Sigma_p$.

$\text{Get}(Q) : R, (t, S_Q)$

1. Search all messages $\mathcal{P}_Q$ satisfying Q.
2. Set timestamp t to the current time.
3. Create the signature S_Q.
4. return the result to the reader.

Register$(e, Q) : c$

1. Generate a unique notification code c.
2. Save the notification and the notification code
3. Return c.

Unregister$(c) : -$

1. Check if there is notification active, that corresponds to the notification code c.
2. Remove the notification.

Multiple instances As there can be multiple instances of UniBoard running, they need to be recognisable. This is achieved by denoting every operation with an application identifier , e.g. $\text{Post}_{UV}(m, \alpha) : \beta$ for UniVote

5. UniCert

UniCert is a certification authority, that issues digital certificates used to authenticate users, to sign and/or encrypt messages. UniCert provides an interface where the user can authenticate themselves and request a certificate corresponding to their needs. UniCert uses UniBoard to publish all issued certificates.

Authentication A digital certificate must be linked to an entity, thus the entity has to authenticate itself to UniCert. Authentication in UniCert can be done via various identity providers (e.g. SwitchAAI, Google, ...). The user has to authenticate themselves against one of these providers which returns the resulting *authentication token* t_{auth} to UniCert. The form and content of the token can be different depending on which identity provider is used. However, it must contain at least one value allowing to uniquely identify the user among all users of this identity provider. UniCert then issues a certificate based on the identity information present in the token and on other properties provided by the requester. Therefore, UniCert uses a function $f(t_{auth})$, to extract the needed information from the authentication token, resulting in a identifier *id*. f must use at least one unique identifier present in the token in order to be able to output different results for two different inputs. UniCert defines a set $\mathcal{F}$ of functions f allowing to select information from the authentication token, or to make some processing on them before putting them in the certificate (e.g anonymization by using a one-way function). The user has to choose one of the functions provided by UniCert. A reference to the identity provider used is inserted in the certificate.

Application Identifier and Role A user should have the possibility to request multiple certificates. This allows them to manage them differently. For example, they can use a certificate for critical tasks and another one for less critical tasks. The way the user manages the private keys of these two certificates is certainly different. In order make it possible for the user to request multiple certificates, there must be a way to distinguish them. In some cases, this could be done using the cryptographic values in the certificate[1] but this is not always possible.[2] Therefore, two additional fields are added to the certificate: an *application identifier* a and a *role* r. The application identifier allows the user to request certificates for different applications. The application where the certificate is used can (but does not have to) check this field if necessary. The role allows to distinguish certificates issued for the same application. This way, the user can use different certificates for different purposes inside the same application. So, if an application wants to use roles, it has to define them. a and r are both strings. The pair (a, r) should be unique among all applications, but UniCert cannot ensure that fact since it does not know all applications.

[1] For example different p, q and generators in discrete logarithm certificates.

[2] For example in RSA certificates: e, n is always different and thus it cannot be defined that such setup is for such task

Certificate revocation Certificates are not explicitly revoked. An application can consider a certificate as implicitly revoked if a newer one has been issued for the same user, the same application identifier a, the same role r and the same identity provider.

Certificate generation procedure The involved parties in certificate generation procedure are UniCert (UC), the user U requesting the certificate, the UniBoard instance (UBC), and the identity providers $\mathsf{U} \in \mathcal{I}$ supported by UC. The process works as follows:

1. U chooses the following parameter: cryptographic setup s, function $f \in \mathcal{F}$, identity provider $\mathsf{U} \in \mathcal{I}$, application identifier a, role r.
2. U requests an authentication token t_{auth} to one U and provides the required credentials

Keys preparation and certificate request

3. Using s, U generates a private key x and compute the corresponding public key y
4. U signs the data sent to UC to prove knowledge of x: $S_{req} = sign_x(s, y, t_{auth}, f, I, a, r)$.[3]
5. U sends s, y, t_{auth}, f, U, a, r, S_{req} to UC

Certificate creation

6. UC checks if a, r, U and f are value of their respective set
7. UC checks if t_{auth} is valid
8. UC checks the validity of the cryptographic setup s
9. UC verifies S_{req} with y
10. UC generates the identifier id for U using f and t_{auth}
11. UC generates the certificate $Z_U = csertify_{sk_{\mathsf{UC}}}(id, y, t, s, a, r, I)$ where t is a timestamp
12. UC posts $(certificates : Z_U)$ to UBC
13. UC returns Z_U to U

UniCert and UniBoard As already said, UniCert publishes the issued certificates on UniBoard. Therefore, UniCert must define the different parameter required by UniBoard when posting and getting messages.

- $\mathcal{S}$: only one section is used to publish all certificates by UniCert. This avoids that a new section has to be created each time a new application identifier is used. So, $\mathcal{S} = \{\text{"unicert"}\}$
- $\mathcal{G}$: one group is needed for stocking the issued certificates. So, $\mathcal{G} = \{\text{"certificates"}\}$
- $\mathcal{K} = \{k_{\mathsf{UC}}\}$

[3]ElGamal encryption keys cannot properly be used to generate a signature. Thus, in case the user request a discrete logarithm certificate, a zero knowledge proof of knowledge of x replaces the signature. The other values used in the signature would be injected in the hash function of the non-interactive proof in order to also provide a commitment to these other values.

- $\mathcal{M}_{certificates}$: Format of a certificate.

6. UniVote

6.1. Introduction

6.1.1. Involved Parties

UniCert. UC

Election Administration. EA

Election Coordinator. EC

UniBoard (UniVote) UBV

UniBoard (UniCert) UBC

Talliers. $T = \{\mathsf{T}_1, \ldots, \mathsf{T}_t\}$

Mixers. $M = \{\mathsf{M}_1, \ldots, \mathsf{M}_m\}$

Voters. $V = \{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$

Number of ballots: $N \leq n$

6.1.2. Public Identifiers and Keys

Certificates for the following identifiers are assumed to be publicly known.

Bulletin Board (UniVote):

- Identifier: UBV
- Public certificate: Z_{UBV}, signed by CA at time t
- Public verification key: vk_{UBV}
- Private signature key: sk_{UBV}

Bulletin Board (UniCert):

- Identifier: UBC
- Public certificate: Z_{UBC}, signed by CA at time t
- Public verification key: vk_{UBC}
- Private signature key: sk_{UBC}

UniCert:

- Identifier: CA
- Public certificate: Z_{CA}, self-signed or certified by public certification authority at time t
- Public verification key: vk_{CA}
- Private signature key: sk_{CA}

Certificates for the following identifiers are assumed to be available on UBC.

Election Coordinator:

- Identifier: EC
- Public certificate: Z_{EC}, signed by CA at time t
- Public verification key: vk_{EC}
- Private signature key: sk_{EC}

Election Administration:

- Identifier: EA
- Public certificate: Z_{EA}, signed by CA at time t
- Public verification key: vk_{EA}
- Private signature key: sk_{EA}

Talliers: (for $1 \leq j \leq t$)

- Identifier: T_j
- Public certificate: Z_j, signed by CA at time t_j
- Public verification key: vk_j
- Private signature key: sk_j

Mixers: (for $1 \leq k \leq m$)

- Identifier: M_k
- Public certificate: Z_k, signed by CA at time t_k
- Public verification key: vk_k
- Private signature key: sk_k

Voters: (for $1 \leq i \leq n$)

- Identifier: V_i
- Personal credentials: $cred_i$ issued by the CA or an affiliated identity provider.

6.1.3. Posting and Getting Messages

Post involves signature and section id

Get involves checking the signature

6.2. Detailed Protocol Specification

6.2.1. Election Setup

The following tasks can be performed in advance, possibly long before the election starts.

a) Initialization

EA requests from EC to run an election. EC chooses a unique election identifier *id* and requests from UBV the initialization of the election. UBV performs the following steps:

1. Initialize a new section with identifier *id*.
2. Define EC to become the section coordinator.

EC performs the following steps:

3. Get $(\mathsf{certificate} : Z_{\mathsf{EA}})$ from UBC, where Z_{EA} is a UniVote election administration certificate.
4. Verify Z_{EA}.
5. Post $(\mathsf{administrationCertificate} : Z_{\mathsf{EA}})$ to UBV.

b) Election Definition

EC performs the following steps:

1. Select vk_{EA} from Z_{EA}.
2. Post $(\mathsf{accessRight} : vk_{\mathsf{EA}}, \mathsf{electionDefinition}, 1)$ to UBV.
3. Post $(\mathsf{accessRight} : vk_{\mathsf{EA}}, \mathsf{trustees}, 1)$ to UBV.
4. Post $(\mathsf{accessRight} : vk_{\mathsf{EA}}, \mathsf{securityLevel}, 1)$ to UBV.

EA performs the following steps:

5. Define election *title*.
6. Define election $period = (t_1, t_2)$.
7. Post $(\mathsf{electionDefinition} : title, period)$ to UBV.
8. Define talliers $T = \{\mathsf{T}_1, \ldots, \mathsf{T}_t\}$.
9. Define mixers $M = \{\mathsf{M}_1 \ldots, \mathsf{M}_m\}$.

10. Post ($\mathsf{trustees} : T, M$) to UBV.
11. Select security $level \in \{0, 1, 2, \ldots\}$.
12. Post ($\mathsf{securityLevel} : level$) to UBV.

EC performs the following steps:

13. Get ($\mathsf{trustees} : T, M$) from UBV.
14. For each $\mathsf{T}_j \in T$:
    a) Get ($\mathsf{certificate} : Z_j$) from UBC, where Z_j is a UniVote tallier certificate.
    b) Verify Z_j.

    Let $\mathcal{Z}_T = \{Z_j : 1 \leq j \leq t\}$ denote the corresponding set of certificates.
15. For each $\mathsf{M}_k \in M$:
    a) Get ($\mathsf{certificate} : Z_k$) from UBC, where Z_k is a UniVote mixer certificate.
    b) Verify Z_k.

    Let $\mathcal{Z}_M = \{Z_k : 1 \leq k \leq m\}$ denote the corresponding set of certificates.
16. Post ($\mathsf{trusteeCertificates} : \mathcal{Z}_T, \mathcal{Z}_M$) to UBV.

c) Cryptographic Setting

EC performs the following steps:

1. Get ($\mathsf{securityLevel} : level$) from UBV.
2. Select
    - $encryptSetting \in \Sigma_{\mathsf{ENCRYPT}}$,
    - $signSetting \in \Sigma_{\mathsf{SIGN}}$,
    - $hashSetting \in \Sigma_{\mathsf{HASH}}$

    according to $level$, where $\Sigma_{\mathsf{ENCRYPT}}$, Σ_{SIGN}, Σ_{HASH} are corresponding sets of predefined cryptographic settings for ElGamal encryptions, Schnorr signatures, and hash functions (see Chapter 7).
3. Post ($\mathsf{cryptoSetting} : encryptSetting, signSetting, hashSetting$) to UBV.

d) Shared Encryption Key

For each $Z_j \in \mathcal{Z}_T$, EC performs the following steps:

1. Select vk_j from Z_j.
2. Post $(\mathsf{accessRight} : vk_j, \mathsf{encryptionKeyShare}, 1)$ to UBV.

Each $\mathsf{T}_j \in T$ performs the following steps:

3. Get $(\mathsf{cryptoSetting} : \mathit{encryptSetting}, \mathit{signSetting}, \mathit{hashSetting})$ from UBV.
4. Select $\mathcal{G}_Q$ and G from $\mathit{encryptSetting}$.
5. Choose $x_j \in_R \mathbb{Z}_Q$.
6. Compute $y_j = G^{x_j}$.
7. Generate $\pi_j = \mathit{NIZKP}\{(x_j) : y_j = G^{x_j}\}$.
8. Post $(\mathsf{encryptionKeyShare} : y_j, \pi_j)$ to UBV.

EC performs the following steps:

9. For each $\mathsf{T}_j \in T$:
   a) Get $(\mathsf{encryptionKeyShare} : y_j, \pi_j)$ from UBV.
   b) Verify π_j.
10. Compute $y = \prod_j y_j$.
11. Post $(\mathsf{encryptionKey} : y)$ to UBV.

6.2.2. Election Preparation

The following tasks are performed shortly before the beginning of the election.

a) Definition of Election Options and Electoral Roll

EC performs the following step:

1. Post $(\mathsf{accessRight} : vk_{\mathsf{EA}}, \mathsf{electionDetails}, 1)$ to UBV.
2. Post $(\mathsf{accessRight} : vk_{\mathsf{EA}}, \mathsf{electoralRoll}, 1)$ to UBV.

EA performs the following steps:

3. Define the set of election options $\mathcal{Q} = \{Q_1, \ldots, Q_s\}$ and a list of election rules $\mathcal{R}$. This determines the set $\mathit{ValidVotes}(\mathcal{Q}, \mathcal{R})$ of valid votes (see Section 6.6 for details).
4. Define a detailed description of the election *issues* (type, title, options, rules, etc.).[1]
5. Select $\mathit{encode} \in \Sigma_{\mathsf{ENCODE}}$.
6. Post $(\mathsf{electionDetails} : \mathcal{Q}, \mathcal{R}, \mathit{issues}, \mathit{encode})$ to UBV.

[1] This information is only relevant for presenting the election options to the voter.

7. Define the *electoral roll* $V = \{\mathsf{V}_1, \dots, \mathsf{V}_n\}$.
8. Post $(\mathsf{electoralRoll} : V)$ to UBV.

EC performs the following steps:

9. Get $(\mathit{electoralRoll} : V)$ from UBV.
10. For every $V_i \in V$:
    - Get $(\mathsf{certificate} : Z_i)$ from UBC, where Z_i is the most recent UniVote voter certificate for *signSetting*.
    - Verify Z_i (if it exists).

    Let $\mathcal{Z}_V = \{Z_1, \dots, Z_{n'}\}$ denote the set of valid voter certificates, where $n' \leq n$ is the number of eligible voters with a valid UniVote voter certificate.
11. Post $(\mathsf{voterCertificates} : \mathcal{Z}_V)$ on UBV.

b) Mixing the Public Keys

EC performs the following steps:

1. Select $VK_0 = \{vk_1, \dots, vk_{n'}\}$ from $\mathcal{Z}_V$.
2. Let $g_0 = g$.

The following steps are repeated for every $\mathsf{M}_k \in M$ (in ascending order for $1 \leq k \leq m$):

3. EC performs the following steps:
    a) Post $(\mathsf{keyMixingRequest} : \mathsf{M}_k, VK_{k-1}, g_{k-1})$ to UBV.
    b) Select vk_k from $Z_k \in \mathcal{Z}_M$.
    c) Post $(\mathsf{accessRight} : vk_k, \mathsf{keyMixingResult}, 1)$ to UBV.
4. M_k performs the following steps:
    a) Get $(\mathsf{cryptoSetting} : \mathit{encryptSetting}, \mathit{signSetting}, \mathit{hashSetting}$ from UBV.
    b) Select G_q and g from *signSetting*.
    c) Get $(\mathsf{keyMixingRequest} : \mathsf{M}_k, VK_{k-1}, g_{k-1})$ from UBV.
    d) Choose $\alpha_k \in_R \mathbb{Z}_q$.
    e) Compute $g_k = g_{k-1}^{\alpha_k}$.
    f) Choose $\psi_k : [1, n] \to [1, n] \in_R \Psi_n$.
    g) Compute $VK_k = \mathit{shuffle}_{\psi_k}(VK_{k-1}, \alpha_k)$.
    h) Generate $\pi_k = \mathit{NIZKP}\{(\psi_k, \alpha_k) : VK_k = \mathit{shuffle}_{\psi_k}(VK_{k-1}, \alpha_k) \wedge g_k = g_{k-1}^{\alpha_k}\}$.
    i) Post $(\mathsf{keyMixingResult} : VK_k, g_k, \pi_k)$ to UBV.
5. EC performs the following steps:

a) Get ($\mathsf{keyMixingResult} : VK_k, g_k, \pi_k$) from UBV.

b) Verify π_k.

Let $\hat{VK} = VK_m$ denote the set of mixed public keys and $\hat{g} = g_m$ the *signature generator* for this election.

c) Finalizing Election Preparation

EC performs the following steps:

1. Post ($\mathsf{mixedKeys} : \hat{VK}, \hat{g}$) to UBV.
2. For every $\hat{vk}_i \in \hat{VK}$, post ($\mathsf{accessRight} : \hat{vk}_i, \mathsf{ballot}, 1, \mathit{period}$).
3. Get ($\mathsf{electionDefinition} : \mathit{title}, \mathit{period}$) from UBV.
4. Get ($\mathsf{electionDetails} : \mathcal{Q}, \mathcal{R}, \mathit{issues}, \mathit{encode}$) from UBV.
5. Post ($\mathsf{votingData} : \mathit{title}, \mathit{period}, \mathcal{Q}, \mathcal{R}, \mathit{issues}, \mathit{encode}, \mathit{encryptSetting}, \mathit{signSetting}, \mathit{hashSetting}, y, \hat{g}$) to UBV.[2]

6.2.3. Election Period

$\mathsf{V}_i \in V$ performs the following steps:

1. Get ($\mathsf{votingData} : \mathit{title}, \mathit{period}, \mathcal{Q}, \mathcal{R}, \mathit{issues}, \mathit{encode}, \mathit{encryptSetting}, \mathit{signSetting}, \mathit{hashSetting}, y, \hat{g}$) from UBV.
2. Select $\mathcal{G}_Q$ and G from *encryptSetting*.
3. Select G_q from *signSetting*.
4. Choose $v_i = (v_{i,1}, \ldots, v_{i,s}) \in \mathit{ValidVotes}(\mathcal{Q}, \mathcal{R})$.
5. Compute $w_i = \mathit{encode}_{\mathcal{Q},\mathcal{R}}(v_i) \in \mathbb{Z}_Q$.
6. Compute $w'_i = \mathit{subGroup}(w_i) \in \mathcal{G}_Q$.
7. Choose $r_i \in_R \mathbb{Z}_Q$.
8. Compute $E_i = \mathit{encrypt}_y(w'_i, r_i) \in \mathcal{G}_Q \times \mathcal{G}_Q$.
9. Generate $\pi_i = \mathit{NIZKP}\{(w'_i, r_i) : E_i = \mathit{encrypt}_y(w'_i, r_i)\}$.[3]
10. Compute $\hat{vk}_i = \hat{g}^{sk_i} \in \hat{VK}$.
11. Post ($\mathsf{ballot} : E_i, \pi_i$) to UBV using $\hat{vk}_i$.

[2]This post is only for improved convenience. It contains all the relevant information for casting a vote. Retrieving this information in a single step is the purpose of this post.

[3]Note that if $E_i = (a_i, b_i) = (G^{r_i}, w'_i \cdot y^{r_i})$ is an ElGamal encryption, then $\mathit{NIZKP}\{(w'_i, r_i) : E_i = \mathit{encrypt}_y(w'_i, r_i)\}$ is equivalent to $\mathit{NIZKP}\{(r_i) : a_i = G^{r_i}\}$, which implies knowledge of w'_i.

6.2.4. Mixing and Tallying

a) Ballot Validation

EC performs the following steps:

1. Get (ballot : $\mathcal{B}$) from UBV.[4]
2. For every $B_i = (E_i, \pi_i) \in \mathcal{B}$:
   a) Verify π_i.
   b) If π_i is valid, collect E_i.

Let $\mathcal{E}_0 = \{E_1, \ldots, E_N\} \in (\mathcal{G}_Q \times \mathcal{G}_Q)^N$ be list of valid encrypted votes for $N \leq n'$.

b) Mixing the Encrypted Votes

The following steps are repeated for every $\mathsf{M}_k \in M$ (in ascending order for $1 \leq k \leq m$):

1. EC performs the following steps:
   a) Select vk_k from $Z_k \in \mathcal{Z}_M$.
   b) Post ($\mathsf{accessRight}$: $vk_k, \mathsf{voteMixingResult}, 1$) to UBV.
   c) Post ($\mathsf{voteMixingRequest}$: $\mathsf{M}_k, \mathcal{E}_{k-1}$) to UBV.
2. M_k performs the following steps:
   a) Get ($\mathsf{encryptionSetting}$: $\mathcal{G}_Q, G$) from UBV.
   b) Get ($\mathsf{voteMixingRequest}$: $\mathsf{M}_k, \mathcal{E}_{k-1}$) from UBV.
   c) Choose $R_k = (r_{k,1}, \ldots, r_{k,N}) \in_R \mathbb{Z}_Q^N$.
   d) Choose $\tau_k : [1, N] \to [1, N] \in_R \Psi_N$.
   e) Compute $\mathcal{E}_k = \mathit{shuffle}_{\tau_k}(\mathcal{E}_{k-1}, R_k)$.
   f) Generate $\pi'_k = \mathit{NIZKP}\{(\tau_k, R_k) : \mathcal{E}_k = \mathit{shuffle}_{\tau_k}(\mathcal{E}_{k-1}, R_k)\}$.
   g) Post ($\mathsf{voteMixingResult}$: $\mathcal{E}_k, \pi'_k$) to UBV.
3. EC performs the following steps:
   a) Get ($\mathsf{voteMixingResult}$: $\mathcal{E}_k, \pi'_k$) from UBV.
   b) Verify π'_k.

Let $\hat{\mathcal{E}} = \mathcal{E}_m$ the set of mixed encrypted votes.

[4] Using an extensive check.

c) Decrypting and Tallying the Votes

EC performs the following steps:

1. Post $(\mathsf{mixedVotes} : \hat{\mathcal{E}})$ to UBV.
2. For each $Z_j \in \mathcal{Z}_T$:
    a) Select vk_j from Z_j.
    b) Post $(\mathsf{accessRight} : vk_j, \mathsf{partialDecryption}, 1)$ to UBV.

Each $\mathsf{T}_j \in T$ performs the following steps:

1. Get $(\mathsf{mixedVotes} : \hat{\mathcal{E}})$ from UBV.
2. Select $A = (a_1, \dots, a_N) \in \mathcal{G}_Q^N$ from $\hat{\mathcal{E}} = \{(a_i, b_i) : 1 \leq i \leq N\}$.
3. Compute $A_j = (a_{1,j}, \dots, a_{N,j})$ for $a_{i,j} = a_i^{-x_j} \in \mathcal{G}_Q$.
4. Generate $\pi'_j = \mathit{NIZKP}\{(x_j) : y_j = G^{x_j} \wedge \left(\bigwedge_i a_{i,j} = a_i^{-x_j}\right)\}$.
5. Post $(\mathsf{partialDecryption} : A_j, \pi'_j)$ to UBV.

EC performs the following steps:

6. For each $\mathsf{T}_j \in T$:
    a) Get $(\mathsf{partialDecryption} : A_j, \pi'_j)$ from UBV.
    b) Verify π'_j.
7. For all $1 \leq i \leq N$:
    a) Compute $w'_i = b_i \cdot \prod_j a_{i,j} \in \mathcal{G}_Q$.
    b) Compute $w_i = \mathit{subGroup}^{-1}(w'_i) \in \mathbb{Z}_Q$.
8. Post $(\mathsf{decryptedVotes} : \mathcal{W})$ to UBV, where $\mathcal{W} = \{w_1, \dots, w_N\}$ denotes the list of decrypted votes.

EA performs the following steps:

9. Get $(\mathsf{decryptedVotes} : \mathcal{W})$ from UBV.
10. For all $1 \leq i \leq N$, compute $v_i = \mathit{encode}^{-1}_{\mathcal{Q},\mathcal{R}}(w_i)$.
11. Let $\mathcal{V} = \{v_1, \dots, v_N\} \cap \mathit{ValidVotes}(\mathcal{Q}, \mathcal{R}) = \{v'_1, \dots, v'_{N'}\}$ be the list of $N' \leq N$ valid plaintext votes $v'_i = (v'_{i,1}, \dots, v'_{i,s})$, where $v'_{i,j}$ denotes the number of votes for Q_j.
12. Post $(\mathsf{decodedVotes} : \mathcal{V})$ to UBV.
13. For each $Q_j \in \mathcal{Q} = \{Q_1, \dots, Q_s\}$, compute $r_j = \sum_{j=1}^{N'} v'_{i,j}$.
14. Post $(\mathsf{electionResult} : \mathit{result})$ to UBV, where $\mathit{result} = \{r_1, \dots, r_s\}$ denotes the *election result*.

6.3. Late Voter Certificates

The previously described protocol requires that all UniVote voter certificates exist prior to an election. In some contexts, however, it will be impossible to enforce the existence of all certificates when the election starts. Eligible voters without a certificate would then be excluded from casting a vote, even if they receive a valid voter certificate during the election. To handle such *late voter certificates*, the following procedure is invoked. Recall that V denotes the electoral roll published on UBV (see Subsection 6.2.2), whereas $\mathcal{Z}_V$ is the list of voter certificates available at the beginning of the voting period.

6.3.1. General Procedure

Upon notification of a newly issued voter certificate Z'_i for V_i, EC performs the following steps:

1. Verify Z'_i.
2. Check if $\mathsf{V}_i \in V$.
3. Post $(\mathsf{newVoterCertificate} : Z'_i)$ to UBV.
4. Get $(\mathsf{addedVoterCertificate} : \mathcal{Z}_A)$ from UBV.
5. Get $(\mathsf{cancelledVoterCertificate} : \mathcal{Z}_C)$ from UBV.
6. Check if $(\mathcal{Z}_V \cup \mathcal{Z}_A) \setminus \mathcal{Z}_C$ contains a voter certificate for V_i. If this is the case, let Z_i denote this certificate.
   a) Select vk_i from Z_i.
   b) Get $\hat{vk}_i$ from calling the sub-routine described in Section 6.3.2 for vk_i.
   c) Get $(\mathsf{ballot} : \hat{vk}_i)$. Abort if a ballot exists for $\hat{vk}_i$.
   d) Post $(\mathsf{cancelledVoterCertificate} : Z_i)$ to UBV.
   e) Post $(\mathsf{accessRight} : \hat{vk}_i, \mathsf{ballot}, 0)$.
7. Select vk'_i from Z'_i.
8. Get $\hat{vk}'_i$ from calling the sub-routine described in Section 6.3.2 for vk'_i.
9. Post $(\mathsf{addedVoterCertificate} : Z'_i)$ to UBV.
10. Post $(\mathsf{accessRight} : \hat{vk}'_i, \mathsf{ballot}, 1, \mathit{period})$ to UBV.

6.3.2. Mixing a Single Public Key

Let $vk_i = vk_{i,0}$ be a single key to be "mixed" in essentially the same way as the mixing of a list of keys as described in Section 6.2.2. A simplified sub-routine for this is called twice in the above general procedure for different input keys. In this sub-routine, the following steps are repeated for every $\mathsf{M}_k \in M$ (in ascending order for $1 \leq k \leq m$):

1. EC performs the following steps:
   a) Post $(\mathsf{singleKeyMixingRequest} : \mathsf{M}_k, vk_i, vk_{i,k-1})$ to UBV.[5]
   b) Select vk_k from $Z_k \in \mathcal{Z}_M$.
   c) Post $(\mathsf{accessRight} : vk_k, \mathsf{singleKeyMixingRequest}, +1)$ to UBV.
2. M_k performs the following steps:
   a) Get $(\mathsf{singleKeyMixingRequest} : \mathsf{M}_k, vk_i, vk_{i,k-1})$ from UBV.
   b) Compute $vk_{i,k} = vk_{i,k-1}^{\alpha_k}$
   c) Generate $\pi_{i,k} = NIZKP\{(\alpha_k) : vk_{i,k} = vk_{i,k-1}^{\alpha_k} \wedge g_k = g_{k-1}^{\alpha_k}\}$.
   d) Post $(\mathsf{singleKeyMixingResult} : vk_i, vk_{i,k-1}, vk_{i,k}, \pi_{i,k})$ to UBV.
3. EC performs the following steps:
   a) Get $(\mathsf{singleKeyMixingResult} : vk_i, vk_{i,k-1}, vk_{i,k}, \pi_{i,k})$ from UBV.
   b) Verify $\pi_{i,k}$.

Let $\hat{vk}_i = vk_{i,m}$ denote the result of this sub-routine.

6.4. Summary of Election Data

6.5. Universal Verification

Let V be a verifier. To verify the correctness of the election result, V performs the following steps:

a) Election Setup

-

[5] We include the original input key vk_i for convenience reasons.

b) Election Preparation

-

c) Mixing and Tallying

-

6.6. Options, Rules, and Votes

To allow a variety of different election types, we consider two finite sets, a set $\mathcal{Q} = \{Q_1, \dots, Q_s\}$ of possible *election options* and a set $\mathcal{R}$ of *election rules*. For each election option $Q_j \in \mathcal{Q}$ in a given election, the UniVote outputs the number r_j of valid votes that Q_j has received from the voters. Each election rule in $\mathcal{R}$ defines some constraints on how voters can distribute their votes among the election options. We use $v_{i,j}$ to denote this number for a particular ballot B_i and election option Q_j. The tuple $v_i = (v_{i,1}, \dots, v_{i,s}) \in \mathit{ValidVotes}(\mathcal{Q}, \mathcal{R})$ represents a valid vote, where $\mathit{ValidVotes}(\mathcal{Q}, \mathcal{R})$ denotes the set of all valid votes for given sets $\mathcal{Q}$ and $\mathcal{R}$.

We distinguish three types of election rules:

- *Summation-Rule*: The sum of votes for election options in a subset $\mathcal{Q}' \subseteq \mathcal{Q}$ is within a certain range $[a, b]$, i.e., $\sum\limits_{Q_j \in \mathcal{Q}'} v_{i,j} \in [a, b]$. Such rules will be denoted by $\Sigma : \mathcal{Q}' \rightarrow [a, b]$.
- *Cumulation-Rule*: For each election option in a subset $\mathcal{Q}' \subseteq \mathcal{Q}$, the number of votes is within a certain range $[a, b]$, i.e., $v_{i,j} \in [a, b]$ for all $Q_j \in \mathcal{Q}'$. Such rules will be denoted by $\forall : \mathcal{Q}' \rightarrow [a, b]$.
- *Distinctness-Rule*: For each election option in a subset $\mathcal{Q}' \subseteq \mathcal{Q}$, the number of votes is either equal to 0 or unique within $\mathcal{Q}'$, i.e., $v_{i,j} > 0$ implies $v_{i,j} \neq v(Q'_j)$ for all other election options $Q'_j \in \mathcal{Q}' \setminus \{c_j\}$. Such rules will be denoted by $\neq : \mathcal{Q}'$.

Two or several sets of options and sets of rules can be combined to describe multiple elections that run in parallel. We call this operation *composition of elections* and denote it by

$$(\mathcal{Q}_1, \mathcal{R}_1) \circ (\mathcal{Q}_2, \mathcal{R}_2) = (\mathcal{Q}_1 \cup \mathcal{Q}_2, \mathcal{R}_1 \cup \mathcal{R}_2)$$

for two sets of options and corresponding sets of rules. Note that this can be used to describe party-list elections (see example below).

Examples

- Referendum: 1-out-of-2

$$\mathcal{Q} = \{\mathsf{yes}, \mathsf{no}\}, \ \mathcal{R} = \left\{ \begin{array}{c} \Sigma : \{\mathsf{yes}, \mathsf{no}\} \rightarrow [1, 1] \\ \forall : \{\mathsf{yes}, \mathsf{no}\} \rightarrow [0, 1] \end{array} \right\}$$

- Referendum with Null Votes: max-1-out-of-2

$$\mathcal{Q} = \{\mathsf{yes}, \mathsf{no}\},\ \mathcal{R} = \begin{Bmatrix} \Sigma : \{\mathsf{yes}, \mathsf{no}\} \rightarrow [0,1] \\ \forall : \{\mathsf{yes}, \mathsf{no}\} \rightarrow [0,1] \end{Bmatrix}$$

or

$$\mathcal{Q} = \{\mathsf{yes}, \mathsf{no}, \mathsf{null}\},\ \mathcal{R} = \begin{Bmatrix} \Sigma : \{\mathsf{yes}, \mathsf{no}, \mathsf{null}\} \rightarrow [1,1] \\ \forall : \{\mathsf{yes}, \mathsf{no}, \mathsf{null}\} \rightarrow [0,1] \end{Bmatrix}$$

- Referendum List: s times 1-out-of-2

$$\mathcal{Q} = \{\mathsf{yes}_1, no_1, \ldots, \mathsf{yes}_s, no_s\},\ \mathcal{R} = \begin{Bmatrix} \Sigma : \{\mathsf{yes}_1, no_1\} \rightarrow [1,1] \\ \vdots \qquad \vdots \qquad \vdots \\ \Sigma : \{\mathsf{yes}_s, no_s\} \rightarrow [1,1] \\ \forall : \{\mathsf{yes}_1, no_1, \ldots, \mathsf{yes}_s, no_s\} \rightarrow [0,1] \end{Bmatrix}$$

- Multiple-Choice Referendum (Plurality Voting): 1-out-of-n

$$\mathcal{Q} = \{Q_1, \ldots, Q_n\},\ \mathcal{R} = \begin{Bmatrix} \Sigma : \{Q_1, \ldots, Q_n\} \rightarrow [1,1] \\ \forall : \{Q_1, \ldots, Q_n\} \rightarrow [0,1] \end{Bmatrix}$$

- Multiple-Choice Referendum with Null Votes: max-1-out-of-n

$$\mathcal{Q} = \{Q_1, \ldots, Q_n\},\ \mathcal{R} = \begin{Bmatrix} \Sigma : \{Q_1, \ldots, Q_n\} \rightarrow [0,1] \\ \forall : \{Q_1, \ldots, Q_n\} \rightarrow [0,1] \end{Bmatrix}$$

or

$$\mathcal{Q} = \{Q_1, \ldots, Q_n, \mathsf{null}\},\ \mathcal{R} = \begin{Bmatrix} \Sigma : \{Q_1, \ldots, Q_n, \mathsf{null}\} \rightarrow [1,1] \\ \forall : \{Q_1, \ldots, Q_n, \mathsf{null}\} \rightarrow [0,1] \end{Bmatrix}$$

- Approval Voting: max-n-out-of-n

$$\mathcal{Q} = \{Q_1, \ldots, Q_n\},\ \mathcal{R} = \big\{ \forall : \{Q_1, \ldots, Q_n\} \rightarrow [0,1] \big\}$$

- Range Voting: Up to s votes per option

$$\mathcal{Q} = \{Q_1, \ldots, Q_n\},\ \mathcal{R} = \big\{ \forall : \{Q_1, \ldots, Q_n\} \rightarrow [0,s] \big\}$$

- Plurality-at-Large Voting / Limited Voting: k-out-of-n

$$\mathcal{Q} = \{Q_1, \ldots, Q_n\},\ \mathcal{R} = \begin{Bmatrix} \Sigma : \{Q_1, \ldots, Q_n\} \rightarrow [0,k] \\ \forall : \{Q_1, \ldots, Q_n\} \rightarrow [0,1] \end{Bmatrix}$$

- Cumulative Voting: k votes in total, up to $s \leq k$ votes per option

$$\mathcal{Q} = \{Q_1, \ldots, Q_n\},\ \mathcal{R} = \left\{ \begin{array}{c} \Sigma : \{Q_1, \ldots, Q_n\} \rightarrow [0, k] \\ \forall : \{Q_1, \ldots, Q_n\} \rightarrow [0, s] \end{array} \right\}$$

  Note that $k > n$ is allowed here.

- Preferential Voting (Borda Count): Ranks from 1 to n

$$\mathcal{Q} = \{Q_1, \ldots, Q_n\},\ \mathcal{R} = \left\{ \begin{array}{l} \ \forall : \{Q_1, \ldots, Q_n\} \rightarrow [1, n] \\ \neq : \{Q_1, \ldots, Q_n\} \end{array} \right\}$$

- Preferential Voting (Borda Count): Ranks from 1 to k only

$$\mathcal{Q} = \{Q_1, \ldots, Q_n\},\ \mathcal{R} = \left\{ \begin{array}{l} \ \forall : \{Q_1, \ldots, Q_n\} \rightarrow [1, k] \\ \neq : \{Q_1, \ldots, Q_n\} \end{array} \right\}$$

- Party-List Election with Cumulation: Composition of cumulative voting over a set of candidates and plurality voting over a set of party-lists

$$\begin{aligned} &\mathcal{Q} = \{C_1, \ldots, C_n\} \cup \{L_1, \ldots, L_m\}, \\ &\mathcal{R} = \left\{ \begin{array}{c} \Sigma : \{C_1, \ldots, C_n\} \rightarrow [0, k] \\ \forall : \{C_1, \ldots, C_n\} \rightarrow [0, s] \end{array} \right\} \cup \left\{ \begin{array}{c} \Sigma : \{L_1, \ldots, L_m\} \rightarrow [1, 1] \\ \forall : \{L_1, \ldots, L_m\} \rightarrow [0, 1] \end{array} \right\} \end{aligned}$$

  Null votes (with respect to party-lists) can be handled as shown above.

Author	Group	Amount	Content	Readers
EC	administrationCertificate	1	Z_{EA}	–
EA	electionDefinition	1	$title, period$	EC
EA	trustees	1	T, M	EC
EA	securityLevel	1	$level$	EC
EC	trusteeCertificates	1	$\mathcal{Z}_T, \mathcal{Z}_M$	–
EC	cryptoSetting	1	$encryptSetting, signSetting, hashSetting$	T_j, M_k
T_j	encryptionKeyShare	r	y_j, π_j	EC
EC	encryptionKey	1	y	–
EA	electionDetails	1	$\mathcal{Q}, \mathcal{R}, issues, encode$	EC
EA	electoralRoll	1	V	EC
EC	voterCertificates	1	$\mathcal{Z}_V$	–
EC	keyMixingRequest	m	M_k, VK_{k-1}, g_{k-1}	M_k
M_k	keyMixingResult	m	VK_k, g_k, π_k	EC
EC	mixedKeys	1	$\hat{VK}, \hat{g}$	–
EC	votingData	1	$title, period, \mathcal{Q}, \mathcal{R}, issues, encode, encryptSetting, signSetting, hashSetting, y, \hat{g}$	V_i
V_i	ballot	$N \leq n$	E_i, π_i	EC
EC	voteMixingRequest	m	$\mathsf{M}_k, \mathcal{E}_k$	M_k
M_k	voteMixingResult	m	$\mathcal{E}_k, \pi'_k$	EC
EC	mixedVotes	1	$\hat{\mathcal{E}}$	T_j
T_j	partialDecryption	r	A_j, π'_j	EC
EC	decryptedVotes	1	$\mathcal{W}$	EA
EA	decodedVotes	1	$\mathcal{V}$	–
EA	electionResult	1	$result$	–
EC	newVoterCertificate	z_n	Z'_i	–
EC	addedVoterCertificate	z_a	Z'_i	–
EC	cancelledVoterCertificate	z_c	Z_i	–
EC	singleKeyMixingRequest	$m(z_a + z_c)$	M_k, $vk_i, vk_{i,k-1}$	M_k
M_k	singleKeyMixingResult	$m(z_a + z_c)$	$vk_i, vk_{i,k-1}, vk_{i,k}, \pi_{i,k}$	EC

Table 6.1.: Summary of election data published on UBV. Corresponding access rights are attributed by EC. The lower part of the table is the data needed to handle late voter certificates.

Symbol	Type	Element of	List Size
Z_{EA}	certificate		–
title	description		–
period	time period		–
T	identifier list		r
M	identifier list		m
level	integer	$\{0,1,2,\ldots\}$	–
$\mathcal{Z}_T$	certificate list		r
$\mathcal{Z}_M$	certificate list		m
encryptSetting	identifier	$\Sigma_{\mathsf{ENCRYPT}}$	–
signSetting	identifier	Σ_{SIGN}	–
hashSetting	identifier	Σ_{HASH}	–
y_j	public key share	$\mathcal{G}_Q$	–
π_j	zero-knowlegde proof		–
y	public key	$\mathcal{G}_Q$	–
$\mathcal{Q}$	option list		s
$\mathcal{R}$	rule list		?
issues	description		–
encode	identifier	Σ_{ENCODE}	–
V	identifier list		n
$\mathcal{Z}_V$	certificate list		n'
M_k	identifier		–
VK_k	public key list	G_q	n'
g_k	generator	G_q	–
π_k	zero-knowledge proof		–
$\hat{VK}$	public key list	G_q	n'
$\hat{g}$	generator	G_q	–
E_i	ElGamal encryption	$\mathcal{G}_Q \times \mathcal{G}_Q$	–
π_i	zero-knowledge proof		–
$\mathcal{E}_k$	ElGamal encryption list	$\mathcal{G}_Q \times \mathcal{G}_Q$	N
π'_k	zero-knowledge proof		–
$\hat{\mathcal{E}}$	ElGamal encryption list	$\mathcal{G}_Q \times \mathcal{G}_Q$	N
A_j	partial decryption list	$\mathcal{G}_Q$	N
π'_j	zero-knowledge proof		–
$\mathcal{W}$	list of decrypted votes	$\mathbb{Z}_Q$	N
$\mathcal{V}$	list of decoded votes	$\mathbb{N}^s$	N'
result	election result	$\mathbb{N}^s$	–
Z_i, Z'_i	voter certificate		–
$vk_i, vk_{i,k}$	public key	G_q	–
$\pi_{i,k}$	zero-knowledge proof		–

Table 6.2.: Summary of election data published on UBV (in order of appearance).

Part III.

Technical Specification

7. Cryptographic Settings

The following parameters are assumed to be known in advance and not to change over time.

Security Level	Strength	Residue Class Group			Hash Value
		modulo	max. order	min. order	
0 (for testing only)	3	8	7	6	–
1	80	1024	1023	160	160
2	112	2048	2047	224	224
3	128	3072	3071	256	256
4	192	7680	7679	384	384
5	256	15360	15359	512	512

Table 7.1.: Except for security level 0, the values in this table correspond to the NIST recommendations [**?** , Table 2].

$$\begin{aligned}\Sigma_{\mathsf{ENCRYPT}} &= \{\mathsf{RC0e}, \mathsf{RC1e}, \mathsf{RC2e}, \mathsf{RC3e}\}\\ \Sigma_{\mathsf{SIGN}} &= \{\mathsf{RC0s}, \mathsf{RC1s}, \mathsf{RC2s}, \mathsf{RC3s}\}\\ \Sigma_{\mathsf{HASH}} &= \{\mathsf{H1}, \mathsf{H2}, \mathsf{H3}, \mathsf{H4}, \mathsf{H5}\}\end{aligned}$$

7.1. Residue Class Groups

In this section, we specify the parameters for default residue class groups satisfying the security levels 0–3. For a given bit length, we choose the smallest possible prime group order $q = (p-1)/k$ and then the smallest possible co-factor k such that p is prime. In each group, we use $g = 2^k \bmod p$ as default generator.

7.1.1. Level 0 (Testing Only)

Name:	RC0e
Strength:	2
Bit Length:	8/7
Modulo:	167
Order:	83
Co-factor:	2
Generator:	4

Name:	RC0s
Strength:	3
Bit Length:	8/6
Modulo:	149
Order:	37
Co-factor:	4
Generator:	16

7.1.2. Level 1

Name:	RC1e
Strength:	80
Bit Length:	1024/1023
Modulo:	898846567431157953864652595394512366808988489471153286367150405788663379027504815663542386612037680105600569399356966788293948844072083112464237153197370621888839467124327426381511098006230470597265414760425028844190753411712314407369565552704136185816752553422931491199736229692398581524176781648121137402 23
Order:	449423283715578976932326297697256183404494244735576643183575202894331689513752407831771193306018840052800284699678483394146974422036041556232118576598685310944419733562163713190755549003115235298632707380212514422095376705856157203684782776352068092908376276711465745599868114846199290762088390824060568701 11
Co-factor:	2
Generator:	4

Name:	RC1s
Strength:	80
Bit Length:	1024/160
Modulo:	898846567431157953864652595394512366808988489471153286367150405788663379027504815663542386612037680105600569399356966788293948844072083112464237153197370621888839467124327426381511098006230470597265414760425028844190753411712314407369565552704136185816752555293653586983287747087757032152193515453296138759 69
Order:	730750818665451459101842416358141509827966271787
Co-factor:	123003155723136208567847447683223664415731869130386085638915865900973164322556421231024717480721040334016119920526406647270704463131395880224179114380096482367962700290567190341485994461413302531904399363966472118709139313335588033867856826773551831300710596864
Generator:	437539662689561586837941410446090480749443994634971186010092600152789079447933968888726547974366791561717048352633420987472298419829635508715574476834043594463776486457518569138292805779343848313812951031823680370011703145311896581202060526440434692755624731609894511408779313681374405241626450736545123040 68

7.1.3. Level 2

Name:	RC2e
Strength:	112
Bit Length:	2048/2047
Modulo:	1615850303565550365035743834433497598022205133485774201606517271376232756943394544659860070576145673184435898046094900974705977957524546054754407619322414156031543868365049804587509887519482605339802881919203378413839610932130987808091904716923808523529082292601815252144378794577053290430377619956196519276095716669483417121034248739328228474742808801766316102903890282966551309635423015707512929643208855836297180185923092867879917557615082295220184880661664361561356284235541010486257855086346566173483927129032834896752299863417649931910776258319471866777180106771661480232265923930247607409677792680552979882487 9
Order:	8079251517827751825178719172167487990111025667428871008032586356881163784716972723299300352880728365922179490230474504873529889787622730273772038096612070780157719341825249022937549437597413026699014409596016892069198054660654939040459523584619042617645411463009076260721893972885266452151888099780982596380478583347417085605171243696641142373714044008831580514519451414832756548177115078537564648216044279181485900929615464339399587788075411476100924403308321807806781421177705052431289275431732830867419635645164174483761499317088249659553881291597359333885900533858307401161329619651238037048388963402764899412439
Co-factor:	2
Generator:	4

Name:	RC2s
Strength:	112
Bit Length:	2048/224
Modulo:	1615850303565550365035743834433497598022205133485774201606517271З 76232756943394544659860070576145673184435898046094900974705977957 52454605475440761932241415603154386836504980458750988751948260533 98028819192033784138396109321309878080919047169238085235290822926 01815252144378794577053290430377619956196519276095716669483417121 03424873932822847474280880176631610290389028296655130963542301570 75129296432088558362971801859230928678799175576150822952201848806 61664361561356284235541010486257855086346566173483927129032834896 75229986341764993191077626018240418147728931658315222274532240351 24084988448041816607879141260367
Order:	13479973333575319897333507543509815336818572211270286240551805124 797
Co-factor:	11987043769150952793874803430453616717306443681505732405127734328 63035265241224448257650272273442117922854235775131183760154655873 23968246762186326385861552475817897567120457263303465676554996254 23218032340803416954001633709653135468880716615073290073688756152 56474280679640515153088963948173493667962666963665508943301551338 71636861394047830280606078949561993991551118388159502705296649888 16235618653175901258299087128056357203226216572397148388165640339 66583550317895814903767193536143522628430634457899343527463601070 671196072017302504100432184678
Generator:	11342698989719396602562214176029926737577815602473387457111420042 92707499263615663726956407344787191388627204394785328286520316952 45737164011970945956715626527265691980740969997148418484440443783 94889427354052771986760362837213568196773331406427909643009846645 18053443525909642640603162099914341539824434934715022408665363634 88072684751689239340161438398581968988314061683179235048497631421 26080527969429510895336688143488146365666904622327058661427606990 21764820760170288154471669270258911504614068561280584855398438862 52597327322851463914826364508484968371863196419968856241101383447 4496797602932228487527202996447

7.1.4. Level 3

Name:	RC3e
Strength:	128
Bit Length:	3072/3071
Modulo:	2904802997684979031429751266652287185343487588181447618330743076 1436018654985551128686680222665592036256630788774902587219952647972 7002356083144283609351620051605581985322024942202492549452581360 0122382903520906197364840270012052413988292184690761146180604389 5223849463716128758690384897844056547895627556665466217597768924 0815319079008093010012374628422407512125765222478859380206821436929 0495086275786967073127915183202957500434821866026609283416272645 5539518614158170692997932033451629798625937235845297704025061551 0481950587537438000854768036711747287870813649742800665430847926497 9152338818509590797044264172530642931949135881728647441773319439 7771558077232231650996271911700081460285453755877669440809594936 4779576576834935064613384273275871895789541157742231739013005144585 9016247698037520949742756905563488653739484537428521855358075060 6579610122783796206195065764598554782342031897214574708071785539 57231283671210463
Order:	145240149884248951571487563332614359267174379409072380916537153807 180093274927755643433401113327960181283153943874512936099763239863 501178041572141804675810025802790992661012471101246274726290680 006119145176045309868242013500602620699414609234538057309030219476 119247318580643793451924489220282739478137783327331087988844620407 659539504046505006187314211203756062882611239429690103410718464 524754313789348353656395759160147875021741093301330464170813632277 697593070790853464989660167258148993129686179226488520125307755240 975293768719000427384018355873643935406824871400332715423963248 957616940925479539852213208626532146597456794086432372088665971988 857790386161158254981359558500407301427268779388347204047974682389 788288417467532306692136637935947894770578871115869506502572292 950812384901876047487137845278174432686974226871426092767903753032 898050613918981030975328822992773911710159486072873540358927697861 5641835605231
Co-factor:	2
Generator:	4

Name:	RC3s
Strength:	128
Bit Length:	3072/256
Modulo:	29048029976849790314297512666522871853434875881814476183307430761 43601865498555112868668022266559203625663078877490258721995264797 27002356083144283609351620051605581985322024942202492549452581360 01223829035209061973648402700120524139882921846907611461806043895 22384946371612875869038489784405654789562755666546621759776892408 15319079008093010012374628422407512125765222478859380206821436929 04950862757869670731279151832029575004348218660266092834162726455 53951861415817069299793203345162979862593723584529770402506155104 81950587537438000854768036711747287870813649742800665430847926497 91523388185095907970442641725306429319491358817286474417733194397 77155807723223165099627191170008146028545375587766944080959493647 79576576834935064613384273275871895789541157742231739013005144585 90162476980375209497427569055634886537394845374285218553580750607 16204433164749666917562781919225495884397992008274673412368259180 131087873830227
Order:	57896044618658097711785492504343953926634992332820282019728792003 956564820063
Co-factor:	50172736614702193476010968685027850385725009984025754056752351897 84354083664546427042972436257364040015705292620894021158968116670 52278736628424524481364173810719294193392884778267766720794587000 33268758995652670581883324087701986841276235090847530457242114785 08003034351863122611819867710701334475463923697518558595814763337 38120977513870008483043885705348802570420665874935680448798942948 90903505723892042077739270991797692949027455367779291040511577691 18129298733070349048604980733968237990616267319488267460569547220 82654284862289405817133424301830948180453001905416614362070755300 03690390071432433248459391309409195565274258951439355986071694366 37121676228334886710349680880099339347176122780121954092271516962 17192571430830984770102301538658775922986807509273192710496833014 02789114847803512055626615982919607247906296604849442088539792542 702
Generator:	92920125731010433218859140801915212957924174648681944569430739424 36368632400955047430931207305295452016977130491459469008272360996 46074823853068234153087337964364835389953295566321414313091913366 22552453366165871600293028420499959362443638427565001966112132064 28090905852024242916147306177433999396043807599713151522484988122 28036889780065062471230154677956427260036586710411937258219627969 44081469356220545634891695825875959220167038653315231810943582373 04642033650185021199766974695754072338180730437660124968128336662 18850668460811529129373410522906961500960262552792104927047576848 06333861136894510516508364094002731978111599401238922650475382242 11593712190511810930349828435945031628877529853083039287986906995 05264113123684970126578784098482558068892187359052293562177396159 55675269924839998252345273839236796953759255859946063157304428704 83484473572273998392773915519546757089681068563198094927055596095 60299469015227

7.2. Hash Functions

Name	H1
Level	1
Strength	80
Bit Length	160
Standard	SHA-1

Name	H2
Level	2
Strength	112
Bit Length	224
Standard	SHA-224 (truncated SHA-256)

Name	H3
Level	3
Strength	128
Bit Length	256
Standard	SHA-256

Name	H4
Level	4
Strength	192
Bit Length	384
Standard	SHA-384 (truncated SHA-512)

Name	H5
Level	5
Strength	256
Bit Length	512
Standard	SHA-512

8. Encoding Votes

For a given set of election options $\mathcal{Q}$ and corresponding election rules $\mathcal{R}$, let ρ_j denote the maximum number of votes that the election rules allow for each option $Q_j \in \mathcal{Q}$ and ρ the maximum total number of votes for all options. For $v_i = (v_{i,1}, \ldots, v_{i,s}) \in \mathit{ValidVotes}(\mathcal{Q}, \mathcal{R})$, this implies $0 \leq v_{i,j} \leq \rho_j$ and $\sum_j v_{i,j} \leq \rho$. To encode v_i as an integer, we need an invertible encoding function $\mathit{encode}_{\mathcal{Q},\mathcal{R}} : \mathit{ValidVotes}(\mathcal{Q}, \mathcal{R}) \rightarrow \mathbb{Z}_q$ for some $q \geq |\mathit{ValidVotes}(\mathcal{Q}, \mathcal{R})|$. We consider two encodings, one based on the election options and one based on the maximum number of votes. We denote them by $\mathsf{E1}$ and $\mathsf{E2}$, respectively, and

$$\Sigma_{\mathsf{ENCODE}} = \{\mathsf{E1}, \mathsf{E2}\}$$

is the set of available encodings. Each of them defines a total bit length B, which implies $q \leq 2^B$. Usually, the encoding with the smaller upper bound for q is the preferred one.

a) Option-Based Encoding

Consider a bit string of length $B = \sum_{j=1}^{s} b_j$, where $b_j = \|\rho_j\|$ denotes the number of bits reserved for each value $v_{i,j}$. Furthermore, let $B_j = \sum_{k=1}^{j-1} b_k$ be the number of bits in the bit string *prior* to $v_{i,j}$, i.e., $B_1 = 0$, $B_2 = b_1$, $B_3 = b_1 + b_2$, $\ldots$, $B_{s+1} = B$. The encoding function $\mathit{encode}_{\mathcal{Q},\mathcal{R}} : \mathit{ValidVotes}(\mathcal{Q}, \mathcal{R}) \rightarrow \mathbb{Z}_{2^B}$ can then be defined as follows:

$$\mathit{encode}_{\mathcal{Q},\mathcal{R}}(v_i) = \sum_{j=1}^{s} v_{i,j} \cdot 2^{B_j}.$$

Note that some integers in $\mathbb{Z}_{2^B}$ do not represent valid votes according to the election rules R. To decode an integer representation $w_i = \mathit{encode}_{\mathcal{Q},\mathcal{R}}(v_i)$ back to $v = (v_{i,1}, \ldots, v_{i,s})$, we must decompose the bit string into its components. Mathematically, this decomposition can be written as follows:

$$\mathit{encode}^{-1}_{\mathcal{Q},\mathcal{R}}(w_i) = (v_{i,1}, \ldots, v_{i,s}), \text{ where } v_{i,j} = \lfloor w_i / 2^{B_j} \rfloor \bmod 2^{b_j}.$$

As an example, let there be $s = 5$ elections options, each of which receiving up to $\rho_j = 2$ votes with a maximum of $\rho = 4$ votes. This implies $b_j = 2$ bits for each election option and therefore $B_1 = 0$, $B_2 = 2$, $B_3 = 4$, $B_4 = 6$, $B_5 = 8$, and $B = 10$. The following table shows some votes and their option-based encodings. The highest possible encoding is $\mathit{encode}_{\mathcal{Q},\mathcal{R}}(0, 0, 0, 2, 2) = 640$, which implies $q = 641$.

v_i	$\sum_{j=1}^{5} v_{i,j} \cdot 2^{B_j}$	$encode_{\mathcal{Q},\mathcal{R}}(v_i)$
$(0,0,0,0,0)$	$0{\cdot}2^0 + 0{\cdot}2^2 + 0{\cdot}2^4 + 0{\cdot}2^6 + 0{\cdot}2^8$	0
$(1,0,0,0,0)$	$1{\cdot}2^0 + 0{\cdot}2^2 + 0{\cdot}2^4 + 0{\cdot}2^6 + 0{\cdot}2^8$	1
$(2,0,0,0,0)$	$2{\cdot}2^0 + 0{\cdot}2^2 + 0{\cdot}2^4 + 0{\cdot}2^6 + 0{\cdot}2^8$	2
$(0,0,0,0,1)$	$0{\cdot}2^0 + 0{\cdot}2^2 + 0{\cdot}2^4 + 0{\cdot}2^6 + 1{\cdot}2^8$	256
$(0,0,0,0,2)$	$0{\cdot}2^0 + 0{\cdot}2^2 + 0{\cdot}2^4 + 0{\cdot}2^6 + 2{\cdot}2^8$	512
$(1,1,0,1,1)$	$1{\cdot}2^0 + 1{\cdot}2^2 + 0{\cdot}2^4 + 1{\cdot}2^6 + 1{\cdot}2^8$	325
$(0,1,2,1,0)$	$0{\cdot}2^0 + 1{\cdot}2^2 + 2{\cdot}2^4 + 1{\cdot}2^6 + 0{\cdot}2^8$	100
$(2,2,0,0,0)$	$2{\cdot}2^0 + 2{\cdot}2^2 + 0{\cdot}2^4 + 0{\cdot}2^6 + 0{\cdot}2^8$	10
$(0,0,0,2,2)$	$0{\cdot}2^0 + 0{\cdot}2^2 + 0{\cdot}2^4 + 2{\cdot}2^6 + 2{\cdot}2^8$	640

b) Vote-Based Encoding

Consider a bit string of length $B = \rho \cdot |s|$. The idea here is to reserve $|s|$ bits for each of the maximal ρ votes and to use the index j to encode a vote for option Q_j. For $v_i = (v_{i,1}, \dots, v_{i,s})$ and $0 \leq j_1 \leq \cdots \leq j_\rho \leq s$, let $q_i = (Q_{j_1}, \dots, Q_{j_\rho})$ be the sorted list (in ascending index order) of the ρ election options $Q_{j_k} \in \mathcal{Q} \cup \{Q_0\}$ selected in v_i, where Q_0 denotes an empty vote in cases in which not all ρ votes are attributed to options. If we use v_i and q_i as equivalent vote representations, then the encoding function $encode_{\mathcal{Q},\mathcal{R}} : ValidVotes(\mathcal{Q},\mathcal{R}) \rightarrow \mathbb{Z}_{2^B}$ can be defined as follows:

$$encode_{\mathcal{Q},\mathcal{R}}(q_i) = \sum_{k=1}^{\rho} j_k \cdot 2^{(k-1)|s|}.$$

Again, some integers in $\mathbb{Z}_{2^B}$ do not represent valid votes according to the election rules R. To decode an integer representation $w_i = encode_{\mathcal{Q},\mathcal{R}}(q_i)$ back to $q_i = (Q_{j_1}, \dots, Q_{j_\rho})$ and then to $v_i = (v_{i,1}, \dots, v_{i,s})$, we must first decompose the bit string into its components. Mathematically, this decomposition can be written as follows:

$$encode^{-1}_{\mathcal{Q},\mathcal{R}}(w_i) = (Q_{j_1}, \dots, Q_{j_\rho}), \text{ where } Q_{j_k} = \lfloor w_i / 2^{(k-1)|s|} \rfloor \bmod 2^{|s|}.$$

As an example, consider the same setting as above, which implies $|s| = 3$ and $B = 12$. The following table shows the same votes and their vote-based encodings. The highest possible encoding is $encode_{\mathcal{Q},\mathcal{R}}(4,4,5,5) = 2916$, which implies $q = 2917$. For the given setting, this encoding is therefore less efficient than the one above.

v_i	q_i	$\sum_{k=1}^{4} j_k \cdot 2^{(k-1)|s|}$	$encode_{\mathcal{Q},\mathcal{R}}(q_i)$
$(0,0,0,0,0)$	$(0,0,0,0)$	$0{\cdot}2^0 + 0{\cdot}2^3 + 0{\cdot}2^6 + 0{\cdot}2^9$	0
$(1,0,0,0,0)$	$(0,0,0,1)$	$0{\cdot}2^0 + 0{\cdot}2^3 + 0{\cdot}2^6 + 1{\cdot}2^9$	512
$(2,0,0,0,0)$	$(0,0,1,1)$	$0{\cdot}2^0 + 0{\cdot}2^3 + 1{\cdot}2^6 + 1{\cdot}2^9$	576
$(0,0,0,0,1)$	$(0,0,0,5)$	$0{\cdot}2^0 + 0{\cdot}2^3 + 0{\cdot}2^6 + 5{\cdot}2^9$	2560
$(0,0,0,0,2)$	$(0,0,5,5)$	$0{\cdot}2^0 + 0{\cdot}2^3 + 5{\cdot}2^6 + 5{\cdot}2^9$	2880
$(1,1,0,1,1)$	$(1,2,4,5)$	$1{\cdot}2^0 + 2{\cdot}2^3 + 4{\cdot}2^6 + 5{\cdot}2^9$	2833
$(0,1,2,1,0)$	$(2,3,3,4)$	$2{\cdot}2^0 + 3{\cdot}2^3 + 3{\cdot}2^6 + 4{\cdot}2^9$	2266
$(2,2,0,0,0)$	$(1,1,2,2)$	$1{\cdot}2^0 + 1{\cdot}2^3 + 2{\cdot}2^6 + 2{\cdot}2^9$	1161
$(0,0,0,2,2)$	$(4,4,5,5)$	$4{\cdot}2^0 + 4{\cdot}2^3 + 5{\cdot}2^6 + 5{\cdot}2^9$	2916

9. UniBoard

9.1. Basic Types to ByteArray

For hashing any part of a post, this needs to be converted to a byte array.

- ByteArray - Does not need any conversion
- String - ByteArray of the UTF-8 String
- Integer - Is a signed big integer which is translated into a bigendian byte array
- Date - Is first converted into a String following "yyyy-MM-dd'T'HH:mm:ss'Z'" in UTC time and then converted as UTF-8 String.

9.2. Post Signature

To create a signature a recursive hash-algorithm is used. To sign the post the author hashes [message,alpha] where alpha = [section, group]. To sign the post UniBoard hashes [message,alpha,beta] where alpha = [section,group, signature,key] and beta = [timestamp,rank].

9.3. Read Signature

To create a signature a recursive hash-algorithm is used. To sign the result of the query UniBoard hashes [query,resultcontainer] where query=[contraints,order,limit] constraint=[type,identifier,value1,... identifier=[type,s1,s2,s3,...] order=[identifier,ascDesc] resultcontainer=[result,gamma] result [p1,p2,...] post=[message,alpha,beta] alpha=[section,group,signature,key] beta=[timestamp,rank,boardSi gamma=[timestamp]

9.4. Error and Rejects

Type	Code	Description
R	BAC-001	There is no authorization for the provided key
R	BAC-002	The provided signature is not valid
R	BAC-003	Authorization is not active yet
R	BAC-004	Authorization expired
R	BAC-005	Amount of allowed posts used up
E	BAC-006	Internal server error
R	BGT-001	Attribute missing
R	BGT-002	Attribute is not a StringValue
R	BGT-003	Configuration for section-bfh is missing
R	BGT-004	Specified section is not known on this UniBoard
R	BGT-005	Message is not valid in typed only modus
R	BGT-006	Message is not valid for the selected group
R	BSE-001	Attribute missing
R	BSE-002	Attribute is not a StringValue
R	BSE-003	Configuration for section-bfh is missing
R	BSE-004	Specified section is not known on this UniBoard
E	BCG-001	Internal server error
E	BCP-001	Internal server error

10. UniCert

10.1. Format of certificate

The issued certificate are published on UniBoard. In order to allow UniBoard to make a search into the fields of the certificate, the certificate is represented as JSON format additionally to its PEM format. The listing below shows the schema corresponding to the JSON format of a certificate.

```
{
"title": "Schema for UniCert certificates",
"description": "This schema describes the format of a UniCert
   certificate in JSON format",
"type":"object",
"$schema": "http://json-schema.org/draft-04/schema",
"properties": {
"commonName": {
"type": "string",
"description": "Common name of certificate owner"
},
"uniqueIdentifier": {
"type": "string",
"description":  "Unique identifier of certificate owner"
},
"organisation": {
"type": "string",
"description": "Organisation of certificate owner"
},
"organisationUnit": {
"type": "string",
"description": "Organisation unit of certificate owner"
},
"countryName": {
"type": "string",
"description": "Country of certificate owner"
},
"state": {
"type": "string",
"description": "State of certificate owner"
},
"locality": {
"type": "string",
```

```
"description": "Locality certificate owner"
},
"surname": {
"type": "string",
"description": "Surname of certificate owner"
},
"givenName": {
"type": "string",
"description": "Given name of certificate owner"
},
"issuer": {
"type": "string",
"description": "Issuer of the certificate"
},
"serialNumber": {
"type":"string",
"description": "Serial number of the certificate"
},
"validFrom": {
"type": "string",
"description": "Date when certificate starts to be valid"
},
"validUntil": {
"type": "string",
"description": "Date when certificate stops to be valid"
},
"applicationIdentifier": {
"type": "string",
"description": "Application the certificate has been issued for"
},
"role": {
"type": "string",
"description": "Role inside an application the certificate has been issued for"
},
"identityProvider": {
"type": "string",
"description": "Identity provider used to verify the identity of the certificate owner"
},
"pem": {
"type": "string",
"description": "Certificate in PEM format"
}
},
"required": ["commonName", "issuer", "serialNumber", "validFrom", "validUntil", "identityProvider", "pem" ]
```

```
}
```

11. UniVote

This chapter describes the detailed design of UniVote. It starts off by defining the actions which make up the core of the components of UniVote. Later, more design details will be provided.

Actions are triggered either by notification (normal case) or by manual intervention. Upon an notification (*Notification* column in the tables), the following tables define the steps an action undertakes (*Work*), the message it sends to the board (*Post*) and the successor action(s) if any (*Successors*) that may occur as a consequence of terminating the current the action.

11.1. EC Setup Phase Actions

Actions relevant of the setup phase.

11.1.1. Initial

Notification	Work	Post	Successors
accessRight	Start the new election	-	DefineEA

11.1.2. DefineEA

Notification	Work	Post	Successors
EA defined	Search and set cert for EA	administrationCertificate: Z_{EA}	GrantElectionDefinition, GrantTrustees, GrantSecurityLevel, GrantElectoralRoll, GrantElectionIssues

11.1.3. GrantElectionDefinition

Notification	Work	Post	Successors
administration-Certificate	Grant EA access to electionDefinition	accessRight: vk_{EA}, *electionDefinition*, 1	CreateVotingData

11.1.4. GrantTrustees

Notification	Work	Post	Successors
administration-Certificate	Grant EA access to trustees	accessRight: vk_{EA}, *trustees*, 1	PublishTrusteeCerts

11.1.5. GrantSecurityLevel

Notification	Work	Post	Successors
administration-Certificate	Grant EA access to securityLevel	accessRight: vk_{EA}, *securityLevel*, 1	SetCryptoSetting

11.1.6. PublishTrusteeCerts

Notification	Work	Post	Successors
trustees	Get Trustee certificates and publish them	trusteeCertificates: $\mathcal{Z}_T$, $\mathcal{Z}_M$	GrantEncyptionKeyShares

11.1.7. SetCryptoSetting

Notification	Work	Post	Successors
securityLevel	Publish the crypto setting for this election	cryptoSetting: *settings*	GrantEncyptionKeyShares

11.1.8. GrantEncyptionKeyShares

Notification	Work	Post	Successors
trustee-Certificates	Grant Trustees access to keyshares	accessRight: vk_t, *encrytptionKeyShare*, 1	CombineEncryptionKey-Shares

11.1.9. CombineEncryptionKeyShares

Notification	Work	Post	Successors
encryption-KeyShare	Validate key share and if all key shares are published combine them	encryptionKey: y	CreateVotingData

11.2. EC Preparation Phase Actions

Actions relevant for the preparation phase of the EC.

11.2.1. GrantElectionIssues

Notification	Work	Post	Successors
administration-Certificate	Grant EA access for EI	accessRight: vk_{EA}, *electionIssues*, 1	CreateVotingData

11.2.2. GrantElectoralRoll

Notification	Work	Post	Successors
administration-Certificate	Grant EA access for ER	accessRight: vk_{EA}, *electoralRoll*, 1	KeyMixing

11.2.3. KeyMixing

Notification	Work	Post	Successors
electoralRoll	Retrieve voter certs and create the first mixing request	voterCertificates: $\mathcal{Z}_V$ keyMixingRequest: M_1, g_0, VK_0 accessRight: vk_1, *keyMixingResult*, 1	GrantBallot, CreateElectionData
keyMixing-Request	Create next request or publish mixed keys	keyMixingRequest: M_k, g_k, VK_k accessRight: vk_k, *keyMixingResult*, 1 mixedKeys: $\hat{VK}$	

11.2.4. GrantBallot

Notification	Work	Post	Successors
mixedKeys	Grant all registered voters access for Ballot	accessRight: $\hat{vk}_i$, *ballot*, 1, *period*	-

11.2.5. CreateVotingData

Notification	Work	Post	Successors
election-Definition, election-Details, cryptoSetting, encryptionKey, signature-Generator	Create the voting data when all parts are available and publish it.	votingData: *title*, *period*, C, R, *electionDetails*, *encryptSetting*, *signSetting*, *hashSetting*, y, $\hat{g}$	-

11.3. Trustees

Actions relevant for the trustees. A trustee can be either a tallier of a mixer.

11.3.1. Tallier: CreateSharedKey

Notification	Work	Post	Successor
accessRight: encryption-KeyShare	Get crypto setting from UBV Store: $x \in_R \mathbb{Z}_Q$ $y = G^x$ $\pi = NIZKP\{(x) : y = G^x\}$ finish.	KeyShare: y Store: $x \in_R \mathbb{Z}_Q$	PartialDecryption

11.3.2. Tallier: PartialDecryption

Notification	Work	Post	Successor
accessRight: partialDecryption	Get mixedVotes $\hat{\mathcal{E}}$ from UBV Retrieve: x $A' = (a_1, \ldots, a_N) \in (G)_Q^N$ from $\hat{\mathcal{E}}$ $\pi = NIZKP\{(x) : y = G^x \wedge (\bigwedge_i a_i = a_i^{-x})\}$ finish.	partialDecryption: A, π	-

11.3.3. Mixer: MixingPublicKey

Notification	Work	Post	Successor
keyMixing-Request	Get crpytoSetting from UBV Store: $\alpha \in_R \mathbb{Z}_Q$ $g = g_{\_}^{\alpha}$ Store: $\psi \in_R \Psi_n$ (permutation vector) $\pi = NIZKP\{(\psi, \alpha)\ g = g_{\_}^{\alpha} \wedge VK = \mathit{shuffle}_{\psi}(VK\_, \alpha)\}$ finish.	keyMixingResult: g, VK, π	Cancelling-ExistingKey

11.3.4. Mixer: MixingVote

Notification	Work	Post	Successor
voteMixing-Request	Get electionSetting: $\mathcal{G}_q, G$ from UBV $R = (r_1, \ldots, r_N) \in_R \mathbb{Z}_Q^N$ $\tau : [1, N] \to [1, N] \in_R \Psi_N$ $\mathcal{E} = \mathit{shuffle}_{\tau}(\mathcal{E}\_, R)$ $\pi = NIZKP\{(\tau, R) : \mathcal{E} = \mathit{shuffle}_{\tau}(\mathcal{E}\_, R)\}$ finish.	voteMixingResult: $\mathcal{E}\_$, π	-

11.3.5. Mixer: AddingNewKey

Notification	Work	Post	Successor
ElectionEnd-Timer	run finish. if ElectionEndTimer $\to$ finish.	-	-

11.3.6. Mixer: CancellingExistingKey

Notification	Work	Post	Successor
keyCancelling-Request, Electio-nEndTimer	Get $V_i, vk_{i,\_}$ from UVB Get α $vk_i = vk_{i,\_}^{\alpha}$ $\pi = NIZKP\{(\alpha) : g = g_{\_}^{\alpha} \wedge vk_i = vk_{i,\_}^{\alpha}\}$ run finish if ElectionEndTimer $\to$ finish.	keyCancelling-Result: V_i, vk_i, π_i	-