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

UniVote2 System Specification

Version 0.6.2

Reto E. Koenig, Rolf Haenni, Severin Hauser

16.06.2015

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	Rolf Haenni	Initial Draft.
0.2	15.08.2012	Rolf Haenni	Complete revision of initial draft.
0.3	10.09.2012	Rolf Haenni	Includes now the case of late registrations.
0.4	13.03.2013	Rolf Haenni	Improved ballot acceptance and closing the urn.
0.5	04.04.2013	Rolf Haenni	Revised section on zero-knowledge proofs.
0.5.1	30.04.2013	Rolf Haenni	Improved notational consistency.
0.5.2	21.05.2013	Rolf Haenni	Two important notational corrections.
0.6	07.01.2015	Reto E. Koenig	Quality: Change of title; re-establishement of the version history.
0.6	15.06.2015	Reto E. Koenig	Revised section on Randomness
0.6.1	16.06.2015	Severin Hauser	Revision of UniVote
0.6.2	16.06.2015	Reto E. Koenig	Revision of UniVote

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 $|x| = \lfloor \log_2(\mathrm{abs}(x)) \rfloor$ 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 $b_i \in \{\mathtt{0}, \mathtt{1}\}^8$. 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 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 $\mathit{bytes}_k(x)$ 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 $k \geq \lceil (|x|+1)/8 \rceil$. We use $\mathit{bytes}(x)$ as a short-cut notation for the shortest possible such byte array representation of length $k = \lceil (|x|+1)/8 \rceil$. Note that the most significant bit of $\mathit{bytes}(x)[0]$ is always the sign bit, also for $\mathit{bytes}(0) = \langle \mathtt{00} \rangle$, 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 $k \leq 4$:

	k				
x	1	2	3	4	$\cdots$
0	$\langle\mathtt{00}\rangle$	$\langle\mathtt{00,00}\rangle$	$\langle\mathtt{00,00,00}\rangle$	$\langle\mathtt{00,00,00,00}\rangle$	
1	$\langle\mathtt{01}\rangle$	$\langle\mathtt{00,01}\rangle$	$\langle\mathtt{00,00,01}\rangle$	$\langle\mathtt{00,00,00,01}\rangle$	
127	$\langle\mathtt{7F}\rangle$	$\langle\mathtt{00,7F}\rangle$	$\langle\mathtt{00,00,7F}\rangle$	$\langle\mathtt{00,00,00,7F}\rangle$	
128	–	$\langle\mathtt{00,80}\rangle$	$\langle\mathtt{00,00,80}\rangle$	$\langle\mathtt{00,00,00,80}\rangle$	
255	–	$\langle\mathtt{00,FF}\rangle$	$\langle\mathtt{00,00,FF}\rangle$	$\langle\mathtt{00,00,00,FF}\rangle$	
256	–	$\langle\mathtt{01,00}\rangle$	$\langle\mathtt{00,01,00}\rangle$	$\langle\mathtt{00,00,01,00}\rangle$	
32767	–	$\langle\mathtt{7F,FF}\rangle$	$\langle\mathtt{00,7F,FF}\rangle$	$\langle\mathtt{00,00,7F,FF}\rangle$	
32768	–	–	$\langle\mathtt{00,80,00}\rangle$	$\langle\mathtt{00,00,80,00}\rangle$	
65535	–	–	$\langle\mathtt{00,FF,FF}\rangle$	$\langle\mathtt{00,00,FF,FF}\rangle$	
-1	$\langle\mathtt{FF}\rangle$	$\langle\mathtt{FF,FF}\rangle$	$\langle\mathtt{FF,FF,FF}\rangle$	$\langle\mathtt{FF,FF,FF,FF}\rangle$	
-2	$\langle\mathtt{FE}\rangle$	$\langle\mathtt{FF,FE}\rangle$	$\langle\mathtt{FF,FF,FE}\rangle$	$\langle\mathtt{FF,FF,FF,FE}\rangle$	
-128	$\langle\mathtt{80}\rangle$	$\langle\mathtt{FF,80}\rangle$	$\langle\mathtt{FF,FF,80}\rangle$	$\langle\mathtt{FF,FF,FF,80}\rangle$	
-129	–	$\langle\mathtt{FF,7F}\rangle$	$\langle\mathtt{FF,FF,7F}\rangle$	$\langle\mathtt{FF,FF,FF,7F}\rangle$	
-256	–	$\langle\mathtt{FF,00}\rangle$	$\langle\mathtt{FF,FF,00}\rangle$	$\langle\mathtt{FF,FF,FF,00}\rangle$	
-257	–	$\langle\mathtt{FE,FF}\rangle$	$\langle\mathtt{FF,FE,FF}\rangle$	$\langle\mathtt{FF,FF,FE,FF}\rangle$	
-32768	–	$\langle\mathtt{80,00}\rangle$	$\langle\mathtt{FF,80,00}\rangle$	$\langle\mathtt{FF,FF,80,00}\rangle$	
-32769	–	–	$\langle\mathtt{FF,7F,FF}\rangle$	$\langle\mathtt{FF,FF,7F,FF}\rangle$	
-65536	–	–	$\langle\mathtt{FF,00,00}\rangle$	$\langle\mathtt{FF,FF,00,00}\rangle$	

The two's complement representation in big-endian byte order is the only integer representation considered in this document.

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 $\mathit{bytes}(S)$ 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*("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*("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.

2.3. Byte Trees

Following the definition given in [10], a *byte tree* is either a *leaf* containing an array of bytes, or a *node* containing other byte trees. The purpose of a byte tree is to provide a simple byte-oriented representation of complex mathematical objects consisting of values of different types (integers, strings, etc.). Constructing a byte tree $T = \mathit{byteTree}(\phi)$ from such a mathematical object ϕ means replacing each individual value by its byte array representation (see previous section).

A byte tree T itself can be transformed into a single byte array $B = \mathit{bytes}(T)$, which may then be given as input to a hash function, a random oracle, or an encryption function. This transformation is defined as follows:

- If T is a node containing byte trees $T_1, \ldots, T_k$, then

$$\mathit{bytes}(T) = \langle\texttt{00}\rangle \;||\; {}_4(k) \;||\; \mathit{bytes}(T_1) \;||\; \cdots \;||\; \mathit{bytes}(T_k).$$

- If T is a leaf containing a byte array B of length n, then

$$\mathit{bytes}(T) = \langle\texttt{01}\rangle \;||\; \mathit{bytes}_4(n) \;||\; B.$$

In each case, the first byte specifies the type of the byte tree and the following four bytes its size. Note that this transformation restricts the maximal number of children in a node and the maximal number of bytes in leaf to $2^{31} - 1$.

As an example, consider the mathematical object $\phi = (\texttt{"Hello"}, (256, -32769))$, which is a pair consisting of a string and a pair of integers. From

$$\begin{aligned}
\mathit{bytes}(\texttt{"Hello"}) &= \langle\texttt{48},\texttt{65},\texttt{6C},\texttt{6C},\texttt{6F}\rangle,\\
\mathit{bytes}(256) &= \langle\texttt{01},\texttt{00}\rangle,\\
\mathit{bytes}(-32769) &= \langle\texttt{FF},\texttt{7F},\texttt{FF}\rangle,
\end{aligned}$$

we obtain the byte tree $T = \mathit{byteTree}(\phi) = (\langle\texttt{48},\texttt{65},\texttt{6C},\texttt{6C},\texttt{6F}\rangle, (\langle\texttt{01},\texttt{00}\rangle, \langle\texttt{FF},\texttt{7F},\texttt{FF}\rangle))$, which finally leads to the following byte array:

$$\begin{aligned}
\mathit{bytes}(T) = \langle &\texttt{00},\texttt{00},\texttt{00},\texttt{00},\texttt{02},\\
&\quad\texttt{01},\texttt{00},\texttt{00},\texttt{00},\texttt{05},\texttt{48},\texttt{65},\texttt{6C},\texttt{6C},\texttt{6F},\\
&\quad\texttt{00},\texttt{00},\texttt{00},\texttt{00},\texttt{02},\\
&\qquad\texttt{01},\texttt{00},\texttt{00},\texttt{00},\texttt{02},\texttt{01},\texttt{00},\\
&\qquad\texttt{01},\texttt{00},\texttt{00},\texttt{00},\texttt{03},\texttt{FF},\texttt{7F},\texttt{FF}\rangle.
\end{aligned}$$

2.4. Pairing

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

One of the simplest pairing function, called *elegant pairing* [9], is defined as

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

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

				x				
		0	1	2	3	4	5	$\cdots$
	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	
	$\vdots$							

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

$$\mathit{unpair}(z) = \begin{cases} (t, s), & \text{if } t < s, \\ (s, t - s), & \text{otherwise,} \end{cases} \tag{2.2}$$

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

There are two important generalizations of a pairing functions, which can be added individually or jointly. The first one 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 $\mathit{fold}(x) \in \mathbb{N}$, where

$$\mathit{fold}(x) = \begin{cases} 2x, & \text{if } x \geq 0, \\ 2|x| - 1, & \text{otherwise,} \end{cases} \tag{2.3}$$

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

$$\mathit{unfold}(x) = \begin{cases} \frac{1}{2}x, & \text{if } x \text{ is even,} \\ -\frac{1}{2}(x + 1), & \text{otherwise.} \end{cases} \tag{2.4}$$

The second generalization extends the domain of the pairing function from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}^k$ for $k \geq 2$. 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:

$$pair_k(x_1, \ldots, x_k) = \begin{cases} pair_{\frac{k}{2}}(pair(x_1, x_2), \ldots, pair(x_{k-1}, x_k)), & \text{if } k > 2 \text{ is even,} \\ pair_{\frac{k+1}{2}}(pair(x_1, x_2), \ldots, pair(x_{k-2}, x_{k-1}), x_k), & \text{if } k > 2 \text{ is odd,} \\ pair(x_1, x_2), & \text{if } k = 2. \end{cases} \tag{2.5}$$

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

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 $\mathcal{B} = \{\mathtt{0},\mathtt{1}\}^8$ 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 practical hash functions we consider in this document are the NIST standards SHA-256, SHA-384, and SHA-512, which output byte arrays of length 32 (256 bits), 48 (384 bits), and 64 (512 bits), respectively.

Let $\phi \in \Phi$ be an arbitrary mathematical object. Then we consider the following two generalized definitions of a hash function with a more general domain Φ:

- *Byte Tree Hashing*: The input object ϕ is first encoded as a byte tree, which is then transformed into an input byte array for the hash function:

$$treeHash(\phi) = hash(bytes(byteTree(\phi))). \tag{3.1}$$

- *Recursive Hashing*: Here we compute the hash value recursively from the hash values of the components of ϕ (unless ϕ is atomic):

$$recHash(\phi) = \begin{cases} hash(bytes(\phi)), & \text{if } \phi \text{ is atomic,} \\ hash(recHash(\phi_1) \;||\; \cdots \;||\; recHash(\phi_k)), & \text{if } \phi = (\phi_1, \ldots, \phi_k). \end{cases} \tag{3.2}$$

Sometimes, if we do not want to specify the concrete hash method applied to ϕ, we simply write $hash(\phi)$ for the hash value of ϕ. Furthermore, if $\phi = (\phi_1, \ldots, \phi_k)$ is a composed object, we sometimes write $hash(\phi)$ as the application $hash(\phi_1, \ldots, \phi_k)$ of a k-ary hash function to multiple arguments.

3.2. Generating Random Bits and Random Numbers

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

3.2.1. Random Oracles

In cryptography, a *random oracle* is mapping $\mathit{randomOracle} : \{0,1\}^* \rightarrow \{0,1\}^\infty$, which responds to each input bit sequence $Q \in \{0,1\}^*$ with an infinitely long output bit sequence $\mathit{randomOracle}(Q) \in \{0,1\}^\infty$, in which every bit is chosen uniformly and independently. Note that if such a *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.*

In practice, random oracles can at most be approximated, for example by hash functions, pseudo-random bit generators, or sponge functions. In this document, we use pseudo-random bit generators in combination with hash functions to approximate random oracles. The following table gives an overview of the fundamental differences between random oracles and the cryptographic primitives used to approximate them.

	Domain	Co-domain	State width	Max. period
Hash function	$\{0,1\}^*$	$\{0,1\}^n$	–	–
Pseudo-random bit generator	$\{0,1\}^n$	$\{0,1\}^\infty$	n	2^n
Sponge function	$\{0,1\}^*$	$\{0,1\}^\infty$	b	2^b
Random oracle	$\{0,1\}^*$	$\{0,1\}^\infty$	∞	∞

To make random oracles more flexible with respect to the type of query they accept, we allow writing $\mathit{randomOracle}(\phi)$ for feeding the oracle with the bits contained in $\mathit{bytes}(\mathit{byteTree}(\phi))$, where ϕ is an arbitrary mathematical object.

3.2.2. Pseudo-Random Bit Generators

A *pseudo-random bit generator* (PRBG) is a deterministic function $\mathit{PRBG} : \{0,1\}^n \rightarrow \{0,1\}^\infty$ for generating infinitely long bit sequences whose properties approximate the properties of true random bit sequences, except for the fact they are periodic. The n input bits $S \in \{0,1\}^n$ of a PRBG are called *seed*, and the output bits $\mathit{PRBG}(S) \in \{0,1\}^\infty$ are called *pseudo-random bit sequence.* The seed determines the initial internal state of the PRNG. Usually, the bit length of the internal state—called *state width*—is equivalent to n, the bit length of the seed. This implies that the period of the resulting pseudo-random bit sequence can be no longer than 2^n.

A simple way of constructing a PRNG is by applying a one-way function $f : \{0,1\}^n \rightarrow \{0,1\}^m$ or $f : \{0,1\}^* \rightarrow \{0,1\}^m$ repeatedly to $S + i$ for $i = 0, 1, 2 \ldots$, which outputs the pseudo-random bit sequence

$$PRBG(S) = f(S) \mid\mid f(S+1) \mid\mid f(S+2) \mid\mid \cdots$$

in chunks of m bits [3]. This general construction is typically instantiated with either a hash function, a block cipher, or a MAC. In this document, this is the only PRNG construction we consider, and we instantiate it with SHA-256 and hence with $n = m = 256$.

To use a PRBG as an approximation of a random oracle, it's input space must be extended from $\{0,1\}^n$ to $\{0,1\}^*$. The simplest solution is to use a hash function for computing $S = hash_n(Q)$ from then random oracle's query $Q \in \{0,1\}^*$, and to use S as seed for the PRBG. In this document, we use SHA-256 for this purpose.

3.2.3. Reseedable Pseudo-Random Bit Generators

The problem with a singly-seeded PRBG is that an adversary who learns the internal state of the PRBG can predict the remaining pseudo-random bit sequence. This problem can be circumvented, if a (possibly compromised) PRGB allows to reseed (update) the internal state with a fresh random string ($\{0,1\}^n$) where $n \geq$ security parameter.

3.2.4. Sponge Functions

Sponge functions generalize both hash functions and pseudo-random bit generators (and many other cryptographic primitives). They are also designed to better approximate random oracles. At the moment, sponge functions are not used in UniVote, but we mention them here, because they are likely to replace the current pseudo-random bit generators in future releases. Note that the forthcoming new hash function standard SHA3 is also based on a sponge function.

In its normal mode of operation, a sponge function $sponge : \{0,1\}^* \rightarrow \{0,1\}^\infty$ defines a mapping between a finite input and an infinite output bit sequence, similar to a random oracle. The difference to a random oracle is the limitation of the state width to b bits. However, since b can be chosen freely, sponge functions are quite flexible in approximating random oracles as precisely as needed. Nevertheless, the output bit sequence remains periodic with an upper bound of 2^b bits.

3.2.5. Creating Randomness by the use of Entropy-Provider

In order to work with randomized cryptographic protocols, random data is required which is entropic for the adversary, whereas entropic data in this context is derived from the definition of Shannon entropy [8]. The immanent problem when dealing with the desire to gain random data from a source, is the lack of knowledge whether the source really provides (enough) data of the desired entropy (denoted as question mark ?). $(\{?\}^?, t) \rightarrow \{0,1\}^*$, where t is the time needed in order to gain the desired amount of entropic randomness.

If the adversary can pretend to be a source for data with entropy, i.e. provides data which only 'looks' entropic to everyone but the adversary, all randomized cryptographic constructs building upon random data will fail their duty. Hence, in order to establish a cryptographic secret, the establisher is urged to gain random data from a source providing data of entropy for everyone else. Using a combination of a reseedable PRBG which is (re-)seeded by one or multiple entropy-provider, the resulting data is entropic even if the entropy-provider fails to deliver the requested amount or a constant stream of entropic data ($\{?\}^? \rightarrow \{0,1\}^\infty$).

3.2.6. 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 $\mathit{prime}(p)$ to check whether p is prime, and repeating this procedure until $\mathit{prime}(p) = \mathit{true}$. 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 $\pi : \mathbb{Z}_n \rightarrow \mathbb{Z}_n$ 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's and Wendy's linear-time unranking algorithm $\mathit{unrank} : \mathbb{Z}_{n!} \rightarrow \Pi_n$ [4]. 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 [1]. 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

$$\mathit{encrypt}_y(m, r) = (g^r, m \cdot y^r) \in G_q \times G_q \tag{3.3}$$

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 $E = (a, b) = \mathit{encrypt}_y(m, r)$, m can be recovered by using the private decryption key x to compute

$$\mathit{decrypt}_x(E) = a^{-x} \cdot b = m. \tag{3.4}$$

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:

$$\mathit{encrypt}_y(m_1, r_1) \cdot \mathit{encrypt}_y(m_2, r_2) = \mathit{encrypt}_y(m_1 \cdot m_2, r_1 + r_2). \tag{3.5}$$

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,

$$\mathit{reEncrypt}_y(E, r') = E \cdot \mathit{encrypt}_y(1, r') = \mathit{encrypt}_y(m, r + r'), \tag{3.6}$$

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:

$$m = \mathit{subGroup}(m') = \begin{cases} m' + 1, & \text{if } (m' + 1)^q = 1, \\ p - (m' + 1), & \text{otherwise.} \end{cases} \tag{3.7}$$

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:

$$m' = \mathit{subGroup}^{-1}(m) = \begin{cases} m - 1, & \text{if } m \leq q, \\ (p - m) - 1, & \text{otherwise.} \end{cases} \tag{3.8}$$

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 [6]. 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

[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.

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 $hash : \{0,1\}^* \rightarrow \{0,1\}^n$. 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 $a = hash(m, g^r) \bmod q$ the integer representation of the hash value of (m, g^r) modulo q, then

$$sign_{sk}(m, r) = (a, r - a \cdot sk) \in \mathbb{Z}_q \times \mathbb{Z}_q \tag{3.9}$$

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

$$verify_{vk}(m, S) = \begin{cases} accept, & \text{if } a = hash(m, g^b \cdot vk^a) \bmod q, \\ reject, & \text{otherwise.} \end{cases} \tag{3.10}$$

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

$$Z_{\mathsf{X}} = certify_{sk_{\mathsf{CA}}}(\mathsf{X}, k), \tag{3.11}$$

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

$$verify_{vk_{\mathsf{CA}}}(Z_{\mathsf{X}}) \in \{reject, reject\}. \tag{3.12}$$

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 $c = \mathit{hash}(b, t, \mathsf{P}) \bmod q$, for $q = |\mathit{image}(\phi)|$.
4. Compute $s = \omega + c \cdot a$.

The triple $(t, c, s) = \mathit{NIZKP}\{(a) : b = \phi(a)\}$ is the resulting non-interactive preimage proof, which can be published without revealing any information about a. Note that $\mathit{image}(\phi) = Y$ 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:

$$\mathit{verify}(\pi) = \begin{cases} \mathit{accept}, & \text{if } c = \mathit{hash}(b, t, \mathsf{P}) \bmod q \text{ and } \phi(s) = t \cdot b^c, \\ \mathit{reject}, & \text{otherwise.} \end{cases} \tag{3.13}$$

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:

$$\begin{aligned} a &= m, \\ b &= c, \\ \phi(x) &= g^x, \end{aligned}$$

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:

$$\begin{aligned} a &= m, \\ b &= (c_1, c_2), \\ \phi(x) &= (g_1^x, g_2^x), \end{aligned}$$

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

- 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 $b_1, \ldots, b_n$ be publicly known, where $b_i = \phi_i(a_i)$
- P proves knowledge of $a_1, \ldots, a_n$ using the Σ-protocol for:

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

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

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

Equality Proof

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

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

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

- Note that $t = (t_1, \ldots, t_n)$, 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 [5, 7]. 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 [2]. 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.[2] 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,

$$\mathit{NIZKP}\{(x_j) : y_j = g^{x_j} \wedge (\bigwedge_i a_{i,j} = a_i^{-x_j})\}, \tag{3.14}$$

[2] 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 $m = b^{1-n} \cdot \prod_i m_i$.

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

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.

- 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.

- 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.

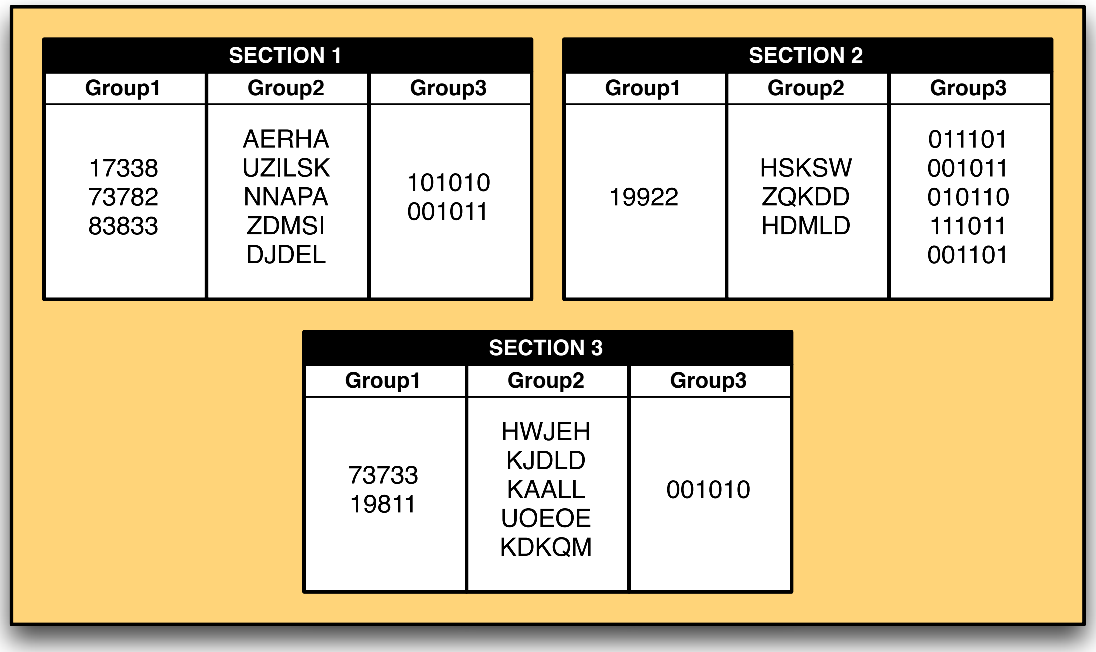
4.2. Posting Properties

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.



SECTION 1		
Group1	Group2	Group3
17338 73782 83833	AERHA UZILSK NNAPA ZDMSI DJDEL	101010 001011

SECTION 2		
Group1	Group2	Group3
19922	HSKSW ZQKDD HDMLD	011101 001011 010110 111011 001101

SECTION 3		
Group1	Group2	Group3
73733 19811	HWJEH KJDLD KAALL UOEOE KDKQM	001010

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 $S_m = \mathit{sign}_{sk_U}(m, \alpha_I)$ 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 $\mathit{verify}_{vk}(m, \alpha_I, S_m)$ 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

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

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. Define the total order n of the message in s .
7. Add the current time t to the message.
8. Create the hash h for the message.
9. Create the signature for the post $S_p = Sign_{sk_{BB}}(m, (s, g, S_m, vk), (n, t, h))$
10. Send notifications for this message if necessary.
11. Save the post.
12. Return (n, t, h, 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.

$\text{Register}(e, Q) : c$

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

$\text{Unregister}(c) : -$

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. Overview

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. Voting Process

6.1.3. 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 $1 \leq j \leq r$)

- 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.

6.1.4. 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 the most recent 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 textual description *election* of the election event (title, description, election rules, languages, etc.).
6. Define $period = (t_1, t_2)$.
7. Post $(\mathsf{electionDefinition} : election, period)$ 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 $\mathit{securityLevel} \in \{1, 2, 3\}$
12. Post $(\mathsf{securityLevel} : \mathit{securityLevel})$ 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 the most recent UniVote tallier certificate.
    b) Verify Z_j.

    Let $\mathcal{Z}_T = \{Z_j : 1 \leq j \leq r\}$ 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 the most recent 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 $(\mathsf{securityLevel} : \mathit{securityLevel})$ from UBV.
2. Select
    - $\mathit{encSetting} \in \Sigma_{\mathsf{ENC}}$,
    - $\mathit{sigSetting} \in \Sigma_{\mathsf{SIG}}$,
    - $\mathit{hashSetting} \in \Sigma_{\mathsf{HASH}}$

    according to $\mathit{securityLevel}$, where Σ_{ENC}, Σ_{SIG}, Σ_{HASH} are corresponding sets of predefined cryptographic settings for ElGamal encryptions, Schnorr signatures, and hash functions, respectively (see Chapter 7).
3. Post $(\mathsf{cryptoSetting} : \mathit{encSetting}, \mathit{sigSetting}, \mathit{hashSetting})$ 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 $(\mathsf{cryptoSetting} : \mathit{encSetting}, \mathit{sigSetting}, \mathit{hashSetting})$ from UBV.
4. Retrieve $\mathcal{G}_Q$ and G from $\mathit{encSetting}$.
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 starting the election.

a) Definition of Election Options and Electoral Roll

EC performs the following step:

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

EA performs the following steps:

3. Define the set of choices $C = \{c_1, \ldots, c_s\}$ and a rule set R describing the set $\mathcal{W} = \mathit{electionOptions}(C, R)$ of valid *election options*, where *electionOptions* is a publicly known function (see Section 6.6.1 for details).
4. Define a detailed description *options* of the election options (election type, lists of choices, etc.).[1]

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

5. Post (electionOptions : $C, R, \mathit{options}$) to UBV.
6. Define the *electoral roll* $V = \{\mathsf{V}_1, \dots, \mathsf{V}_n\}$.
7. Post (electoralRoll : V) to UBV.

EC performs the following steps:

8. Get (*electoralRoll* : V) from UBV.
9. For every $V_i \in V$:
   - Get (certificate : Z_i) from UBC, where Z_i is the most recent UniVote voter certificate for *sigSetting*.
   - Verify Z_i.

   Let $\mathcal{Z}_V = \{Z_1, \dots, Z_n\}$ denote the set of valid voter certificates.[2]
10. Post (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 (keyMixingRequest : $\mathsf{M}_k, g_{k-1}, VK_{k-1}$) to UBV.
   b) Select vk_k from $Z_k \in \mathcal{Z}_M$.
   c) Post (accessRight : vk_k, keyMixingResult, 1) to UBV.
4. M_k performs the following steps:
   a) Get (cryptoSetting : *encSetting*, *sigSetting*, *hashSetting* from UBV.
   b) Retrieve G_q and g from *sigSetting*.
   c) Get (keyMixingRequest : $\mathsf{M}_k, g_{k-1}, VK_{k-1}$) 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 $\pi_k = \mathit{NIZKP}\{(\psi_k, \alpha_k) : g_k = g_{k-1}^{\alpha_k} \wedge VK_k = \mathit{shuffle}_{\psi_k}(VK_{k-1}, \alpha_k)\}$.
   i) Post (keyMixingResult : g_k, VK_k, π_k) to UBV.

[2] To simplify the notation, we assume that $|V| = |\mathcal{Z}_V| = n$. In general, the number of valid certificates will be smaller or equal than the number of eligible voters.

5. EC performs the following steps:
   a) Get (keyMixingResult : g_k, VK_k, π_k) from UBV.
   b) Verify π_k.

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

c) Finalizing Election Preparation

EC performs the following steps:

1. Post (mixedKeys : $\hat{VK}$) to UBV.
2. For every $\hat{vk}_i \in \hat{VK}$, post (accessRight : $\hat{vk}_i$, ballot, $1, \mathit{period}$).
3. Get (electionDefinition : $\mathit{election}, \mathit{period}$) from UBV.
4. Get (electionOptions : $C, R, \mathit{options}$) from UBV.
5. Post (electionData : $\mathit{election}, \mathit{period}, C, R, \mathit{options}, \mathit{encSetting}, \mathit{sigSetting}, \mathit{hashSetting}, y, \hat{g}$) to UBV.[3]

6.2.3. Election Period

a) Vote Creation and Casting

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

1. Get (electionData : $\mathit{election}, \mathit{period}, C, R, \mathit{options}, \mathit{encSetting}, \mathit{sigSetting}, \mathit{hashSetting}, y, \hat{g}$) from UBV.
2. Retrieve $\mathcal{G}_Q$ and G from *encSetting*.
3. Retrieve G_q from *sigSetting*.
4. Choose $v_i \in \mathit{electionOptions}(C, R)$.
5. Compute $m'_i = \mathit{encode}_{C,R}(v_i) \in \mathbb{Z}_Q$.
6. Compute $m_i = \mathit{subGroup}(m'_i) \in \mathcal{G}_Q$.
7. Choose $r_i \in_R \mathbb{Z}_Q$.
8. Compute $E_i = \mathit{encrypt}_y(m_i, r_i) \in \mathcal{G}_Q \times \mathcal{G}_Q$.
9. Generate $\pi_i = \mathit{NIZKP}\{(m_i, r_i) : E_i = \mathit{encrypt}_y(m_i, r_i)\}$.[4]
10. Compute $\hat{vk}_i = \hat{g}^{sk_i} \in \hat{VK}$.
11. Post (ballot : E_i, π_i) to UBV using $\hat{vk}_i$.

[3] 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.

[4] Note that if $E = (a, b) = (G^r, m \cdot y^r)$ is an ElGamal encryption, then $\mathit{NIZKP}\{(m, r) : E = \mathit{encrypt}_y(m, r)\}$ is equivalent to $\mathit{NIZKP}\{(r) : a = G^r\}$, which implies knowledge of m.

b) Finalizing Election Period

When the election period is over, EC performs the following steps:

1. Get $(\mathsf{ballot} : \mathcal{B})$ from UBV.[5]
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 $\mathcal{E}_0 = \{E_1, \dots, E_N\} \in (\mathcal{G}_Q \times \mathcal{G}_Q)^N$ be list of valid encrypted votes for $N \leq n$.

6.2.4. Mixing and Tallying

a) 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 $R_k = (r_{k,1}, \dots, r_{k,N}) \in_R \mathbb{Z}_Q^N$.
    d) Choose $\tau_k : [1, N] \rightarrow [1, N] \in_R \Psi_N$.
    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.

[5] Using an extensive check.

b) 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 $A = (a_1, \ldots, a_N) \in \mathcal{G}_Q^N$ from $\hat{\mathcal{E}} = \{(a_i, b_i) : 1 \leq i \leq N\}$.
3. Compute $A_j = (a_{1,j}, \ldots, a_{N,j})$ 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 $m_i = b_i \cdot \prod_j a_{i,j} \in \mathcal{G}_Q$.
   b) Compute $m'_i = \mathit{subGroup}^{-1}(m_i) \in \mathbb{Z}_Q$.
   c) Compute $v_i = \mathit{decode}_{C,R}(m'_i)$.
8. For $\mathcal{V} = \{v'_1, \ldots, v'_N\}$, let $\mathcal{V}' = \mathcal{V} \cap \mathcal{W} = \{v'_1, \ldots, v'_{N'}\}$ be the list of valid plaintext votes $v'_i = (v'_{i,1}, \ldots, v'_{i,s})$, where $v'_{i,j}$ denotes the number of votes for c_j.
9. For each $c_j \in C = \{c_1, \ldots, c_s\}$, compute $r_j = \sum_{j=1}^{N'} v'_{i,j}$.
10. Post ($\mathsf{decryptedVotes} : \mathcal{V}', \mathit{result}$) to UBV, where $\mathit{result} = (r_1, \ldots, r_s)$ 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).

6.3.1. Preparation

Upon notification of a newly issued voter certificate $\bar{Z}_i$ for V_i, EC performs the following steps:

1. Check if $\mathsf{V}_i \in V$.
2. Verify $\bar{Z}_i$.
3. Post $(\mathsf{newVoterCertificate} : \bar{Z}_i)$ to UBV.

6.3.2. Cancelling an Existing Key

The existence of a new certificate $\bar{Z}_i$ requires EC to cancel an existing public key (if one exists). For this, EC performs the following steps:

1. Check if $\mathcal{Z}_V$ contains a voter certificate for V_i. If this is the case, let $Z_i \in \mathcal{Z}_V$ denote this certificate. If this is not the case, discard the following steps and continue with the procedure in Section 6.3.3.
2. Select $vk_{i,0}$ from Z_i.

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

Let $\hat{vk}_i = vk_{i,m}$ denote the cancelled anonymized public key. EC performs the following steps:

1. Post $(\mathsf{cancelledKey} : \hat{vk}_i)$ to UBV.
2. Post $(\mathsf{accessRight} : \hat{vk}_i, \mathsf{ballot}, 0, period)$. to UBV.

6.3.3. Adding the New Certificate

EC performs the following steps:

1. Check if $\mathsf{V}_i \in V$.
2. Get $(\mathsf{certificate} : \bar{Z}_i)$ from UBC, where $\bar{Z}_i$ is the most recent issued UniVote voter certificate for *sigSetting*.
3. Post $(\mathsf{newVoterCertificate} : \bar{Z}_i)$ to UBV.

Let $\bar{\mathcal{Z}}_V$ denote the current set of certificates added during the election period.

Let $\bar{vk}_{i,0} = \bar{vk}_i$ be the new verification key from $\bar{Z}_i$. Repeat the following steps for each $\mathsf{M}_k \in M$ (in ascending order for $1 \leq k \leq m$):[6]

4. Compute $\bar{vk}_{i,k} = \bar{vk}_{i,k-1}^{\alpha_k}$.
5. Generate $\pi_{\bar{vk}_{i,k}} = \mathit{NIZKP}\{(\alpha_k) : g_k = g_{k-1}^{\alpha_k} \wedge \bar{vk}_{i,k} = \bar{vk}_{i,k-1}^{\alpha_k}\}$.
6. Post $(\mathsf{M}_k, id, \bar{vk}_{i,k}, \pi_{\bar{vk}_{i,k}})$ to UBV.

EC performs the following steps:

7. For each $\mathsf{M}_k \in M$, do the following:
   a) Check that $\mathit{verify}_{vk_k}(id, \bar{vk}_{i,k}, \pi_{\bar{vk}_{i,k}}, S_{\bar{vk}_k}) = \mathit{accept}$.
   b) Check correctness of $\pi_{\bar{vk}_{i,k}}$ (see Subsection **??** for details).
8. Let $\bar{vk}'_i = \bar{vk}_{i,m} = \bar{vk}_i^{\alpha}$.
9. Post $(\mathsf{EC}, id, \mathsf{V}_i, \bar{vk}'_i)$ to UBV. Let $\bar{VK}'$ denote the set of all public keys $\bar{vk}'_i$ added during the election period.

6.3.4. Late Renewal of Registration

Essentially the same steps are repeated, if the current set $\mathcal{Z}_V \cup \bar{\mathcal{Z}}_V$ contains an earlier certificate $\hat{Z}_i = (\mathsf{V}_i, \hat{vk}_i, \hat{t}_i, \mathsf{CA}, \hat{C}_i)$ of V_i. Note that the first part of the above procedure is not repeated, since $\hat{Z}_i \in \mathcal{Z}_V \cup \bar{\mathcal{Z}}_V$ implies that $\hat{Z}_i$ has already been verified. Let $\hat{vk}_{i,0} = \hat{vk}_i$ be the former verification key from $\hat{Z}_i$. Repeat the following steps for each $\mathsf{M}_k \in M$ (in ascending order for $1 \leq k \leq m$):[7]

1. Compute $\hat{vk}_{i,k} = \hat{vk}_{i,k-1}^{\alpha_k}$.
2. Generate $\pi_{\hat{vk}_{i,k}} = \mathit{NIZKP}\{(\alpha_k) : g_k = g_{k-1}^{\alpha_k} \wedge \hat{vk}_{i,k} = \hat{vk}_{i,k-1}^{\alpha_k}\}$ (see Subsection **??** for details).
3. Generate signature $S_{\hat{vk}_{i,k}} = \mathit{sign}_{sk_k}(id, \hat{vk}_{i,k}, \pi_{\hat{vk}_{i,k}})$.

[6] Note that this procedure corresponds to the borderline case of the general mixing procedure for a single innput public key (with a simplified proof).

[7] Again, this procedure corresponds to the borderline case of the general mixing procedure for a single innput public key (with a simplified proof).

4. Publish $(\mathsf{M}_k, id, \widehat{vk}_{i,k}, \pi_{\widehat{vk}_{i,k}}, S_{\widehat{vk}_{i,k}})$ on UBV.

EC performs the following steps:

5. For each $\mathsf{M}_k \in M$, do the following:
   a) Check that $verify_{vk_k}(id, \widehat{vk}_{i,k}, \pi_{\widehat{vk}_{i,k}}, S_{\widehat{vk}_{i,k}}) = accept$.

   b) Check correctness of $\pi_{\widehat{vk}_{i,k}}$ (see Subsection **??** for details).
6. Let $\widehat{vk}'_i = \widehat{vk}_m = \widehat{vk}_i^{\alpha}$.
7. Generate signature $S_{\widehat{vk}'_i} = sign_{sk_{\mathsf{EC}}}(id, \mathsf{V}_i, \widehat{vk}'_i)$.
8. Publish $(\mathsf{EC}, id, \mathsf{V}_i, \widehat{vk}'_i, S_{\widehat{vk}'_i})$ on UBV. Let $\widehat{VK}'$ denote the set of all public keys $\widehat{vk}'_i$, which have been replaced by a new one during the election period.

6.4. Summary of Election Data

Author	Group	Amount	Content	Readers
EC	administrationCertificate	1	Z_{EA}	–
EA	electionDefinition	1	$election, period$	EC
EA	trustees	1	T, M	EC
EA	securityLevel	1	$securityLevel$	EC
EC	trusteeCertificates	1	$\mathcal{Z}_T, \mathcal{Z}_M$	–
EC	cryptoSetting	1	$encSetting, sigSetting, hashSetting$	T_j, M_k
T_j	encryptionKeyShare	r	y_j, π_j	EC
EC	encryptionKey	1	y	–
EA	electionOptions	1	$C, R, options$	EC
EA	electoralRoll	1	V	EC
EC	voterCertificates	1	$\mathcal{Z}_V$	–
EC	keyMixingRequest	m	$\mathsf{M}_k, g_{k-1}, VK_{k-1}$	M_k
M_k	keyMixingResult	1	g_k, VK_k, π_k	EC
EC	mixedKeys	m	$\widehat{VK}$	–
EC	electionData	1	$election, period, C, R, options, encSetting,$ $sigSetting, hashSetting, y, \hat{g}$	V_i
V_i	ballot	1	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	$\mathcal{V}', result$	–

Table 6.1.: Summary of election data published on UBV. Corresponding access rights are attributed by EC.

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

-

b) Election Preparation

-

c) Mixing and Tallying

-

6.6. Encoding Choices, Rules, and Votes

6.6.1. Choices and Rules

To allow a variety of different election types, we consider two finite sets, a set $C = \{c_1, \ldots, c_s\}$ of possible *election choices* and a set R of *election rules*. For each election choice $c_j \in C$ in a given election, the election system outputs the number r_j of votes that c_j has received from the voters. Each election rule in R defines some constraints on how voters can distribute their votes among the election choices. We use $v_{i,j}$ to denote this number for a particular voter V_i. We distinguish three types of election rules:

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

Two or several sets of candidates 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

$$(C_1, R_1) \circ (C_2, R_2) = (C_1 \cup C_2, R_1 \cup R_2)$$

for two sets of choices C_1, C_2 and corresponding sets of rules R_1, R_2. Note that this can be used to describe party-list elections (see example below).

a) Examples

- Referendum: 1-out-of-2

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

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

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

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

$$C = \{c_1, \ldots, c_n\},\ R = \left\{ \begin{array}{c} \Sigma : \{c_1, \ldots, c_n\} \rightarrow [1,1] \\ \forall : \{c_1, \ldots, c_n\} \rightarrow [0,1] \end{array} \right\}$$

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

$$C = \{c_1, \ldots, c_n\},\ R = \left\{ \begin{array}{c} \Sigma : \{c_1, \ldots, c_n\} \rightarrow [0,1] \\ \forall : \{c_1, \ldots, c_n\} \rightarrow [0,1] \end{array} \right\}$$

  or

$$C = \{c_1, \ldots, c_n, null\},\ R = \left\{ \begin{array}{c} \Sigma : \{c_1, \ldots, c_n, null\} \rightarrow [1,1] \\ \forall : \{c_1, \ldots, c_n, null\} \rightarrow [0,1] \end{array} \right\}$$

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

$$C = \{c_1, \ldots, c_n\},\ R = \big\{ \forall : \{c_1, \ldots, c_n\} \rightarrow [0,1] \big\}$$

- Range Voting: Up to s votes per choice

$$C = \{c_1, \ldots, c_n\},\ R = \big\{ \forall : \{c_1, \ldots, c_n\} \rightarrow [0,s] \big\}$$

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

$$C = \{c_1, \ldots, c_n\},\ R = \left\{ \begin{array}{c} \Sigma : \{c_1, \ldots, c_n\} \rightarrow [0,k] \\ \forall : \{c_1, \ldots, c_n\} \rightarrow [0,1] \end{array} \right\}$$

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

$$C = \{c_1, \ldots, c_n\},\ R = \left\{ \begin{array}{c} \Sigma : \{c_1, \ldots, c_n\} \rightarrow [0,k] \\ \forall : \{c_1, \ldots, c_n\} \rightarrow [0,s] \end{array} \right\}$$

  Note that $k > n$ is allowed here.

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

$$C = \{c_1, \ldots, c_n\},\ R = \left\{ \begin{array}{l} \forall : \{c_1, \ldots, c_n\} \to [1, n] \\ \neq : \{c_1, \ldots, c_n\} \end{array} \right\}$$

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

$$C = \{c_1, \ldots, c_n\},\ R = \left\{ \begin{array}{l} \forall : \{c_1, \ldots, c_n\} \to [1, k] \\ \neq : \{c_1, \ldots, c_n\} \end{array} \right\}$$

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

$$\begin{aligned} C &= \{c_1, \ldots, c_n\} \cup \{\ell_1, \ldots, \ell_m\}, \\ R &= \left\{ \begin{array}{l} \Sigma : \{c_1, \ldots, c_n\} \to [0, k] \\ \forall : \{c_1, \ldots, c_n\} \to [0, s] \end{array} \right\} \cup \left\{ \begin{array}{l} \Sigma : \{\ell_1, \ldots, \ell_m\} \to [1, 1] \\ \forall : \{\ell_1, \ldots, \ell_m\} \to [0, 1] \end{array} \right\} \end{aligned}$$

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

6.6.2. Encoding Votes

For $C = \{c_1, \ldots, c_s\}$, let $v_{i,j} \leq \rho_j$ the number of votes attributed to c_j by a given voter V_i. With ρ_j we denote the maximum number of votes that the election rules in R allow for c_j. The tuple $v_i = (v_{i,1}, \ldots, v_{i,s}) \in \mathcal{W}$ represents a valid vote of V_i, where $\mathcal{W} = \mathit{electionOptions}(C, R)$ denotes the set of all valid votes for given sets C and R.

To encode $v_i \in \mathcal{W}$ as an integer, consider a bit string of length $B = \sum_{j=1}^{n} b_j$, where $b_j = \lfloor \log_2 \rho_j \rfloor + 1$ 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., $B_1 = 0$, $B_2 = b_1$, $B_3 = b_1 + b_2$, $\ldots$, $B_{n+1} = B$. The encoding function $\mathit{encode}_{C,R} : \mathcal{W} \to \{0, \ldots, 2^B - 1\}$ can then be defined as follows:

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

Since some integers in $\{0, \ldots, 2^B - 1\}$ do not represent valid votes according to the election rules R, this encoding is not optimal in terms of memory space consumption. In the vast majority of practical cases, however, we believe that the size of the message space G_Q of the ElGamal cryptosystem is large enough to support this encoding.

To decode an integer representation $x = \mathit{encode}_{C,R}(v_i)$ 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:

$$\mathit{decode}_{C,R}(x) = (v_{i,1}, \ldots, v_{i,s}),\ \text{where } v_{i,j} = \lfloor x / 2^{B_j} \rfloor \bmod 2^{b_j}.$$

Symbol	Type	Element of
Z_{EA}	certificate	
$\mathit{election}$	description	
$\mathit{period} = (t_1, t_2)$	time period	
$T = \{\mathsf{T}_1, \ldots, \mathsf{T}_t\}$	identifier list	
$M = \{\mathsf{M}_1, \ldots, \mathsf{M}_m\}$	identifier list	
$\mathit{securityLevel}$	integer	$\{1, 2, 3\}$
$\mathcal{Z}_T$	certificate list	
$\mathcal{Z}_M$	certificate list	
$\mathit{encSetting}$	identifier	Σ_{ENC}
$\mathit{sigSetting}$	identifier	Σ_{SIG}
$\mathit{hashSetting}$	identifier	Σ_{HASH}
y_j	public key share	$\mathcal{G}_Q$
π_j	zero-knowlegde proof	
y	public key	$\mathcal{G}_Q$
$C = \{c_1, \ldots, c_s\}$	choice list	
R	rule list	
$\mathit{options}$	description	
$V = \{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$	identifier list	
$\mathcal{Z}_V$	certificate list	
g_k	generator	G_q
VK_k	public key list	
$\hat{VK}$	public key list	
π_k	zero-knowledge proof	
$\hat{g}$	generator	G_q
$E_i = (a_i, b_i)$	ElGamal encryption	$\mathcal{G}_Q \times \mathcal{G}_Q$
π_i	zero-knowledge proof	
$\mathcal{E}_k$	ElGamal encryption list	
$\hat{\mathcal{E}}$	ElGamal encryption list	
π'_k	zero-knowledge proof	
$A_j = (a_{1,j}, \ldots, a_{N,j})$	partial decryption list	$\mathcal{G}_Q^N$
π'_j	zero-knowledge proof	
$\mathcal{V}' = \{v'_1, \ldots, v'_{N'}\}$	list of votes	
$\mathit{result} = (r_1, \ldots, r_s)$	integer list	$\mathbb{N}^s$

Table 6.2.: Summary of election data published on UBV.

Part III.

Technical Specification

7. Cryptographic Settings

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

Level	Strength	Residue Class			Elliptic Curve	Hash Value
		Modulo	Group max.	Group min.		
1	80	1024	1023	256	–	–
2	112	2048	2047	256	–	–
3	128	3072	3071	256	256	256
4	192	–	–	–	384	384
5	256	–	–	–	521	512

$$\begin{aligned} \Sigma_{\mathsf{ENC}} &= \{\mathsf{RC1}, \mathsf{RC2}, \mathsf{RC3}, \mathsf{EC3}, \mathsf{EC4}, \mathsf{EC5}\} \\ \Sigma_{\mathsf{SIG}} &= \{\mathsf{RC1s}, \mathsf{RC2s}, \mathsf{RC3s}, \mathsf{EC3}, \mathsf{EC4}, \mathsf{EC5}\} \\ \Sigma_{\mathsf{HASH}} &= \{\mathsf{H3}, \mathsf{H4}, \mathsf{H5}\} \end{aligned}$$

7.1. Residue Classes

Name	RC1
Level	1
Strength	80
Type	Residue Classes
Bit Length	1024/1023
Modulo	
Group Size	
Co-factor	2
Generator	

Name	RC2
Level	2
Strength	112
Type	Residue Classes
Bit Length	2048/2047
Modulo	
Group Size	
Co-factor	2
Generator	

Name	RC3
Level	3
Strength	128
Type	Residue Classes
Bit Length	3072/3071
Modulo	
Group Size	
Co-factor	2
Generator	

Name	RC1s
Level	1
Strength	80
Type	Residue Classes
Bit Length	1024/256
Modulo	161931481198080639220214033595931441094586304918402813506510547237223787775475425991443924977419330663170224569788019900180050114468430413908687329871251101280878786588515668012772798298511621634145464600626619548823238185390034868354933050128115662663653841842699535282987363300852550784188180264807606304297
Group Size	65133683824381501983523684796057614145070427752690897588060462960319251776021
Co-factor	24861403760716632875247331778465318559945273192732440781453938775554007545712348715294434346682772598368660726428686204682262081347281812139231616555387573862842271111716580558969211505099354230364743581681223815751132821487026727776
Generator	109291242937709414881219423205417309207119127359359243049468707782004862682441897432780127734395596275377218236442035534825283725782836026439537687695084410797228793004739671835061419040912157583607422965551428749149162882960112513332411954585778903685207256083057895070357159920203407651236651002676481874709

Name	RC2s
Level	2
Strength	112
Type	Residue Classes
Bit Length	2048/256
Modulo	
Group Size	
Co-factor	
Generator	

Name	RC3s
Level	3
Strength	128
Type	Residue Classes
Bit Length	3072/256
Modulo	
Group Size	
Co-factor	
Generator	

7.2. Elliptic Curves

Name	EC3
Level	3
Strength	128
Type	Elliptic Curve
Bit Length	256
Standard	

Name	EC4
Level	4
Strength	192
Type	Elliptic Curve
Bit Length	384
Standard	

Name	EC5
Level	5
Strength	256
Type	Elliptic Curve
Bit Length	521
Standard	

7.3. Hash Functions

Name	H3
Level	3
Strength	128
Bit Length	256
Standard	

Name	H4
Level	4
Strength	192
Bit Length	384
Standard	

Name	H4
Level	5
Strength	256
Bit Length	512
Standard	

8. UniBoard

8.1. 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.

8.2. 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, hash].

8.3. 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,hash,bo gamma=[timestamp]

9. UniCert

9.1. 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"]
}
```

10. UniVote

10.1. EC Setup Phase Actions

10.1.1. Initial

Notification	Work	Post	Successors
accessRight	Start the new election	-	DefineEA

10.1.2. DefineEA

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

10.1.3. GrantElectionDefinition

Notification	Work	Post	Successors
-	Grant EA access to electionDefinition	accessRight : $vk_{EA}, electionDefinition, 1$	-

10.1.4. GrantTrustees

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

10.1.5. GrantSecurityLevel

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

10.1.6. GrantElectionDefinition

Notification	Work	Post	Successors
-	Grant EA access to electionDefinition	accessRight : $vk_{EA}, electionDefinition, 1$	-

10.1.7. PublishTrusteeCerts

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

10.1.8. SetCryptoSetting

Notification	Work	Post	Successors
securityLevel	Set cryptoSetting	cryptoSetting : *settings*	GrantEncyptionKeyShares

10.1.9. GrantEncyptionKeyShares

Notification	Work	Post	Successors
-	Grant Trustees access to keyshares	accessRight : $vk_t, encrytptionKeyShare, 1$	

10.1.10. CombineEncryptionKeyShares

Notification	Work	Post	Successors
encryptionKeyShare	Validate keyshare and if all keyshare are published combine them	encyptionKey : y_j, π_j	

10.2. EC Perparation Phase Actions

10.2.1. GrantEAAccess

Notification	Action	Unregister	Register	Post
encrytptionKeyShare	Check keyShare			
	Post encryptionKey	encrytptionKeyShare		encryptionKey : y
	Grant access for EO			accessRight : $vk_{EA}, electionOptions, 1$
	Grant access for ER		electoralRoll	accessRight : $vk_{EA}, electoralRoll, 1$

10.2.2. InitiateKeyMixing (Trustee: Mixer)

Notification	Action	Unregister	Register	Post
electoralRoll	Retrieve voter certs	electoralRoll		voterCertificates : $\mathcal{Z}_V$
	Post keyMixingRequest			keyMixingRequest : M_1, g_0, VK_0
	Grant access for keyMixingResult		keyMixingResult	accessRight : $vk_1, keyMixingResult, 1$

10.2.3. MixKeys (Trustee: Mixer)

Notification	Action	Unregister	Register	Post
keyMixingResult	Post keyMixingRequest			keyMixingRequest : M_k, g_{k-1}, VK_{k-1}
	Grant access for keyMixingResult			accessRight : $vk_k, keyMixingResult, 1$

10.2.4. FinishSetup

Notification	Action	Unregister	Register	Post
	Post mixedKey	keyMixingResult		mixedKeys : $\hat{VK}$
	Post electionData			electionData : $data$
	Grant access for Ballots			accessRight : $\hat{vk}_i, ballot, 1, period$

10.3. Trustees

10.3.1. Tallier: CreateSharedKey

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

10.3.2. Tallier: PartialDecryption

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

10.3.3. Mixer: MixingPublicKey

Notification	Work	Post	Successor
keyMixingRequest	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	CancellingExistingKey AddingNewKey MixingVote

10.3.4. Mixer: MixingVote

Notification	Work	Post	Successor
voteMixingRequest	Get electionSetting:$\mathcal{G}_q, G$ from UBV $R = (r_1, \ldots, r_N) \in_R \mathbb{Z}_Q^N$ $\tau : [1, N] \rightarrow [1, N] \in_R \Psi_N$ $\mathcal{E} = \mathit{shuffle}_{\tau}(\mathcal{E}\_, R)$ $\pi = NIZKP\{(\tau, R)\ \mathcal{E} = shuffle_{\tau}(\mathcal{E}\_, R)\}$ finish.	voteMixingResult: $\mathcal{E}\_, \pi$ Store: $\psi \in_R \Psi_n$ (permutation vector)	-

10.3.5. Mixer: AddingNewKey

Notification	Work	Post	Successor
ElectionEndTimer	run finish. if ElectionEndTimer → finish.		

10.3.6. Mixer: CancellingExistingKey

Notification	Work	Post	Successor
keyCancellingRequest ElectionEndTimer	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 → finish.	keyCancellingResult: V_i, vk_i, π_i	-

Bibliography

[1] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and D. Chaum, editors, *CRYPTO'84, Advances in Cryptology*, LNCS 196, pages 10–18, Santa Barbara, USA, 1984. Springer.

[2] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure distributed key generation for discrete-log based cryptosystems. In J. Stern, editor, *EUROCRYPT'99, 18th International Conference on the Theory and Application of Cryptographic Techniques*, LNCS 1592, pages 295–310, Prague, Czech Republic, 1999.

[3] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, USA, 1996.

[4] W. Myrvold and F. Wendy. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.

[5] T. P. Pedersen. A threshold cryptosystem without a trusted party. In D. W. Davies, editor, *EUROCRYPT'91, 10th Workshop on the Theory and Application of Cryptographic Techniques*, volume 547 of *LNCS 547*, pages 522–526, Brigthon, U.K., 1991.

[6] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[7] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[8] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[9] M. Szudzik. An elegant pairing function. In *NKS'06, 3rd Wolfram Science Conference*, Washington, USA, 2006.

[10] D. Wikström. *How to Implement a Stand-alone Verifier for the Verificatum Mix-Net.* Verificatum AB, Stockholm, Sweden, 2012.