CHVote System Specification

Version 1.3

Rolf Haenni, Reto E. Koenig, Philipp Locher, Eric Dubuis

{rolf.haenni,reto.koenig,philipp.locher,eric.dubuis}@bfh.ch

October 31, 2017

Bern University of Applied Sciences
CH-2501 Biel, Switzerland

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Revision History

Revision	Date	Author(s)	Description
0.1	14.07.2016	Rolf Haenni	Initial Draft.
0.2	11.10.2016	Rolf Haenni	Draft to present at meeting.
0.3	17.10.2016	Rolf Haenni, Reto E. Koenig	Vote casting and confirmation algorithms finished.
0.4	24.10.2016	Rolf Haenni, Reto E. Koenig	Update of vote casting and confirmation algorithms.
0.5	18.11.2016	Rolf Haenni, Philipp Locher	Mixing process finished.
0.6	25.11.2016	Rolf Haenni	String conversion introduced, tallying finished.
0.7	07.12.2016	Rolf Haenni	Section 5 finished.
0.8	17.12.2016	Rolf Haenni	Hashing algorithms and cryptographic parameters added.
0.9	10.01.2017	Rolf Haenni	Section 8 finished.
0.10	06.02.2017	Rolf Haenni	Security parameters finished.
0.11	21.02.2017	Rolf Haenni	Section 6 finished, reorganization of Section 7.
0.11	14.03.2017	Rolf Haenni	Section 7 finished.
0.12	21.03.2017	Rolf Haenni	Section 8 finished.
0.13	30.03.2017	Rolf Haenni	Minor corrections, Section 1 finished.
1.0	12.04.2017	Rolf Haenni	Minor corrections, Section 2 finished.
1.1	19.04.2017	Rolf Haenni	Conclusion added.
1.1.1	24.05.2017	Rolf Haenni	Parameters p' changed.
1.2	14.07.2017	Rolf Haenni	Major protocol revision, full sender privacy added to oblivious transfer.
1.2.1	14.09.2017	Rolf Haenni	Various minor corrections.
1.3	28.09.2017	Rolf Haenni	Description and algorithms for channel security added.

Contents

Part I.

Project Context

1. Introduction

The State of Geneva is one of the worldwide pioneers in offering Internet elections to their citizens. The project, which was initiated in 2001, was one of first and most ambitious attempts in the world of developing an electronic voting procedure that allows the submission of votes over the Internet in referendums and elections. For this, a large number of technical, legal, and administrative problems had to be solved. Despite the complexity of these problems and the difficulties of finding appropriate solutions, first legally binding referendums had been conducted in 2003 in two suburbs of the City of Geneva. Referendums on cantonal and national levels followed in 2004 and 2005. In a popular referendum in in 2009, a new constitutional provision on Internet voting had been approved by a 70.2% majority. At more or less the same time, Geneva started to host referendums and elections for other Swiss cantons. The main purpose of these collaborations was—and still is—to provide Internet voting to Swiss citizens living abroad.

While the Geneva Internet voting project continued to expand, concerns about possible vulnerabilities had been raised by security experts and scientists. There were two main points of criticism: the lack of transparency and verifiability and the *insecure platform problem* [43]. The concept of *verifiable elections* has been known in the scientific literature for quite some time [11], but the Geneva e-voting system—like most other e-voting systems in the world at that time—remained completely unverifiable. The awareness of the insecure platform problem was given from the beginning of the project [42], but so-called *code voting* approaches and other possible solutions were rejected due to usability concerns and legal problems [40].

In the cryptographic literature on remote electronic voting, a large amount of solutions have been proposed for both problems. One of the most interesting approaches, which solves the insecure platform problem by adding a verification step to the vote casting procedure, was implemented in the Norwegian Internet voting system and tested in legally binding municipal and county council elections in 2011 and 2013 [8, 24, 25, 46]. The Norwegian project was one of the first in the world that tried to achieve a maximum degree of transparency and verifiability from the very beginning of the project. Despite the fact that the project has been stopped in 2014 (mainly due to the lack of increase in turnout), it still serves as a model for future projects and second-generation systems.

As a response to the third report on *Vote électronique* by the Swiss Federal Council and the new requirements of the Swiss Federal Chancellery [39, 5], the State of Geneva decided to introduce a radical strategic change towards maximum transparency and full verifiability. For this, they invited leading scientific researchers and security experts to contribute to the development of their second-generation system, in particular by designing a cryptographic voting protocol that satisfies the requirements to the best possible degree. In this context, a collaboration contract between the State of Geneva and the Bern University of Applied

Sciences was signed in 2016. The goal of this collaboration is to lay the foundation for an entirely new system, which will be implemented from scratch.

As a first significant outcome of this collaboration, a scientific publication with a proposal for a cryptographic voting protocol was published in 2016 at the *12th International Joint Conference on Electronic Voting* [27]. The proposed approach is the basis for the specification presented in this document. Compared to the protocol as presented in the publication, the level of technical details in this document is considerably higher. By providing more background information and a broader coverage of relevant aspects, this text is also more self-contained and comprehensive than its predecessor.

The core of this document is a set of approximately 60 algorithms in pseudo-code, which are executed by the protocol parties during the election process. The presentation of these algorithms is sufficiently detailed for an experienced software developer to implement the protocol in a modern programming language.[1] Cryptographic libraries are only required for standard primitives such as hash algorithms and pseudo-random generators. For one important sub-task of the protocol—the mixing of the encrypted votes—a second scientific publication was published in 2017 at the *21th International Conference on Financial Cryptography* [26]. By facilitating the implementation of a complex cryptographic primitive by non-specialists, this paper created a useful link between the theory of cryptographic research and the practice of implementing cryptographic systems. The comprehensive specification of this document, which encompasses all technical details of a fully-featured cryptographic voting protocol, provides a similar, but much broader link between theory and practice.

1.1. Principal Requirements

In 2013, the introduction of the new legal ordinance by the Swiss Federal Chancellery, *Ordinance on Electronic Voting* (VEleS), created a new situation for the developers and providers of Internet voting systems in Switzerland [4, 5]. Several additional security requirements have been introduced, in particular requirements related to the aforementioned concept of verifiable elections. The legal ordinance proposes a two-step procedure for expanding the electorate allowed of using the electronic channel. A system that meets the requirements of the first expansion stage may serve up to 50% of the cantonal and 30% of the federal electorate, whereas a system that meets the requirements of the second (full) expansion stage may serve up to 100% of both the cantonal and the federal electorate. Current systems may serve up to 30% of the cantonal and 10% of the federal electorate [5, 6].

The cryptographic protocol presented in this document is designed to meet the security requirements of the full expansion stage. From a conceptual point of view, the most important requirements are the following:

- *End-to-End Encryption*: The voter's intention is protected by strong encryption along the path from the voting client to the tally. To guarantee vote privacy even after decrypting the votes, a cryptographically secure anonymization method must be part of the post-election process.

[1] See https://github.com/republique-et-canton-de-geneve/chvote-protocol-pocfor a complete proof of concept implementation in Java by a developer of the CHVote project.

- *Individual Verifiability*: After submitting an encrypted vote, the voter receives conclusive evidence that the vote has been cast and recorded as intended. This evidence enables the voter to exclude with high probability the possibility that the vote has been manipulated by a compromised voting client. According to [4, Paragraph 4.2.4], this is the proposed countermeasure against the insecure platform problem. The probability of detecting a compromised vote must be 99.9% or higher.
- *Universal Verifiability*: The correctness of the election result can be tested by independent verifiers. The verification includes checks that only votes cast by eligible voters have been tallied, that every eligible voter has voted at most once, and that every vote cast by an eligible voter has been tallied as recorded.
- *Distribution of Trust*: Several independent *control components* participate in the election process, for example by sharing the private decryption key or by performing individual anonymization steps. While single control components are not fully trusted, it is assumed that they are trustworthy as a group, i.e., that at least one of them will prevent or detect any type of attack or failure. The general goal of distributing trust in this way is to prevent single points of failures.

In this document, we call the control components *election authorities* (see Section 6.1). They are jointly responsible for generating the necessary elements of the implemented cast-as-intended mechanism. They also generate the public encryption key and use corresponding shares of the private key for the decryption. Finally, they are responsible for the anonymization process consisting of a series of cryptographic shuffles. By publishing corresponding cryptographic proofs, they demonstrate that the shuffle and decryption process has been conducted correctly. Checking these proof is part of the universal verification.

While verifiability and distributed trust are mandatory security measures at the full expansion stage, measures related to some other security aspects are not explicitly requested by the legal ordinance. For example, regarding the problem of vote buying and coercion, the legal ordinance only states that the risk must not be significantly higher compared to voting by postal mail [4, Paragraph 4.2.2]. Other problems of lower significance in the legal ordinance are the possibility of privacy attacks by malware on the voting client, the lack of long-term security of today's cryptographic standards, or the difficulty of printing highly confidential information and sending them securely to the voters. We adopt corresponding assumptions in this document without questioning them.

1.2. Goal and Content of Document

The goal of this document is to provide a self-contained, comprehensive, and fully-detailed specification of a new cryptographic voting protocol for the future system of the State of Geneva. The document should therefore describe every relevant aspect and every necessary technical detail of the computations and communications performed by the participants during the protocol execution. To support the general understanding of the cryptographic protocol, the document should also accommodate the necessary mathematical and cryptographic background information. By providing this information to the maximal possible extent, we see this document as the ultimate companion for the developers in charge of implementing the future Internet voting system of the State of Geneva. It may also serve as a manual for developers trying to implement an independent election verification software. The decision of making this document public will even enable implementations by third parties, for example by students trying to develop a clone of the Geneva system for scientific evaluations or to implement protocol extensions to achieve additional security properties. In any case, the target audience of this document are system designers, software developers, and cryptographic experts.

What is currently entirely missing in this document are proper definitions of the security properties and corresponding formal proofs that these properties hold in this protocol. An informal discussion of such properties is included in the predecessor document [27], but this is not sufficient from a cryptographic point of view. However, the development of proper security proofs, which is an explicit requirement of the legal ordinance, has been excluded from this collaboration. The goal is to outsource the formal proofs to a separate project by an external third party, which will at the same time conduct a review of the specification. Results from this sister project will be published in a separate document as soon as they are available. It is likely that their feedback will lead to a revision of this document.

This document is divided into five parts. In Part I, we describe the general project context, the goal of this work and the purpose of this document (Chapter 1). We also give a first outline of the election procedure, an overview of the supported election types, and a discussion of the expected electorate size (Chapter 2). In Part II, we first introduce notational conventions and some basic mathematical concepts (Chapter 4). We also describe conversion methods for some basic data types and propose a general method for computing hash values of composed mathematical objects (Chapter 3). Finally, we summarize the cryptographic primitives used in the protocol (Chapter 5). In Part III, we first provide a comprehensive protocol description with detailed discussions of many relevant aspects (Chapter 6). This description is the core and the major contribution of this document. Further details about the necessary computations during a protocol execution are given in form of an exhaustive list of pseudo-code algorithms (Chapter 7). Looking at these algorithms is not mandatory for understanding the protocol and the general concepts of our approach, but for developers, they provide a useful link from the theory towards an actual implementation. In Part IV, we propose three security levels and corresponding system parameters, which we recommend to use in an actual implementation of the protocol (Chapter 8). Finally, in Part V, we summarize the main achievements and conclusions of this work and discuss some open problem and future work.

2. Election Context

The election context, for which the protocol presented in this document has been designed, is limited to the particular case of the direct democracy as implemented and practices in Switzerland. Up to four times a year, multiple referendums or multiple elections are held simultaneously on a single election day, sometimes on up to four different political levels (federal, cantonal, municipal, pastoral). In this document, we use „election“ as a general term for referendums and elections and *election event* for an arbitrary combinations of such elections taking place simultaneously. Responsible for conducting an election event are the cantons, but the election results are published for each municipality. Note that two residents of the same municipality do not necessarily have the same rights to vote in a given election event. For example, some canton or municipalities accept votes from residents without a Swiss citizenship, provided that they have been living there long enough. Swiss citizens living abroad are not residents in a municipality, but the are still allowed to voter in federal or cantonal issues.

Since voting has a long tradition in Switzerland and is practiced by its citizens very often, providing efficient voting channels has always been an important consideration for election organizers to increase turnout and to reduce costs. For this reason, some cantons started to accept votes by postal mail in 1978, and later in 1994, postal voting for federal issues was introduced in all cantons. Today, voting by postal mail is the dominant voting channel, which is used by approximately 90% of the voters. Given the stability of the political system in Switzerland and the high reliability of most governmental authorities, concerns about manipulations when voting from a remote place are relatively low. Therefore, with the broad acceptance and availability of information and communications technologies today, moving towards an electronic voting channel seems to be the natural next step. This one of the principal reasons for the Swiss government to support the introduction of Internet voting. The relatively slow pace of the introduction is a strategic decision to limit the security risks.

2.1. General Election Procedure

In the general setting of the CHVote system, voters submit their electronic vote using a regular web browser on their own computer. To circumvent the problem of malware attacks on these machines, some approaches suggest using an out-of-band channel as a trust anchor, over which additional information is transmitted securely to the voters. In the particular setting considered in this document, each voter receives a *voting card* from the election authorities by postal mail. Each voting card contains different *verification codes* for every voting option and a single *finalization code*. These codes are different for every voting card. An example of such a voting card is shown in Figure 2.1. As we will discuss below, the voting card also contains two authentication codes, which the voter must enter during vote casting. Note that the length of all codes must be chosen carefully to meet the system's security requirements (see Section 6.3.1).

Voting Card Nr. 3587

	Yes	No	Blank
Question 1: Etiam dictum sem pulvinar elit con vallis vehicula. Duis vitae purus ac tortor volut pat iaculis at sed mauris at tempor quam?	A34C	18F5	76BC
Question 2: Donec at consectetur ex. Quisque fermentum ipsum sed est pharetra molestie. Sed at nisl malesuada ex mollis consequat?	91F3	71BD	034A
Question 3: Mauris rutrum tellus et lorem vehicula, quis ornare tortor vestibulum. In tempor, quam sit amet sodales sagittis, nib quam placerat?	774C	CB4A	76F2

Voting code:	**Confirmation code**:	**Finalization code**:
eZ54-gr4B-3pAQ-Zh8q	uw4M-QL91-jZ9N-nXA2	87483172

Figure 2.1.: Example of a voting card for an election event consisting of three referendums. Verification codes are printed as 4-digit numbers in hexadecimal notation, whereas the finalization code is printed as an 8-digit decimal number. The two authentication codes are printed as alphanumeric strings.

After submitting the ballot, verification codes for the chosen voting options are displayed by the voting application and voters are instructed to check if the displayed codes match with the codes printed on the voting card. Matching codes imply with high probability that a correct ballot has been submitted. This step—called *cast-as-intended verification*—is the proposed counter-measure against integrity attacks by malware on the voter's insecure platform, but it obviously does not prevent privacy attacks. Nevertheless, as long as integrity attacks by malware are detectable with probability higher than 99.9%, the Swiss Federal Chancellery has approved this approach as a sufficient solution for conducting elections over the Internet [5, Paragraph 4.2.4]. To provide a guideline to system designers, a description of an example voting procedure based on verification codes is given in [3, Appendix 7]. The procedure proposed in this document follows the given guideline to a considerable degree.

In addition to the verification and finalization codes, voter's are also supplied with two authentication codes called *voting code* and *confirmation code*. In the context of this document, we consider the case where authentication, verification, and finalization codes are all printed on the same voting card, but we do not rule out the possibility that some codes are printed on a separate paper. In addition to these codes, a voting card has a unique identifier. If N_E denotes the size of the electorate, the unique voting card identifier will simply be an integer $i \in \{1, \ldots, N_E\}$, the same number that we will use to identify voters in the electorate (see Section 6.1).

In the Swiss context, since any form of vote updating is prohibited by election laws, voters cannot re-submit the ballot from a different platform in case of non-matching verification codes. From the voter's perspective, the voting process is therefore an *all-or-nothing* procedure, which terminates with either a successfully submitted valid vote (success case) or an abort (failure case). The procedure in the success case consists of five steps:

1. The voter selects the allowed number of voting options and enters the voting code.
2. The voting system[1] checks the voting code and returns the verification codes of the selected voting options for inspection.
3. The voter checks the correctness of the verification codes and enters the confirmation code.
4. The voting system checks the confirmation code and returns the finalization code for inspection.
5. The voter checks the correctness of the finalization code.

From the perspective of the voting system, votes are accepted after receiving the voter's confirmation in Step 4. From the voter's perspective, vote casting was successful after receiving correct verification codes in Step 3 and a correct finalization code in Step 5. In case of an incorrect or missing finalization code, the voter is instructed to trigger an investigation by contacting the election hotline. In any other failure case, voters are instructed to abort the process immediately and use postal mail as a backup voting channel.

2.2. Election Uses Cases

The voting protocol presented in this document is designed to support election events consisting of $t \geqslant 1$ simultaneous elections. Every election $j \in \{1, \dots, t\}$ is modeled as an independent k_j-out-of-n_j election with $n_j \geqslant 2$ candidates, of which (exactly) $0 < k_j < n_j$ can be selected by the voters. Note that we use *candidate* as a general term for all types of voting options, in a similar way as using *election* for various types of elections and referendums. Over all t elections, $n = \sum_{j=1}^{t} n_j$ denotes the total number of candidates, whereas $k = \sum_{j=1}^{t} k_j$ denotes the total number of candidates for voters to select, provided that they are eligible in every election. A single selected candidate is denoted by a value $s \in \{1, \dots, n\}$.

As stated earlier, we also have to take into account that voters may not be eligible in all t elections of an election event. If N_E denotes the size of the electorate, we set $e_{ij} = 1$ if voter $i \in \{1, \dots, N_E\}$ is eligible in election $j \in \{1, \dots, t\}$ and $e_{ij} = 0$ otherwise. These values define the *eligibility matrix* (an N_E-by-t Boolean matrix satisfying $\sum_{i=1}^{N_E} e_{ij} > 0$ and $\sum_{j=1}^{t} e_{ij} > 0$), which must be specified prior to every election event by the election administrator. For voter i, the product $k'_{ij} = e_{ij} k_j \in \{0, k_j\}$ denotes the number of allowed selections in election j, and $k'_i = \sum_{j=1}^{t} k'_{ij}$ denotes the total number of selections over all t elections of the given election event. In Section 6.3.2, this general model of an election event will be discussed in further detail.

2.2.1. Electorate

In the political system in Switzerland, all votes submitted in an election event are tallied in so-called *counting circles*. In smaller municipalities, the counting circle is identical to the municipality itself, but larger cities may consist of multiple counting circles. For statistical

[1] Here we use *voting system* as a general term for all server-sider parties involved in the election phase of the protocol.

reasons, the results of each counting circle must be published separately for elections on all four political levels, i.e., the final election results on federal, cantonal, communal, or pastoral issues are obtained by summing of the results of all involved counting circles. Counting circles will typically consist of several hundred or several thousand eligible voters. Even in the largest counting circle, we expect not more than 100'000 voters.

To comply with this setting, every submitted ballot will need to be assigned to a counting circle. Let $w \geqslant 1$ denote the total number of counting circles in an election event, and $w_i \in \{1, \ldots, w\}$ the counting circle of voter $i \in \{1, \ldots, N_E\}$, i.e., w_i is the number that needs to be attached to a ballot submitted by voter i. By including the information about each voter's counting circle and eligibility into the protocol specification, a single protocol instance will be sufficient to run all sorts of mixed election events on the level of the cantons, which by law are in charge of organizing and conducting elections in Switzerland. Regarding the number of counting circles in a canton, we expect an upper bound of $w \leqslant 380$. As we will see in Section 8.3.2, we limit the total number of candidates in an election event to $n \leqslant 1678$, which should be sufficient to cover all practically relevant combinations of simultaneous elections on all four political levels and for all municipalities of a given canton. Running a single protocol instance with exactly the same election parameters is also a desirable property form an organizational point of view, since it greatly facilitates the system setup in such a canton.

2.2.2. Type of Elections

In the elections that we consider voters must always select exactly k different candidates from a list of n candidates. At first glance, such k-out-of-n elections may seems too restrictive to cover all necessary election use cases in the given context, but they are actually flexible enough to support more general election types, for example elections with the option of submitting blank votes. In general, it is possible to substitute any $(k_{\min}, k_{\max})$-out-of-n election, in which voters are allowed to select between $k_{\min}$ and $k_{\max}$ different candidates from the candidate list, by an equivalent k'-out-of-n' election for $k' = k_{\max}$ and $n' = n + b$, where $b = k_{\max} - k_{\min}$ denotes the number of additional *blank candidates*. An important special case of this augmented setting arises for $k_{\min} = 0$, in which a completely blank ballot is possible by selecting all $b = k_{\max}$ blank candidates.

In another generalization of basic k-out-of-n elections, voters are allowed to give up to $c \leqslant k$ votes to the same candidate. This is called *cumulation*. In the most flexible case of cumulation, the k votes can be distributed among the n candidates in an arbitrary manner. This case can be handled by increasing the size of the candidate list from n to $n' = cn$, i.e., each candidate obtains c distinct entries in the extended candidate list. This leads to an equivalent k-out-of-n' election, in which voters may select the same candidate up to c times by selecting all its entries in the extended list. At the end of the election, an additional accumulation step is necessary to determine the exact number of votes of a given candidate from the final tally. By combining this technique of handling cumulations with the above way of handling blank votes, we obtain k'-out-of-n' elections with $k' = k_{\max}$ and $n' = cn + b$.

In Table 2.1 we give a non-exhaustive list of some common election types with corresponding election parameters to handle blank votes and cumulations as explained above. In this list, we assume that blank votes are always allowed up to the maximal possible number. The last entry in the list, which describes the case of party-list elections, is thought to cover elections of the Swiss National Council. This particular election type can be understood as two independent elections in parallel, one 1-out-of-n_p party election and one cumulative k-out-of-n_c candidate election, where n_p and n_c denote the number of parties and candidates, respectively. Cumulation is usally restricted to maximal $c = 2$ voter per candidate. Blank votes are allowed for both the party and the candidate election. In some cases, a completely blank candidate ballot is prohibited together with a party vote. This particular case can be covered by reducing the number of blank candidates from $b = k$ to $b = k - 1$ and by introducing two *blank parties* instead of one, one for a blank party vote with at least one non-blank candidate vote and one for an entirely blank vote. In the latter case, candidate votes are discarded inthe final tally.

Election Type	k	n	b	c	k'	n'
Referendum, popular initiative, direct counter-proposal	1	2	1	1	1	3
Deciding question	1	2	1	1	1	3
Single non-transferable vote	1	n	1	1	1	$n+1$
Multiple non-transferable vote	k	n	k	1	k	$n+k$
Approval voting	n	n	n	1	n	$2n$
Cumulative voting	k	n	k	c	k	$cn+k$
Party-list election	$(1,k)$	(n_p,n_c)	$(1,k)$	$(1,2)$	$(1,k)$	$(n_p+1,2n_c+k)$

Table 2.1.: Election parameters for common types of elections. Party-list elections (last line) are modeled as two independent elections in parallel, one for the parties and one for the candidates.

Even in the largest possible use case in the context of elections in Switzerland, we expect k' to be less than 100 and n' to be less than 1000 for a single election. Since multiple complex elections are rarely combined in a single election event, we expect the accumulations of these values over all elections to be less than 150 for $k' = \sum_{j=1}^{t} k'_j$ and less than 1500 for $n' = \sum_{j=1}^{t} n'_j$. This estimation of the largest possible list of candidates is consistent with the supported number of candidates $n_{\max} = 1678$ (see Section 8.3.2).

Part II.

Theoretical Background

3. Mathematical Preliminaries

3.1. Notational Conventions

As a general rule, we use upper-case latin or greek letters for sets and lower-case latin or greek letters for their elements, for example $X = \{x_1, \ldots, x_n\}$. For composed sets or subsets of composed sets, we use calligraphic upper-case latin letters, for example $\mathcal{X} \subseteq X \times Y \times Z$ for the set or a subset of triples (x, y, z). $|X|$ denotes the cardinality of a finite set X. For general tuples, we use lower-case latin or greek letters in normal font, for example $t = (x, y, z)$ for triples from $X \times Y \times Z$. For sequences (arrays, lists, strings), we use upper-case latin letters and indices starting from 0, for example $S = \langle s_0, \ldots, s_{n-1} \rangle \in A^*$ for a string of characters $s_i \in A$, where A is a given alphabet. We write $|S| = n$ for the length of S and use standard array notation $S[i] = s_i$ to select the element at index $i \in \{0, \ldots, n-1\}$. $S_1 \parallel S_2$ denotes the concatenation of two sequences. For vectors, we use lower-case latin letters in bold font, for example $\mathbf{x} = (x_1, \ldots, x_n) \in X^n$ for a vector of length $|\mathbf{x}| = n$. For two-dimensional (or higher-dimensional) matrices, we use upper-case latin letters in bold font, for example

$$\mathbf{X} = \begin{pmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \cdots & x_{m,n} \end{pmatrix} \in X^{mn}$$

for an m-by-n matrix of values $x_{ij} \in X$. We use $\mathbf{X} = (x_{ij})_{m \times n} \in X^{mn}$ as a shortcut notation. Similarly, $\mathbf{X} = (x_{ijk})_{m \times n \times r} \in X^{mnr}$ is a shortcut notation for a three-dimensional m-by-n-by-r matrix of values $x_{ijk} \in X$.

The set of integers is denoted by $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, the set of natural numbers by $\mathbb{N} = \{0, 1, 2, \ldots\}$, and the set of positive natural numbers by $\mathbb{N}^+ = \{1, 2, \ldots\}$. The set of the n smallest natural numbers is denoted by $\mathbb{Z}_n = \{0, \ldots, n-1\}$, where $\mathbb{B} = \{0, 1\} = \mathbb{Z}_2$ denotes the special case of the Boolean domain. The set of all prime numbers is denoted by $\mathbb{P}$. A prime number $p = 2q + 1 \in \mathbb{P}$ is called *safe prime*, if $q \in \mathbb{P}$, and the set of all safe primes is denoted by $\mathbb{S}$.

For an integer $x \in \mathbb{Z}$, we write $\mathrm{abs}(x)$ for the absolute value of x and $\|x\| = \lfloor \log_2(\mathrm{abs}(x)) \rfloor + 1$ for the *bit length* of $x \neq 0$ (let $\|0\| = 0$ by definition). The set of all natural numbers of a given bit length $l \geqslant 1$ is denoted by $\mathbb{Z}_{[l]} = \{x \in \mathbb{N} : \|x\| = l\} = \mathbb{Z}_{2^l} \backslash \mathbb{Z}_{2^{l-1}}$ and the cardinality of this set is $|\mathbb{Z}_{[l]}| = 2^{l-1}$. For example, $\mathbb{Z}_{[3]} = \{4, 5, 6, 7\}$ has cardinality $2^{3-1} = 4$. Similarly, we write $\mathbb{P}_{[l]} = \mathbb{P} \cap \mathbb{Z}_{[l]}$ and $\mathbb{S}_{[l]} = \mathbb{S} \cap \mathbb{Z}_{[l]}$ for corresponding sets of prime numbers and safe primes, respectively.

To denote mathematical functions, we generally use one italic or multiple non-italic lower-case latin letters, for example $f(x)$ or $\gcd(x, y)$. For algorithms, we use single or multiple words starting with an upper-case letter in sans-serif font, for example $\mathsf{Euclid}(x, y)$ or

$\mathsf{ExtendedEuclid}(x, y)$. Algorithms can be deterministic or randomized. We use $\leftarrow$ for assigning the return value of an algorithm call to a variable, for example $z \leftarrow \mathsf{Euclid}(x, y)$. Picking a value uniformly at random from a finite set X is denoted by $x \in_R X$.

3.2. Mathematical Groups

In mathematics, a *group* $\mathcal{G} = (G, \circ, \mathrm{inv}, e)$ is an algebraic structure consisting of a set G of elements, a (binary) operation $\circ : G \times G \rightarrow G$, a (unary) operation $\mathrm{inv} : G \rightarrow G$, and a neutral element $e \in G$. The following properties must be satisfied for $\mathcal{G}$ to qualify as a group:

- $x \circ y \in G$ (closure),
- $x \circ (y \circ z) = (x \circ y) \circ z$ (associativity),
- $e \circ x = x \circ e = x$ (identity element),
- $x \circ \mathrm{inv}(x) = e$ (inverse element),

for all $x, y, z \in G$.

Usually, groups are written either additively as $\mathcal{G} = (G, +, -, 0)$ or multiplicatively as $\mathcal{G} = (G, \times, ^{-1}, 1)$, but this is just a matter of convention. We write $k \cdot x$ in an additive group and x^k in a multiplicative group for applying the group operator $k - 1$ times to x. We define $0 \cdot x = 0$ and $x^0 = 1$ and handle negative values as $-k \cdot x = k \cdot (-x) = -(k \cdot x)$ and $x^{-k} = (x^{-1})^k = (x^k)^{-1}$, respectively. A fundamental law of group theory states that if $n = |G|$ is the *group order* of a finite group, then $n \cdot x = 0$ and $x^n = 1$, which implies $k \cdot x = (k \bmod n) \cdot x$ and $x^k = x^{k \bmod n}$. In other words, scalars or exponents such as k can be restricted to elements of the additive group $\mathbb{Z}_n$, in which additions are computed modulo n (see below). Often, the term group is used for both the algebraic structure $\mathcal{G}$ and its set of elements G.

3.2.1. The Multiplicative Group of Integers Modulo p

With $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$ we denote the multiplicative group of integers modulo a prime $p \in \mathbb{P}$, in which multiplications are computed modulo p. The group order is $|\mathbb{Z}_p^*| = p-1$, i.e., operations on the exponents can be computed modulo $p - 1$. An element $g \in \mathbb{Z}_p^*$ is called *generator* of $\mathbb{Z}_p^*$, if $\{g^1, \ldots, g^{p-1}\} = \mathbb{Z}_p^*$. Such generators always exist for $\mathbb{Z}_p^*$ if p is prime. Generally, groups for which generators exist are called *cyclic*.

Let g be a generator of $\mathbb{Z}_p^*$ and $x \in \mathbb{Z}_p^*$ an arbitrary group element. The problem of finding a value k such that $x = g^k$ is believed to be a hard. The value $k = \log_g x$ is called *discrete logarithm* of x to base g and the problem of finding k is called *discrete logarithm problem* (DL). It is widely believed that DL is hard in $\mathbb{Z}_p^*$. A related problem, called *decisional Diffie-Hellman problem* (DDH), consists in distinguishing two triples (g^a, g^b, g^{ab}) and (g^a, g^b, g^c) for random exponents a, b, c. While DDH is known to be easy in $\mathbb{Z}_p^*$, it is believed that DDH is hard in large subgroups of $\mathbb{Z}_p^*$.

A subset $\mathbb{G}_q \subset \mathbb{Z}_p^*$ forms a *subgroup* of $\mathbb{Z}_p^*$, if $(\mathbb{G}_q, \times, ^{-1}, 1)$ satisfies the above properties of a group. An important theorem of group theory states that the order $q = |\mathbb{G}_q|$ of every such subgroup divides the order of $\mathbb{Z}_p^*$, i.e., $q | p-1$. If q is a large prime factor of $p-1$, then it is believed that DL in $\mathbb{G}_q$ is as hard as in $\mathbb{Z}_p^*$. In fact, even DDH seems be hard in a large subgroup $\mathbb{G}_q$, which is not the case in $\mathbb{Z}_p^*$.

A particular case arises when $p = 2q+1 \in \mathbb{S}$ is a safe prime. In this case, $\mathbb{G}_q$ is equivalent to the group of so-called *quadratic residues* modulo p, which we obtain by squaring all elements of $\mathbb{Z}_p^*$. Since q is prime, it follows that every $x \in \mathbb{G}_q \backslash \{1\}$ is a generator of $\mathbb{G}_q$, i.e., generators of $\mathbb{G}_q$ can be found easily by squaring arbitrary elements of $\mathbb{Z}_p^* \backslash \{1, p-1\}$.

3.2.2. The Field of Integers Modulo p

With $\mathbb{Z}_n = \{0, \dots, n-1\}$ we denote the additive group of integers, in which additions are computed modulo n. This group as such is not interesting for cryptographic purposes (no hard problems are known), but for $n = p-1$, it serves as the natural additive group when working with exponents in applications of $\mathbb{Z}_p^*$. The same holds for groups of prime order q, for example for subgroups $\mathbb{G}_q \subset \mathbb{Z}_p^*$. In this case, all calculations in the exponent take place in $\mathbb{Z}_q$.

Generally, when $\mathbb{Z}_p$ is an additive group modulo a prime $p \in \mathbb{P}$, then $(\mathbb{Z}_p, +, \times, -, ^{-1}, 0, 1)$ is a *prime-order field* with two binary operations $+$ and $\times$. This particular field combines the additive group $(\mathbb{Z}_p, +, -, 0)$ and the multiplicative group $(\mathbb{Z}_p^*, \times, ^{-1}, 1)$ in one algebraic structure with an additional property:

- $x \times (y+z) = (x \times y) + (x \times z)$, for all $x, y, z \in \mathbb{Z}_p$ (distributivity of multiplication over addition).

For a given prime-order field $\mathbb{Z}_p$, it is possible to define univariate polynomials

$$A(X) = \sum_{i=0}^{d} a_i X^i \in \mathbb{Z}_p[X]$$

of degree $d \geqslant 0$ and with coefficients $a_i \in \mathbb{Z}_p$ (degree d means $a_d \neq 0$). Clearly, such polynomials are fully determined by the list $\mathbf{a} = (a_0, \dots, a_d)$ of all coefficients. Another representation results from picking distinct points $p_i = (x_i, y_i)$, $y_i = A(x_i)$, from the polynomial. Using Lagrange's interpolation method, the coefficients can then be reconstructed if at least $d+1$ such points are available. Reconstructing the coefficient $a_0 = A(0)$ is of particular interest in many applications. For given points $\mathbf{p} = (p_1, \dots, p_d)$, $p_i \in (x_i, y_i) \in \mathbb{Z}_p^2$, we obtain

$$a_0 = \sum_{i=0}^{d} y_i \cdot \Big[\prod_{\substack{0 \leqslant j \leqslant d \\ j \neq i}} \frac{x_j}{x_j - x_i} \Big].$$

by applying Lagrange's general method to $X = 0$.

4. Type Conversion and Hash Algorithms

4.1. Byte Arrays

Let $B = \langle b_0, \ldots, b_{n-1} \rangle$ denote an array of bytes $b_i \in \mathcal{B}$, where $\mathcal{B} = \mathbb{B}^8$ denotes the set of all 256 bytes. We identify individual bytes as integers $b_i \in \mathbb{Z}_{256}$ and use hexadecimal or binary notation to denote them. For example, $B = \langle \texttt{0A}, \texttt{23}, \texttt{EF} \rangle$ denotes a byte array containing three bytes $B[0] = \texttt{0x0A} = \texttt{00001010}_2$, $B[1] = \texttt{0x23} = \texttt{001000011}_2$, and $B[2] = \texttt{0xEF} = \texttt{11101111}_2$.

For two byte arrays B_1 and B_2 of equal length $n = |B_1| = |B_2|$, we write $B_1 \oplus B_2$ for the results of applying the XOR operator $\oplus$ bit-wise to B_1 and B_2. For truncating a byte array B of length $n = |B|$ to the first $m \leqslant n$ bytes, and for skipping the first m bytes from B, we write

$$\begin{aligned} \mathsf{Truncate}(B, m) &= \langle B[0], \ldots, B[m-1] \rangle, \\ \mathsf{Skip}(B, m) &= \langle B[m], \ldots, B[n-1] \rangle, \end{aligned}$$

respectively. Clearly, $B = \mathsf{Truncate}(B, m) \,\|\, \mathsf{Skip}(B, m)$ holds for all $B \in \mathcal{B}^*$ and all $0 \leqslant m \leqslant n$.

Another basic byte array operation is needed for generating unique verification codes on every voting card (see Section 6.3.1 and Algs. 7.13 and 7.28). The goal of this operation is similar to a digital watermark, which we use here for making verification codes unique on each voting card. Below we define an algorithm $\mathsf{MarkByteArray}(B, m, m_{\max})$, which adds an integer watermark m, $0 \leqslant m \leqslant m_{\max}$, to the bits of a byte array B.

Algorithm: $\mathsf{MarkByteArray}(B, m, m_{\max})$
Input: Byte arrays $B \in \mathcal{B}^*$
Watermark m, $0 \leqslant m \leqslant m_{\max}$
Maximal watermark $m_{\max}$, $\|m_{\max}\| \leqslant 8 \cdot |B|$
$l \leftarrow \|m_{\max}\|$
$s \leftarrow \frac{8 \cdot |B|}{l}$
for $i = 0, \ldots, l-1$ **do**
  $B \leftarrow \mathsf{SetBit}(B, \lfloor i \cdot s \rfloor, m \bmod 2)$ // see Alg. 4.2
  $m \leftarrow \lfloor m/2 \rfloor$
return B // $B \in \mathcal{B}^*$

Algorithm 4.1: Adds an integer watermark m to the bits of a given byte array. The bits of the watermark are spread equally across the bits of the byte array.

Algorithm: $\mathsf{SetBit}(B, i, b)$
Input: ByteArray $B \in \mathcal{B}^*$
Index i, $0 \leqslant i < 8 \cdot |B|$
Bit $b \in \mathbb{B}$
$j \leftarrow \lfloor i/8 \rfloor$
$x \leftarrow 2^{i \bmod 8}$
if $b = 0$ **then**
  $B[j] \leftarrow B[j] \wedge (255 - x)$ // $\wedge$ denotes the bitwise AND operator
else
  $B[j] \leftarrow B[j] \vee x$ // $\vee$ denotes the bitwise OR operator
return B // $B \in \mathcal{B}^*$

Algorithm 4.2: Sets the i-th bit of a byte array B to $b \in \mathbb{B}$.

4.1.1. Converting Integers to Byte Arrays

Let $x \in \mathbb{N}$ be a non-negative integer. We use $B \leftarrow \mathsf{ToByteArray}(x, n)$ to denote the algorithm which returns the byte array $B \in \mathcal{B}^n$ obtained from truncating the $n \geqslant \frac{\|x\|}{8}$ least significant bytes from the (infinitely long) binary representation of x in big-endian order:

$$B = \langle b_0, \ldots, b_{n-1} \rangle, \text{ where } b_i = \left\lfloor \frac{x}{256^{n-i-1}} \right\rfloor \bmod 256.$$

We use $\mathsf{ToByteArray}(x)$ as a short-cut notation for $\mathsf{ToByteArray}(x, n_{min})$, which returns the shortest possible such byte array representation of length $n_{min} = \lceil \frac{\|x\|}{8} \rceil$. Table 4.1 shows the byte array representations for different integers x and $n \leqslant 4$.

	$\mathsf{ToByteArray}(x,n)$						
x	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	n_{min}	$\mathsf{ToByteArray}(x)$
0	$\langle\rangle$	$\langle 00 \rangle$	$\langle 00,00 \rangle$	$\langle 00,00,00 \rangle$	$\langle 00,00,00,00 \rangle$	0	$\langle\rangle$
1	–	$\langle 01 \rangle$	$\langle 00,01 \rangle$	$\langle 00,00,01 \rangle$	$\langle 00,00,00,01 \rangle$	1	$\langle 01 \rangle$
255	–	$\langle \mathrm{FF} \rangle$	$\langle 00,\mathrm{FF} \rangle$	$\langle 00,00,\mathrm{FF} \rangle$	$\langle 00,00,00,\mathrm{FF} \rangle$	1	$\langle \mathrm{FF} \rangle$
256	–	–	$\langle 01,00 \rangle$	$\langle 00,01,00 \rangle$	$\langle 00,00,01,00 \rangle$	2	$\langle 01,00 \rangle$
65,535	–	–	$\langle \mathrm{FF},\mathrm{FF} \rangle$	$\langle 00,\mathrm{FF},\mathrm{FF} \rangle$	$\langle 00,00,\mathrm{FF},\mathrm{FF} \rangle$	2	$\langle \mathrm{FF},\mathrm{FF} \rangle$
65,536	–	–	–	$\langle 01,00,00 \rangle$	$\langle 00,01,00,00 \rangle$	3	$\langle 01,00,00 \rangle$
16,777,215	–	–	–	$\langle \mathrm{FF},\mathrm{FF},\mathrm{FF} \rangle$	$\langle 00,\mathrm{FF},\mathrm{FF},\mathrm{FF} \rangle$	3	$\langle \mathrm{FF},\mathrm{FF},\mathrm{FF} \rangle$
16,777,216	–	–	–	–	$\langle 01,00,00,00 \rangle$	4	$\langle 01,00,00,00 \rangle$

Table 4.1.: Byte array representation for different integers and different output lengths.

The shortest byte array representation in big-endian byte order, $B \leftarrow \mathsf{ToByteArray}(x)$, is the default byte array representation of non-negative integers considered in this document. It will be used for computing cryptographic hash values for integer inputs (see Section 4.3).

4.1.2. Converting Byte Arrays to Integers

Since $\mathsf{ToByteArray}(x)$ from the previous subsection is not bijective relative to $\mathcal{B}^*$, it does not define a unique way of converting an arbitrary byte array $B \in \mathcal{B}^*$ into an integer $x \in \mathbb{N}$.

Algorithm: $\mathsf{ToByteArray}(x)$
Input: Non-negative integer $x \in \mathbb{N}$
$n_{min} \leftarrow \lceil \frac{\|x\|}{8} \rceil$
$B \leftarrow \mathsf{ToByteArray}(x, n_{min})$ // see Alg. 4.4
return B // $B \in \mathcal{B}^*$

Algorithm 4.3: Computes the shortest byte array representation in big-endian byte order of a given non-negative integer $x \in \mathbb{N}$.

Algorithm: $\mathsf{ToByteArray}(x, n)$
Input: Non-negative integer $x \in \mathbb{N}$
Length of byte array $n \geqslant \frac{\|x\|}{8}$
for $i = 1, \ldots, n$ **do**
    $b_{n-i} \leftarrow x \bmod 256$
    $x \leftarrow \lfloor \frac{x}{256} \rfloor$
$B \leftarrow \langle b_0, \ldots, b_{n-1} \rangle$
return B // $B \in \mathcal{B}^n$

Algorithm 4.4: Computes the byte array representation in big-endian byte order of a given non-negative integer $x \in \mathbb{N}$. The given length $n \geqslant \frac{\|x\|}{8}$ of the output byte array B implies that the first $n - \lceil \frac{\|x\|}{8} \rceil$ bytes of B are zeros.

Defining such a conversion depends on whether the conversion needs to be injective or not. In this document, we only need the following non-injective conversion,

$$x = \sum_{i=0}^{n-1} B[i] \cdot 256^{n-i-1}, \text{ for } n = |B|,$$

in which leading zeros are ignored. With $x \leftarrow \mathsf{ToInteger}(B)$ we denote a call to an algorithm, which computes this conversion for all $B \in \mathcal{B}^*$. It will be used in non-interactive zero-knowledge proofs to generate integer challenges from Fiat-Shamir hash values (see Alg. 7.4 and Alg. 7.5). Note that $x \leftarrow \mathsf{ToInteger}(\mathsf{ToByteArray}(x))$ holds for all $x \in \mathbb{N}$, but $B \leftarrow \mathsf{ToByteArray}(\mathsf{ToInteger}(B))$ only holds for byte arrays without any leading zeros (i.e., only when $B[0] \neq 0$). One the other hand, $B \leftarrow \mathsf{ToByteArray}(\mathsf{ToInteger}(B), n)$ holds for all byte arrays $B \in \mathcal{B}^n$ of length n.

4.1.3. Converting UCS Strings to Byte Arrays

Let A_{ucs} denote the *Universal Character Set* (UCS) as defined by ISO/IEC 10646, which contains about $128,000$ abstract characters. A sequence $S = \langle s_0, \ldots, s_{n-1} \rangle \in A_{\mathsf{ucs}}^*$ of characters $s_i \in A_{\mathsf{ucs}}$ is called *UCS string* of length n. A_{ucs}^* denotes the set of all UCS strings, including the empty string. Concrete string instances are written in the usual string notation, for example `""` (empty string), `"x"` (string consisting of a single character `'x'`), or `"Hello"`.

Algorithm: $\mathsf{ToInteger}(B)$
Input: Byte array $B \in \mathcal{B}^*$
$x \leftarrow 0$
for $i = 0, \ldots, |B| - 1$ **do**
  $x \leftarrow 256 \cdot x + B[i]$
return x // $x \in \mathbb{N}$

Algorithm 4.5: Computes a non-negative integer from a given byte array B. Leading zeros of B are ignored.

To encode a string $S \in A^*_{\mathsf{ucs}}$ as byte array, we use the UTF-8 character encoding as defined in ISO/IEC 10646 (Annex D). Let $B \leftarrow \mathtt{UTF8}(S)$ denote an algorithm that computes corresponding byte arrays $B \in \mathcal{B}^*$, in which characters use 1, 2, 3, or 4 bytes of space depending on the type of character. For example, $\langle \mathtt{48}, \mathtt{65}, \mathtt{6C}, \mathtt{6C}, \mathtt{6F} \rangle \leftarrow$ `UTF8("Hello")` is a byte array of length 5, because it only consists of Basic Latin characters, whereas $\langle \mathtt{56}, \mathtt{6F}, \mathtt{69}, \mathtt{6C}, \mathtt{C3}, \mathtt{A0} \rangle \leftarrow$ `UTF8("Voilà")` 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 for general UCS strings. It will be used for computing cryptographic hash values of given input strings (see Section 4.3). Since implementations of UTF-8 character encoding are widely available, we do not provide an explicit pseudo-code algorithm.

4.2. Strings

Let $A = \{c_1, \ldots, c_N\}$ be an alphabet of size $N \geqslant 2$. The characters in A are totally ordered, let's say as $c_1 \prec \cdots \prec c_N$, which we express by defining a ranking function $rank_A(c_i) = i - 1$ together with its inverse $rank_A^{-1}(i) = c_{i+1}$. A string $S \in A^*$ is a sequence $S = \langle s_0, \ldots, s_{k-1} \rangle$ of characters $s_i \in A$.

4.2.1. Converting Integers to Strings

Let $x \in \mathbb{N}$ be a non-negative integer. We use $S \leftarrow \mathsf{ToString}(x, k, A)$ to denote an algorithm that returns the following string of length $k \geqslant \log_N x$ in big-endian order:

$$S = \langle s_0, \ldots, s_{k-1} \rangle, \text{ where } s_i = rank_A^{-1}\left(\left\lfloor \frac{x}{N^{k-i-1}} \right\rfloor \bmod N\right).$$

We will use this conversion in Alg. 7.13 to print long integers in a more compact form. Note that the following algorithm is almost identical to Alg. 4.4 given in Section 4.1.1 to obtain byte arrays from integers.

4.2.2. Converting Strings to Integers

In Algs. 7.18 and 7.30, string representations $S \leftarrow \mathsf{ToString}(x, k, A)$ of length k must be reconverted into their original integers $x \in \mathbb{N}$. In a similar way as in Section 4.1.2, we obtain

Algorithm: $\mathsf{ToString}(x, k, A)$
Input: Integer $x \in \mathbb{N}$
String length $k \geqslant \log_N x$
Alphabet $A = \{c_1, \dots, c_N\}$
for $i = 1, \dots, k$ **do**
  $s_{k-i} \leftarrow \mathit{rank}_A^{-1}(x \bmod N)$
  $x \leftarrow \lfloor \frac{x}{N} \rfloor$
$S \leftarrow \langle s_0, \dots, s_{k-1} \rangle$
return S // $S \in A^k$

Algorithm 4.6: Computes a string representation of length k in big-endian order of a given non-negative integer $x \in \mathbb{N}$ and relative to some alphabet A.

the inverse of $\mathsf{ToString}(x, k, A)$ by

$$x = \sum_{i=0}^{k-1} \mathit{rank}_A(S[i]) \cdot N^{k-i-1} < N^k,$$

in which leading characters with rank 0 are ignored. The following algorithm is an adaptation of Alg. 4.5.

Algorithm: $\mathsf{ToInteger}(S, A)$
Input: String $S \in A^*$
Alphabet $A = \{c_1, \dots, c_N\}$
$x \leftarrow 0$
for $i = 0, \dots, |S| - 1$ **do**
  $x \leftarrow N \cdot x + \mathit{rank}_A(S[i])$
return x // $x \in \mathbb{N}$

Algorithm 4.7: Computes a non-negative integer from a given string S.

4.2.3. Converting Byte Arrays to Strings

Let $B \in \mathcal{B}^n$ be a byte array of length n. The goal is to represent B by a unique string $S \in A^k$ of length k, such that k is as small as possible. We will use this conversion in Algs. 7.13, 7.28 and 7.36 to print and display byte arrays in human-readable form. Since there are $|\mathcal{B}^n| = 256^n = 2^{8n}$ byte arrays of length n and $|A^k| = N^k$ strings of length k, we derive $k = \lceil \frac{8n}{\log_2 N} \rceil$ from the inequality $2^{8n} \leqslant N^k$. To obtain an optimal string representation of B, let $x_B \leftarrow \mathsf{ToInteger}(B) < 2^{8n}$ be the representation of B as a non-negative integer. This leads to the following length-optimal mapping from $\mathcal{B}^n$ to A^k.

Algorithm: $\mathsf{ToString}(B, A)$
Input: Byte array $B \in \mathcal{B}^n$
Alphabet $A = \{c_1, \ldots, c_N\}$
$x_B \leftarrow \mathsf{ToInteger}(B)$ // see Alg. 4.5
$k \leftarrow \left\lceil \frac{8n}{\log_2 N} \right\rceil$
$S \leftarrow \mathsf{ToString}(x_B, k, A)$ // see Alg. 4.6
return S // $S \in A^*$

Algorithm 4.8: Computes the shortest string representation of a given byte array B relative to some alphabet A.

4.3. Hash Algorithms

A cryptographic hash algorithm defines a mapping $h : \mathbb{B}^* \rightarrow \mathbb{B}^\ell$, which transforms an input bit array $B \in \mathbb{B}^*$ of arbitrary length into an output bit array $h(B) \in \mathbb{B}^\ell$ of length ℓ, called the *hash value* of B. In practice, hash algorithms such as SHA-1 or SHA-256 operate on byte arrays rather than bit arrays, which implies that the length of the input and output bit arrays is a multiple of 8. We denote such practical algorithms by $H \leftarrow \mathsf{Hash}_L(B)$, where $B \in \mathcal{B}^*$ and $H \in \mathcal{B}^L$ are byte arrays of length $L = \frac{\ell}{8}$. Throughout this document, we do not specify which of the available practical hash algorithms that is compatible with the output bit length ℓ is used. For this we refer to the technical specification in Chapter 8.

4.3.1. Hash Values of Integers and Strings

To compute the hash value of a non-negative integer $x \in \mathbb{N}$, it is first encoded as a byte array $B \leftarrow \mathsf{ToByteArray}(x)$ using Alg. 4.3 and then hashed into $\mathsf{Hash}_L(B)$. The whole process defines a mapping $h : \mathbb{N} \rightarrow \mathcal{B}^L$. Similarly, for an input string $S \in A^*_{\mathsf{ucs}}$, we compute the hash value $\mathsf{Hash}_L(B)$ of the byte array $B \leftarrow \mathsf{UTF8}(S)$ using UTF-8 character encoding (see Section 4.1.3). In this case, we obtain a mapping $h : A^*_{\mathsf{ucs}} \rightarrow \mathcal{B}^L$. Both cases are included as special cases in Alg. 4.9.

4.3.2. Hash Values of Multiple Inputs

Let $\mathbf{b} = (B_1, \ldots, B_k)$ be a vector of multiple input byte arrays $B_i \in \mathcal{B}^*$ of arbitrary length. The hash value of $\mathbf{b}$ can be defined recursively by

$$h(\mathbf{b}) = \begin{cases} h(\langle\rangle), \text{ if } k = 0, \\ h(B_1), \text{ if } k = 1, \\ h(h(B_1) \,\|\, \cdots \,\|\, h(B_k)), \text{ if } k > 1. \end{cases}$$

We distinguish the special case of $k = 1$ to avoid computing $h(h(B_1))$ for a single input and to be able to use $h(B_1, \ldots, B_k)$ as a consistent alternative notation for $h(\mathbf{b})$.

This definition can be generalized to multiple input values of various types. Let $(v_1, \ldots, v_k)$ be such a tuple of general input values, where v_i is either a byte array, an integer, a string, or another tuple of general input values. As above, we define the hash value recursively as

$$h(v_1, \ldots, v_k) = \begin{cases} h(\langle\rangle), \text{ if } k = 0, \\ h(v_1), \text{ if } k = 1, \\ h(h(v_1) \parallel \cdots \parallel h(v_k)), \text{ if } k > 1. \end{cases}$$

Note that an arbitrary tree containing byte arrays, integers, or strings in its leaves can be hashed in this way. Calling such a general hash algorithm is denoted by

$$H \leftarrow \mathsf{RecHash}_L(v_1, \ldots, v_k),$$

where subscript L indicates that the algorithm is instantiated with a cryptographic hash algorithm of output length L. The details of the recursion are given in Alg. 4.9. Note that the special case $k = 0$ is included in the general case $k \neq 1$, in which the empty byte array is assigned to B. Alg. 4.9 also specifies a row-wise recursion for hashing two-dimensional matrix.

Algorithm: $\mathsf{RecHash}_L(v_1, \ldots, v_k)$
Input: Input values $v_i \in V_i$, V_i unspecified, $k \geqslant 0$
if $k = 1$ **then**
  $w \leftarrow v_1$
  if $w \in \mathcal{B}^*$ **then**
    return $\mathsf{Hash}_L(w)$
  if $w \in \mathbb{N}$ **then**
    return $\mathsf{Hash}_L(\mathsf{ToByteArray}(w))$ // see Alg. 4.3
  if $w \in A^*_{\mathsf{ucs}}$ **then**
    return $\mathsf{Hash}_L(\mathsf{UTF8}(w))$ // see Section 4.1.3
  if $w = (w_1, \ldots, w_n)$ **then**
    return $\mathsf{RecHash}_L(w_1, \ldots, w_n)$
  if $w = (w_{ij})_{n \times m}$ **then**
    for $i = 1, \ldots, n$ **do**
      $\mathbf{w}_i \leftarrow (w_{i,1}, \ldots, w_{i,m})$
    return $\mathsf{RecHash}_L(\mathbf{w}_1, \ldots, \mathbf{w}_n)$
  return $\perp$ // type of w not supported
else
  $B \leftarrow \parallel_{i=1}^{k} \mathsf{RecHash}_L(v_i)$
  return $\mathsf{Hash}_L(B)$

Algorithm 4.9: Computes the hash value $h(v_1, \ldots, v_k) \in \mathcal{B}^L$ of multiple inputs $v_1, \ldots, v_k$ in a recursive manner.

5. Cryptographic Primitives

5.1. ElGamal Encryption

An *ElGamal encryption scheme* is a triple $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ of algorithms, which operate on a cyclic group for which the DDH problem is believed to be hard [21]. The most common choice for such a group is the subgroup of quadratic residues $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of prime order q, where $p = 2q + 1$ is a *safe prime* large enough to resist index calculus and other methods for solving the discrete logarithm problem. The public parameters of an ElGamal encryption scheme are thus p, q, and a generator $g \in \mathbb{G}_q \backslash \{1\}$.

5.1.1. Using a Single Key Pair

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

$$\mathsf{Enc}_{pk}(m, r) = (m \cdot pk^r, g^r) \in \mathbb{G}_q \times \mathbb{G}_q$$

denotes the ElGamal encryption of m with randomization $r \in_R \mathbb{Z}_q$. Note that the bit length of an encryption $e \leftarrow \mathsf{Enc}_{pk}(m, r)$ is twice the bit length of p. For a given encryption $e = (a, b)$, the plaintext m can be recovered by using the private decryption key sk to compute

$$m \leftarrow \mathsf{Dec}_{sk}(e) = a \cdot b^{-sk}.$$

For any given key pair $(sk, pk) \leftarrow \mathsf{KeyGen}()$, it is easy to show that $\mathsf{Dec}_{sk}(\mathsf{Enc}_{pk}(m, r)) = m$ holds for all $m \in \mathbb{G}_q$ and $r \in \mathbb{Z}_q$.

The ElGamal encryption scheme is provably IND-CPA secure under the DDH assumption and homomorphic with respect to multiplication. Therefore, component-wise multiplication of two ciphertexts yields an encryption of the product of respective plaintexts:

$$\mathsf{Enc}_{pk}(m_1, r_1) \cdot \mathsf{Enc}_{pk}(m_2, r_2) = \mathsf{Enc}_{pk}(m_1 m_2, r_1 + r_2).$$

In a homomorphic encryption scheme like ElGamal, a given encryption $e \leftarrow \mathsf{Enc}_{pk}(m, r)$ can be *re-encrypted* by multiplying e with an encryption of the neutral element 1. The resulting re-encryption,

$$\mathsf{ReEnc}_{pk}(e, r') = e \cdot \mathsf{Enc}_{pk}(1, r') = \mathsf{Enc}_{pk}(m, r + r'),$$

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

5.1.2. Using a Shared Key Pair

If multiple parties generate ElGamal key pairs as described above, let's say $(sk_j, pk_j) \leftarrow \mathsf{KeyGen}()$ for parties $j \in \{1, \ldots, s\}$, then it is possible to aggregate the public encryption keys into a common public key $pk = \prod_{j=1}^{s} pk_j$, which can be used to encrypt messages as described above. The corresponding private keys sk_j can then be regarded as *key shares* of the private key $sk = \sum_{j=1}^{s} sk_j$, which is not known to anyone. This means that an encryption $e = enc_{pk}(m, r)$ can only be decrypted if all parties collaborate. This idea can be generalized such that only a threshold number $t \leqslant s$ of parties is required to decrypt a message, but this property is not needed in this document.

In the setting where s parties hold shares of a common key pair (sk, pk), the decryption of $e \leftarrow \mathsf{Enc}_{pk}(m, r)$ can be conducted without revealing the key shares sk_j:

$$\mathsf{Dec}_{sk}(e) = a \cdot b^{-sk} = a \cdot b^{-\sum_{j=1}^{s} sk_j} = a \cdot (\prod_{j=1}^{s} b^{s_j})^{-1} = a \cdot (\prod_{j=1}^{s} b_j)^{-1},$$

where each *partial decryption* $b_j = b^{sk_j}$ can be computed individually by the respective holder of the key share sk_j.

5.2. Pedersen Commitment

The (extended) *Pedersen commitment scheme* is based on a cyclic group for which the DL problem is believed to be hard. In this document, we use the same q-order subgroup $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of integers modulo $p = 2q + 1$ as in the ElGamal encryption scheme. Let $g, h_1, \ldots, h_n \in \mathbb{G}_q \backslash \{1\}$ be independent generators of $\mathbb{G}_q$, which means that their relative logarithms are provably not known to anyone. For a deterministic algorithm that generates an arbitrary number of independent generators, we refer to the NIST standard FIPS PUB 186-4 [2, Appendix A.2.3]. Note that the deterministic nature of this algorithm enables the verification of the generators by the public.

The Pedersen commitment scheme consists of two deterministic algorithms, one for computing a commitment

$$\mathsf{Com}(\mathbf{m}, r) = g^r h_1^{m_1} \cdots h_n^{m_n} \in \mathbb{G}_q$$

to n messages $\mathbf{m} = (m_1, \ldots, m_n) \in \mathbb{Z}_q^n$ with randomization $r \in_R \mathbb{Z}_q$, and one for checking the validity of $c \leftarrow \mathsf{Com}(\mathbf{m}, r)$ when $\mathbf{m}$ and r are revealed. In the special case of a single message m, we write $\mathsf{Com}(m, r) = g^r h^m$ using a second generator h independent from g. The Pedersen commitment scheme is perfectly hiding and computationally binding under the DL assumption.

In this document, we will also require commitments to permutations $\psi : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$. Let $\mathbf{B}_\psi = (b_{ij})_{n \times n}$ be the *permutation matrix* of ψ, which consists of bits

$$b_{ij} = \begin{cases} 1, & \text{if } \psi(i) = j, \\ 0, & \text{otherwise.} \end{cases}$$

Note that each row and each column in $\mathbf{B}_\psi$ has exactly one 1-bit. If $\mathbf{b}_j = (b_{1,j}, \ldots, b_{n,j})$ denotes the j-th column of $\mathbf{B}_\psi$, then

$$\mathsf{Com}(\mathbf{b}_j, r_j) = g^{r_j} \prod_{i=1}^{n} h_i^{b_{ij}} = g^{r_j} h_i, \text{ for } i = \psi^{-1}(j),$$

is a commitment to $\mathbf{b}_j$ with randomization r_j. By computing such commitments to all columns,

$$\mathsf{Com}(\psi, \mathbf{r}) = (\mathsf{Com}(\mathbf{b}_1, r_1), \ldots, \mathsf{Com}(\mathbf{b}_n, r_n)),$$

we obtain a commitment to ψ with randomizations $\mathbf{r} = (r_1, \ldots, r_n)$. Note that the size of such a *permutation commitment* $\mathbf{c} \leftarrow \mathsf{Com}(\psi, \mathbf{r})$ is $O(n)$.

5.3. Oblivious Transfer

An oblivious transfer results from the execution of a protocol between two parties called *sender* and *receiver*. In a k-out-of-n oblivious transfer, denoted by OT_n^k, the sender holds a list $\mathbf{m} = (M_1, \ldots, M_n)$ of messages $M_i \in \mathbb{B}^\ell$ (bit strings of length ℓ), of which $k \leqslant n$ can be selected by the receiver. The selected messages are transferred to the receiver such that the sender remains oblivious about the receiver's selections and that the receiver remains oblivious about the $n - k$ other messages. We write $\mathbf{s} = (s_1, \ldots, s_k)$ for the k selections $s_j \in \{1, \ldots, n\}$ of the receiver and $\mathbf{m_s} = (M_{s_1}, \ldots, M_{s_k})$ for the k messages to transfer.

In the simplest possible case of a two-round protocol, the receiver sends a randomized query $\alpha \leftarrow \mathsf{Query}(\mathbf{s}, \mathbf{r})$ to the sender, the sender replies with $\beta \leftarrow \mathsf{Reply}(\alpha, \mathbf{m})$, and the receiver obtains $\mathbf{m_s} \leftarrow \mathsf{Open}(\beta, \mathbf{s}, \mathbf{r})$ by removing the randomization $\mathbf{r}$ from β. For the correctness of the protocol, $\mathsf{Open}(\mathsf{Reply}(\mathsf{Query}(\mathbf{s}, \mathbf{r}), \mathbf{m}), \mathbf{s}, \mathbf{r}) = \mathbf{m_s}$ must hold for all possible values of $\mathbf{m}$, $\mathbf{s}$, and $\mathbf{r}$. A triple of algorithms $(\mathsf{Query}, \mathsf{Reply}, \mathsf{Open})$ satisfying this property is called (two-round) OT_n^k*-scheme*.

An OT_n^k-scheme is called *secure*, if the three algorithms guarantee both *receiver privacy* and *sender privacy*. Receiver privacy is defined in terms of indistinguishable selections $\mathbf{s}_1$ and $\mathbf{s}_2$ relative to corresponding queries q_1 and q_2, whereas sender privacy is defined in terms of indistinguishable transcripts obtained from executing the real protocol and a simulation of the ideal protocol in the presence of a malicious receiver. In the ideal protocol, $\mathbf{s}$ and $\mathbf{m}$ are sent to an incorruptible trusted third party, which forwards $\mathbf{m_s}$ to the simulator. In the literature, there is a subtle but important distinction between *sender privacy* and *weak sender privacy* [35]. In the latter case, by selecting out-of-bounds indices, the receiver may still learn up to k messages.

5.3.1. OT-Scheme by Chu and Tzeng

There are many general ways of constructing OT_n^k schemes, for example on the basis of a less complex OT_n^1- or OT_2^1-scheme, but such general constructions are usually not very efficient. In this document, we use the second OT_n^k-scheme presented in [17].[1] We instantiate

[1]The modified protocol as presented in [18] is slightly more efficient, but fits less into the particular context of this document.

the protocol to the same q-order subgroup $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of integers modulo $p = 2q + 1$ as in the ElGamal encryption scheme. Besides the description of this group, there are several public parameters: a generator $g \in \mathbb{G}_q \backslash \{1\}$, an encoding $\Gamma : \{1, \ldots, n\} \to \mathbb{G}_q$ of the possible selections into $\mathbb{G}_q$, and a collision-resistant hash function $h : \mathbb{B}^* \to \mathbb{B}^\ell$ with output length ℓ. In Prot. 5.1, we provide a detailed formal description of the protocol. The query is a vector $\mathbf{a} \in \mathbb{G}_q^k$ of length k and the response is a tuple $(\mathbf{b}, \mathbf{c}, d)$ consisting of a vector $\mathbf{b} \in \mathcal{G}^k$ of length k, a vector $\mathbf{c} \in (\mathbb{B}^\ell)^n$ of length n, and a single value $d \in \mathbb{G}_q$, i.e.,

$$\begin{aligned} \mathbf{a} &\leftarrow \mathsf{Query}(\mathbf{s}, \mathbf{r}), \\ (\mathbf{b}, \mathbf{c}, d) &\leftarrow \mathsf{Reply}(\mathbf{a}, \mathbf{m}, z), \\ \mathbf{m_s} &\leftarrow \mathsf{Open}(\mathbf{b}, \mathbf{c}, d, \mathbf{s}, \mathbf{r}), \end{aligned}$$

where $\mathbf{r} = (r_1, \ldots, r_k) \in_R \mathbb{Z}_q^k$ is the randomization vector used for computing the query and $z \in_R \mathbb{Z}_q$ an additional randomization used for computing the response.

Receiver		Sender
knows $\mathbf{s} = (s_1, \ldots, s_k)$		knows $\mathbf{m} = (M_1, \ldots, M_n)$
for $j = 1, \ldots, k$ – pick random $r_j \in_R \mathbb{Z}_q$ – compute $a_j = \Gamma(s_j) \cdot g^{r_j}$		
	$\xrightarrow{\mathbf{a} = (a_1, \ldots, a_k)}$	
		pick random $z \in_R \mathbb{Z}_q$ for $j = 1, \ldots, k$ – compute $b_j = a_j^z$ for $i = 1, \ldots, n$ – compute $k_i = \Gamma(i)^z$ – compute $C_i = M_i \oplus h(k_i)$ compute $d = g^z$
	$\mathbf{b} = (b_1, \ldots, b_k)$, $\mathbf{c} = (C_1, \ldots, C_n), d$ $\longleftarrow$	
for $j = 1, \ldots, k$ – compute $k_j = b_j \cdot d^{-r_j}$ – compute $M_{s_j} = C_{s_j} \oplus h(k_j)$		

Protocol 5.1: Two-round OT_n^k-scheme with weak sender privacy, where $g \in \mathbb{G}_q \backslash \{1\}$ is a generator of $\mathbb{G}_q \subset \mathbb{Z}_p^*$, $\Gamma : \{1, \ldots, n\} \to \mathbb{G}_q$ an encoding of the selections into $\mathbb{G}_q$, and $h : \mathbb{B}^* \to \mathbb{B}^\ell$ a collision-resistant hash function with output length ℓ.

Executing Query and Open requires k fixed-base exponentiations in $\mathbb{G}_q$ each, whereas Reply requires $n + k + 1$ fixed-exponent exponentiations in $\mathbb{G}_q$. Note that among the $2k$ exponentiations of the receiver, k can be precomputed, and among the $n + k + 1$ exponentiations of the sender, $n+1$ can be precomputed. Therefore, only k online exponentiations remain for both the receiver and the sender, i.e., the protocol is very efficient in terms of computation and communication costs. In the random oracle model, the scheme is provably secure against a malicious receiver and a semi-honest sender. Receiver privacy is unconditional and weak sender privacy is computational under the *chosen-target computational Diffie-Hellman* (CT-CDH) assumption. Note that the CT-CDH assumption is weaker than standard CDH [13].

5.3.2. Full Sender Privacy in the OT-Scheme by Chu and Tzeng

As discussed above, the two major properties of an OT-scheme—receiver privacy and weak sender privacy—are given under reasonable assumptions in Chu and Tzeng's scheme. However, full sender privacy, which guarantees that by submitting $t \leqslant k$ invalid queries $a_j \notin \{\Gamma(i) \cdot g^r : 1 \leqslant i \leqslant n, r \in \mathbb{Z}_q\}$, the receiver learns only up to $k-t$ messages, is not provided. For example, by submitting an invalid query $a_j = \Gamma(s_j)^z g^{r_j}$ for $z > 1$, the scheme by Chu and Tzeng allows the receiver to obtain a correct message $M_{s_j} = C_{s_j} \oplus h((b_i \cdot d^{-r_j})^{-z})$, i.e., Chu and Tzeng's scheme is clearly not fully sender-private. Various similar deviations from the protocol exist for obtaining correct messages. While such deviations are not a problem for many OT applications, they can lead to severe vote integrity attacks in the e-voting application context of this document.[2]

In Prot. 5.2 we present an extension of Chu and Tzeng's scheme that provides full sender privacy. The main difference to the basic scheme is the size of the reply to a query, which consists now of a matrix $\mathbf{C} \in (\mathbb{B}^\ell)^{nk}$ of size nk instead of a vector $\mathbf{c} \in (\mathbb{B}^\ell)^n$ of size n. There are also more random values involved in the computation of the reply. The signatures of the three algorithms are as follows:

$$\begin{aligned} \mathbf{a} &\leftarrow \mathsf{Query}(\mathbf{s}, \mathbf{r}), \\ (\mathbf{b}, \mathbf{C}, d) &\leftarrow \mathsf{Reply}(\mathbf{a}, \mathbf{m}, z_1, z_2, \beta_1, \ldots, \beta_k), \\ \mathbf{m_s} &\leftarrow \mathsf{Open}(\mathbf{b}, \mathbf{C}, d, \mathbf{s}, \mathbf{r}). \end{aligned}$$

Another important difference of the extended scheme is the shape of the queries $a_j = (\Gamma(s_j) \cdot g_1^{r_j}, g_2^{r_j})$, which correspond to ElGamal encryptions for a public key $g_1 = g_2^x$. As a consequence, receiver privacy depends now on the decisional Diffie-Hellman assumption, i.e., it is no longer unconditional. However, the close connection between OT queries and ElGamal encryptions is a key property that we use for submitting ballots (see Section 6.4.2).

The performance of the extended scheme is slightly inferior compared to the basic scheme. On the receiver's side, executing Query requires $2k$ fixed-base exponentiations in $\mathbb{G}_q$ (which can all be precomputed), and Open requires k fixed-base exponentiations in $\mathbb{G}_q$. On the sender's side, Reply requires $n+2k+2$ fixed-exponent exponentiations in $\mathbb{G}_q$ (of which $n+2$ are precomputable). Therefore, k online exponentiations remain for the receiver and $2k$ for the sender. Note that due to the size of th resulting matrix $\mathbf{C}$, the overall asymptotic running time for the sender is $O(nk)$.

[2] The existence of such attacks against the protocol presented in an earlier version of this document have been discovered by Tomasz Truderung [48, Appendix B].

Receiver		Sender
knows $\mathbf{s} = (s_1, \dots, s_k)$		knows $\mathbf{m} = (M_1, \dots, M_n)$
for $j = 1, \dots, k$ – pick random $r_j \in_R \mathbb{Z}_q$ – compute $a_{j,1} = \Gamma(s_j) \cdot g_1^{r_j}$ – compute $a_{j,2} = g_2^{r_j}$ – let $a_j = (a_{j,1}, a_{j,2})$		
	$\xrightarrow{\mathbf{a} = (a_1, \dots, a_k)}$	

		pick random $z_1, z_2 \in_R \mathbb{Z}_q$
		for $j = 1, \dots, k$
		– pick random $\beta_j \in_R \mathbb{G}_q$
		– compute $b_j = a_{j,1}^{z_1} a_{j,2}^{z_2} \beta_j$
		for $i = 1, \dots, n$
		– compute $k_i = \Gamma(i)^{z_1}$
		– for $j = 1, \dots, k$
		  – compute $k_{ij} = k_i \beta_j$
		  – compute $C_{ij} = M_i \oplus h(k_{ij})$
		compute $d = g_1^{z_1} g_2^{z_2}$

	$\xleftarrow{\mathbf{b} = (b_1, \dots, b_k),\ \mathbf{C} = (C_{ij})_{n \times k}, d}$	
for $j = 1, \dots, k$ – compute $k_j = b_j \cdot d^{-r_j}$ – compute $M_{s_j} = C_{s_j,j} \oplus h(k_j)$		

Protocol 5.2: Two-round OT_n^k-scheme with sender privacy receiver, where $g_1, g_2 \in \mathbb{G}_q \backslash \{1\}$ are independent generators of $\mathbb{G}_q \subset \mathbb{Z}_p^*$, $\Gamma : \{1, \dots, n\} \to \mathbb{G}_q$ an encoding of the selections into $\mathbb{G}_q$, and $h : \mathbb{B}^* \to \mathbb{B}^\ell$ a collision-resistant hash function with output length ℓ.

5.3.3. Simultaneous Oblivious Transfers

The OT_n^k-scheme from the previous subsection can be extended to the case of a sender holding multiple lists $\mathbf{m}_l$ of length n_l, from which the receiver selects $k_l \leqslant n_l$ in each case. If t is the total number of such lists, then $n = \sum_{l=1}^{t} n_l$ is the total number of available messages and $k = \sum_{l=1}^{t} k_l$ the total number of selections. A simultaneous oblivious transfer of this kind is denoted by $\mathrm{OT}_{\mathbf{n}}^{\mathbf{k}}$ for vectors $\mathbf{n} = (n_1, \dots, n_t)$ and $\mathbf{k} = (k_1, \dots, k_t)$. It can be realized in two ways, either by conducting t such k_l-out-of-n_l oblivious transfers in parallel, for example using the scheme from the previous subsection, or by conducting a single k-out-of-n oblivious transfer relative to $\mathbf{m} = \mathbf{m}_1 \| \cdots \| \mathbf{m}_t = (M_1, \dots, M_n)$ with some additional constraints relative to the choice of $\mathbf{s} = (s_1, \dots, s_k)$.

To define these constraints, let $k'_l = \sum_{i=1}^{l-1} k_i$ and $n'_l = \sum_{i=1}^{l-1} n_i$ for $1 \leqslant l \leqslant t+1$. This determines for each $j \in \{1, \ldots, k\}$ a unique index $l \in \{1, \ldots, t\}$ satisfying $k'_l < j \leqslant k'_{l+1}$, which we can use to define a constraint

$$n'_l < s_j \leqslant n'_{l+1} \tag{5.1}$$

for every selection s_j in $\mathbf{s}$. This guarantees that the first k_1 messages are selected from $\mathbf{m}_1$, the next k_2 messages from $\mathbf{m}_2$, and so on.

Starting from Prot. 5.2, the sender's algorithm Reply can be generalized in a natural way by introducing an additional outer loop over $1 \leqslant l \leqslant t$ and by iterating the inner loops from $n'_l + 1$ to $n'_l + n_l$ and from $k'_l + 1$ to $k'_l + k_l$, respectively, as shown in Prot. 5.3. Note that the receiver's algorithms Query and Open are not affected by this change. It is easy to demonstrate that this generalization of the OT_n^k-scheme of the previous subsection is equivalent to performing t individual oblivious transfers in parallel. Note that the total number of exponentiations in $\mathbb{G}_q$ remains the same for all three algorithms.

In this extended version of the protocol, the resulting matrix $\mathbf{C} = (C_{ij})_{n \times k}$ of ciphertexts contains only $\sum_{l=1}^{t} k_l n_l$ non-trivial entries, which can be considerably less than its full size kn. As an example, consider the case of $t = 3$ simultaneous oblivious transfers with $\mathbf{k} = (2, 3, 1)$ and $\mathbf{n} = (3, 4, 2)$. The resulting 9-by-6 matrix $\mathbf{C}$ will then look as follows:

$$\mathbf{C} = \begin{pmatrix} C_{1,1} & C_{1,2} & & & & \\ C_{2,1} & C_{2,2} & & & & \\ C_{3,1} & C_{3,2} & & & & \\ & & C_{4,3} & C_{4,4} & C_{4,5} & \\ & & C_{5,3} & C_{5,4} & C_{5,5} & \\ & & C_{6,3} & C_{6,4} & C_{6,5} & \\ & & C_{7,3} & C_{7,4} & C_{7,5} & \\ & & & & & C_{8,6} \\ & & & & & C_{9,6} \end{pmatrix}$$

In this particular case, the matrix contains $2 \cdot 3 + 3 \cdot 4 + 1 \cdot 2 = 20$ regular entries C_{ij} and 34 empty entries.

5.3.4. Oblivious Transfer of Long Messages

If the output length ℓ of the available hash function $h : \mathbb{B}^* \rightarrow \mathbb{B}^\ell$ is shorter than the messages M_i known to the sender, the methods of the previous subsections can not be applied directly. The problem is the computation of the values $C_i = M_i \oplus h(k_i)$ by the sender, for which equally long hash values $h(k_i)$ are needed. In general, for messages $M_i \in \mathbb{B}^{\ell_m}$ of length $\ell_m > \ell$, we can circumvent this problem by applying the counter mode of operation (CTR) from block ciphers. If we suppose that $\ell_m = r\ell$ is a multiple of ℓ, we can split each message M_i into r blocks $M_{ij} \in \mathbb{B}^\ell$ of length ℓ and process them individually using hash values $h(k_i, j)$. Here, the index $j \in \{1, \ldots, k\}$ plays the role of the counter. This is identical to applying a single concatenated hash value $h(k_i, 1) \,\|\, \cdots \,\|\, h(k_i, k)$ of length ℓ_m to M_i. If ℓ_m is not an exact multiple of ℓ, we do the same for $r = \lceil \ell_m / \ell \rceil$ block, but then truncate the first ℓ_m bits from the resulting concatenated hash value value to obtain the desired length.

Receiver		Sender
knows $\mathbf{s} = (s_1, \ldots, s_k)$		knows $\mathbf{m} = (M_1, \ldots, M_n)$
for $j = 1, \ldots, k$ – pick random $r_j \in_R \mathbb{Z}_q$ – compute $a_{j,1} = \Gamma(s_j) \cdot g_1^{r_j}$ – compute $a_{j,2} = g_2^{r_j}$ – let $a_j = (a_{j,1}, a_{j,2})$		
	$\xrightarrow{\mathbf{a} = (a_1, \ldots, a_k)}$	

		pick random $z_1, z_2 \in_R \mathbb{Z}_q$
		for $j = 1, \ldots, k$
		– pick random $\beta_j \in_R \mathbb{G}_q$
		– compute $b_j = a_{j,1}^{z_1} a_{j,2}^{z_2} \beta_j$
		for $l = 1, \ldots, t$
		– for $i = n'_l + 1, \ldots, n'_l + n_l$
		  – compute $k_i = \Gamma(i)^{z_1}$
		  – for $j = k'_l + 1, \ldots, k'_l + k_l$
		    – compute $k_{ij} = k_i \beta_j$
		    – compute $C_{ij} = M_i \oplus h(k_{ij})$
		compute $d = g_1^{z_1} g_2^{z_2}$

	$\mathbf{b} = (b_1, \ldots, b_k)$, $\mathbf{C} = (C_{ij})_{n \times k}, d$ $\longleftarrow$	
for $j = 1, \ldots, k$ – compute $k_j = b_j \cdot d^{-r_j}$ – compute $M_{s_j} = C_{s_j,j} \oplus h(k_j)$		

Protocol 5.3: Two-round $\mathrm{OT}_{\mathbf{n}}^{\mathbf{k}}$-scheme with sender privacy, where $g_1, g_2 \in \mathbb{G}_q \backslash \{1\}$ are independent generators of $\mathbb{G}_q \subset \mathbb{Z}_p^*$, $\Gamma : \{1, \ldots, n\} \to \mathbb{G}_q$ an encoding of the selections into $\mathbb{G}_q$, and $h : \mathbb{B}^* \to \mathbb{B}^\ell$ a collision-resistant hash function with output length ℓ.

5.4. Non-Interactive Preimage Proofs

Non-interactive zero-knowledge proofs of knowledge are important building blocks in cryptographic protocol design. In a non-interactive *preimage proof*

$$NIZKP[(x) : y = \phi(x)]$$

for a one-way group homomorphism $\phi : X \to Y$, the prover proves knowledge of a secret preimage $x = \phi^{-1}(y) \in X$ for a public value $y \in Y$ [38]. The most common construction of a non-interactive preimage proof results from combining the Σ-protocol with the Fiat-Shamir heuristic [22]. Proofs constructed in this way are perfect zero-knowledge in the random oracle model. In practical implementations, the random oracle is approximated with a collision-resistant hash function h.

Generating a preimage proof $(t, s) \leftarrow \mathsf{GenProof}_\phi(x, y)$ for ϕ consists of picking a random value $w \in_R X$ and computing a commitment $t = \phi(w) \in Y$, a challenge $c = h(y, t)$, and a response $s = w + c \cdot x \in X$. Verifying a proof includes computing $c = h(y, t)$ and checking $t = \phi(s) \cdot y^{-c}$. For a given proof $\pi = (t, s)$, this process is denoted by $b \leftarrow \mathsf{CheckProof}_\phi(\pi, y)$ for $b \in \mathbb{B}$. Clearly, we have

$$\mathsf{CheckProof}_\phi(\mathsf{GenProof}_\phi(x, y), y) = 1$$

for all $x \in X$ and $y = \phi(x) \in Y$.

5.4.1. Composition of Preimage Proofs

Preimage proofs for two (or more) one-way homomorphisms $\phi_1 : X_1 \to Y_1$ and $\phi_2 : X_2 \to Y_2$ can be reduced to a single preimage proof for $\phi : X_1 \times X_2 \to Y_1 \times Y_2$ defined by $\phi(x_1, x_2) = (\phi_1(x_1), \phi_2(x_2))$. In this case, $w = (w_1, w_2) \in X_1 \times X_2$, $t = (t_1, t_2) \in Y_1 \times Y_2$, and $s = (s_1, s_2) \in X_1 \times X_2$ are pairs of values, whereas c remains a singe value. This way of combining multiple preimage proofs into a single preimage proof is sometimes called *AND-composition*. The following two equivalent notations are therefore equivalent and can be used interchangeably:

$$\mathit{NIZKP}[(x_1, x_2) : y_1 = \phi_1(x_1) \wedge y_2 = \phi_2(x_2)] = \mathit{NIZKP}[(x_1, x_2) : (y_1, y_2) = \phi(x_1, x_2)].$$

An important special case of an AND-composition arises when $\phi_1 : X \to Y_1$ and $\phi_2 : X \to Y_2$ have a common domain X and when the $y_1 = \phi_1(x)$ and $y_2 = \phi_2(x)$ have the same preimage $x \in X$. The corresponding *equality proof*,

$$\mathit{NIZKP}[(x) : y_1 = \phi_1(x) \wedge y_2 = \phi_2(x)] = \mathit{NIZKP}[(x) : (y_1, y_2) = \phi(x)],$$

shows that y_1 and y_2 have an equal preimage. In the special case of two exponential functions $\phi_1(x) = g^x$ and $\phi_2(x) = h^x$, this demonstrates the equality of discrete logarithms [14].

5.4.2. Applications of Preimage Proofs

Let us look at some concrete instantiations of the above preimage proof. Each of them will be used later in this document.

Schnorr Identification. In a Schnorr identification scheme, the holder of a private credential $x \in X$ proves knowledge of $x = \phi^{-1}(y) = \log_g y$, where g is a generator in a suitable group Y in which the DL assumption holds [45]. This leads to one of the simplest and most fundamental instantiation of the above preimage proof,

$$\mathit{NIZKP}[(x) : y = g^x],$$

where $\phi(x) = g^x$ is the exponential function to base g. For $w \in_R X$, the prover computes $t = g^w$, $c = h(t, y)$, and $s = w + c \cdot x$, and the verifier checks $\pi = (t, s)$ by $t = y^{-c} \cdot g^s$.

Proof of Knowledge of ElGamal Plaintext. Another application of a preimage proof results from the ElGamal encryption scheme. The goal is to prove knowledge of the plaintext m and the randomization r for a given ElGamal ciphertext $(a, b) \leftarrow \mathsf{Enc}_{pk}(m, r)$, which we can denote as

$$NIZKP[(m, r) : e = \mathsf{Enc}_{pk}(m, r)] = NIZKP[(m, r) : (a, b) = (g^r, m \cdot pk^r)].$$

Since Enc_{pk} defines a homomorphism from $\mathbb{G}_q \times \mathbb{Z}_q$ to $\mathbb{G}_q \times \mathbb{G}_q$, both the commitment $t = (t_1, t_2) \in \mathbb{G}_q \times \mathbb{G}_q$ and the response $s = (s_1, s_2) \in \mathbb{G}_q \times \mathbb{Z}_q$ are pairs of values. Generating the proof requires two and verifying the proof four exponentiations in $\mathbb{G}_q$.

ElGamal Decryption Proof. The decryption $m \leftarrow \mathsf{Dec}_{sk}(e)$ of an ElGamal ciphertext $e = (a, b)$ defines a mapping from $\mathbb{G}_q \times \mathbb{G}_q$ to $\mathbb{G}_q$, but this mapping is not homomorphic. The desired decryption proof,

$$NIZKP[(sk) : m = \mathsf{Dec}_{sk}(e) \wedge pk = g^{sk}] = NIZKP[(sk) : (m, pk) = (a \cdot b^{-sk}, g^{sk})],$$

which demonstrates that the correct decryption key sk has been used, can therefore not be treated directly as an application of a preimage proof. However, since $m = a \cdot b^{-sk}$ can be rewritten as $a/m = b^{sk}$, we can achieve the same goal by

$$NIZKP[(sk) : (a/m, pk) = (b^{sk}, g^{sk})].$$

Note that this proof is a standard proof of equality of discrete logarithms. We will use it to prove the correctness of a partial decryption $b_j = b^{sk_j}$, where sk_j is a share of the private key sk (see Section 5.1.2).

5.5. Wikström's Shuffle Proof

A *cryptographic shuffle* of a list $\mathbf{e} = (e_1, \ldots, e_N)$ of ElGamal encryptions $e_i \leftarrow \mathsf{Enc}_{pk}(m_i, r_i)$ is another list of ElGamal encryptions $\mathbf{e}' = (e'_1, \ldots, e'_N)$, which contains the same plaintexts m_i in permuted order. Such a shuffle can be generated by selecting a random permutation $\psi : \{1, \ldots, N\} \rightarrow \{1, \ldots, N\}$ from the set Ψ_N of all such permutations (e.g., using Knuth's shuffle algorithm [33]) and by computing re-encryptions $e'_i \leftarrow \mathsf{ReEnc}_{pk}(e_j, r'_j)$ for $j = \psi(i)$. We write

$$\mathbf{e}' \leftarrow \mathsf{Shuffle}_{pk}(\mathbf{e}, \mathbf{r}', \psi)$$

for an algorithm performing this task, where $\mathbf{r}' = (r'_1, \ldots, r'_N)$ denotes the randomization used to re-encrypt the input ciphertexts.

Proving the correctness of a cryptographic shuffle can be realized by proving knowledge of ψ and $\mathbf{r}'$, which generate $\mathbf{e}'$ from $\mathbf{e}$ in a cryptographic shuffle:

$$NIZKP[(\psi, \mathbf{r}') : \mathbf{e}' = \mathsf{Shuffle}_{pk}(\mathbf{e}, \mathbf{r}', \psi)].$$

Unfortunately, since $\mathsf{Shuffle}_{pk}$ does not define a homomorphism, we can not apply the standard technique for preimage proofs. Therefore, the strategy of what follows is to find an equivalent formulation using a homomorphism.

The shuffle proof according to Wikström and Terelius consists of two parts, an offline and an online proof. In the offline proof, the prover computes a commitment $c \leftarrow \mathsf{Com}(\psi, \mathbf{r})$

and proves that c is a commitment to a permutation matrix. In the online proof, the prover demonstrates that the committed permutation matrix has been used in the shuffle to obtain $\mathbf{e}'$ from $\mathbf{e}$. The two proofs can be kept separate, but combining them into a single proof results in a slightly more efficient method. Here, we only present the combined version of the two proofs and we restrict ourselves to the case of shuffling ElGamal ciphertexts.

From a top-down perspective, Wikström's shuffle proof can be seen as a two-layer proof consisting of a top layer responsible for preparatory work such as computing the commitment $\mathbf{c} \leftarrow \mathsf{Com}(\psi, \mathbf{r})$ and a bottom layer computing a standard preimage proof.

5.5.1. Preparatory Work

There are two fundamental ideas behind Wikström's shuffle proof. The first idea is based on a simple theorem that states that if $\mathbf{B}_\psi = (b_{ij})_{N \times N}$ is an N-by-N matrix over $\mathbb{Z}_q$ and $(x_1, ..., x_N)$ a vector of N independent variables, then $\mathbf{B}_\psi$ is a permutation matrix if and only if $\sum_{j=1}^N b_{ij} = 1$, for all $i \in \{1, \dots, N\}$, and $\prod_{i=1}^N \sum_{j=1}^N b_{ij} x_i = \prod_{i=1}^N x_i$. The first condition means that the elements of each row of $\mathbf{B}_\psi$ must sum up to one, while the second condition requires that $\mathbf{B}_\psi$ has exactly one non-zero element in each row.

Based on this theorem, the general proof strategy is to compute a permutation commitment $\mathbf{c} \leftarrow \mathsf{Com}(\psi, \mathbf{r})$ and to construct a zero-knowledge argument that the two conditions of the theorem hold for $\mathbf{B}_\psi$. This implies then that $\mathbf{c}$ is a commitment to a permutation matrix without revealing ψ or $\mathbf{B}_\psi$.

For $\mathbf{c} = (c_1, \dots, c_N)$, $\mathbf{r} = (r_1, \dots, r_N)$, and $\bar{r} = \sum_{j=1}^N r_j$, the first condition leads to the following equality:

$$\prod_{j=1}^N c_j = \prod_{j=1}^N g^{r_j} \prod_{i=1}^N h_i^{b_{ij}} = g^{\sum_{j=1}^N r_j} \prod_{i=1}^N h_i^{\sum_{j=1}^N b_{ij}} = g^{\bar{r}} \prod_{i=1}^N h_i = \mathsf{Com}(\mathbf{1}, \bar{r}). \tag{5.2}$$

Similarly, for arbitrary values $\mathbf{u} = (u_1, \dots, u_N) \in \mathbb{Z}_q^N$, $\mathbf{u}' = (u'_1, \dots, u'_N) \in \mathbb{Z}_q^N$, with $u'_i = \sum_{j=1}^N b_{ij} u_j = u_j$ for $j = \psi(i)$, and $\tilde{r} = \sum_{j=1}^N r_j u_j$, the second condition leads to two equalities:

$$\prod_{i=1}^N u'_i = \prod_{j=1}^N u_j, \tag{5.3}$$

$$\begin{aligned}\prod_{j=1}^N c_j^{u_j} &= \prod_{j=1}^N (g^{r_j} \prod_{i=1}^N h_i^{b_{ij}})^{u_j} = g^{\sum_{j=1}^N r_j u_j} \prod_{i=1}^N h_i^{\sum_{j=1}^N b_{ij} u_j} = g^{\tilde{r}} \prod_{i=1}^N h_i^{u'_i} \\ &= \mathsf{Com}(\mathbf{u}', \tilde{r}),\end{aligned} \tag{5.4}$$

By proving that (5.2), (5.3), and (5.4) hold, and from the independence of the generators, it follows that both conditions of the theorem are true and finally that $\mathbf{c}$ is a commitment to a permutation matrix. In the interactive version of Wikström's proof, the prover obtains $\mathbf{u} = (u_1, \dots, u_N) \in \mathbb{Z}_q^N$ in an initial message from the verifier, but in the non-interactive version we derive these values from the public inputs, for example by computing $u_i \leftarrow \mathsf{Hash}((\mathbf{e}, \mathbf{e}', \mathbf{c}), i)$.

The second fundamental idea of Wikström's proof is based on the homomorphic property of the ElGamal encryption scheme and the following observation for values $\mathbf{u}$ and $\mathbf{u}'$ defined in the same way as above:

$$\begin{aligned}\prod_{i=1}^{N}(e'_i)^{u'_i} &= \prod_{j=1}^{N}\mathsf{ReEnc}_{pk}(e_j, r'_j)^{u_j} = \prod_{j=1}^{N}\mathsf{ReEnc}_{pk}(e_j^{u_j}, r'_j u_j)\\ &= \mathsf{ReEnc}_{pk}(\prod_{j=1}^{N} e_j^{u_j}, \sum_{j=1}^{N} r'_j u_j) = \mathsf{Enc}_{pk}(1, r')\cdot\prod_{j=1}^{N} e_j^{u_j},\end{aligned} \tag{5.5}$$

for $r' = \sum_{j=1}^{N} r'_j u_j$. By proving (5.5), it follows that every e'_i is a re-encryption of e_j for $j = \psi(i)$. This is the desired property of the cryptographic shuffle. By putting (5.2) to (5.5) together, the shuffle proof can therefore be rewritten as follows:

$$NIZKP\left[(\bar{r}, \tilde{r}, r', \mathbf{u}') : \begin{array}{l}\prod_{j=1}^{N} c_j = \mathsf{Com}(\mathbf{1}, \bar{r})\\ \wedge \prod_{i=1}^{N} u'_i = \prod_{j=1}^{N} u_j\\ \wedge \prod_{j=1}^{N} c_j^{u_j} = \mathsf{Com}(\mathbf{u}', \tilde{r})\\ \wedge \prod_{i=1}^{N}(e'_i)^{u'_i} = \mathsf{Enc}_{pk}(1, r')\cdot\prod_{j=1}^{N} e_j^{u_j}\end{array}\right].$$

The last step of the preparatory work results from replacing in the above expression the equality of products, $\prod_{i=1}^{N} u'_i = \prod_{j=1}^{N} u_j$, by an equivalent expression based on a chained list $\hat{\mathbf{c}} = \{\hat{c}_1, \ldots, \hat{c}_N\}$ of Pedersen commitments with different generators. For $\hat{c}_0 = h$ and random values $\hat{\mathbf{r}} = (\hat{r}_1, \ldots, \hat{r}_N) \in \mathbb{Z}_q^N$, we define $\hat{c}_i = g^{\hat{r}_i}\hat{c}_{i-1}^{u'_i}$, which leads to $\hat{c}_N = \mathsf{Com}(u, \hat{r})$ for $u = \prod_{i=1}^{N} u_i$ and

$$\hat{r} = \sum_{i=1}^{N}\hat{r}_i\prod_{j=i+1}^{N} u'_j.$$

Applying this replacement leads to the following final result, on which the proof construction is based:

$$NIZKP\left[(\bar{r}, \hat{r}, \tilde{r}, r', \hat{\mathbf{r}}, \mathbf{u}') : \begin{array}{l}\prod_{j=1}^{N} c_j = \mathsf{Com}(\mathbf{1}, \bar{r})\\ \wedge\, \hat{c}_N = \mathsf{Com}(u, \hat{r}) \wedge \left[\bigwedge_{i=1}^{N}(\hat{c}_i = g^{\hat{r}_i}\hat{c}_{i-1}^{u'_i})\right]\\ \wedge \prod_{j=1}^{N} c_j^{u_j} = \mathsf{Com}(\mathbf{u}', \tilde{r})\\ \wedge \prod_{i=1}^{N}(e'_i)^{u'_i} = \mathsf{Enc}_{pk}(1, r')\cdot\prod_{j=1}^{N} e_j^{u_j}\end{array}\right].$$

To summarize the preparatory work for the proof generation, we give a list of all necessary computations:

- Pick $\mathbf{r} = (r_1, \ldots, r_N) \in_R \mathbb{Z}_q^N$ and compute $\mathbf{c} \leftarrow \mathsf{Com}(\psi, \mathbf{r})$.
- For $i = 1, \ldots, N$, compute $u_i \leftarrow \mathsf{Hash}((\mathbf{e}, \mathbf{e}', \mathbf{c}), i)$, let $u'_i = u_{\psi(i)}$, pick $\hat{r}_i \in_R \mathbb{Z}_q$, and compute $\hat{c}_i = g^{\hat{r}_i}\hat{c}_{i-1}^{u'_i}$.
- Let $\hat{\mathbf{r}} = (\hat{r}_1, \ldots, \hat{r}_N)$ and $\hat{\mathbf{c}} = (\hat{c}_1, \ldots, \hat{c}_N)$.
- Compute $\bar{r} = \sum_{j=1}^{N} r_j$, $\hat{r} = \sum_{i=1}^{N}\hat{r}_i\prod_{j=i+1}^{N} u'_j$, $\tilde{r} = \sum_{j=1}^{N} r_j u_j$, and $r' = \sum_{j=1}^{N} r'_j u_j$.

Note that $\hat{r}$ can be computed in linear time by generating the values $\prod_{j=i+1}^{N} u'_j$ in an incremental manner by looping backwards over $j = N, \ldots, 1$.

5.5.2. Preimage Proof

By rearranging all public values to the left-hand side and all secret values to the right-hand side of each equation, we can derive a homomorphic one-way function from the final expression of the previous subsection. In this way, we obtain the homomorphic function

$$\begin{aligned}
&\phi(x_1, x_2, x_3, x_4, \hat{\mathbf{x}}, \mathbf{x}') \\
&\quad = (g^{x_1}, g^{x_2}, \mathsf{Com}(\mathbf{x}', x_3), \mathsf{ReEnc}_{pk}(\prod_{i=1}^{N}(e'_i)^{x'_i}, -x_4), (g^{\hat{x}_1}\hat{c}_0^{x'_1}, \ldots, g^{\hat{x}_N}\hat{c}_{N-1}^{x'_N})),
\end{aligned}$$

which maps inputs $(x_1, x_2, x_3, x_4, \hat{\mathbf{x}}, \mathbf{x}') \in X$ of length $2N + 4$ into outputs

$$(y_1, y_2, y_3, y_4, \hat{\mathbf{y}}) = \phi(x_1, x_2, x_3, x_4, \hat{\mathbf{x}}, \mathbf{x}') \in Y$$

of length $N + 5$, i.e., $X = \mathbb{Z}_q^4 \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N$ is the domain and $Y = \mathbb{G}_q^3 \times \mathbb{G}_q^2 \times \mathbb{G}_q^N$ the co-domain of ϕ. Note that we slightly modified the order of the five sub-functions of ϕ for better readability. By applying this function to the secret values $(\bar{r}, \hat{r}, \tilde{r}, r', \hat{\mathbf{r}}, \mathbf{u}')$, we get a tuple of public values,

$$(\bar{c}, \hat{c}, \tilde{c}, e', \hat{\mathbf{c}}) = (\frac{\prod_{j=1}^{N} c_j}{\prod_{j=1}^{N} h_j}, \frac{\hat{c}_N}{h^u}, \prod_{j=1}^{N} c_j^{u_j}, \prod_{j=1}^{N} e_j^{u_j}, (\hat{c}_1, \ldots, \hat{c}_N)),$$

which can be derived from the public values $\mathbf{e}$, $\mathbf{e}'$, $\mathbf{c}$, $\hat{\mathbf{c}}$, and pk (and from $\mathbf{u}$, which is derived from $\mathbf{e}$, $\mathbf{e}'$, and $\mathbf{c}$).

To summarize, we have a homomorphic one-way function $\phi : X \to Y$, secret values $x = (\bar{r}, \hat{r}, \tilde{r}, r', \hat{\mathbf{r}}, \mathbf{u}') \in X$, and public values $y = (\bar{c}, \hat{c}, \tilde{c}, e', \hat{\mathbf{c}}) = \phi(x) \in Y$. We can therefore generate a non-interactive preimage proof

$$NIZKP\left[(\bar{r}, \hat{r}, \tilde{r}, r', \hat{\mathbf{r}}, \mathbf{u}') : \begin{array}{r} \bar{c} = g^{\bar{r}} \wedge \hat{c} = g^{\hat{r}} \wedge \tilde{c} = \mathsf{Com}(\mathbf{u}', \tilde{r}) \\ \wedge\ e' = \mathsf{ReEnc}_{pk}(\prod_{i=1}^{N}(e'_i)^{u'_i}, -r') \\ \wedge \left[\bigwedge_{i=1}^{N}(\hat{c}_i = g^{\hat{r}_i}\hat{c}_{i-1}^{u'_i})\right] \end{array} \right],$$

using the standard procedure from Section 5.4. The result of such a proof generation, $(t, s) \leftarrow \mathsf{GenProof}_{\phi}(x, y)$, consists of two values $t = \phi(w) \in Y$ of length $N + 5$ and $s = \omega + c \cdot x \in X$ of length $2N + 4$, which we obtain from picking $w \in_R X$ (of length $2N + 4$) and computing $c = \mathsf{Hash}(y, t)$. Alternatively, a different $c = \mathsf{Hash}(y', t)$ could be derived directly from the public values $y' = (\mathbf{e}, \mathbf{e}', \mathbf{c}, \hat{\mathbf{c}}, pk)$, which has the advantage that $y = (\bar{c}, \hat{c}, \tilde{c}, e', \hat{\mathbf{c}})$ needs not to be computed explicitly during the proof generation.

This preimage proof, together with the two lists of commitments $\mathbf{c}$ and $\hat{\mathbf{c}}$, leads to the desired non-interactive shuffle proof $NIZKP[(\psi, \mathbf{r}') : \mathbf{e}' = \mathsf{Shuffle}_{pk}(\mathbf{e}, \mathbf{r}', \psi)]$. We denote the generation and verification of a such proof $\pi = (t, s, \mathbf{c}, \hat{\mathbf{c}})$ by

$$\begin{aligned}
\pi &\leftarrow \mathsf{GenProof}_{pk}(\mathbf{e}, \mathbf{e}', \mathbf{r}', \psi) \\
b &\leftarrow \mathsf{CheckProof}_{pk}(\pi, \mathbf{e}, \mathbf{e}').
\end{aligned}$$

respectively. Corresponding algorithms are depicted in Alg. 7.43 and Alg. 7.47. Note that generating the proof requires $7N+4$ and verifying the proof $9N+11$ modular exponentiations in $\mathbb{G}_q$. The proof itself consists of $5N + 9$ elements ($2N + 4$ elements from $\mathbb{Z}_q$ and $3N + 5$ elements from $\mathbb{G}_q$).

5.6. Schnorr Signatures

The *Schnorr signature scheme* consists of a triple $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ of algorithms, which operate on a cyclic group for which the DL problem is believed to be hard [45]. A common choice is a prime-order subgroup $\mathbb{G}_q$ of the multiplicative group $\mathbb{Z}_p^*$ of integers modulo p, where the primes $p = kq+1$ (for $k \geqslant 2$) and q are large enough to resist all known methods for solving the discrete logarithm problem. In this particular setting, the public parameters of a Schnorr signature scheme are the values p and q, a generator $g \in \mathbb{G}_q \backslash \{1\}$, and a cryptographic hash function $h : \mathbb{B}^* \rightarrow \mathbb{B}^\ell$. Note that the output length ℓ of the hash function depends on the scheme's security parameter.

A key pair in the Schnorr signature scheme is a tuple $(sk, pk) \leftarrow \mathsf{KeyGen}()$, where $sk \in_R \mathbb{Z}_q$ is the randomly chosen private signature key and $pk = g^{sk} \in \mathbb{G}_q$ the corresponding public verification key. If $m \in \mathbb{B}^*$ denotes the message to sign and $r \in_R \mathbb{Z}_q$ a random value, then a Schnorr signature

$$(c, s) \leftarrow \mathsf{Sign}_{sk}(m, r) \in \mathbb{B}^\ell \times \mathbb{Z}_q$$

consists of two values $c = h(g^r, m)$ and $s = r - c \cdot sk$. Using the public key sk, a given signature $\sigma = (c, s)$ of m can be verified by

$$b \leftarrow \mathsf{Verify}_{pk}(\sigma, m) = \begin{cases} 1, & \text{if } h(g^s \cdot pk^c, m) = c, \\ 0, & \text{otherwise.} \end{cases} \quad .$$

For any given key pair $(sk, pk) \leftarrow \mathsf{KeyGen}()$, it is easy to show that $\mathsf{Verify}_{pk}(\mathsf{Sign}_{sk}(m, r), m) = 1$ holds for all $m \in \mathbb{B}^*$ and $r \in \mathbb{Z}_q$. Note that a Schnorr signature is very similar to a non-interactive zero-knowledge proof $NIZKP[(sk) : pk = g^{sk}]$, in which m is passed as an additional input to the Fiat-Shamir hash function (a few other subtle differences are due to different traditions of describing Schnorr signatures and non-interactive zero-knowledge proofs in the literature).

Assuming that the DL problem is hard in the chosen group, the Schnorr signature scheme is provably EUF-CMA secure in the random oracle model. Due to (expired) patent restrictions, Schnorr signatures have been standardized only recently and only for elliptic curves [1, 7]. As a consequence, despite multiple advantages over other DL-based schemes such as DSA (which is not provably secure in the random oracle model), they are not yet very common in practical applications.

5.7. Hybrid Encryption and Key-Encapsulation

For large messages $m \in \mathcal{B}^*$, public-key encryption schemes such as ElGamal are often not efficient enough. This is the motivation for constructing hybrid encryption schemes, which combine the advantages of (asymmetric) public-key encryption schemes with the advantages of (symmetric) secret-key encryption schemes. The idea is to use a *key-encapsulation mechanism* (KEM) to generate and encapsulate an ephemeral secret key $k \in \mathbb{B}^\ell$, which is used to encrypt m symmetrically. For a key pair $(sk, pk) \leftarrow \mathsf{KeyGen}()$, the result of a hybrid encryption is a ciphertext $(c, c') \leftarrow \mathsf{Enc}_{pk}(m)$, which consists of the encapsulated key c obtained from $(c, k) \leftarrow \mathsf{Encaps}_{pk}()$ and the symmetric ciphertext $c' \leftarrow \mathsf{Enc}'_{\mathsf{k}}(m)$. The decryption $m \leftarrow \mathsf{Dec}_{sk}(c, c')$ works in the opposite manner, i.e., first the symmetric key

$k \leftarrow \mathsf{Decaps}_{sk}(c)$ is reconstructed from c and then the plaintext message $m \leftarrow \mathsf{Dec}'_k(c')$ is decrypted from c' using k. Note that a triple of algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ constructed in this way from a key-encapsulation mechanism $(\mathsf{Encaps}, \mathsf{Decaps})$ and a secret-key encryption scheme $(\mathsf{Enc}', \mathsf{Dec}')$ is a public-key encryption scheme. For this general construction, IND-CPA and IND-CCA security can be proven depending on the properties of the underlying schemes [32].

A simple KEM construction operates on a cyclic group for which at least the CDH problem is believed to be hard. A common choice is a prime-order subgroup $\mathbb{G}_q$ of the multiplicative group $\mathbb{Z}_p^*$ of integers modulo p, where the $p = kq + 1$ (for $k \geqslant 2$) and q are large primes. In this particular setting, the public parameters of the KEM are the values p and q, a generator $g \in \mathbb{G}_q \backslash \{1\}$, and a cryptographic hash function $h : \mathbb{B}^* \rightarrow \mathbb{B}^\ell$ with output length ℓ (which corresponds to the length of the symmetric key k and therefore depends on the security parameter). Note that this setting is identical to the setting of the above Schnorr signature scheme, except for the slightly stronger computational assumption. A key pair in this setting consists of two values $sk \in_R \mathbb{Z}_q$ and $pk = g^{sk} \in \mathbb{G}_q$, and key encapsulation generates a pair of values $c = g^r$ and $k = h(pk^r)$, where $r \in_R \mathbb{Z}_q$ is chosen at random. Using the private key sk, the symmetric key $k = h(c^{sk}) = h(pk^r)$ can then be reconstructed from c. Note that both key encapsulation and decapsulation require a single exponentiation in $\mathbb{G}_q$.

Relative to the above KEM algorithms Encaps and Decaps, a proof for CPA-security can be based either on DDH (standard model) or CDH (random oracle model) [32]. However, by combining this KEM with a practical block cipher such as AES, an appropriate mode of operation such as CTR, and a suitable padding algorithm, provable security is replaced by practical security, i.e., it is assumed that the practical block cipher is a good approximation of an ideal block cipher [19]. Nevertheless, given the significant efficiency benefits, instantiations based on current standards such a as AES are commonly accepted and widely used in practice.

Part III.

Protocol Specification

6. Protocol Description

The goal of this chapter is to describe the cryptographic voting protocol from various perspectives. We introduce the involved parties, describe their roles, and define the communication channels over which they exchange messages during a protocol execution. The protocol itself has various phases—each with multiple sub-phases—which we describe with sufficient technical details for understanding the general protocol design and the most important computational details. A comprehensive list of security and election parameters is introduced beforehand. We also model the adversary and give a list of underlying trust assumptions. Finally, we discuss the security properties that we obtain from applying the adversary model and trust assumptions to the protocol. For further details in form of low-level pseudo-code algorithms, we refer to Chapter 7. The protocol itself is an extension of the protocol introduced in [27].

6.1. Parties and Communication Channels

In our protocol, we consider six different types of parties. A party can be a human being, a computer, a human being controlling a computer, or even a combination of multiple human beings and computers. In each of these cases, we consider them as atomic entities with distinct tasks and responsibilities. Here is the list of parties we consider:

- The *election administrator* is responsible for setting up an election event. This includes tasks such as defining the electoral roll, the number of elections, the set of candidates in each election, and the eligibility of each voter in each election (see Section 6.3.2). At the end of the election process, the election administrator determines and publishes the final election result.
- A group of *election authorities* guarantees the integrity and privacy of the votes submitted during the election period. They are numbered with indices $j \in \{1, \ldots, s\}$, $s \geqslant 1$. Before every election event, they establish jointly a public ElGamal encryption key pk. They also generate the credentials and codes to be printed on the voting cards. During vote casting, they respond to the submitted ballots and confirmations. At the end of the election period, they perform a cryptographic shuffle of the encrypted votes. Finally, they use their private key shares sk_j to decrypt the votes in a distributed manner.
- The *printing authority* is responsible for printing the voting cards and delivering them to the voters. They receive the data necessary for generating the voting cards from the bulletin board and the election authorities.

- The *voters* are the actual human users of the system. They are numbered with indices $i \in \{1, \ldots, N_E\}$, $N_E \geqslant 0$. Prior to an election event, they receive the voting card from the printing authority, which they can use to cast and confirm a vote during the election period using their voting client.
- The *voting client* is a machine used by some voter to conduct the vote casting and confirmation process. Typically, this machine is either a desktop, notebook, or tablet computer with a network connection and enough computational power to perform cryptographic computations. The strict separation between voter and voting client is an important precondition for the protocol's security concept.
- The *bulletin board* is the central communication unit of the system. It implements a broadcast channel with memory among the parties involved in the protocol [29]. For this, it keeps track of all the messages reveived during the protocol execution. The messages from the election administrator and the election authorities are kept in separate dedicated sections, which implies that bulletin board can authenticate them unambiguously. The entire election data stored by the bulletin board defines the input of the verification process.

An overview of the involved parties is given in Figure 6.1, together with the necessary communication channels between them. It depicts the central role of the bulletin board as a communication hub. The election administrator, for example, only communicates with the bulletin board. Since only public messages are sent to the bulletin board, none of its input or output channels is confidential. As indicated in Figure 6.1 by means of a padlock, confidential channels only exist from the election authorities to the printing authority and from the printing authority to the voters (and between the voter and the voting client). The channel from the printing authority to the voters consists of sending a personalized voting card by postal mail.

We assume that the election administrator and the election authorities are in possession of a private signature key, which they use to sign all messages sent to the bulletin board. Corresponding output channels are therefore authentic. In Section 6.6, we give further details on how the presumed channel security can be achieved in practice, and in Section 7.6, we give corresponding pseudo-code algorithms.

A special case is the channel between the voter and the voting client, which exists in form of the device's user interface and the voter's interaction with the device. We assume that this channel is confidential. Note that the bandwidth of this channel is obviously not very high. All other channels are assumed to be efficient enough for transmitting the messages and the signatures sufficiently fast.

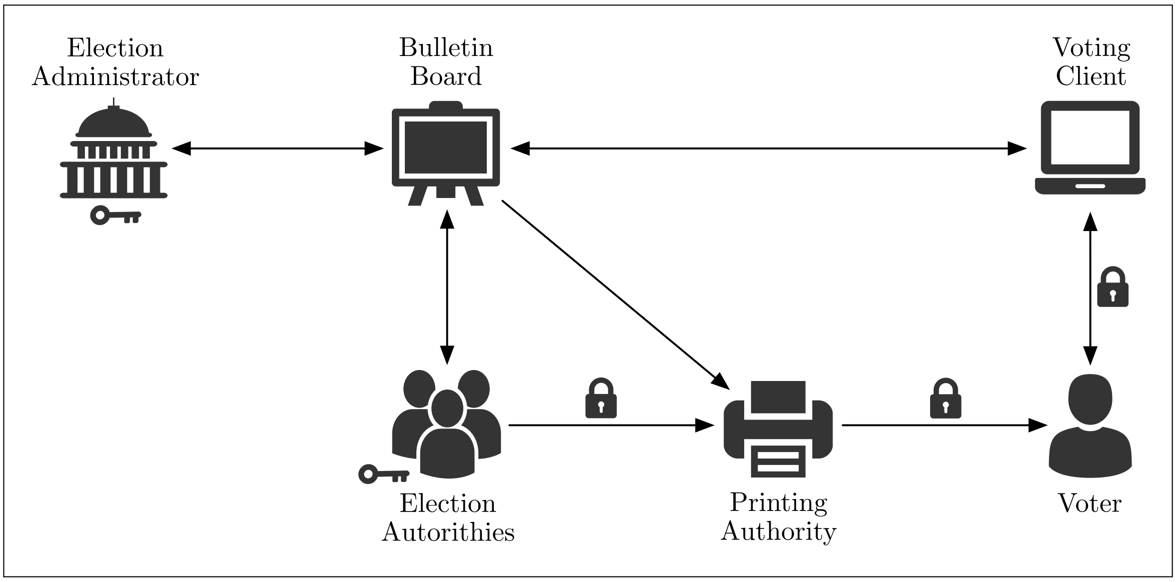

Figure 6.1.: Overview of the parties and communication channels.

6.2. Adversary Model and Trust Assumptions

We assume that the general adversarial goal is to break the integrity or secrecy of the votes, but not to influence the election outcome via bribery or coercion. We consider *covert adversaries*, which may arbitrarily interfere with the voting process or deviate from the protocol specification to reach their goals, but only if such attempts are likely to remain undetected [9]. Voters and authorities are potential covert adversaries, as well as any external party. This includes adversaries trying to spread dedicated malware to gain control over the voting clients or to break into the systems operated by the election administrator, the election authorities, or the bulletin board.

All parties are polynomially bounded and thus incapable of solving supposedly hard problems such as the DDH problem or breaking cryptographic primitives such as contemporary hash algorithms. This implies that adversaries cannot efficiently decrypt ElGamal ciphertexts or generate valid non-interactive zero-knowledge proofs without knowing the secret inputs. For making the system resistant against attacks of that kind, it is necessary to select the cryptographic parameters of Section 6.3 with much care and in accordance with current recommendations (see Chapter 8).

For preparing and conducting an election event, as well as for computing the final election result, we assume that at least one honest election authority is following the protocol faithfully. In other words, we take into account that dishonest election authorities may collude with the adversary (willingly or unwillingly), but not all of them in the same election event. Trust assumptions like this are common in cryptographic voting protocols, but they may be difficult to implement in practice. A difficult practical problem is to guarantee that the authorities act independently, which implies, for example, that they use software written by independent developers and run them on hardware from independent manufacturers. This document does not specify conditions for the election authorities to reach a satisfactory degree of independence.

There are two very strong trust assumptions in our protocol. The first one is attributed to the voting client, which is assumed not to be corrupted by an adversary trying to attack vote privacy. Since the voting client learns the plaintext vote from the voter during the vote casting process, it is obvious that vote privacy can not be guaranteed in the presence of a corrupted device, for instance one that is infiltrated with malware. This is one of the most important unsolved problems in any approach, in which voter's are allowed to prepare and submit their votes on their own (insecure) devices.

The second very strong trust assumption in our protocol is attributed to the printing authority. For printing the voting cards in the pre-election phase, the printing authority receives very sensitive information from the election authorities, for example the credentials for submitting a vote or the verification codes for the candidates. In principle, knowing this information allows the submission of votes on behalf of eligible voters. Exploiting this knowledge would be noticed by the voters when trying to submit a ballot, but obviously not by voters abstaining from voting. Even worse, if check is given access to the verification codes, it can easily bypass the cast-as-intended verification mechanism, i.e., voters can no longer detect vote manipulations on the voting client. These scenarios exemplify the strength of the trust assumptions towards the printing authority, which after all constitutes a single-point-of-failure in the system. Given the potential security impact in case of a failure, it is important to use extra care when selecting the people, the technical infrastructure (computers, software, network, printers, etc.), and the business processes for providing this service. In this document, we will give a detailed functional specification of the printing authority (see Section 7.3), but we will not recommend measures for establishing a sufficient amount of trust.

6.3. System Parameters

The specification of the cryptographic voting protocol relies on a number of system parameters, which need to be fixed for every election event. There are two categories of parameters. The first category consists of *security parameters*, which define the security of the system from a cryptographic point of view. They are likely to remain unchanged over multiple election events until external requirements such as the desired level of protection or key length recommendations from well-known organizations are revised. The second category of *election parameters* define the particularities of every election event such as the number of eligible voters or the candidate list. In our subsequent description of the protocol, we assume that the security parameters are known to everyone, whereas the election parameters are published on the bulletin board by the election administrator. Knowing the full set of all parameters is a precondition for verifying an election result based on the data published on the bulletin board.

6.3.1. Security Parameters

The security of the system is determined by four principal security parameters. As the resistance of the system against attackers of all kind depends strongly on the actual choice of these parameters, they need to be selected with much care. Note that they impose strict lower bounds for all other security parameters.

- The *minimal privacy* σ defines the amount of computational work for a polynomially bounded adversary to break the privacy of the votes to be greater or equal to $c \cdot 2^\sigma$ for some constant value $c > 0$ (under the given trust assumptions of Section 6.2). This is equivalent to brute-force searching a key of length σ bits. Recommended values today are $\sigma = 112$, $\sigma = 128$, or higher.

- The *minimal integrity* τ defines the amount of computational work for breaking the integrity of a vote in the same way as σ for breaking the privacy of the vote. In other words, the actual choice of τ determines the risk that an adversary succeeds in manipulating an election. Recommendations for τ are similar to the above-mentioned values for σ, but since manipulating an election is only possible during the election period or during tallying, a less conservative value may be chosen.

- The *deterrence factor* $0 < \epsilon \leqslant 1$ defines a lower bound for the probability that an attempt to cheat by an adversary is detected by some honest party. Clearly, the higher the value of ϵ, the greater the probability for an adversary of getting caught and therefore the greater the deterrent to perform an attack. There are no general recommendations, but values such as $\epsilon = 0.99$ or $\epsilon = 0.999$ seem appropriate for most applications.

- The *number of election authorities* $s \geqslant 1$ determines the amount of trust that needs to be attributed to each of them. This is a consequence of our assumption that at least one election authority is honest, i.e., in the extreme case of $s = 1$, full trust is attributed to a single authority. Generally, increasing the number of authorities means to decrease the chance that they are all malicious. On the other hand, finding a large number of independent and trustworthy authorities is a difficult problem in practice. There is no general rule, but $3 \leqslant s \leqslant 5$ authorities seems to be a reasonable choice in practice.

In the following paragraphs, we introduce the complete set of security parameters that can be derived from σ, τ, and ϵ. A summary of all parameters and constraints to consider when selecting them will be given in Table 6.1 at the end of this subsection.

a) Hash Algorithm Parameters

At multiple places in our voting protocol, we require a collision-resistant hash functions $h : \mathbb{B}^* \rightarrow \mathbb{B}^\ell$ for various purposes. In principle, we could work with different output lengths ℓ, depending on whether the use of the hash function affects the privacy or integrity of the system. However, for reasons of simplicity, we propose to use a single hash algorithm $\mathsf{Hash}_L(B)$ throughout the entire document. Its output length $L = 8\ell$ must therefore be adjusted to both σ and τ. The general rule for a hash algorithm to resist against birthday attacks is that its output length should at least double the desired security strength, i.e., $\ell \geqslant 2 \cdot \max(\sigma, \tau)$ bits (resp. $L \geqslant \frac{\max(\sigma,\tau)}{4}$ bytes) in our particular case.

b) Group and Field Parameters

Other important building blocks in our protocol are the algebraic structures (two multiplicative groups, one prime field), on which the cryptographic primitives operate. Selecting appropriate group and field parameters is important to guarantee the minimal privacy σ and the minimal integrity τ. We follow the current NIST recommendations [10, Table 2], which defines minimal bit lengths for corresponding moduli and orders.

- The *encryption group* $\mathbb{G}_q \subset \mathbb{Z}_p$ is a q-order subgroup of the multiplicative group of integers modulo a safe prime $p = 2q + 1 \in \mathbb{S}$. Since $\mathbb{G}_q$ is used for the ElGamal encryption scheme and the oblivious transfer, i.e., it is only used to protect the privacy of the votes, the minimal bit length of p (and q) depends on σ only. The following constraints are consistent with the NIST recommendations:

$$\|p\| \geqslant \begin{cases} 1024, & \text{for } \sigma = 80, \\ 2048, & \text{for } \sigma = 112, \\ 3072, & \text{for } \sigma = 128, \\ 7680, & \text{for } \sigma = 192, \\ 15360, & \text{for } \sigma = 256. \end{cases} \tag{6.1}$$

  In addition to p and q, two independent generators $g, h \in \mathbb{G}_q \backslash \{1\}$ of this group must be known to everyone. The only constraint when selecting them is that their independence is guaranteed in a verifiable manner.

- The *identification group* $\mathbb{G}_{\hat{q}} \subset \mathbb{Z}_{\hat{p}}$ is a $\hat{q}$-order subgroup of the multiplicative group of integers modulo a prime $\hat{p} = k\hat{q} + 1 \in \mathbb{P}$, where $\hat{q} \in \mathbb{P}$ is prime and $k \geqslant 2$ the co-factor. Since this group is used for voter identification using Schnorr's identification scheme, i.e., it is only used to protect the integrity of the votes, the bit length of $\hat{p}$ and $\hat{q}$ depend on τ only. The constraints for the bit length of $\hat{p}$ are therefore identical to the constraints for the bit length of p,

$$\|\hat{p}\| \geqslant \begin{cases} 1024, & \text{for } \tau = 80, \\ 2048, & \text{for } \tau = 112, \\ 3072, & \text{for } \tau = 128, \\ 7680, & \text{for } \tau = 192, \\ 15360, & \text{for } \tau = 256, \end{cases} \tag{6.2}$$

  but the NIST recommendations also define a minimal bit length for $\hat{q}$. For reasons similar to those defining the minimal output length of a collision-resistant hash function, the desired security strength τ must be doubled. This implies that $\|\hat{q}\| \geqslant 2\tau$ is the constraint to consider when choosing $\hat{q}$. Finally, an arbitrary generator $\hat{g} \in \mathbb{G}_{\hat{q}} \backslash \{1\}$ must be known to everyone.

- A *prime field* $\mathbb{Z}_{p'}$ is required in our protocol for polynomial interpolation during the vote confirmation process. The goal of working with polynomials is to prove the validity of a submitted vote in an efficient way. For maximal efficiency, we connect this proof to Schnorr's identification scheme in the vote confirmation process. This connection requires that the constraint for $\mathbb{G}_{\hat{q}}$ also apply to $\mathbb{Z}_{p'}$, i.e., we must consider $\|p'\| \geqslant 2\tau$ when choosing p'. Maximal simplicity can be reached by setting $p' = \hat{q}$. An additional parameter that follows directly from p' is the length L_M of the messages transferred by the OT-protocol. Since each of these messages represents a point in $\mathbb{Z}_{p'}^2$, we obtain $L_M = 2 \cdot \lceil \frac{\|p'\|}{8} \rceil$ bytes.

c) Parameters for Voting and Confirmation Codes

As we will see in Section 6.5.2, Schnorr's identification scheme is used twice in the vote casting and confirmation process. For this, voter i obtains a random pair of secret values $(x_i, y_i) \in \mathbb{Z}_{\hat{q}_x} \times \mathbb{Z}_{\hat{q}_y}$ in form of a pair of fixed-length strings $(X_i, Y_i) \in A_X^{\ell_X} \times A_Y^{\ell_Y}$, which are printed on the voting card. The values $\hat{q}_x \leqslant \hat{q}$ and $\hat{q}_y \leqslant \hat{q}$ are the upper bounds for x_i and y_i, respectively. If $|A_X| \geqslant 2$ and $|A_Y| \geqslant 2$ denote the sizes of corresponding alphabets, we can derive the string lengths of X_i and Y_i as follows:

$$\ell_X = \left\lceil \frac{\|\hat{q}_x\|}{\log_2 |A_X|} \right\rceil, \qquad \ell_Y = \left\lceil \frac{\|\hat{q}_y\|}{\log_2 |A_Y|} \right\rceil.$$

For reasons similar to the ones mentioned above, it is critical to choose values $\hat{q}_x$ and $\hat{q}_y$ satisfying $\|\hat{q}_x\| \geqslant 2\tau$ and $\|\hat{q}_y\| \geqslant 2\tau$ to guarantee the security of Schnorr's identification scheme. In the simplest possible case, i.e., by setting $\hat{q}_x = \hat{q}_y = \hat{q}$, all constraints are automatically satisfied. The selection of the alphabets A_X and A_Y is mainly a trade-off between conflicting usability parameters, for example the number of character versus the number of *different* characters to enter. Typical alphabets for such purposes are the sets $\{\mathtt{0},\ldots,\mathtt{9}\}$, $\{\mathtt{0},\ldots,\mathtt{9},\mathtt{A},\ldots,\mathtt{Z}\}$, $\{\mathtt{0},\ldots,\mathtt{9},\mathtt{A},\ldots,\mathtt{Z},\mathtt{a},\ldots,\mathtt{z}\}$, or other combinations of the most common characters. Each character will then contribute between 3 to 6 entropy bits to the entropy of x_i or y_i. While even larger alphabets may be problematical from a usability point of view, standardized word lists such as *Diceware*[1] are available in many natural languages. These lists have been designed for optimizing the quality of passphrases. In the English Diceware list, the average word length is 4.2 characters, and each word contributes approximately 13 entropy bits. With this, the values x_i and y_i would by represented by passphrases consisting of at least $\frac{2\tau}{13}$ English words.

d) Parameters for Verification and Finalization Codes

Other elements printed on the voting card of voter i are the verification codes RC_{ij} and the finalization code FC_i. Their purpose is the detection of attacks by corrupt voting clients. The length of these codes is therefore a function of the deterrence factor ϵ. They are generated in two steps, first as byte arrays R_{ij} of length L_R and F_i of length L_F, respectively, which are then converted into strings RC_{ij} of length ℓ_R and FC_i of length ℓ_F (for given alphabets A_R and A_F). To provide the security defined by the deterrence factor, the following general constraints must be satisfied:

$$8L_R \geqslant \log \frac{1}{1-\epsilon}, \qquad 8L_F \geqslant \log \frac{1}{1-\epsilon}.$$

For $\epsilon = 0.999$ (0.001 chance of an undetected attack), for example, $L_R = L_F = 2$ would be appropriate. In the case of the finalization code, the string length ℓ_F follows directly from L_F and the size of the alphabet A_F. For the verification codes, an additional usability constraint needs to be considered, namely that each code should appear at most once on each voting card. This problem can be solved by increasing the length of the byte arrays and to watermark them with $j-1 \in \{0,\ldots,n-1\}$ before converting them into a string (see Alg. 4.1). Note that this creates a minor technical problem, namely that L_R is no longer

[1] See http://world.std.com/~reinhold/diceware.html.

independent of the election parameters (see next subsection). We can solve this problem by defining $n_{\max}$ to be the maximal number of candidates in every possible election event and to extend the constraint for L_R into

$$8L_R \geqslant \log \frac{n_{\max} - 1}{1 - \epsilon}.$$

For $\epsilon = 0.999$ and $n_{\max} = 1000$, for example, $L_R = 3$ would satisfy this extended constraint. For given lengths L_R and L_F, we can calculate the lengths ℓ_R and ℓ_F of corresponding strings using the alphabet sizes:

$$\ell_R = \left\lceil \frac{8L_R}{\log_2 |A_R|} \right\rceil, \qquad \ell_F = \left\lceil \frac{8L_F}{\log_2 |A_F|} \right\rceil.$$

For $L_R = 3$, $L_F = 2$, and alphabet sizes $|A_R| = |A_F| = 64$ (6 bits), $\ell_R = 4$ characters are required for the verification codes and $\ell_F = 3$ characters for the finalization code.

6.3.2. Election Parameters

A second category of parameters defines the details of a concrete election event. Defining such *election parameters* is the responsibility of the election administrator. For making them accessible to every participating party, they are published on the bulletin board. This is the initial step of the election preparation phase (see Section 6.5.1). At the end of this subsection, Table 6.2 summarizes the list of all election parameters and constraints to consider when selecting them.

In Chapter 2, we already discussed that our definition of an election event, which constitutes of multiple simultaneous k-out-of-n elections over multiple counting circles, covers all election use cases in the given context. The most important parameters of an election event are therefore the number t of simultaneous elections and the number w of counting circles. By assuming $t \geqslant 1$ and $w \geqslant 1$, we exclude the meaningless limiting cases of an election event with no elections or no counting circles. Most other election parameters are directly or indirectly influenced by the actual values of t and w.

Different election events are distinguished by associating a unique *election event identifier* $U \in A^*_{\mathsf{ucs}}$. While the protocol is not designed to run multiple election events in parallel, it is important to strictly separate the election data of successive election events. By introducing a unique election event identifier and by adding it to every digital signature issued during the protocol execution (see Section 6.6), the data of a given election event is unanimously tied together. This is the main purpose of the election event identifier. To avoid that the data of multiple elections is inadvertently tied together when the same identifier U is used multiple times, we assume U to contain enough information (e.g., the date of the election day) to allow participating parties to judge whether U is a fresh value or not.

a) Candidates

Let $n_j \geqslant 2$ denote the number of candidates in the j-th election of an election event. By requiring at least two candidates, we exclude trivial or meaningless elections with $n = 1$ or $n = 0$ candidates. The sum of such values, $n = \sum_{j=1}^{t} n_j$, represents the total number

Parameters		Constraints
L	Output length of hash function (bytes)	$L \geqslant \frac{\max(\sigma,\tau)}{4}$
p	Modulo of encryption group $\mathbb{G}_q$	see (6.1)
g, h	Independent generators of $\mathbb{G}_q$	$g, h \in \mathbb{G}_q \backslash 1$
$\hat{p}$	Modulo of identification group $\mathbb{G}_{\hat{q}}$	see (6.2)
$\hat{q}$	Order of $\mathbb{G}_{\hat{q}}$	$\\|\hat{q}\\| \geqslant 2\tau$
$\hat{g}$	Generator of $\mathbb{G}_{\hat{q}}$	$g \in \mathbb{G}_{\hat{q}} \backslash 1$
p'	Modulo of prime field $\mathbb{Z}_{p'}$	$\\|p'\\| \geqslant 2\tau$
L_M	Length of OT messages (bytes)	$L_M = 2 \cdot \lceil \frac{\\|p'\\|}{8} \rceil$
$\hat{q}_x$	Upper bound of secret voting credential x	$\\|\hat{q}_x\\| \geqslant 2\tau$, $\hat{q}_x \leqslant \hat{q}$
A_X	Voting code alphabet	$\|A_X\| \geqslant 2$
ℓ_X	Length of voting codes (characters)	$\ell_X = \left\lceil \frac{\\|\hat{q}_x\\|}{\log_2 \|A_X\|} \right\rceil$
$\hat{q}_y$	Upper bound of secret confirmation credential y	$\\|\hat{q}_y\\| \geqslant 2\tau$, $\hat{q}_y \leqslant \hat{q}$
A_Y	Confirmation code alphabet	$\|A_Y\| \geqslant 2$
ℓ_Y	Length of confirmation codes (characters)	$\ell_Y = \left\lceil \frac{\\|\hat{q}_y\\|}{\log_2 \|A_Y\|} \right\rceil$
$n_{\max}$	Maximal number of candidates	$n_{\max} \geqslant 2$
L_R	Length of verification codes R_{ij} (bytes)	$8L_R \geqslant \log \frac{n_{\max}-1}{1-\epsilon}$
A_R	Verification code alphabet	$\|A_R\| \geqslant 2$
ℓ_R	Length of verification codes RC_{ij} (characters)	$\ell_R = \left\lceil \frac{8L_R}{\log_2 \|A_R\|} \right\rceil$
L_F	Length of finalization codes F_i (bytes)	$8L_F \geqslant \log \frac{1}{1-\epsilon}$
A_F	Finalization code alphabet	$\|A_F\| \geqslant 2$
ℓ_F	Length of finalization codes FC_i (characters)	$\ell_F = \left\lceil \frac{8L_F}{\log_2 \|A_F\|} \right\rceil$

Table 6.1.: List of security parameters derived from the principal security parameters σ, τ, and ϵ. We assume that these values are fixed and publicly known to every party participating in the protocol.

of candidates in an election event. For each such candidate $i \in \{1, \ldots, n\}$, a *candidate description* $C_i \in A_{\mathsf{ucs}}^*$ must be provided. In this document, by assuming that candidate descriptions are given as arbitrary UCS strings, we do not further specify the type and format of the information given for each candidate. Other important parameters of an election event are the numbers of candidates k_j, $0 < k_j < n_j$, which a voter can select in each election j. We exclude the two meaningless limiting cases of $k_j = 0$ and $k_j = n_j$. The total number of selections over all elections, $k = \sum_{j=1}^{t} k_j$, is limited by a constraint that follows from our particular vote encoding method (see Section 6.4.1).

b) Electorate

A second category of election parameters specifies the details of the electorate. With $N_E \geqslant 0$ we denote the number of eligible voters in an election event and use $i \in \{1, \ldots, N_E\}$ as identifier.[2] For each voter i, a *voter description* $V_i \in A^*_{\mathsf{ucs}}$ and a counting circle $w_i \in \{1,\ldots,w\}$ must be provided. As for the candidate descriptions, we do not further specify the type and format of the given information. Note that in the given election use cases of Section 2.2, voter i is not automatically eligible in every election of an election event. We use single bits $e_{ij} \in \mathbb{B}$ to define whether voter i is eligible in election j or not, and we exclude completely ineligible voters by $\sum_{j=1}^{t} e_{ij} > 0$. The matrix $\mathbf{E} = (e_{ij})_{N_E \times t}$ of all such values is called *eligibility matrix*.

Parameters		Constraints
U	Unique election event identifier	$U \in A^*_{\mathsf{ucs}}$
t	Number of elections	$t \geqslant 1$
w	Number of counting circles	$w \geqslant 1$
$\mathbf{n} = (n_1,\ldots,n_t)$	Number of candidates in each election	$n_j \geqslant 2$
n	Total number of candidates	$n = \sum_{j=1}^{t} n_j$
$\mathbf{c} = (C_1,\ldots,C_n)$	Candidate descriptions	$C_i \in A^*_{\mathsf{ucs}}$
$\mathbf{k} = (k_1,\ldots,k_t)$	Number of selections in each election	$0 < k_j < n_j$
k	Total number of selections	$k = \sum_{j=1}^{t} k_j$, $p_{n+w} \prod_{j=1}^{k} p_{n-j+1} < q$
N_E	Number of eligible voters	$N_E \geqslant 0$
$\mathbf{v} = (V_1,\ldots,V_{N_E})$	Voter descriptions	$V_i \in A^*_{\mathsf{ucs}}$
$\mathbf{w} = (w_1,\ldots,w_{N_E})$	Assigned counting circles	$w_i \in \{1,\ldots,w\}$
$\mathbf{E} = (e_{ij})_{N_E \times t}$	Eligibility matrix	$e_{ij} \in \mathbb{B}$, $\sum_{j=1}^{t} e_{ij} \geqslant 1$

Table 6.2.: List of election parameters.

6.4. Technical Preliminaries

From a cryptographic point of view, our protocol exploits a few non-trivial technical tricks. In order to facilitate the exposition of the protocol in the next section, we introduce them beforehand. Some of them have been used in other cryptographic voting protocols and are well documented.

6.4.1. Encoding of Votes and Counting Circles

In an election that allows votes for multiple candidates, it is usually more efficient to incorporate all votes into a single encryption. In the case of the ElGamal encryption scheme with $\mathbb{G}_q$ as message space, we must define an invertible mapping Γ from the set of all possible

[2]Related election parameters will be formed during vote casting and confirmation. The number of submitted ballots will be denoted by $N_B \leqslant N_E$, the number of confirmed ballots by $N_C \leqslant N_B$, and the number of valid votes by $N \leqslant N_C$.

votes into $\mathbb{G}_q$. A common technique for encoding a selection $\mathbf{s} = (s_1, \ldots, s_k)$ of k candidates out of n candidates, $1 \leqslant s_j \leqslant n$, is to encode each selection s_j by a prime number $\Gamma(s_j) \in \mathbb{P} \cap \mathbb{G}_q$ and to multiply them into $\Gamma(\mathbf{s}) = \prod_{j=1}^{k} \Gamma(s_j)$. Inverting $\Gamma(\mathbf{s})$ by factorization is unique as long as $\Gamma(\mathbf{s}) < q$ and efficient when n is small [25]. For optimal capacity, we choose the n smallest prime numbers $p_1, \ldots, p_n \in \mathbb{P} \cap \mathbb{G}_q$, $p_i < p_{i+1}$, and define $\Gamma(s_j) = p_{s_j}$ for $j \in \{1, \ldots, k\}$.

Since each encrypted votes is attributed to a counting circle, we extend the above invertible mapping $\Gamma : \{1, \ldots, n\}^k \to \mathbb{G}_q$ into $\Gamma' : \{1, \ldots, n\}^k \times \{1, \ldots, w\} \to \mathbb{G}_q$ by considering the w next smallest prime numbers $p_{n+1}, \ldots, p_{n+w} \in \mathbb{P} \cap \mathbb{G}_q$. A selection $\mathbf{s}$ and a counting circle $w_i \in \{1, \ldots, w\}$ can then be encoded together as $\Gamma'(\mathbf{s}, w_i) = p_{n+w_i} \cdot \Gamma(\mathbf{s})$. This mapping is invertible, if the product of p_{n+w} with the k largest primes $p_{n-k+1}, \ldots, p_n$ is smaller than q, i.e., if $p_{n+w} \prod_{j=1}^{k} p_{n-j+1} < q$. This is an important constraint when choosing the security and election parameters of an election event (see Table 6.2 in Section 6.3). Note that in this way, due to the homomorphic property of ElGamal, assigning a counting circle w_i to an encoded vote can also be conducted under encryption: let $(a, b) = \mathsf{Enc}_{pk}(\Gamma(\mathbf{s}), r)$ be an ElGamal encryption of $\Gamma(\mathbf{s})$, then $(p_{n+w_i} \cdot a, b) = \mathsf{Enc}_{pk}(p_{n+w_i}, 0) \cdot \mathsf{Enc}_{pk}(\Gamma(\mathbf{s}), r) = \mathsf{Enc}_{pk}(\Gamma'(\mathbf{s}, w_i), r)$ is an ElGamal encryption of $\Gamma'(\mathbf{s}, w_i)$. We will use this property in the protocol to assign in a verifiable manner the counting circles to the encrypted votes before processing them through the mix-net.

6.4.2. Linking OT Queries to ElGamal Encryptions

If the same encoding $\Gamma : \{1, \ldots, n\} \to \mathbb{G}_q$ is used for the $\mathrm{OT}^{\mathbf{k}}_{\mathbf{n}}$-scheme (see Section 5.3.3) and for encoding plaintext votes, we obtain a natural link between an OT query $\mathbf{a} = (a_1, \ldots, a_k)$ and an ElGamal encryption $(a, b) \leftarrow \mathsf{Enc}_{pk}(\Gamma(\mathbf{s}), r)$. The link arises by substituting the first generator g_1 in the OT-scheme with the public encryption key $pk = g^{sk} \bmod p$ and the second generator g_2 by g. In this case, we obtain $a_j = (\Gamma(s_j) {\cdot} pk^{r_j}, g^{r_j})$ and therefore $a = \prod_{j=1}^{k} a_j = (\Gamma(\mathbf{s}) {\cdot} pk^r, g^r)$ for $r = \sum_{j=1}^{k} r_j$. This simple technical link between the OT query and the encrypted vote is crucial for making our protocol efficient [27]. It means that submitting $\mathbf{a}$ as part of the ballot solves two problems at the same time: sending an OT query and an encrypted vote to the election authorities and guaranteeing that they contain exactly the same selection of candidates.

6.4.3. Validity of Encrypted Votes

The main purpose of the verification codes in our protocol is to provide evidence to the voters that their votes have been cast and recorded as intended. However, our way of constructing the verification codes solves another important problem, namely to guarantee that every submitted encrypted vote satisfies exactly the constraints given by the election parameters $\mathbf{k}$, $\mathbf{n}$, and $\mathbf{E}$, i.e., that every encryption contains a valid vote. Let $RC_1, \ldots, RC_n \in A_R^{\ell_R}$ be the verification codes for the $n = \sum_{j=1}^{t} n_j$ candidates of a given voting card. In our scheme, they are constructed as follows [27]:

- For every voter $i \in \{1, \ldots, N_E\}$ and $k'_i = \sum_{j=1}^{t} e_{ij} k_j$, each authority picks a random polynomial $A_i(X) \in_R \mathbb{Z}_{p'}[X]$ of degree $k'_i - 1$. From this polynomial, the authority selects n random points $p_{ij} = (x_{ij}, A_i(x_{ij}))$ by picking n distinct random values

$x_{ij} \in_R \mathbb{Z}_{p'}$. The result is a vector of points, $\mathbf{p}_i = (p_{i,1}, \dots, p_{i,n})$, of length n. Over all N_E voting cards, each authority generates a matrix $(p_{ij})_{N_E \times n}$ of such points. Computing this matrix is part of the election preparation of every election authority. In the remaining of this document, the matrix generated by authority j will be denoted by $\mathbf{P}_j$.

- During vote casting, every authority transfers exactly k'_i points from $\mathbf{P}_j$ obliviously to the voting client of voter i, i.e., the voting client receives a matrix $\mathbf{P_s} = (p_{ij})_{s \times k'_i}$ of such points, which depends on the voter's selection $\mathbf{s}$. The verification code RC_{s_j} for the selected candidate s_j is derived from the points $p_{1,j}, \dots, p_{s,j}$ by truncating corresponding hash values $h(p_{ij})$ to the desired length L_R, combining them with an exclusive-or into a single value, and finally converting this value into a string RC_{s_j} of length ℓ_R. The same happens simultaneously for all of the voter's k'_i selections, which leads to a vector $\mathbf{rc_s} = (RC_{s_1}, \dots, RC_{s_{k'_i}})$. During the printing of the voting card, exactly the same calculations are performed for the verification codes of all n candidates.
- By obtaining k'_i points from a particular election authority, the voting client can reconstruct the polynomial $A_i(X)$ of degree $k'_i - 1$, if at least k'_i distinct points from $A_i(X)$ are available (see Section 3.2.2). If this is the case, the simultaneous $\mathrm{OT}_{\mathbf{n}}^{\mathbf{k}'_i}$ query must have been formed properly under the constraints given by $\mathbf{n}$, $\mathbf{k}$, and $\mathbf{E}$. The voting client can therefore prove the validity of the encrypted vote by proving knowledge of this polynomial. For this, it evaluates the polynomial for $X = 0$ to obtain a *secret vote validity credential* $y'_i = A_i(0)$, which can not be guessed efficiently without knowing the polynomial. In this way, the voting client obtains a secret vote validity credential y'_{ij} from every authority j. Their integer sum $y'_i = \sum_{j=1}^{s} y'_{ij}$ is incorporated into the voter's public confirmation credential $\hat{y}$ by adding it to the secret confirmation credential y_i derived from the conformation code Y_i (see next subsection). Knowing correct values y'_{ij} is therefore a prerequisite for the voting client to successfully confirm the vote (see following subsection).

The finalization code $FC \in A_F^{\ell_F}$ of a given voting card is also derived from the random points generated by each authority. The procedure is similar to the generation of the verification codes. First, election authority j computes the hash value of the voter's n points in $\mathbf{P}_j$ and truncates it to the desired length L_F. The resulting s hash values—one from every authority—are combined with an exclusive-or into a single value, which is then converted into a string of length ℓ_F. These last steps are the same for the printing authority during the election preparation and for the voting client at the end of the vote casting process.

6.4.4. Voter Identification

During the vote casting process, the voter needs to be identified twice as an eligible voter, first to submit the initial ballot and to obtain corresponding verification codes, and second to confirm the vote after checking the verification codes. A given voting card contains two secret codes for this purpose, the voting code $X \in A_X^{\ell_X}$ and the confirmation code $Y \in A_Y^{\ell_Y}$. By entering these codes into the voting client, the voter expresses the intention to proceed to the next step in the vote casting process. In both cases, a Schnorr identification is performed between the voting client and the election authorities (see Section 5.4.2). Without entering these codes, or by entering incorrect codes, the identification fails and the process stops.

The voting code X is a string representation of a secret value $x \in \mathbb{Z}_{\hat{q}}$ called *secret voting credential*. This value is generated by the election authorities in a distributed way, such that no one except the printing authority learns it. For this, each election authority contributes a random value $x_j \in_R \mathbb{Z}_{\hat{q}}$, which the printing authority combines into $x = \sum_{j=1}^{s} x_j \bmod \hat{q}$. The corresponding *public voting credential* $\hat{x} \in \mathbb{G}_{\hat{q}}$ is derived from the values $\hat{x}_j = \hat{g}^{x_j} \bmod \hat{p}$, which are published by the election authorities:

$$\hat{x} = \prod_{j=1}^{s} \hat{x}_j \bmod \hat{p} = \prod_{j=1}^{s} \hat{g}^{x_j} \bmod \hat{p} = \hat{g}^{\sum_{j=1}^{s} x_j} \bmod \hat{p} = \hat{g}^{x} \bmod \hat{p}.$$

For a given pair $(x, \hat{x}) \in \mathbb{Z}_{\hat{q}} \times \mathbb{G}_{\hat{q}}$ of secret and public voting credentials, executing the Schnorr identification protocol corresponds to computing a non-interactive zero-knowledge proof $\mathit{NIZKP}[(x) : \hat{x} = \hat{g}^x \bmod \hat{p}]$. In our protocol, we combine this proof with a proof of knowledge of the plaintext vote contained in the submitted ballot (see Section 5.4.2).

The generation of the confirmation code Y is very similar. It is a string representation of the *secret confirmation credential* $y \in \mathbb{Z}_{\hat{q}}$, which is generated by the election authorities in exactly the same way as x. However, for the corresponding *public confirmation credential* $\hat{y} \in \mathbb{G}_{\hat{q}}$, the method is slightly different. After picking $y_j \in_R \mathbb{Z}_{\hat{q}}$ at random, the authority computes $\hat{y}_j = \hat{g}^{y_j + y'_j} \bmod \hat{p}$, where y'_j denotes the vote validity credential from the previous subsection. The public credential can be computed by

$$\hat{y} = \prod_{j=1}^{s} \hat{y}_j \bmod \hat{p} = \prod_{j=1}^{s} \hat{g}^{y_j + y'_j} \bmod \hat{p} = \hat{g}^{\sum_{j=1}^{s} y_j + \sum_{j=1}^{s} y'_j} \bmod \hat{p} = \hat{g}^{y+y'} \bmod \hat{p},$$

for $y = \sum_{j=1}^{s} y_j \bmod \hat{q}$ and $y' = \sum_{j=1}^{s} y'_j \bmod \hat{q}$. Therefore, performing a Schnorr identification relative to $\hat{y}$ requires knowledge of $y + y'$. The corresponding zero-knowledge proof, $\mathit{NIZKP}[(y, y') : \hat{y} = \hat{g}^{y+y'} \bmod \hat{p}]$, is more efficient than conducting a conjunction of two separate proofs for y and y'.

6.5. Protocol Description

Based on the preceding sections about parties, channels, adversaries, trust assumptions, system parameters, and technical preliminaries, we are now ready to present the cryptographic protocol in greater detail. As mentioned earlier, the protocol itself has three phases, which we describe in corresponding subsections with sufficient technical details for understanding the general protocol design. By exhibiting the involved parties in each phase and sub-phase, a first overview of the protocol is given in Table 6.3. This overview illustrates the central role of the bulletin board as a communication hub and the strong involvement of the election authorities in almost every step of the whole process.

In each of the following subsections, we provide comprehensive illustrations of corresponding protocol sub-phases. The illustrations are numbered from Prot. 6.3 to Prot. 6.9. Each illustration depicts the involved parties, the necessary information known to each party prior to executing the protocol sub-phase, the computations performed by each party during the

Phase	Election Admin.	Election Authority	Printing Authority	Voter	Voting Client	Bulletin Board	Protocol Nr.
1. Pre-Election	•	•	•	•		•	
1.1 Election Preparation	•	•				•	6.1
1.2 Printing of Voting Cards		•	•	•		•	6.2
1.3 Key Generation		•				•	6.3
2. Election		•		•	•	•	
2.1 Candidate Selection				•	•	•	6.4
2.2 Vote Casting		•			•	•	6.5
2.3 Vote Confirmation		•		•	•	•	6.6
3. Post-Election	•	•				•	
3.1 Mixing		•				•	6.7
3.2 Decryption		•				•	6.8
3.3 Tallying	•					•	6.9

Table 6.3.: Overview of the protocol phases and sub-phases with the involved parties.

protocol sub-phase, and the exchanged messages. Together, these illustration define a precise and complete skeleton of the entire protocol. The details of the algorithms called by the parties when performing their computations are given in Chapter 7. Note that the illustrations do not show the signatures that are generated by the election administrator and the election authorities. These signatures are important to provide authenticity, i.e., they must be generated whenever a message is sent to the bulletin board and verified whenever a message is retrieved from there. As already discussed in Section 6.3.2, a unique election event identifier U is included in every signature. The distribution of U is included in the protocol illustrations, but other details of the signature generation are discussed in Section 6.6. Corresponding algorithms are given in Section 7.6.

6.5.1. Pre-Election Phase

The pre-election phase of the protocol involves all necessary tasks to setup an election event. The main goal is to equip each eligible voter with a personalized voting card, which we identify with an index $i \in \{1, \dots, N_E\}$. Without loss of generality, we assume that voting card i is sent to voter i. We understand a voting card as a string $S_i \in A_{\mathsf{ucs}}^*$, which is printed on paper by the printing authority. This string contains the voter index i, the voter description $V_i \in A_{\mathsf{ucs}}^*$, the counting circle $w_i \in \{1, \dots, w\}$, the voting code $X_i \in A_X^{\ell_X}$, the confirmation code $Y_i \in A_Y^{\ell_Y}$, the finalization code $FC_i \in A_F^{\ell_F}$, and the candidate descriptions $C_j \in A_{\mathsf{ucs}}^*$ with corresponding verification codes $RC_{ij} \in A_R^{\ell_R}$ for each candidate $j \in \{1, \dots, n\}$. The information printed on voting card i is therefore a tuple

$$(i, V_i, w_i, X_i, Y_i, FC_i, \{(C_j, RC_{ij})\}_{j=1}^n).$$

a) Election Preparation

The codes printed on the voting cards are generated by the s election authorities in a distributed manner (see Sections 6.4.2 and 6.4.3 for technical background). For this, each election authority j calls an algorithm $\mathsf{GenElectorateData}(\mathbf{n}, \mathbf{k}, \mathbf{E})$ with the election parameters $\mathbf{n}$, $\mathbf{k}$, and $\mathbf{E}$, which are published beforehand by the election administrator. The result obtained from calling this algorithm consists of a private part $\mathbf{d}_j$, a public part $\hat{\mathbf{d}}_j$, and the matrix of random points $\mathbf{P}_j$. Further details of the algorithm are given in Alg. 7.6. These first steps are depicted in the upper part of Prot. 6.1.

The public part $\hat{\mathbf{d}}_j$, which contains the authority's partial information for deriving the public voter credentials $\hat{x}_i$ and $\hat{y}_i$, is submitted via the bulletin board to all other election authorities. At the end of this process, every election authority knows the public data of the whole electorate, $\hat{\mathbf{D}} = (\hat{\mathbf{d}}_1, \ldots, \hat{\mathbf{d}}_s)$, which they can use for calling $\mathsf{GetPublicCredentials}(\hat{\mathbf{D}})$. This algorithm outputs the two lists $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ of all public credentials, which are used to identify the voters during the vote casting and vote confirmation phases (see Section 6.4.4 and Alg. 7.12 for further details).

Election Administrator		Bulletin Board		Election Authority $j \in \{1, \ldots, s\}$
knows $U, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}$				
	$\xrightarrow{U, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}}$			
			$\xrightarrow{U, \mathbf{n}, \mathbf{k}, \mathbf{E}}$	
				$(\mathbf{d}_j, \hat{\mathbf{d}}_j, \mathbf{P}_j) \leftarrow \mathsf{GenElectorateData}(\mathbf{n}, \mathbf{k}, \mathbf{E})$
			$\xleftarrow{\hat{\mathbf{d}}_j}$	
		$\hat{\mathbf{D}} \leftarrow (\hat{\mathbf{d}}_1, \ldots, \hat{\mathbf{d}}_s)$		
			$\xrightarrow{\hat{\mathbf{D}}}$	
				$(\hat{\mathbf{x}}, \hat{\mathbf{y}}) \leftarrow \mathsf{GetPublicCredentials}(\hat{\mathbf{D}})$

Protocol 6.1: Election Preparation.

b) Printing of Code Sheets

The private part $\mathbf{d}_j$ of the electorate data generated by authority j contains the authority's partial information about the secret voting, confirmation, finalization, and verification codes of every voting card. This information is very sensitive and can only be shared with the printing authority. The process of sending $\mathbf{d}_j$ to the printing authority is depicted in Prot. 6.2. Recall that this channel is confidential, i.e., it must be secured by cryptographic means. This can be achieved by sending $\mathbf{d}_j$ in encrypted form using the key-encapsulation mechanism in combination with a symmetric encryption scheme as described in Section 5.7. We denote the resulting ciphertext, which results from calling $\mathsf{GenCiphertext}_\phi(pk_{\mathsf{Print}}, \mathbf{d}_j)$ using the printing authority's public encryption key pk_{Print}, by $[\mathbf{d}_j]$. Using the corresponding private key sk_{Print}, the printing authority can then call $\mathsf{GetPlaintext}_\phi(sk_{\mathsf{Print}}, [\mathbf{d}_j])$ to decrypt $[\mathbf{d}_j]$ into $\mathbf{d}_j$. See Alg. 7.56 and Alg. 7.57 for further details.

The actual voting cards can be generated from the collected private data $\mathbf{D} \leftarrow (\mathbf{d}_1, \ldots, \mathbf{d}_s)$ and the elections parameters $\mathbf{v}$, $\mathbf{w}$, $\mathbf{c}$, $\mathbf{n}$, $\mathbf{k}$, and $\mathbf{E}$. The printing authority uses them as inputs for the algorithm $\mathsf{GetVotingCards}(\mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{D})$, which produces corresponding strings $\mathbf{s} = (S_1, \ldots, S_{N_E})$, $S_i \in A_{\mathsf{ucs}}^*$ (see Alg. 7.13). A printout of such a string is sent to every voter, for example using a trusted postal service.

Bulletin Board		Printing Authority		Election Authority $j \in \{1, \ldots, s\}$
knows $U, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}$				knows $U, \mathbf{d}_j$
	$\xrightarrow{U, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}}$			
			$\xleftarrow{[\mathbf{d}_j]}$	
		$\mathbf{D} \leftarrow (\mathbf{d}_1, \ldots, \mathbf{d}_s)$ $\mathbf{s} \leftarrow \mathsf{GetVotingCards}(\mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{D})$		
Voter $i \in \{1, \ldots, N_E\}$				
	$\xleftarrow{S_i}$			

Protocol 6.2: Printing of Voting Cards.

c) Key Generation

In the last step of the election preparation, a public ElGamal encryption key $pk \in \mathbb{G}_q$ is generated jointly by the election authorities. As shown in Prot. 6.3, this is a simple process between the election authorities and the bulletin board. At the end of the protocol, pk is known to every authority, and each of them holds a share $sk_j \in \mathbb{Z}_q$ of the corresponding private key. It involves calls to two algorithms $\mathsf{GenKeyPair}()$ for generating the key shares and $\mathsf{GetPublicKey}(\mathbf{pk})$ for combining the resulting public keys. For details of these algorithms, we refer to Section 5.1.2 and Algs. 7.15 and 7.16.

6.5.2. Election Phase

The election phase is the core of the cryptographic voting protocol. The start and end of this phase are given by the official election period. These are two very critical events in every election. To prevent or detect the submission of early or late votes, it is very important to handle these events accurately. Since there are multiply ways for dealing with this problem, we do not propose a solution in this document. We only assume that the bulletin board and the election authorities will always agree whether a particular vote (or vote confirmation) has been submitted within the election period, and only accept it if this is the case.

Bulletin Board		Election Authority $j \in \{1,\dots,s\}$
		knows U
		$(sk_j, pk_j) \leftarrow \mathsf{GenKeyPair}()$
	$\xleftarrow{\quad pk_j \quad}$	
$\mathbf{pk} \leftarrow (pk_1,\dots,pk_s)$		
	$\xrightarrow{\quad \mathbf{pk} \quad}$	
		$pk \leftarrow \mathsf{GetPublicKey}(\mathbf{pk})$

Protocol 6.3: Key Generation

The main actors of the election phase are the voters and the election authorities, which communicate over the bulletin board. The main goal of the voters is to submit a valid vote for the selected candidates using the untrusted voting client, whereas the goal of the election authorities is to collect all valid votes from eligible voters. The submission of a single vote takes place in three subsequent steps.

a) Candidate Selection

The first step for the voter is the selection of the candidates. In an election event with t simultaneous elections, voter v must select exactly $e_{vj}k_j$ candidates for each election $j \in \{1,\dots,t\}$ and $k'_v = \sum_{j=1}^{t} e_{vj}k_j$ candidates in total. These values can be derived from the election parameters $\mathbf{k}$ and $\mathbf{E}$, which the voting client retrieves from the bulletin board together with the candidate descriptions $\mathbf{c}$ and the number of candidates $\mathbf{n}$. This preparatory step is shown in the upper part of Prot. 6.4. By calling $\mathsf{GetVotingPage}(v, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E})$, the voting client then generates a *voting page* $P_v \in A^*_{\mathsf{ucs}}$, which represents the visual interface displayed to voter v for selecting the candidates (see Alg. 7.17). The voter's selection $\mathbf{s} = (s_1,\dots,s_{k'_v})$ is a vector of values s_j satisfying the constraint in (5.1) from Section 5.3.3. The voter enters these values together with the voting code X_v from the voting card.

b) Vote Casting

Based on the voter's selection $\mathbf{s} = (s_1,\dots,s_k)$, the voting client generates a ballot $\alpha = (\hat{x}_v, \mathbf{a}, \pi)$ by calling an algorithm $\mathsf{GenBallot}(X_v, \mathbf{s}, pk)$. The ballot contains an $\mathrm{OT}^{\mathbf{k}}_{\mathbf{n}}$ query $\mathbf{a} = (a_1,\dots,a_k) \in \mathbb{G}_q^k$ for corresponding return codes. By using the public encryption key pk in the oblivious transfer as a generator of the group $\mathbb{G}_q$ (see Section 6.4.2), each query a_j is an ElGamal encryption of the voter's selection s_j. The ballot α also contains the voter's public credential $\hat{x}_v$, which is derived from the secret voting code X_v, and a non-interactive zero-knowledge proof

$$\pi_\alpha = NIZKP[(x_v, \mathbf{s}, r) : \hat{x}_v = \hat{g}^{x_v} \bmod \hat{p} \wedge \prod_{j=1}^{k} a_j = \mathsf{Enc}_{pk}(\Gamma(\mathbf{s}), r)],$$

Voter $v \in \{1, \ldots, N_E\}$		Voting Client		Bulletin Board
knows v, X_v				knows $U, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}$
			$\xleftarrow{U, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}}$	
	$\xrightarrow{v}$			
		$P_v \leftarrow \mathsf{GetVotingPage}(v, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E})$		
	$\xleftarrow{P_v}$			
$\mathbf{s} \leftarrow (s_1, \ldots, s_k)$				
	$\xrightarrow{X_v, \mathbf{s}}$			

Protocol 6.4: Candidate Selection

that demonstrates the well-formedness of the ballot. This proof includes all elements of a Schnorr identification relative to $\hat{x}_v$ (see Section 6.4.4).

The ballot is submitted to the election authorities via the bulletin board. Each authority checks its validity by calling $\mathsf{CheckBallot}(v, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B)$. This algorithm verifies that the size of $\mathbf{a}$ is exactly $k'_v = \sum_{j=1}^{t} e_{vj} k_j$, that the public voting credential $\hat{x}_v$ is included in $\hat{\mathbf{x}}$, that the zero-knowledge proof π_α is valid (which implies that the voter is in possession of a valid voting code X_v), and that the same voter has not submitted a valid ballot before. To detect multiple ballots from the same voter, each authority keeps track of a list B_j of valid ballots submitted so far. If one of the above checks fails, the ballot is rejected and the process aborts.

If the ballot α passes all checks, the election authorities respond to the OT query $\mathbf{a}$ included in α. Each of them computes its OT response β_j by calling $\mathsf{GenResponse}(v, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_j)$. The selected points from the matrix $\mathbf{P}_j$ are the messages to transfer obliviously to the voter via the bulletin board (see Section 6.4.3). By calling $\mathsf{GetPointMatrix}(\boldsymbol{\beta}, \mathbf{s}, \mathbf{r})$ for $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_s)$, the voting client derives the s-by-k matrix $\mathbf{P_s}$ of selected points from every β_j. Finally, by calling $\mathsf{GetReturnCodes}(\mathbf{s}, \mathbf{P_s})$, it computes the verification codes $\mathbf{rc_s} = (RC_{s_1}, \ldots, RC_{s_k})$ for the selected candidates. This whole procedure is depicted in Prot. 6.5.

c) Vote Confirmation

The voting client displays the verification codes $\mathbf{rc_s} = (RC_{s_1}, \ldots, RC_{s_k})$ for the selected candidates to the voter for comparing them with the codes $\mathbf{rc}_v$ printed on voter v' voting card. We describe this process by an algorithm call $\mathsf{CheckReturnCodes}(\mathbf{rc}_v, \mathbf{rc_s}, \mathbf{s})$, which is executed by the human voter. In case of a match, the voter enters the confirmation code Y_v, from which the voting client computes the *confirmation* $\gamma = (\hat{y}_v, \pi_\beta)$ consisting of the voter's public confirmation credential $\hat{y}_v$ and a non-interactive zero-knowledge proof

$$\pi_\beta = NIZKP[(y_v, y'_v) : \hat{y}_v = \hat{g}^{y_v + y'_v} \bmod \hat{p}].$$

Voting Client		Bulletin Board		Election Authority $j \in \{1, \ldots, s\}$
knows $U, v, X_v, \mathbf{s}$		knows $\mathbf{pk}$		knows $U, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_j, \hat{\mathbf{x}}, B_j \leftarrow \langle\rangle$
	$\xleftarrow{\mathbf{pk}}$			
$pk \leftarrow \mathsf{GetPublicKey}(\mathbf{pk})$ $(\alpha, \mathbf{r}) \leftarrow \mathsf{GenBallot}(X_v, \mathbf{s}, pk)$				
	$\xrightarrow{v, \alpha}$			
			$\xrightarrow{v, \alpha}$	
				if $\neg\mathsf{CheckBallot}(v, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B)$ **abort** $(\beta_j, z) \leftarrow \mathsf{GenResponse}(v, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_j)$ $B_j \leftarrow B_j \,\|\, \langle(v, \alpha, z)\rangle$
			$\xleftarrow{v, \beta_j}$	
		$\boldsymbol{\beta} = (\beta_1, \ldots, \beta_s)$		
	$\xleftarrow{\boldsymbol{\beta}}$			
$\mathbf{P_s} \leftarrow \mathsf{GetPointMatrix}(\boldsymbol{\beta}, \mathbf{s}, \mathbf{r})$ $\mathbf{rc_s} \leftarrow \mathsf{GetReturnCodes}(\mathbf{s}, \mathbf{P_s})$				

Protocol 6.5: Vote Casting

In this way, the voting client proves knowledge of a sum $y_v + y'_v$ of values y_v (derived from Y_v) and y'_v (derived from $\mathbf{P_s}$). The motivation and details of this particular construction have been discussed in Section 6.4.4.

After submitting γ via the bulletin board to every authority, they check the validity of the zero-knowledge proof included. In the success case, they respond with their *finalization* $\delta_j = (F_{vj}, z_{vj})$. The voting client retrieves the finalization code FC from the values $(F_{v,1}, \ldots, F_{v,s})$ included in $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_s)$ by calling $\mathsf{GetFinalizationCode}(\boldsymbol{\delta})$ and displays it to the voter for comparison. As above, we describe this process by an algorithm call $\mathsf{CheckFinalizationCode}(FC_v, FC)$ executed by the human voter. The whole process is depicted in Prot. 6.6. Note that the randomizations $(z_{v,1}, \ldots, z_{v,s})$ included in $\boldsymbol{\delta}$ are not needed for computing the finalization code. But their publication enables the verification of the OT responses by external verifiers [27].

6.5.3. Post-Election Phase

In the post-election phase, all $N \leqslant N_E$ submitted and confirmed ballots are processed through a mixing and decryption process. The main actors are the election authorities, which perform the mixing in a serial and the decryption in a parallel process. For the decryption, they require their shares sk_j of the private encryption key, which the have generated during the pre-election phase. Before applying their key shares to the output of the mixing, they verify all previous steps by checking the validity of every ballot collected during the election phase and the correctness of the shuffle proofs. In addition to performing the decryption,

Voter $v \in \{1, \dots, N_E\}$		Voting Client		Bulletin Board		Election Authority $j \in \{1, \dots, s\}$
knows $Y_v, FC_v, \mathbf{rc}_v, \mathbf{s}$		knows $U, v, \mathbf{P}_\mathbf{s}, \mathbf{rc}_\mathbf{s}$				knows $U, \mathbf{P}_j, \hat{\mathbf{y}}, B_j \leftarrow \langle\rangle, C_j \leftarrow \langle\rangle$
	$\xleftarrow{\mathbf{rc}_\mathbf{s}}$					
if $\neg\mathsf{CheckReturnCodes}(\mathbf{rc}_v, \mathbf{rc}_\mathbf{s}, \mathbf{s})$ **abort**						
	$\xrightarrow{Y_v}$					
		$\gamma \leftarrow \mathsf{GenConfirmation}(Y_v, \mathbf{P}_\mathbf{s})$				
			$\xrightarrow{v, \gamma}$			
					$\xrightarrow{v, \gamma}$	
						if $\neg\mathsf{CheckConfirmation}(v, \gamma, \hat{\mathbf{y}}, B_j, C_j)$ **abort**
						$\delta_j \leftarrow \mathsf{GetFinalization}(v, \mathbf{P}_j, B_j)$
						$C_j \leftarrow C_j \,\Vert\, \langle(v, \gamma)\rangle$
					$\xleftarrow{v, \delta_j}$	
				$\boldsymbol{\delta} \leftarrow (\delta_1, \dots, \delta_s)$		
			$\xleftarrow{\boldsymbol{\delta}}$			
		$FC \leftarrow \mathsf{GetFinalizationCode}(\boldsymbol{\delta})$				
	$\xleftarrow{FC}$					
if $\neg\mathsf{CheckFinalizationCode}(FC_v, FC)$ **abort**						

Protocol 6.6: Vote Confirmation

they need to demonstrate its correctness with a non-interactive zero-knowledge proof. The very last step of the entire election process is the computation and announcement of the final election result by the election administrator.

a) Mixing

The mixing is a serial process, in which all election authorities are involved. Without loss of generality, we assume that the first mix is performed by the Authority 1, the second by Authority 2, and so on. The process is the same for everyone, except for the first authority, which needs to extract the list of encrypted votes from the submitted ballots. Recall that during vote casting, each authority keeps track of all submitted ballots and confirmation. In case of Authority 1, corresponding lists are denoted by B_1 and C_1, respectively. By calling $\mathsf{GetEncryptions}(B_1, C_1, \mathbf{n}, \mathbf{w})$, the first authority retrieves the list $\mathbf{e}_0$ of encrypted votes, and by calling $\mathsf{GenShuffle}(\mathbf{e}_0, pk)$, this list is shuffled into $\mathbf{e}_1 \leftarrow \mathsf{Shuffle}_{pk}(\mathbf{e}_0, \mathbf{r}_1, \psi_1)$, where $\mathbf{r}_1$ denotes the re-encryption randomizations and ψ_1 the random permutation. These values are the secret inputs for a non-interactice proof

$$\pi_1 = \mathit{NIZKP}[(\psi_1, \mathbf{r}_1) : \mathbf{e}_1 = \mathsf{Shuffle}_{pk}(\mathbf{e}_0, \mathbf{r}_1, \psi_1)],$$

which proves the correctness of the shuffle. This proof results from calling the algorithm $\mathsf{GenShuffleProof}(\mathbf{e}_0, \mathbf{e}_1, \mathbf{r}_1, \psi_1, pk)$. The results from conducting the first schuffle—the shuffled list of encryptions $\mathbf{e}_1$ and the zero-knowledge proof π_1—are sent to the bulletin board. This is depicted in the upper part of Prot. 6.7.

Exactly the same shuffling procedure is repeated s times, where the output list $\mathbf{e}_{j-1}$ of the shuffle performed by authority $j-1$ becomes the input list for the shuffle $\mathbf{e}_j \leftarrow \mathsf{Shuffle}_{pk}(\mathbf{e}_{j-1}, \mathbf{r}_j, \psi_j)$ performed of authority j. The whole process over all s authorities realizes the functionality of a re-encryption mix-net. The final result of the mix-net consists of s lists of encryption $\mathbf{E} = (\mathbf{e}_1, \ldots, \mathbf{e}_s)$ with corresponding shuffle proofs $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_s)$.

b) Decryption

After the mixing, every authority retrieves the complete output of the mix-net—the shuffled lists of encryptions $\mathbf{E}$ and the shuffle proofs $\boldsymbol{\pi}$—from the bulletin board. The input $\mathbf{e}_0$ of the first shuffle is retrieved from the submitted ballots by calling $\mathsf{GetEncryptions}(B_j, C_j, \mathbf{n}, \mathbf{w})$. Before starting the decryption, $\mathsf{CheckShuffleProofs}(\boldsymbol{\pi}, \mathbf{e}_0, \mathbf{E}, pk, j)$ is called the to verify the correctness of all shuffles. For authority j, this algorithm loops over all shuffle proofs π_i, $i \neq j$, and checks them individually. As shown in Prot. 6.8, the process aborts in case any of these check fails.

In the success case, the encryptions $\mathbf{e}_s = ((a_1, b_1), \ldots (a_n, b_N))$ obtained from authority s (the last mixer in the mix-net) are partially decrypted using the share sk_j of the private decryption key. Calling $\mathsf{GetPartialDecryptions}(\mathbf{e}_s, sk_j)$ returns a list $\mathbf{b}'_j = (b'_{1,j}, \ldots, b'_{N,j})$ of partial decryptions $b'_{ij} = b_i^{sk_j}$, which are published on the bulletin board. To guarantee the correctness of the decryption, a non-interactive decryption proof

$$\pi'_j = \mathit{NIZKP}[(sk_j) : (b'_{1,j}, \ldots, b'_{N,j}, pk_j) = (b_1^{sk_j}, \ldots, b_N^{sk_j}, g^{sk_j})]$$

Election Authority $j = 1$		Bulletin Board
knows $U, pk, B_1, C_1, \mathbf{n}$		knows $\mathbf{w}$
	$\xleftarrow{\mathbf{w}}$	
$\mathbf{e}_0 \leftarrow \mathsf{GetEncryptions}(B_1, C_1, \mathbf{n}, \mathbf{w})$ $(\mathbf{e}_1, \mathbf{r}_1, \psi_1) \leftarrow \mathsf{GenShuffle}(\mathbf{e}_0, pk)$ $\pi_1 \leftarrow \mathsf{GenShuffleProof}(\mathbf{e}_0, \mathbf{e}_1, \mathbf{r}_1, \psi_1, pk)$		
	$\xrightarrow{\mathbf{e}_1, \pi_1}$	
Election Authority $j \in \{2, \ldots, s\}$		
knows U, pk		
	$\xleftarrow{\mathbf{e}_{j-1}}$	
$(\mathbf{e}_j, \mathbf{r}_j, \psi_j) \leftarrow \mathsf{GenShuffle}(\mathbf{e}_{j-1}, pk)$ $\pi_j \leftarrow \mathsf{GenShuffleProof}(\mathbf{e}_{j-1}, \mathbf{e}_j, \mathbf{r}_j, \psi_j, pk)$		
	$\xrightarrow{\mathbf{e}_j, \pi_j}$	
		$\mathbf{E} \leftarrow (\mathbf{e}_1, \ldots, \mathbf{e}_s)$ $\boldsymbol{\pi} \leftarrow (\pi_1, \ldots, \pi_s)$

Protocol 6.7: Mixing

is computed by calling $\mathsf{GenDecryptionProof}(sk_j, pk_j, \mathbf{e}_s, \mathbf{b}'_j)$ and published along with $\mathbf{b}'_j$. Note that this is a proof of equality of multiple discrete logarithms (see Section 5.4.2). At the end of this process, the partial decryptions and the decryption proofs from all election authorities are available on the bulletin board.

c) Tallying

To conclude an election, the election administrator retrieves the partial decryptions of every election authority from the bulletin board. The attached decryption proofs are checked by calling $\mathsf{CheckDecryptionProofs}(\boldsymbol{\pi}', \mathbf{pk}, \mathbf{e}_s, \mathbf{B}')$. The process aborts if one or more than one check fails. Otherwise, by calling $\mathsf{GetDecryptions}(\mathbf{e}_s, \mathbf{B}')$, the partial decryptions are assembled and the plaintexts are determined. Recall from Section 6.4.2 that every such plaintext is an encoding $\Gamma'(\mathbf{s}, w_i) \in \mathbb{G}_q$ of some voter's selection of candidates and the voter's counting circle, and that the individual votes can be retrieved by factorizing this number. By calling $\mathsf{GetVotes}(\mathbf{m}, \mathbf{n}, \mathbf{w})$, this process is performed for all plaintexts.

The whole tallying process is depicted in Prot. 6.9. The resulting *election result matrix* $\mathbf{V} = (v_{ij})_{N \times n}$ and the *counting circle matrix* $\mathbf{W} = (w_{ij})_{N \times w}$ represent the outcome of the election. The value $v_{ij} \in \mathbb{B}$ is set to 1, if plaintext vote i contains a vote for candidate $j \in \{1, \ldots, n\}$, and to 0, if this is not the case. Similarly, $w_{ij} \in \mathbb{B}$ is set to 1, if plaintext vote i contains a vote for counting circle $j \in \{1, \ldots, w\}$, and to 0, if this is not the case. These

Election Authority $j \in \{1, \ldots, s\}$		Bulletin Board
knows $U, sk_j, pk_j, pk, B_j, C_j, \mathbf{n}$		knows $\mathbf{w}, \mathbf{E} = (\mathbf{e}_1, \ldots, \mathbf{e}_s), \boldsymbol{\pi}$
	$\xleftarrow{\mathbf{w}, \mathbf{E}, \boldsymbol{\pi}}$	
$\mathbf{e}_0 \leftarrow \mathsf{GetEncryptions}(B_j, C_j, \mathbf{n}, \mathbf{w})$ **if** $\neg\mathsf{CheckShuffleProofs}(\boldsymbol{\pi}, e_0, \mathbf{E}, pk, j)$ **abort** $\mathbf{b}'_j \leftarrow \mathsf{GetPartialDecryptions}(\mathbf{e}_s, sk_j)$ $\pi'_j \leftarrow \mathsf{GenDecryptionProof}(sk_j, pk_j, \mathbf{e}_s, \mathbf{b}'_j)$		
	$\xrightarrow{\mathbf{b}'_j, \pi'_j}$	
		$\mathbf{B}' \leftarrow (\mathbf{b}'_1, \ldots, \mathbf{b}'_s)$ $\boldsymbol{\pi}' \leftarrow (\pi'_1, \ldots, \pi'_s)$

Protocol 6.8: Decryption

matrices can be used to compute the following aggregated election results:

$$V_j = \sum_{i=1}^{N} v_{ij} = \text{ total number of votes for candidate } j,$$

$$W_j = \sum_{i=1}^{N} w_{ij} = \text{ total number of submitted votes in counting circle } j,$$

$$V_{jj'} = \sum_{i=1}^{N} v_{ij} w_{ij'} = \text{ total number of votes for candidate } j \text{ in counting circle } j'.$$

Election Administrator		Bulletin Board
knows $U, \mathbf{n}, \mathbf{w}$		knows $\mathbf{pk}, \mathbf{e}_s, \mathbf{B}', \boldsymbol{\pi}'$
	$\xleftarrow{\mathbf{pk}, \mathbf{e}_s, \mathbf{B}', \boldsymbol{\pi}'}$	
if $\neg\mathsf{CheckDecryptionProofs}(\boldsymbol{\pi}', \mathbf{pk}, \mathbf{e}_s, \mathbf{B}')$ **abort** $\mathbf{m} \leftarrow \mathsf{GetDecryptions}(\mathbf{e}_s, \mathbf{B}')$ $(\mathbf{V}, \mathbf{W}) \leftarrow \mathsf{GetVotes}(\mathbf{m}, \mathbf{n}, \mathbf{w})$		
	$\xrightarrow{\mathbf{V}, \mathbf{W}}$	

Protocol 6.9: Tallying

6.6. Channel Security

In Section 6.1, we have already identified the channels that need to be secured by cryptographic means. Most importantly, we require all messages sent to the bulletin board by either the election administrator or the election authorities to be digitally signed. For this, we assume that each of these parties possesses a Schnorr signature key pair (sk_X, pk_X), and that a certificate C_X that binds the public key pk_X to party $X \in \{\mathsf{Admin}, \mathsf{Auth}_1, \ldots, \mathsf{Auth}_s\}$ is publicly available. We assume that checking the validity of certificates is part of checking a signature, i.e., without explicitly describing this process. Therefore, we do not further specify the type, format, and issuer of the certificates and the algorithms for checking them. For this, we refer to current standards such as X.509 and corresponding software libraries and best practices.

Table 6.4 gives an overview of all signatures generated during the protocol execution. For the reasons discussed earlier in Section 6.3.2, we include the election event identifier U as a message prefix in every signature. Generally, for generating a signature for multiple messages $m = (m_1, \ldots, m_r)$, we call $\mathsf{GenSignature}(sk_X, m)$ using the party's public key pk_X. This algorithm implements Schnorr's signature scheme as described in Section 5.6 (see Alg. 7.54 for further details). Note that according to Table 6.4, redundant signatures $\sigma_1^{\mathsf{param}}$, $\sigma_2^{\mathsf{param}}$, and $\sigma_3^{\mathsf{param}}$ are generated by the election administrator during the preparation phase. The reason for this redundancy is to provide tailor-made signatures for all involved parties, i.e., depending on the information they retrieve from the bulletin board during the protocol run.

A special case in the list of signatures shown in Table 6.4 is the entry for Prot. 6.2, which describes the only signature not submitted to the bulletin board. Recall that the private part $\mathbf{d}_j$ of the electorate data generated by election authority j must be sent over a confidential channel to the printing authority. We realize this confidential channel using a symmetric encryption scheme in combination with a key-encapsulation mechanism. Instead of signing $\mathbf{d}_j$, the result of this hybrid encryption, $[\mathbf{d}_j] \leftarrow \mathsf{GenCiphertext}_\phi(pk_{\mathsf{Print}}, \mathbf{d}_j)$, is signed and sent to the printing authority. pk_{Print} denotes the public encryption key of the printing authority. Again, we assume that a certificate for this key exists and is known to everyone.

Issuer	Nr.	Protocol	Parameters	Signatures	Range
Election administrator	6.1	Election preparation	$U, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}$	$\sigma_1^{\mathsf{param}}$	
			$U, \mathbf{n}, \mathbf{k}, \mathbf{E}$	$\sigma_2^{\mathsf{param}}$	
			$U, \mathbf{w}$	$\sigma_3^{\mathsf{param}}$	
	6.9	Tallying	$U, \mathbf{V}, \mathbf{W}$	σ^{tally}	
Election authority $j \in \{1, \ldots, s\}$	6.1	Election preparation	$U, \hat{\mathbf{d}}_j$	σ_j^{prep}	
	6.2	Printing	$U, [\mathbf{d}_j]$	$\sigma_j^{\mathsf{print}}$	
	6.3	Key generation	U, pk_j	σ_j^{kgen}	
	6.5	Vote casting	U, v, β_j	$\sigma_{ij}^{\mathsf{cast}}$	$i \in \{1, \ldots, N_B\}$
	6.6	Vote confirmation	U, v, δ_j	$\sigma_{ij}^{\mathsf{conf}}$	$i \in \{1, \ldots, N_C\}$
	6.7	Mixing	$U, \mathbf{e}_j, \pi_j$	σ_j^{mix}	
	6.8	Decryption	$U, \mathbf{b}'_j, \pi'_j$	σ_j^{dec}	

Table 6.4.: Overview of the signatures generated during the protocol execution.

In Table 6.5, which provides the counterpart of the above list of signatures, we show the necessary signature verifications performed during a complete protocol run. In principle, each time a signed message is retrieved from the bulletin board or received over a direct channel, its attached signature is verified. There is only one exception from this general rule. In Prot. 6.7, i.e., during the mixing process, checking the signatures for the data retrieved from the bulletin board is not mandatory. The mixing process, as implemented in this protocol, is an optimistic procedure, in which each participating election authority performs its task without questioning the correctness of the mixing steps executed previously by other authorities. Since checking the overall correctness of the mix-net is done in the beginning of the decryption process (see Prot. 6.8), no harm can result from this way of performing the mixing. The same holds for the checking the signatures issued for the data involved in this protocol step, i.e., for $\mathbf{w}, \mathbf{e}_1, \ldots, \mathbf{e}_{s-1}$, which is done by every election authority as an initial step of the decryption process.

<table>
<tr><th>Verifier</th><th>Nr.</th><th>Protocol</th><th>Parameters</th><th>Signatures</th><th>Range</th></tr>
<tr><td rowspan="3">Election administrator</td><td rowspan="3">6.9</td><td rowspan="3">Tallying</td><td>U, pk_j</td><td>σ_j^{kgen}</td><td rowspan="16">$j \in \{1, \ldots, s\}$</td></tr>
<tr><td>$U, \mathbf{e}_s, \pi_s$</td><td>σ_s^{mix}</td></tr>
<tr><td>$U, \mathbf{b}'_j, \pi'_j$</td><td>σ_j^{dec}</td></tr>
<tr><td rowspan="5">Election authority $j \in \{1, \ldots, s\}$</td><td rowspan="2">6.1</td><td rowspan="2">Election preparation</td><td>$U, \mathbf{n}, \mathbf{k}, \mathbf{E}$</td><td>$\sigma_2^{\mathsf{param}}$</td></tr>
<tr><td>$U, \hat{\mathbf{d}}_j$</td><td>σ_j^{prep}</td></tr>
<tr><td>6.3</td><td>Key generation</td><td>U, pk_j</td><td>σ_j^{kgen}</td></tr>
<tr><td rowspan="2">6.8</td><td rowspan="2">Decryption</td><td>$U, \mathbf{w}$</td><td>$\sigma_3^{\mathsf{param}}$</td></tr>
<tr><td>$U, \mathbf{e}_j, \pi_j$</td><td>σ_j^{mix}</td></tr>
<tr><td rowspan="4">Voting client</td><td>6.4</td><td>Candidate selection</td><td>$U, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}$</td><td>$\sigma_1^{\mathsf{param}}$</td></tr>
<tr><td rowspan="2">6.5</td><td rowspan="2">Vote casting</td><td>U, pk_j</td><td>σ_j^{kgen}</td></tr>
<tr><td>U, v, β_j</td><td>$\sigma_{ij}^{\mathsf{cast}}$</td></tr>
<tr><td>6.6</td><td>Vote confirmation</td><td>U, v, δ_j</td><td>$\sigma_{ij}^{\mathsf{conf}}$</td></tr>
<tr><td rowspan="2">Printing authority</td><td rowspan="2">6.2</td><td rowspan="2">Printing</td><td>$U, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}$</td><td>$\sigma_1^{\mathsf{param}}$</td></tr>
<tr><td>$U, [\mathbf{d}_j]$</td><td>$\sigma_j^{\mathsf{print}}$</td></tr>
</table>

Table 6.5.: Overview of the signatures verified during the election process.

7. Pseudo-Code Algorithms

To complete the formal description of the cryptographic voting protocol from the previous chapter, we will now present all necessary algorithms in pseudo-code. This will provide an even closer look at the details of the computations performed during the entire election process. The algorithms are numbered according to their appearance in the protocol. To avoid code redundancy and for improved clarity, some algorithms delegate certain tasks to sub-algorithms. An overview of all algorithms and sub-algorithms is given at the beginning of every subsection. Every algorithm is commented in the caption below the pseudo-code, but apart from that, we do not give further explanations. In Section 7.2, we start with some general algorithms for specific tasks, which are needed at multiple places. In Sections 7.3 to 7.5, we specify the algorithms of the respective protocol phases.

7.1. Conventions and Assumptions

With respect to the names attributed to the algorithms, we apply the convention of using the prefix „Gen" for non-deterministic algorithms, the prefix „Get" for general deterministic algorithms, and the prefixes „Is", „Has", or „Check" for predicates. In the case of non-deterministic algorithms, we assume the existence of a cryptographically secure pseudo-random number generator (PRNG) and access to a high-entropy seed. We require such a PRNG for picking elements $r \in_R \mathbb{Z}_q$, $r \in_R \mathbb{G}_q$, $r \in_R \mathbb{Z}_{\hat{q}}$, $r \in_R \mathbb{Z}_{p'}$, and $r \in_R [a, b]$ uniformly at random. Since implementing a PRNG is a difficult problem on its own, it cannot be addressed in this document. Corresponding algorithms are usually available in standard cryptographic libraries of modern programming languages.

The public security parameters from Section 6.3.1 are assumed to be known in every algorithm, i.e., we do not pass them explicitly as parameters. Most numeric calculations in the algorithms are performed modulo p, q, $\hat{p}$, $\hat{q}$, or p'. For maximal clarity, we indicate the modulus in each individual case. We suppose that efficient algorithms are available for computing modular exponentiations $x^y \bmod p$ and modular inverses $x^{-1} \bmod p$. Divisions $x/y \bmod p$ are handled as $xy^{-1} \bmod p$ and exponentiations $x^{-y} \bmod p$ with negative exponents as $(x^{-1})^y \bmod p$ or $(x^y)^{-1} \bmod p$. We also assume that readers are familiar with mathematical notations for sums and products, such that implementing expressions like $\sum_{i=1}^N x_i$ or $\prod_{i=1}^N x_i$ is straightforward.

An important precondition for every algorithm is the validity of the input parameters, for example that an ElGamal encryption $e = (a, b)$ is an element of $\mathbb{G}_q \times \mathbb{G}_q$ or that a given input lists has the desired length. We specify all preconditions for every algorithm, but we do not give explicit code to perform corresponding checks. However, as many attacks—for example on mix-nets—are based on infiltrating invalid parameters, we stress the importance of conducting such checks in an actual implementation. For an efficient way of testing group memberships $x \in \mathbb{G}_q$, we refer to Alg. 7.2.

7.2. General Algorithms

We start with some general algorithms that are called by at least two other algorithms in at least two different protocol phases. They are all deterministic. In Table 7.1 we give an overview. The algorithm $\mathsf{IsMember}(x)$, which is called by $\mathsf{getPrimes}(n)$ for checking the set membership of values $x \in \mathbb{Z}_p$, can also be used for checking the validity of such parameters in other algorithms. As mentioned before, our algorithms do not contain explicit codes for making such checks.

Nr.	Algorithm	Called by	Protocol
7.1	$\mathsf{getPrimes}(n)$	Algs. 7.19, 7.25 and 7.53	6.5, 6.9
7.2	$\hookrightarrow$ $\mathsf{IsMember}(x)$		
7.3	$\mathsf{GetGenerators}(n)$	Algs. 7.43 and 7.47	6.7, 6.8
7.4	$\mathsf{GetNIZKPChallenge}(y, t, \kappa)$	Algs. 7.21, 7.24, 7.32, 7.35, 7.43, 7.47, 7.49 and 7.51	6.5, 6.6, 6.7, 6.8, 6.9
7.5	$\mathsf{GetNIZKPChallenges}(n, y, \kappa)$	Algs. 7.43 and 7.47	6.7, 6.8

Table 7.1.: Overview of general algorithms for specific tasks.

Other general algorithms have been introduced in the Chapter 4 for converting integers, strings, and byte arrays and for hash value computations. We do not repeat them here. There are four algorithms in total, for which we not give explicit pseudo-code: $\mathsf{Sort}_{\leq}(S)$ for sorting a list S according to some total order $\leq$, $\mathsf{UTF8}(S)$ for converting a string S into a byte array according to the UTF-8 character encoding, $\mathsf{Hash}_L(B)$ for computing the hash value of length L (bytes) of an input byte array B (see Section 8.1), and $\mathsf{JacobiSymbol}(x, p)$ for computing the Jacobi symbol $\left(\frac{x}{p}\right) \in \{-1, 0, 1\}$ for two integers x and p. A proposal for $\mathsf{Hash}_L(B)$ based on the SHA-256 hash algorithm is given in Section 8.1.

For the first three algorithms, standard implementations are available in most modern programming languages. Algorithms to compute the Jacobi symbol are not so widely available, but GMPLib[1], one of the fastest and most widely used libraries for multiple-precision arithmetic, provides an implementation of the Kronecker symbol, which includes the Jacobi symbol as special case. If no off-the-shelf implementation is available, we refer to existing pseudo-code algorithms such as [2, pp. 76–77].

[1]See https://gmplib.org

```
Algorithm: getPrimes(n)
Input: Number of primes n ⩾ 0
x ← 1
for i = 1, ..., n do
    repeat
        if x ⩽ 2 then
            x ← x + 1
        else
            x ← x + 2
        if x ⩾ p then
            return ⊥                                    // n is incompatible with p
    until IsPrime(x) and IsMember(x)                    // see Alg. 7.2
    p_i ← x
p ← (p_1, ..., p_n)
return p                                                // p ∈ (G_q ∩ P)^n
```

Algorithm 7.1: Computes the first n prime numbers from $\mathbb{G}_q \subset \mathbb{Z}_p^*$. The computation possibly fails if n is too large or p is too small, but this case is very unlikely in practice. In a more efficient implementation of this algorithm, the list of resulting primes is accumulated in a cache or precomputed for the largest expected value $n_{\max} \geqslant n$.

```
Algorithm: IsMember(x)
Input: Number to test x ∈ N
if 1 ⩽ x < p then
    j ← JacobiSymbol(x, p)                              // j ∈ {−1, 0, 1}
    if j = 1 then
        return true
return false
```

Algorithm 7.2: Checks if a positive integer $x \in \mathbb{N}$ is an element of $\mathbb{G}_q \subset \mathbb{Z}_p^*$. The core of the algorithm is the computation of the Jacobi symbol $\left(\frac{x}{p}\right) \in \{-1, 0, 1\}$, for which we refer to existing algorithms such as [2, pp. 76–77] or implementations in libraries such as GMPLib.

Algorithm: $\mathsf{GetGenerators}(n)$

Input: Number of independent geneators $n \geqslant 0$

for $i = 1, \ldots, n$ **do**

  $x \leftarrow 0$

  repeat

    $x \leftarrow x + 1$

    $h_i \leftarrow \mathsf{ToInteger}(\mathsf{RecHash}_L(\texttt{"chVote"}, \texttt{"ggen"}, i, x)) \bmod p$ // see Algs. 4.5 and 4.9

    $h_i \leftarrow h_i^2 \bmod p$

  until $h_i \notin \{0, 1\}$ // these cases are very unlikely

$\mathbf{h} \leftarrow (h_1, \ldots, h_n)$

return h // $\mathbf{h} \in (\mathbb{G}_q \backslash \{1\})^n$

Algorithm 7.3: Computes n independent generators of $\mathbb{G}_q \subset \mathbb{Z}_p^*$. The algorithm is an adaption of the NIST standard FIPS PUB 186-4 [2, Appendix A.2.3]. The string `"chVote"` guarantees that the resulting values are specific to the chVote project. In a more efficient implementation of this algorithm, the list of resulting generators is accumulated in a cache or precomputed for the largest expected value $n_{\max} \geqslant n$.

Algorithm: $\mathsf{GetNIZKPChallenge}(y, t, \kappa)$

Input: Public value $y \in Y$, Y unspecified
  Commitment $t \in T$, T unspecified
  Soundness strength $1 \leqslant \kappa \leqslant 8L$

$c \leftarrow \mathsf{ToInteger}(\mathsf{RecHash}_L(y, t)) \bmod 2^\kappa$ // see Algs. 4.5 and 4.9

return c // $c \in \mathbb{Z}_{2^\kappa}$

Algorithm 7.4: Computes a NIZKP challenge $0 \leqslant c < 2^\kappa$ for a given public value y and a public commitment t. The domains Y and T of the input values are unspecified.

Algorithm: $\mathsf{GetNIZKPChallenges}(n, y, \kappa)$

Input: Number of challenges $n \geqslant 0$
  Public value $y \in Y$, Y unspecified
  Soundness strength $1 \leqslant \kappa \leqslant 8L$

$H \leftarrow \mathsf{RecHash}_L(y)$ // see Alg. 4.9

for $i = 1, \ldots, n$ **do**

  $I \leftarrow \mathsf{RecHash}_L(i)$ // see Alg. 4.9

  $c_i \leftarrow \mathsf{ToInteger}(\mathsf{Hash}_L(H \,\|\, I)) \bmod 2^\kappa$ // see Alg. 4.5

$\mathbf{c} \leftarrow (c_1, \ldots, c_n)$

return c // $\mathbf{c} \in \mathbb{Z}_{2^\kappa}^n$

Algorithm 7.5: Computes n challenges $0 \leqslant c_i < 2^\kappa$ for a given of public value y. The domain Y of the input value is unspecified. The results in $\mathbf{c} = (c_1, \ldots, c_n)$ are identical to $c_i = \mathsf{ToInteger}(\mathsf{RecHash}_L(y, i)) \bmod 2^\kappa$, but precomputing H makes the algorithm more efficient, especially if y is a complex mathematical object.

7.3. Pre-Election Phase

The main actors in the pre-election phase are the election authorities. For the given election definition consisting of values $\mathbf{n}$, $\mathbf{k}$, and $\mathbf{E}$, each election authority generates a share of the electorate data by calling Alg. 7.6. This is the main algorithm of the election preparation, which invokes several sub-algorithms for more specific tasks. Table 7.2 gives an overview of all algorithms of the pre-election phase. The public parts of the electorate data from every authority, which are exchanged using the bulletin board, are assembled by the election authorities by calling Alg. 7.12. The private parts of the electorate data, which are sent to the printing authority over a confidential channel, are assembled to create the voting cards by calling Alg. 7.13. The corresponding sub-task for creating a single voting card is delegated to Alg. 7.14, but the formating details are not specified explicitly. Two other algorithms are required for generating shares of the encryption key and for assembling the shares of the public key. For a more detailed description of the pre-election phase, we refer to Section 6.5.1.

Nr.	Algorithm	Called by	Protocol
7.6	$\mathsf{GenElectorateData}(\mathbf{n}, \mathbf{k}, \mathbf{E})$	Election authority	6.1
7.7	$\hookrightarrow$ $\mathsf{GenPoints}(n, k)$		
7.8	$\quad\hookrightarrow$ $\mathsf{GenPolynomial}(d)$		
7.9	$\quad\hookrightarrow$ $\mathsf{GetYValue}(x, \mathbf{a})$		
7.10	$\hookrightarrow$ $\mathsf{GenSecretVoterData}(\mathbf{p})$		
7.11	$\hookrightarrow$ $\mathsf{GetPublicVoterData}(x, y)$		
7.12	$\mathsf{GetPublicCredentials}(\hat{\mathbf{D}})$	Election authority	
7.13	$\mathsf{GetVotingCards}(\mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{D})$	Printing authority	6.2
7.14	$\hookrightarrow$ $\mathsf{GetVotingCard}(v, V, w, \mathbf{c}, \mathbf{n}, \mathbf{k}, X, Y, FC, \mathbf{rc})$		
7.15	$\mathsf{GenKeyPair}()$	Election authority	6.3
7.16	$\mathsf{GetPublicKey}(\mathbf{pk})$	Election authority	

Table 7.2.: Overview of algorithms and sub-algorithms of the pre-election phase.

Algorithm: $\mathsf{GenElectorateData}(\mathbf{n}, \mathbf{k}, \mathbf{E})$

Input: Number of candidates $\mathbf{n} = (n_1, \ldots, n_t)$, $n_j \geqslant 2$
Number of selections $\mathbf{k} = (k_1, \ldots, k_t)$, $0 < k_j < n_j$
Eligibility matrix $\mathbf{E} = (e_{ij})_{N_E \times t}$, $e_{ij} \in \mathbb{B}$

$n \leftarrow \sum_{j=1}^{t} n_j$

for $i = 1, \ldots, N_E$ **do**

- $k'_i \leftarrow \sum_{j=1}^{t} e_{ij} k_j$
- $(\mathbf{p}_i, y'_i) \leftarrow \mathsf{GenPoints}(n, k'_i)$ // $\mathbf{p}_i = (p_{i,1}, \ldots, p_{i,n})$, see Alg. 7.7
- $d_i \leftarrow \mathsf{GenSecretVoterData}(\mathbf{p}_i)$ // $d_i = (x_i, y_i, F_i, \mathbf{r}_i)$, see Alg. 7.10
- $\hat{d}_i \leftarrow \mathsf{GetPublicVoterData}(x_i, y_i, y'_i)$ // $\hat{d}_i = (\hat{x}_i, \hat{y}_i)$, see Alg. 7.11

$\mathbf{d} \leftarrow (d_1, \ldots, d_{N_E})$

$\hat{\mathbf{d}} \leftarrow (\hat{d}_1, \ldots, \hat{d}_{N_E})$

$\mathbf{P} \leftarrow (p_{ij})_{N_E \times n}$

return $(\mathbf{d}, \hat{\mathbf{d}}, \mathbf{P})$ // $\mathbf{d} \in (\mathbb{Z}_{\hat{q}_x} \times \mathbb{Z}_{\hat{q}_y} \times \mathcal{B}^{L_F} \times (\mathcal{B}^{L_R})^n)^{N_E}$, $\hat{\mathbf{d}} \in (\mathbb{G}_{\hat{q}}^2)^{N_E}$, $\mathbf{P} \in (\mathbb{Z}_{p'}^2)^{N_E n}$

Algorithm 7.6: Generates the voting card data for the whole electorate. For this, the algorithm loops over all voters and computes for each voter i the permitted number $k'_i = \sum_{j=1}^{t} e_{ij} k_j$ of selections of the current election event. Alg. 7.10 and Alg. 7.11 are called to generate the voter data for each single voter. At the end, the responses of these calls are grouped into a secret part $\mathbf{d}$ sent to the voters prior to an election event via the printing authority (see Prot. 6.2), a public part $\hat{\mathbf{d}}$ sent to the bulletin board to allow voter identification during vote casting (see Prot. 6.1 and Prot. 6.5), and the matrix $\mathbf{P} = (p_{ij})_{N_E \times n}$ of random points $p_{ij} = (x_{ij}, y_{ij})$, of which k'_i will be transferred obliviously to the voters during vote casting (see Prot. 6.5).

Algorithm: $\mathsf{GenPoints}(n, k)$

Input: Number of candidates $n \geqslant 2$
Number of selections $0 < k < n$

$\mathbf{a} \leftarrow \mathsf{GenPolynomial}(k-1)$ // $\mathbf{a} = (a_0, \ldots, a_{k-1})$, see Alg. 7.8

$X \leftarrow \varnothing$

for $i = 1, \ldots, n$ **do**

- $x \in_R \mathbb{Z}_{p'} \setminus X$ // different from values picked previously
- $X \leftarrow X \cup \{x\}$
- $y \leftarrow \mathsf{GetYValue}(x, \mathbf{a})$ // see Alg. 7.9
- $p_i \leftarrow (x, y)$

$y' \leftarrow \mathsf{GetYValue}(0, \mathbf{a})$ // see Alg. 7.9

$\mathbf{p} \leftarrow (p_1, \ldots, p_n)$

return $(\mathbf{p}, y')$ // $\mathbf{p} \in (\mathbb{Z}_{p'}^2)^n$, $y' \in \mathbb{Z}_{p'}$

Algorithm 7.7: Generates a list of n random points picked from a random polynomial $A(X) \in_R \mathbb{Z}_{p'}[X]$ of degree $k-1$. The random polynomial is obtained from calling Alg. 7.8. Additionally, using Alg. 7.9, the value $y' = A(0)$ is computed and returned together with the random points.

Algorithm: GenPolynomial(d)
Input: Degree $d \geqslant -1$
if $d = -1$ **then**
    $\mathbf{a} \leftarrow (0)$
else
    for $i = 0, \ldots, d-1$ **do**
        $a_i \in_R \mathbb{Z}_{p'}$
    $a_d \in_R \mathbb{Z}_{p'} \backslash \{0\}$
    $\mathbf{a} \leftarrow (a_0, \ldots, a_d)$
return a // $\mathbf{a} \in \mathbb{Z}_{p'}^{d+1}$

Algorithm 7.8: Generates the coefficients $a_0, \ldots, a_d$ of a random polynomial $A(X) = \sum_{i=0}^{d} a_i X^i \bmod p'$ of degree $d \geqslant 0$. The algorithm also accepts $d = -1$ as input, which we interpret as the polynomial $A(X) = 0$. In this case, the algorithm returns the coefficient list $\mathbf{a} = (0)$.

Algorithm: GetYValue($x, \mathbf{a}$)
Input: Value $x \in \mathbb{Z}_{p'}$
Coefficients $\mathbf{a} = (a_0, \ldots, a_d)$, $a_i \in \mathbb{Z}_{p'}$, $d \geqslant 0$
if $x = 0$ **then**
    $y \leftarrow a_0$
else
    $y \leftarrow 0$
    for $i = d, \ldots, 0$ **do**
        $y \leftarrow a_i + x \cdot y \bmod p'$
return y // $y \in \mathbb{Z}_{p'}$

Algorithm 7.9: Computes the value $y = A(x) \in \mathbb{Z}_{p'}$ obtained from evaluating the polynomial $A(X) = \sum_{i=0}^{d} a_i X^i \bmod p'$ at position x. The algorithm is an implementation of Horner's method.

Algorithm: $\mathsf{GenSecretVoterData}(\mathbf{p})$

Input: Points $\mathbf{p} = (p_1, \dots, p_n)$, $p_i \in \mathbb{Z}_{p'}^2$

$\hat{q}'_x \leftarrow \lfloor \hat{q}_x / s \rfloor$, $\hat{q}'_y \leftarrow \lfloor \hat{q}_y / s \rfloor$

$x \in_R \mathbb{Z}_{\hat{q}'_x}$, $y \in_R \mathbb{Z}_{\hat{q}'_y}$

$F \leftarrow \mathsf{Truncate}(\mathsf{RecHash}_L(\mathbf{p}), L_F)$ // see Alg. 4.9

for $i = 1, \dots, n$ **do**

  $R_i \leftarrow \mathsf{Truncate}(\mathsf{RecHash}_L(p_i), L_R)$ // see Alg. 4.9

$\mathbf{r} \leftarrow (R_1, \dots, R_n)$

$d \leftarrow (x, y, F, \mathbf{r})$

return d // $d \in \mathbb{Z}_{\hat{q}_x} \times \mathbb{Z}_{\hat{q}_y} \times \mathcal{B}^{L_F} \times (\mathcal{B}^{L_R})^n$

Algorithm 7.10: Generates an authority's share of the secret data for a single voter, which is sent to the voter prior to an election event via the printing authority.

Algorithm: $\mathsf{GetPublicVoterData}(x, y, y')$

Input: Secret voting credential $x \in \mathbb{Z}_{\hat{q}}$
Secret confirmation credential $y \in \mathbb{Z}_{\hat{q}}$
Secret vote validity credential $y' \in \mathbb{Z}_{p'}$

$\hat{x} \leftarrow \hat{g}^x \bmod \hat{p}$, $\hat{y} \leftarrow \hat{g}^{y + y' \bmod \hat{q}} \bmod \hat{p}$

$\hat{d} \leftarrow (\hat{x}, \hat{y})$

return $\hat{d}$ // $\hat{d} \in \mathbb{G}_{\hat{q}}^2$

Algorithm 7.11: Generates an authority's share of the public data for a single voter, which is sent to the bulletin board.

Algorithm: $\mathsf{GetPublicCredentials}(\hat{\mathbf{D}})$

Input: Public voter credentials $\hat{\mathbf{D}} = (\hat{d}_{ij})_{N_E \times s}$, $\hat{d}_{ij} = (\hat{x}_{ij}, \hat{y}_{ij})$, $\hat{x}_{ij} \in \mathbb{G}_{\hat{q}}$, $\hat{y}_{ij} \in \mathbb{G}_{\hat{q}}$

for $i = 1, \dots, N_E$ **do**

  $\hat{x}_i \leftarrow \prod_{j=1}^{s} \hat{x}_{ij} \bmod \hat{p}$

  $\hat{y}_i \leftarrow \prod_{j=1}^{s} \hat{y}_{ij} \bmod \hat{p}$

$\hat{\mathbf{x}} \leftarrow (\hat{x}_1, \dots, \hat{x}_{N_E})$

$\hat{\mathbf{y}} \leftarrow (\hat{y}_1, \dots, \hat{y}_{N_E})$

return $(\hat{\mathbf{x}}, \hat{\mathbf{y}})$ // $\hat{\mathbf{x}} \in \mathbb{G}_{\hat{q}}^{N_E}$, $\hat{\mathbf{y}} \in \mathbb{G}_{\hat{q}}^{N_E}$

Algorithm 7.12: Computes lists $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ of public voter credentials, which are obtained by multiplying corresponding values from the public parts of the electorate data generated by the election authorities. The values in $\hat{\mathbf{x}}$ are used in Prot. 6.5 to verify if a submitted ballot belongs to an eligible voter, whereas the values in $\hat{\mathbf{y}}$ are used in Prot. 6.6 to verify that the vote confirmation has been invoked by the same eligible voter.

Algorithm: $\mathsf{GetVotingCards}(\mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{D})$

Input: Voter descriptions $\mathbf{v} = (V_1, \ldots, V_{N_E})$, $V_i \in A^*_{\mathsf{ucs}}$
Counting circles $\mathbf{w} = (w_1, \ldots, w_{N_E})$, $w_i \in \mathbb{N}$
Candidate descriptions $\mathbf{c} = (C_1, \ldots, C_n)$, $C_i \in A^*_{\mathsf{ucs}}$
Number of candidates $\mathbf{n} = (n_1, \ldots, n_t)$, $n_j \geqslant 2$, $n = \sum_{j=1}^{t} n_j$
Number of selections $\mathbf{k} = (k_1, \ldots, k_t)$, $0 < k_j < n_j$
Eligibility matrix $\mathbf{E} = (e_{ij})_{N_E \times t}$, $e_{ij} \in \mathbb{B}$
Voting card data $\mathbf{D} = (d_{ij})_{N_E \times s}$, $d_{ij} = (x_{ij}, y_{ij}, F_{ij}, \mathbf{r}_{ij})$, $x_{ij} \in \mathbb{Z}_{\hat{q}_x}$,
$\sum_{j=1}^{s} x_{ij} < \hat{q}_x$, $y_{ij} \in \mathbb{Z}_{\hat{q}_y}$, $\sum_{j=1}^{s} y_{ij} < \hat{q}_y$, $F_{ij} \in \mathcal{B}^{L_F}$, $\mathbf{r}_{ij} = (R_{i,j,1}, \ldots, R_{i,j,n})$,
$R_{ijk} \in \mathcal{B}^{L_R}$

for $i = 1, \ldots, N_E$ **do**
  $\mathbf{k} = (e_{i,1}k_1, \ldots, e_{i,t}k_t)$
  $X \leftarrow \mathsf{ToString}(\sum_{j=1}^{s} x_{ij}, \ell_X, A_X)$ // see Alg. 4.6
  $Y \leftarrow \mathsf{ToString}(\sum_{j=1}^{s} y_{ij}, \ell_Y, A_Y)$ // see Alg. 4.6
  $FC \leftarrow \mathsf{ToString}(\bigoplus_{j=1}^{s} F_{ij}, A_F)$ // see Alg. 4.8
  for $k = 1, \ldots, n$ **do**
    $R \leftarrow \mathsf{MarkByteArray}(\bigoplus_{j=1}^{s} R_{ijk}, k-1, n_{\max})$ // see Alg. 4.1
    $RC_k \leftarrow \mathsf{ToString}(R, A_R)$ // see Alg. 4.8
  $\mathbf{rc} \leftarrow (RC_1, \ldots, RC_n)$
  $S_i \leftarrow \mathsf{GetVotingCard}(i, V_i, w_i, \mathbf{c}, \mathbf{n}, \mathbf{k}, X, Y, FC, \mathbf{rc})$ // see Alg. 7.14
$\mathbf{s} \leftarrow (S_1, \ldots, S_{N_E})$
return s // $\mathbf{s} \in (A^*_{\mathsf{ucs}})^{N_E}$

Algorithm 7.13: Computes the list $\mathbf{s} = (S_1, \ldots, S_{N_E})$ of voting cards for every voter. A single voting card is represented as a string $S_i \in A^*_{\mathsf{ucs}}$, which is generated by Alg. 7.14.

Algorithm: $\mathsf{GetVotingCard}(v, V, w, \mathbf{c}, \mathbf{n}, \mathbf{k}, X, Y, FC, \mathbf{rc})$

Input: Voter index $v \in \mathbb{N}$
Voter description $V \in A^*_{\mathsf{ucs}}$
Counting circle $w \in \mathbb{N}$
Candidate descriptions $\mathbf{c} = (C_1, \ldots, C_n)$, $C_i \in A^*_{\mathsf{ucs}}$
Number of candidates $\mathbf{n} = (n_1, \ldots, n_t)$, $n_j \geqslant 2$, $n = \sum_{j=1}^{t} n_j$
Number of selections $\mathbf{k} = (k_1, \ldots, k_t)$, $0 \leqslant k_j < n_j$ // $k_j = 0$ means ineligible
Voting code $X \in A_X^{\ell_X}$
Confirmation code $Y \in A_Y^{\ell_Y}$
Finalization code $FC \in A_F^{\ell_F}$
Verification codes $\mathbf{rc} = (RC_1, \ldots, RC_n)$, $RC_i \in A_R^{\ell_R}$
$S \leftarrow \cdots$ // compose string to be printed on voting card
return S // $S \in A^*_{\mathsf{ucs}}$

Algorithm 7.14: Computes a string $S \in A^*_{\mathsf{ucs}}$, which represent a voting card that can be printed on paper and sent to voter v. Specifying the formatting details of presenting the information on the printed voting card is beyond the scope of this document.

Algorithm: $\mathsf{GenKeyPair}()$

$sk \in_R \mathbb{Z}_q$
$pk \leftarrow g^{sk} \bmod p$
return (sk, pk) // $(sk, pk) \in \mathbb{Z}_q \times \mathbb{G}_q$

Algorithm 7.15: Generates a random ElGamal encryption key pair $(sk, pk) \in \mathbb{Z}_q \times \mathbb{G}_q$ or a shares of such a key pair. This algorithm is used in Prot. 6.3 by the authorities to generate private shares of a common public encryption key.

Algorithm: $\mathsf{GetPublicKey}(\mathbf{pk})$

Input: Public keys $\mathbf{pk} = (pk_1, \ldots, pk_s)$, $pk_j \in \mathbb{G}_q$
$pk \leftarrow \prod_{j=1}^{s} pk_j \bmod p$
return pk // $pk \in \mathbb{G}_q$

Algorithm 7.16: Computes a public ElGamal encryption key $pk \in \mathbb{G}_q$ from given shares $pk_j \in \mathbb{G}_q$.

7.4. Election Phase

The election phase is the most complex part of the cryptographic protocol, in which each of the involved parties (voter, voting client, election authorities) calls several algorithms. An overview of all algorithms is given in Table 7.3. To submit a ballot containing the voter's selections $\mathbf{s}$, the voting client calls Alg. 7.17 to obtain the voting page that is presented to the voter and Alg. 7.16 to obtain the public encryption key. Using the voter's inputs X and $\mathbf{s}$, the ballot is constructed by calling Alg. 7.18, which internally invokes several sub-algorithms. The authorities call Alg. 7.22 to check the validity of the ballot and Alg. 7.25 to generate the response to the OT query included in the ballot. The voting client unpacks the responses by calling Alg. 7.26 and assembles the resulting point matrix into the verification codes of the selected candidates by calling Alg. 7.28. The voter then compares the displayed verification codes with the ones on the voting card and enters the confirmation code Y. We describe

Nr.	Algorithm	Called by	Protocol
7.17	$\mathsf{GetVotingPage}(i, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E})$	Voting client	6.4
7.16	$\mathsf{GetPublicKey}(\mathbf{pk})$	Voting client	
7.18	$\mathsf{GenBallot}(X, \mathbf{s}, pk)$	Voting client	
7.19	$\hookrightarrow$ $\mathsf{GetSelectedPrimes}(\mathbf{s})$		
7.20	$\hookrightarrow$ $\mathsf{GenQuery}(\mathbf{q}, pk)$		
7.21	$\hookrightarrow$ $\mathsf{GenBallotProof}(x, m, r, \hat{x}, e, pk)$		
7.22	$\mathsf{CheckBallot}(v, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B)$	Election authority	6.5
7.23	$\hookrightarrow$ $\mathsf{HasBallot}(v, B)$		
7.24	$\hookrightarrow$ $\mathsf{CheckBallotProof}(\pi, \hat{x}, e, pk)$		
7.25	$\mathsf{GenResponse}(v, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P})$	Election authority	
7.26	$\mathsf{GetPointMatrix}(\boldsymbol{\beta}, \mathbf{s}, \mathbf{r})$	Voting client	
7.27	$\hookrightarrow$ $\mathsf{GetPoints}(\beta, \mathbf{s}, \mathbf{r})$		
7.28	$\mathsf{GetReturnCodes}(\mathbf{s}, \mathbf{P_s})$	Voting client	
7.29	$\mathsf{CheckReturnCodes}(\mathbf{rc}, \mathbf{rc}', \mathbf{s})$	Voter	
7.30	$\mathsf{GenConfirmation}(Y, \mathbf{P})$	Voting client	
7.31	$\hookrightarrow$ $\mathsf{GetValue}(\mathbf{p})$		
7.32	$\hookrightarrow$ $\mathsf{GenConfirmationProof}(y, y', \hat{y})$		
7.33	$\mathsf{CheckConfirmation}(v, \gamma, \hat{\mathbf{y}}, B, C)$	Election authority	
7.23	$\hookrightarrow$ $\mathsf{HasBallot}(v, B)$		6.6
7.34	$\hookrightarrow$ $\mathsf{HasConfirmation}(i, C)$		
7.35	$\hookrightarrow$ $\mathsf{CheckConfirmationProof}(\pi, \hat{y})$		
7.36	$\mathsf{GetFinalization}(v, \mathbf{P}, B)$	Election authority	
7.37	$\mathsf{GetFinalizationCode}(\boldsymbol{\delta})$	Voting client	
7.38	$\mathsf{CheckFinalizationCode}(FC, FC')$	Voter	

Table 7.3.: Overview of algorithms and sub-algorithms of the election phase.

the (human) execution of this task by a call to Alg. 7.29. The voting client then generates the confirmation message using Alg. 7.30, which invokes several sub-algorithms. By calling Algs. 7.33 and 7.36, the authorities check the confirmation and return their shares of the finalization code. Using 7.37, the voting client assembles the finalization code and displays it to the voter, which finally executes Alg. 7.38 to compare it with the finalization code printed on the voting card. Section 6.5.2 describes the election phase in more details.

Algorithm: $\mathsf{GetVotingPage}(v, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E})$
Input: Voter index $v \in \{1, \ldots, N_E\}$
Voter descriptions $\mathbf{v} = (V_1, \ldots, V_{N_E})$, $V_i \in A^*_{\mathsf{ucs}}$
Counting circles $\mathbf{w} = (w_1, \ldots, w_{N_E})$, $w_i \in \mathbb{N}$
Candidate descriptions $\mathbf{c} = (C_1, \ldots, C_n)$, $C_i \in A^*_{\mathsf{ucs}}$
Number of candidates $\mathbf{n} = (n_1, \ldots, n_t)$, $n_j \geqslant 2$, $n = \sum_{j=1}^{t} n_j$
Number of selections $\mathbf{k} = (k_1, \ldots, k_t)$, $0 < k_j < n_j$
Eligibility matrix $\mathbf{E} = (e_{ij})_{N_E \times t}$, $e_{ij} \in \mathbb{B}$
$P \leftarrow \cdots$ // compose string to be displayed to the voter
return P // $P \in A^*_{\mathsf{ucs}}$

Algorithm 7.17: Computes a string $P \in A^*_{\mathsf{ucs}}$, which represents the voting page displayed to voter v. Specifying the details of presenting the information on the voting page is beyond the scope of this document.

Algorithm: $\mathsf{GenBallot}(X, \mathbf{s}, pk)$
Input: Voting code $X \in A_X^{\ell_X}$
Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k$
Encryption key $pk \in \mathbb{G}_q$
$x \leftarrow \mathsf{ToInteger}(X)$ // see Alg. 4.7
$\hat{x} \leftarrow \hat{g}^x \bmod \hat{p}$
$\mathbf{q} \leftarrow \mathsf{GetSelectedPrimes}(\mathbf{s})$ // $\mathbf{q} = (q_1, \ldots, q_k)$, see Alg. 7.19
$m \leftarrow \prod_{j=1}^{k} q_j$
if $m \geqslant p$ **then**
  return $\perp$ // (k, n) is incompatible with p
$(\mathbf{a}, \mathbf{r}) \leftarrow \mathsf{GenQuery}(\mathbf{q}, pk)$ // $\mathbf{a} = (a_1, \ldots, a_k)$, $\mathbf{r} = (r_1, \ldots, r_k)$, see Alg. 7.20
$e \leftarrow (\prod_{j=1}^{k} a_{j,1} \bmod p, \prod_{j=1}^{k} a_{j,2} \bmod p)$
$r \leftarrow \sum_{j=1}^{k} r_j \bmod q$
$\pi \leftarrow \mathsf{GenBallotProof}(x, m, r, \hat{x}, e, pk)$ // $\pi = (t, s)$, see Alg. 7.21
$\alpha \leftarrow (\hat{x}, \mathbf{a}, \pi)$
return $(\alpha, \mathbf{r})$ // $\alpha \in \mathbb{Z}_{\hat{q}} \times (\mathbb{G}_q \times \mathbb{G}_q)^k \times ((\mathbb{G}_{\hat{q}} \times \mathbb{G}_q^2) \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q))$, $\mathbf{r} \in \mathbb{Z}_q^k$

Algorithm 7.18: Generates a ballot based on the selection $\mathbf{s}$ and the voting code X. The ballot includes an OT query $\mathbf{a}$ and a NIZKP π. The algorithm also returns the randomizations $\mathbf{r}$ of the OT query, which are required in Alg. 7.27 to derive the transferred messages from the OT response.

Algorithm: $\mathsf{GetSelectedPrimes}(\mathbf{s})$
Input: Selections $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k$
$\mathbf{p} \leftarrow \mathsf{getPrimes}(s_k)$ // see Alg. 7.1
for $j = 1, \ldots, k$ **do**
  $q_j \leftarrow p_{s_j}$
$\mathbf{q} \leftarrow (q_1, \ldots, q_k)$
return q // $\mathbf{q} \in (\mathbb{G}_q \cap \mathbb{P})^k$

Algorithm 7.19: Selects k prime numbers from $\mathbb{G}_q$ corresponding to the given indices $\mathbf{s} = (s_1, \ldots, s_k)$. For example, $\mathbf{s} = (1, 3, 7)$ means selecting the first, the third, and the seventh prime from $\mathbb{G}_q$.

Algorithm: $\mathsf{GenQuery}(\mathbf{q}, pk)$
Input: Selected primes $\mathbf{q} = (q_1, \ldots, q_k)$
  Encryption key $pk \in \mathbb{G}_q$
for $j = 1, \ldots, k$ **do**
  $r_j \in_R \mathbb{Z}_q$
  $a_{j,1} \leftarrow q_j \cdot pk^{r_j} \bmod p$
  $a_{j,2} \leftarrow g^{r_j} \bmod p$
  $a_j \leftarrow (a_{j,1}, a_{j,2})$
$\mathbf{a} \leftarrow (a_1, \ldots, a_k)$
$\mathbf{r} \leftarrow (r_1, \ldots, r_k)$
return $(\mathbf{a}, \mathbf{r})$ // $\mathbf{a} \in (\mathbb{G}_q \times \mathbb{G}_q)^k$, $\mathbf{r} \in \mathbb{Z}_q^k$

Algorithm 7.20: Generates an OT query **a** from the prime numbers representing the voter's selections and a for a given public encryption key (which serves as a generator of $\mathbb{Z}_p$).

Algorithm: $\mathsf{GenBallotProof}(x, m, r, \hat{x}, e, pk)$

Input: Voting credentials $(x, \hat{x}) \in \mathbb{Z}_{\hat{q}} \times \mathbb{G}_{\hat{q}}$
Product of selected primes $m \in \mathbb{G}_q$
Randomization $r \in \mathbb{Z}_q$
ElGamal encryption $e = (a, b) \in \mathbb{G}_q \times \mathbb{G}_q$
Encryption key $pk \in \mathbb{G}_q$

$\omega_1 \in_R \mathbb{Z}_{\hat{q}}$, $\omega_2 \in_R \mathbb{G}_q$, $\omega_3 \in_R \mathbb{Z}_q$
$t_1 \leftarrow \hat{g}^{\omega_1} \bmod \hat{p}$, $t_2 \leftarrow \omega_2 \cdot pk^{\omega_3} \bmod p$, $t_3 \leftarrow g^{\omega_3} \bmod p$
$y \leftarrow (\hat{x}, a, b)$, $t \leftarrow (t_1, t_2, t_3)$
$c \leftarrow \mathsf{GetNIZKPChallenge}(y, t, \tau)$ // see Alg. 7.4
$s_1 \leftarrow \omega_1 + c \cdot x \bmod \hat{q}$, $s_2 \leftarrow \omega_2 \cdot m^c \bmod p$, $s_3 \leftarrow \omega_3 + c \cdot r \bmod q$
$s \leftarrow (s_1, s_2, s_3)$
$\pi \leftarrow (t, s)$
return π // $\pi \in (\mathbb{G}_{\hat{q}} \times \mathbb{G}_q^2) \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q)$

Algorithm 7.21: Generates a NIZKP, which proves that the ballot has been formed properly. This proof includes a proof of knowledge of the secret voting credential x that matches with the public voting credential $\hat{x}$. Note that this is equivalent to a Schnorr identification proof [45]. For the verification of this proof, see Alg. 7.24.

Algorithm: $\mathsf{CheckBallot}(v, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}}, B)$

Input: Voter index $v \in \{1, \ldots N_E\}$
Ballot $\alpha = (\hat{x}, \mathbf{a}, \pi)$, $\hat{x} \in \mathbb{Z}_{\hat{q}}$, $\mathbf{a} = (a_1, \ldots, a_k)$, $a_j = (a_{j,1}, a_{j,2}) \in \mathbb{G}_q^2$, $k > 0$
Encryption key $pk \in \mathbb{G}_q$
Number of selections $\mathbf{k} = (k_1, \ldots, k_t)$, $0 < k_j < n_j$
Eligibility matrix $\mathbf{E} = (e_{ij})_{N_E \times t}$, $e_{ij} \in \mathbb{B}$
Public voting credentials $\hat{\mathbf{x}} = \{\hat{x}_1, \ldots, \hat{x}_{N_E}\}$, $\hat{x}_i \in \mathbb{G}_{\hat{q}}$
Ballot list $B = \langle (v_i, \alpha_i, z_i) \rangle_{i=0}^{N_B - 1}$, $v_i \in \{1, \ldots, N_E\}$

$k' \leftarrow \sum_{j=1}^{t} e_{vj} k_j$
if $\neg\mathsf{HasBallot}(v, B)$ **and** $\hat{x} = \hat{x}_v$ **and** $k = k'$ **then** // see Alg. 7.23
  $e \leftarrow (\prod_{j=1}^{k} a_{j,1} \bmod p, \prod_{j=1}^{k} a_{j,2} \bmod p)$
  if $\mathsf{CheckBallotProof}(\pi, \hat{x}, e, pk)$ **then** // see Alg. 7.24
    return *true*
return *false*

Algorithm 7.22: Checks if a ballot α obtained from voter v is valid. For this, voter v must not have submitted a valid ballot before, $\hat{x}$ must be the public voting credential of voter v, the length $k = |\mathbf{a}|$ must be equal to $k' = \sum_{j=1}^{t} k_{vj}$, and π must be valid.

Algorithm: $\mathsf{HasBallot}(v, B)$

Input: Voter index $v \in \mathbb{N}$
Ballot list $B = \langle(v_i, \alpha_i, z_i)\rangle_{i=0}^{N_B-1}$, $v_i \in \mathbb{N}$

foreach $(v_i, \alpha_i, z_i) \in B$ **do** // use binary search or hash table for better performance
  if $v = v_i$ **then**
    return *true*

return *false*

Algorithm 7.23: Checks if the ballot list B contains an entry for voter v.

Algorithm: $\mathsf{CheckBallotProof}(\pi, \hat{x}, e, pk)$

Input: Ballot proof $\pi = (t, s)$, $t = (t_1, t_2, t_3) \in \mathbb{G}_{\hat{q}} \times \mathbb{G}_q^2$, $s = (s_1, s_2, s_3) \in \mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q$
Public voting credential $\hat{x} \in \mathbb{Z}_{\hat{q}}$
ElGamal encryption $e = (a, b) \in \mathbb{G}_q \times \mathbb{G}_q$
Encryption key $pk \in \mathbb{G}_q$

$y \leftarrow (\hat{x}, a, b)$
$c \leftarrow \mathsf{GetNIZKPChallenge}(y, t, \tau)$ // see Alg. 7.4
$t'_1 \leftarrow \hat{x}^{-c} \cdot \hat{g}^{s_1} \bmod \hat{p}$
$t'_2 \leftarrow a^{-c} \cdot s_2 \cdot pk^{s_3} \bmod p$
$t'_3 \leftarrow b^{-c} \cdot g^{s_3} \bmod p$
return $(t_1 = t'_1) \wedge (t_2 = t'_2) \wedge (t_3 = t'_3)$

Algorithm 7.24: Checks the correctness of a NIZKP π generated by Alg. 7.21. The public values of this proof are the public voting credential $\hat{x}$ and the ElGamal encryption $e = (a, b)$.

Algorithm: $\mathsf{GenResponse}(v, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P})$

Input: Voter index $v \in \{1, \ldots, N_E\}$
Queries $\mathbf{a} = (a_1, \ldots, a_k)$, $a_j \in \mathbb{G}_q$
Encryption key $pk \in \mathbb{G}_q$
Number of candidates $\mathbf{n} = (n_1, \ldots, n_t)$, $n_j \geqslant 2$, $n = \sum_{j=1}^{t} n_j$
Number of selections $\mathbf{k} = (k_1, \ldots, k_t)$, $0 < k_j < n_j$
Eligibility matrix $\mathbf{E} = (e_{ij})_{N_E \times t}$, $e_{ij} \in \mathbb{B}$
Points $\mathbf{P} = (p_{ij})_{N_E \times n}$, $p_{ij} = (x_{ij}, y_{ij})$, $x_{ij} \in \mathbb{Z}_{p'}$, $y_{ij} \in \mathbb{Z}_{p'}$

for $j = 1, \ldots, n$ **do**
- $M_j \leftarrow \mathsf{ToByteArray}(x_{vj}, \frac{L_M}{2}) \parallel \mathsf{ToByteArray}(y_{vj}, \frac{L_M}{2})$ // see Alg. 4.4

$z_1, z_2 \in_R \mathbb{Z}_q$

for $j = 1, \ldots, k$ **do**
- $\beta_j \in_R \mathbb{G}_q$
- $b_j \leftarrow a_{j,1}^{z_1} a_{j,2}^{z_2} \beta_j \bmod p$

$\ell_M \leftarrow \lceil L_M / L \rceil$
$\mathbf{p} \leftarrow \mathsf{getPrimes}(n)$ // $\mathbf{p} = (p_1, \ldots, p_n)$, see Alg. 7.1
$n' \leftarrow 0$, $k' \leftarrow 0$

for $l = 1, \ldots, t$ **do**
- **for** $i = n' + 1, \ldots, n' + n_l$ **do**
  - $p'_i \leftarrow p_i^{z_1} \bmod p$
  - **for** $j = k' + 1, \ldots, k' + e_{vl} k_l$ **do**
    - $k_{ij} \leftarrow p'_i \beta_j \bmod p$
    - $K_{ij} \leftarrow \mathsf{Truncate}(\parallel_{c=1}^{\ell_M} \mathsf{RecHash}_L(k_{ij}, c), L_M)$ // see Alg. 4.9
    - $C_{ij} \leftarrow M_i \oplus K_{ij}$
- $n' \leftarrow n' + n_l$, $k' \leftarrow k' + e_{vl} k_l$

$\mathbf{b} \leftarrow (b_1, \ldots, b_k)$, $\mathbf{C} \leftarrow (C_{ij})_{n \times k}$, $d \leftarrow pk^{z_1} g^{z_2} \bmod p$
$\beta \leftarrow (\mathbf{b}, \mathbf{C}, d)$
$z = (z_1, z_2)$
return (β, z) // $\beta \in \mathbb{G}_q^k \times (\mathcal{B}^{L_M})^{nk} \times \mathbb{G}_q$, $z \in \mathbb{Z}_q^2$

Algorithm 7.25: Generates the response β for the given OT query $\mathbf{a}$. The messages to transfer are byte array representations of the n points $(p_{v,1}, \ldots, p_{v,n})$. Along with β, the algorithm also returns the randomizations z used to generate the response.

Algorithm: $\mathsf{GetPointMatrix}(\boldsymbol{\beta}, \mathbf{s}, \mathbf{r})$

Input: OT responses $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_s)$, $\beta_j \in \mathbb{G}_q^k \times (\mathcal{B}^{L_M})^{nk} \times \mathbb{G}_q$
Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$
Randomizations $\mathbf{r} = (r_1, \ldots, r_k)$, $r_j \in \mathbb{Z}_q$

for $i = 1, \ldots, s$ **do**
  $\mathbf{p}_i \leftarrow \mathsf{GetPoints}(\beta_i, \mathbf{s}, \mathbf{r})$ // $\mathbf{p}_j = (p_{i,1}, \ldots, p_{i,k})$, see Alg. 7.27

$\mathbf{P_s} \leftarrow (p_{ij})_{s \times k}$
return $\mathbf{P_s}$ // $\mathbf{P_s} \in (\mathbb{Z}_p^2)^{sk}$

Algorithm 7.26: Computes the s-by-k matrix $\mathbf{P_s} = (p_{ij})_{s \times k}$ of the points obtained from the s authorities for the selection $\mathbf{s}$. The points are derived from the messages included in the OT responses $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_s)$.

Algorithm: $\mathsf{GetPoints}(\beta, \mathbf{s}, \mathbf{r})$

Input: OT response $\beta = (\mathbf{b}, \mathbf{C}, d)$, $\mathbf{b} = (b_1, \ldots, b_k)$, $b_j \in \mathbb{G}_q$, $\mathbf{C} = (C_{ij})_{n \times k}$, $C_{ij} \in \mathcal{B}^{L_M}$, $d \in \mathbb{G}_q$
Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$
Randomizations $\mathbf{r} = (r_1, \ldots, r_k)$, $r_j \in \mathbb{Z}_q$

$\ell_M \leftarrow \lceil L_M / L \rceil$
for $j = 1, \ldots, k$ **do**
  $k_j \leftarrow b_j \cdot d^{-r_j} \bmod p$
  $K_j \leftarrow \mathsf{Truncate}(\|_{c=1}^{\ell_M} \mathsf{RecHash}_L(k_j, c), L_M)$ // see Alg. 4.9
  $M_j \leftarrow C_{s_j, j} \oplus K_j$
  $x_j \leftarrow \mathsf{ToInteger}(\mathsf{Truncate}(M_j, \frac{L_M}{2}))$ // see Alg. 4.5
  $y_j \leftarrow \mathsf{ToInteger}(\mathsf{Skip}(M_j, \frac{L_M}{2}))$ // see Alg. 4.5
  if $x_j \geqslant p'$ **or** $y_j \geqslant p'$ **then**
    return $\perp$ // point not in $\mathbb{Z}_{p'}^2$
  $p_j \leftarrow (x_j, y_j)$

$\mathbf{p} \leftarrow (p_1, \ldots, p_k)$
return $\mathbf{p}$ // $\mathbf{p} \in (\mathbb{Z}_{p'}^2)^k$

Algorithm 7.27: Computes the k transferred points $\mathbf{p} = (p_1, \ldots, p_k)$ from the OT response β using the random values $\mathbf{r}$ from the OT query and the selection $\mathbf{s}$. The algorithm returns $\perp$, if some transfered point lies outside $\mathbb{Z}_{p'}^2$. By selecting the largest possible prime p' for a given bit length, this exception is very unlikely (see Section 8.2).

Algorithm: $\mathsf{GetReturnCodes}(\mathbf{s}, \mathbf{P_s})$

Input: Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$
Points $\mathbf{P_s} = (p_{ij})_{s \times k}$, $p_{ij} \in \mathbb{Z}^2_{p'}$

for $j = 1, \ldots, k$ **do**
  for $i = 1, \ldots, s$ **do**
    $R_{ij} \leftarrow \mathsf{Truncate}(\mathsf{RecHash}_L(p_{ij}), L_R)$ // see Alg. 4.9
  $R_j \leftarrow \mathsf{MarkByteArray}(\bigoplus_{i=1}^{s} R_{ij}, s_j - 1, n_{\max})$ // see Alg. 4.1
  $RC_{s_j} \leftarrow \mathsf{ToString}(R, A_R)$ // see Alg. 4.8

$\mathbf{rc_s} \leftarrow (RC_{s_1}, \ldots, RC_{s_k})$

return $\mathbf{rc_s}$ // $\mathbf{rc} \in (A_F^{\ell_F})^k$

Algorithm 7.28: Computes the k verification codes $\mathbf{rc_s} = (RC_{s_1}, \ldots, RC_{s_k})$ for the selected candidates by combining the hash values of the transferred points $p_{ij} \in \mathbf{P_s}$ from different authorities.

Algorithm: $\mathsf{CheckReturnCodes}(\mathbf{rc}, \mathbf{rc}', \mathbf{s})$

Input: Printed verification codes $\mathbf{rc} = (RC_1, \ldots, RC_n)$, $RC_i \in A_R^{\ell_R}$
Displayed verification codes $\mathbf{rc}' = (RC'_1, \ldots, RC'_k)$, $RC'_j \in A_R^{\ell_R}$
Selections $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$

return $\bigwedge_{j=1}^{k}(RC_{s_j} = RC'_j)$

Algorithm 7.29: Checks if every displayed verification code RC'_i matches with the verification code RC_{s_i} of the selected candidate s_i as printed on the voting card. Note that this algorithm is executed by humans.

Algorithm: $\mathsf{GenConfirmation}(Y, \mathbf{P})$

Input: Confirmation code $Y \in A_Y^{\ell_Y}$
Points $\mathbf{P} = (p_{ij})_{s \times k}$, $p_{ij} \in \mathbb{Z}^2_{p'}$

for $i = 1 \ldots, s$ **do**
  $\mathbf{p}_i \leftarrow (p_{i,1}, \ldots, p_{i,k})$
  $y'_i \leftarrow \mathsf{GetValue}(\mathbf{p}_i)$ // see Alg. 7.31

$y \leftarrow \mathsf{ToInteger}(Y) \bmod \hat{q}$, $y' \leftarrow \sum_{i=1}^{s} y'_i \bmod \hat{q}$ // see Alg. 4.7

$\hat{y} \leftarrow \hat{g}^{y+y' \bmod \hat{q}} \bmod \hat{p}$

$\pi \leftarrow \mathsf{GenConfirmationProof}(y, y', \hat{y})$ // $\pi = (t, s)$, see Alg. 7.32

$\gamma \leftarrow (\hat{y}, \pi)$

return γ // $\gamma \in \mathbb{G}_{\hat{q}} \times (\mathbb{G}_q \times \mathbb{Z}_{\hat{q}})$

Algorithm 7.30: Generates the confirmation γ, which consists of the public confirmation credential $\hat{y}$ and a NIZKP of knowledge π of the secret confirmation and validity credentials y and y'.

Algorithm: $\mathsf{GetValue}(\mathbf{p})$
Input: Points $\mathbf{p} = (p_1, \ldots, p_k)$, $p_j = (x_j, y_j) \in \mathbb{Z}^2_{p'}$, $k \geqslant 0$

$y \leftarrow 0$
for $i = 1, \ldots, k$ **do**
  $n \leftarrow 1$, $d \leftarrow 1$
  for $j = 1, \ldots, k$ **do**
    if $i \neq j$ **then**
      $n \leftarrow n \cdot x_j \bmod p'$
      $d \leftarrow d \cdot (x_j - x_i) \bmod p'$
  $y \leftarrow y + y_i \cdot \frac{n}{d} \bmod p'$
return y // $y \in \mathbb{Z}_{p'}$

Algorithm 7.31: Computes a polynomial $A(X)$ of degree $k-1$ from given points $\mathbf{p} = (p_1, \ldots, p_k)$ using Lagrange's interpolation method and returns the value $y = A(0)$.

Algorithm: $\mathsf{GenConfirmationProof}(y, y', \hat{y})$
Input: Secret confirmation credential $y \in \mathbb{Z}_{\hat{q}}$
Secret validity credential $y' \in \mathbb{Z}_{\hat{q}}$
Public confirmation credential $\hat{y} \in \mathbb{G}_{\hat{q}}$

$\omega \in_R \mathbb{Z}_{\hat{q}}$
$t \leftarrow \hat{g}^{\omega} \bmod \hat{p}$
$c \leftarrow \mathsf{GetNIZKPChallenge}(\hat{y}, t, \tau)$ // see Alg. 7.4
$s \leftarrow \omega + c \cdot (y + y') \bmod \hat{q}$
$\pi \leftarrow (t, s)$
return π // $\pi \in \mathbb{G}_{\hat{q}} \times \mathbb{Z}_{\hat{q}}$

Algorithm 7.32: Generates a NIZKP of knowledge of the secret confirmation and validity credentials y and y' that matches with a given public confirmation credential $\hat{y}$. Note that this proof is equivalent to a Schnorr identification proof [45]. For the verification of π, see Alg. 7.35.

Algorithm: $\mathsf{CheckConfirmation}(v, \gamma, \hat{\mathbf{y}}, B, C)$

Input: Voter index $v \in \{1, \ldots, N_E\}$
Confirmation $\gamma = (\hat{y}, \pi)$, $\hat{y} \in \mathbb{G}_{\hat{q}}$, $\pi \in \mathbb{G}_{\hat{q}} \times \mathbb{Z}_{\hat{q}}$
Public confirmation credentials $\hat{\mathbf{y}} = (\hat{y}_1, \ldots, \hat{y}_{N_E})$, $\hat{y}_i \in \mathbb{G}_{\hat{q}}$
Ballot list $B = \langle (v_i, \alpha_i, z_i) \rangle_{i=0}^{N_B-1}$, $v_i \in \{1, \ldots, N_E\}$
Confirmation list $C = \langle (v_i, \gamma_i) \rangle_{i=0}^{N_C-1}$, $v_i \in \{1, \ldots, N_E\}$

if $\mathsf{HasBallot}(v, B)$ **and** $\neg\mathsf{HasConfirmation}(v, C)$ **and** $\hat{y} = \hat{y}_v$ **then** // see Alg. 7.23, 7.34
  if $\mathsf{CheckConfirmationProof}(\pi, \hat{y})$ **then** // see Alg. 7.35
    return *true*
return *false*

Algorithm 7.33: Checks if a confirmation γ obtained from voter i is valid. For this, voter v must have submitted a valid ballot before, but not a valid confirmation. The check then succeeds if π is valid and if $\hat{y}$ is the public confirmation credential of voter v.

Algorithm: $\mathsf{HasConfirmation}(v, C)$

Input: Voter index $v \in \mathbb{N}$
Confirmation list $C = \langle (v_j, \gamma_j) \rangle_{j=0}^{N_C-1}$, $v_j \in \mathbb{N}$

foreach $(v_j, \gamma_j) \in C$ **do** // use binary search or hash table for better performance
  if $v = v_j$ **then**
    return *true*
return *false*

Algorithm 7.34: Checks if the confirmation list C contains an entry for voter v.

Algorithm: $\mathsf{CheckConfirmationProof}(\pi, \hat{y})$

Input: Confirmation proof $\pi = (t, s)$, $t \in \mathbb{G}_{\hat{q}}$, $s \in \mathbb{Z}_{\hat{q}}$
Public confirmation credential $\hat{y} \in \mathbb{G}_{\hat{q}}$

$c \leftarrow \mathsf{GetNIZKPChallenge}(\hat{y}, t, \tau)$ // see Alg. 7.4
$t' \leftarrow \hat{y}^{-c} \cdot \hat{g}^{s} \bmod \hat{p}$
return $(t = t')$

Algorithm 7.35: Checks the correctness of a NIZKP π generated by Alg. 7.32. The public value of this proof is the public confirmation credential $\hat{y}$.

Algorithm: $\mathsf{GetFinalization}(v, \mathbf{P}, B)$

Input: Voter index $v \in \{1, \dots, N_E\}$
Points $\mathbf{P} = (p_{ij})_{N_E \times n}$, $p_{ij} \in \mathbb{Z}_{p'}^2$
Ballot list $B = \langle (v_i, \alpha_i, z_i) \rangle_{i=0}^{N_B - 1}$, $v_i \in \{1, \dots, N_E\}$

$\mathbf{p} \leftarrow (p_{v,1}, \dots, p_{v,n})$
$F \leftarrow \mathsf{Truncate}(\mathsf{RecHash}_L(\mathbf{p}), L_F)$ // see Alg. 4.9
foreach $(v_i, \alpha_i, z_i) \in B$ **do** // use binary search or hash table for better performance
  if $v = v_i$ **then**
    $\delta \leftarrow (F, z_i)$
    return δ // $\delta \in \mathcal{B}^{L_F} \times \mathbb{Z}_q^2$
return $\perp$ // no entry for v in B

Algorithm 7.36: Computes the finalization code F for voter v from the given points $(p_{v,1}, \dots, p_{v,n})$ and returns F together with the randomizations used in the creation of the OT response.

Algorithm: $\mathsf{GetFinalizationCode}(\boldsymbol{\delta})$

Input: Finalizations $\boldsymbol{\delta} = (\delta_1, \dots, \delta_s)$, $\delta_j = (F_j, z_j)$, $F_j \in \mathcal{B}^{L_F}$, $z_j \in \mathbb{Z}_q^2$

$FC \leftarrow \mathsf{ToString}(\bigoplus_{j=1}^{s} F_j, A_F)$ // see Alg. 4.8
return FC // $FC \in A_F^{\ell_F}$

Algorithm 7.37: Computes a finalization code FC by combining the values F_j received from the authorities.

Algorithm: $\mathsf{CheckFinalizationCode}(FC, FC')$

Input: Printed finalization code $FC \in A_F^{\ell_F}$
Displayed finalization code $FC' \in A_F^{\ell_F}$

return $FC = FC'$

Algorithm 7.38: Checks if the displayed finalization code FC' matches with the finalization code FC from the voting card. Note that this algorithm is executed by humans.

7.5. Post-Election Phase

The main actors in the process at the end of an election are the election authorities. Corresponding algorithms are shown in Table 7.4. To initiate the mixing process, the first election authority calls Alg. 7.39 to cleanse the list of submitted ballots and to extract a sorted list of encrypted votes to shuffle. By calling Algs. 7.40 and 7.43, this list is shuffled according to a random permutation and a NIZKP of shuffle is generated. This step is repeated by every election authority. The final result obtained from the last shuffle is the list of encrypted votes that will be decrypted. Before computing corresponding partial decryptions, each election authority calls Alg. 7.46 to check the correctness of the whole shuffle process. The partial decryptions are then computed using Alg. 7.48 and corresponding decryption proofs are generated using Alg. 7.49. The information exchange during this whole process goes over the bulletin board. After terminating all tasks, the process is handed over from the election authorities to the election administrator, who calls Alg. 7.50 to check all decryption proofs and Alg. 7.53 to obtain the final election result. We refer to Section 6.5.3 for a more detailed description of this process.

Nr.	Algorithm	Called by	Protocol
7.39	GetEncryptions$(B, C, \mathbf{n}, \mathbf{w})$	Election authority	6.7
7.34	↳ HasConfirmation(v, C)		
7.40	GenShuffle$(\mathbf{e}, pk)$	Election authority	
7.41	↳ GenPermutation(N)		
7.42	↳ GenReEncryption(e, pk)		
7.43	GenShuffleProof$(\mathbf{e}, \mathbf{e}', \mathbf{r}', \psi, pk)$	Election authority	
7.44	↳ GenPermutationCommitment$(\psi, \mathbf{h})$		
7.45	↳ GenCommitmentChain$(c_0, \mathbf{u})$		
7.39	GetEncryptions$(B, C, \mathbf{n}, \mathbf{w})$	Election authority	6.8
7.46	CheckShuffleProofs$(\boldsymbol{\pi}, e_0, \mathbf{E}, pk, j)$	Election authority	
7.47	↳ CheckShuffleProof$(\pi, \mathbf{e}, \mathbf{e}', pk)$		
7.48	GetPartialDecryptions$(\mathbf{e}, sk_j)$	Election authority	
7.49	GenDecryptionProof$(sk_j, pk_j, \mathbf{e}, \mathbf{b}')$	Election authority	
7.50	CheckDecryptionProofs$(\boldsymbol{\pi}', \mathbf{pk}, \mathbf{e}, \mathbf{B}')$	Election administrator	6.9
7.51	↳ CheckDecryptionProof$(\pi', pk_j, \mathbf{e}, \mathbf{b}')$		
7.52	GetDecryptions$(\mathbf{e}, \mathbf{B}')$	Election administrator	
7.53	GetVotes$(\mathbf{m}, \mathbf{n}, \mathbf{w})$	Election administrator	

Table 7.4.: Overview of algorithms and sub-algorithms of the post-election phase.

Algorithm: $\mathsf{GetEncryptions}(B, C, \mathbf{n}, \mathbf{w})$

Input: Ballot list $B = \langle(v_i, \alpha_i, z_i)\rangle_{i=0}^{N_B-1}$, $v_i \in \{1, \ldots, N_E\}$
Confirmation list $C = \langle(v_i, \gamma_i)\rangle_{i=0}^{N_C-1}$, $v_i \in \{1, \ldots, N_E\}$
Number of candidates $\mathbf{n} = (n_1, \ldots, n_t)$, $n_j \geqslant 2$
Counting circles $\mathbf{w} = (w_1, \ldots, w_{N_E})$, $w_i \in \mathbb{N}$

$n \leftarrow \sum_{j=1}^{t} n_j$
$w \leftarrow \max_{i=1}^{N_E} w_i$
$\mathbf{p} \leftarrow \mathsf{getPrimes}(n + w)$ // $\mathbf{p} = (p_1, \ldots, p_{n+w})$, see Alg. 7.1
$i \leftarrow 1$ // loop over $i = 1, \ldots, N_C$
foreach $(v, \alpha, z) \in B$ **do** // $\alpha = (\hat{x}, \mathbf{a}, \pi)$, $\mathbf{a} = (a_1, \ldots, a_k)$, $a_j = (a_{j,1}, a_{j,2}) \in \mathbb{G}_q^2$
  if $\mathsf{HasConfirmation}(v, C)$ **then** // see Alg. 7.34
    $a_1 \leftarrow p_{n+w_v} \prod_{j=1}^{k} a_{j,1} \bmod p$
    $a_2 \leftarrow \prod_{j=1}^{k} a_{j,2} \bmod p$
    $e_i \leftarrow (a_1, a_2)$
    $i \leftarrow i + 1$

$\mathbf{e} \leftarrow \mathsf{Sort}_{\preceq}(e_1, \ldots, e_{N_C})$
return $\mathbf{e}$ // $\mathbf{e} \in (\mathbb{G}_q^2)^{N_C}$

Algorithm 7.39: Computes a sorted list of ElGamal encryptions from the list of submitted ballots, for which a valid confirmation is available. The counting circles w_v are added to the encryptions. Sorting the resulting list is necessary to guarantee a unique order. For this, we define a total order over $\mathbb{G}_q^2$ by $e_i \preceq e_j \Leftrightarrow (a_i < a_j) \vee (a_i = a_j \wedge b_i \leqslant b_j)$, for $e_i = (a_i, b_i)$ and $e_j = (a_j, b_j)$.

Algorithm: $\mathsf{GenShuffle}(\mathbf{e}, pk)$

Input: ElGamal encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i \in \mathbb{G}_q^2$
Encryption key $pk \in \mathbb{G}_q$

$\psi \leftarrow \mathsf{GenPermutation}(N)$ // $\psi = (j_1, \ldots, j_N) \in \Psi_N$, see Alg. 7.41
for $i = 1, \ldots, N$ **do**
  $(e'_i, r'_i) \leftarrow \mathsf{GenReEncryption}(e_i, pk)$ // see Alg. 7.42

$\mathbf{e}' \leftarrow (e'_{j_1}, \ldots, e'_{j_N})$
$\mathbf{r}' \leftarrow (r'_1, \ldots, r'_N)$
return $(\mathbf{e}', \mathbf{r}', \psi)$ // $\mathbf{e}' \in (\mathbb{G}_q^2)^N$, $\mathbf{r}' \in \mathbb{Z}_q^N$, $\psi \in \Psi_N$

Algorithm 7.40: Generates a random permutation $\psi \in \Psi_N$ and uses it to shuffle a given list $\mathbf{e} = (e_1, \ldots, e_N)$ of ElGamal encryptions $e_i = (a_i, b_i) \in \mathbb{G}_q^2$. With $\Psi_N = \{(j_1, \ldots, j_N) : j_i \in \{1, \ldots, N\}, j_{i_1} \neq j_{i_2}, \forall i_1 \neq i_2\}$ we denote the set of all $N!$ possible permutations of the indices $\{1, \ldots, N\}$.

Algorithm: $\mathsf{GenPermutation}(N)$
Input: Permutation size $N \in \mathbb{N}$
$I \leftarrow \langle 1, \ldots, N \rangle$
for $i = 0, \ldots, N-1$ **do**
  $k \in_R \{i, \ldots, N-1\}$
  $j_{i+1} \leftarrow I[k]$
  $I[k] \leftarrow I[i]$
$\psi \leftarrow (j_1, \ldots, j_N)$
return ψ // $\psi \in \Psi_N$

Algorithm 7.41: Generates a random permutation $\psi \in \Psi_N$ following Knuth's shuffle algorithm [33, pp. 139–140].

Algorithm: $\mathsf{GenReEncryption}(e, pk)$
Input: ElGamal encryption $e = (a, b)$, $a \in \mathbb{G}_q$, $b \in \mathbb{G}_q$
Encryption key $pk \in \mathbb{G}_q$
$r' \in_R \mathbb{Z}_q$
$a' \leftarrow a \cdot pk^{r'} \bmod p$
$b' \leftarrow b \cdot g^{r'} \bmod p$
$e' \leftarrow (a', b')$
return (e', r') // $e' \in \mathbb{G}_q^2$, $r' \in \mathbb{Z}_q$

Algorithm 7.42: Generates a re-encryption $e' = (a \cdot pk^{r'}, b \cdot g^{r'})$ of the given ElGamal encryption $e = (a, b) \in \mathbb{G}_q^2$. The re-encryption e' is returned together with the randomization $r' \in \mathbb{Z}_q$.

Algorithm: $\mathsf{GenShuffleProof}(\mathbf{e}, \mathbf{e}', \mathbf{r}', \psi, pk)$

Input: ElGamal encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i) \in \mathbb{G}_q^2$
Shuffled ElGamal encryptions $\mathbf{e}' = (e'_1, \ldots, e'_N)$, $e'_i = (a'_i, b'_i) \in \mathbb{G}_q^2$
Re-encryption randomizations $\mathbf{r}' = (r'_1, \ldots, r'_N)$, $r'_i \in \mathbb{Z}_q$
Permutation $\psi = (j_1, \ldots, j_N) \in \Psi_N$
Encryption key $pk \in \mathbb{G}_q$

$\mathbf{h} \leftarrow \mathsf{GetGenerators}(N)$ // see Alg. 7.3
$(\mathbf{c}, \mathbf{r}) \leftarrow \mathsf{GenPermutationCommitment}(\psi, \mathbf{h})$ // $\mathbf{c} = (c_1, \ldots, c_N)$, see Alg. 7.44
$\mathbf{u} \leftarrow \mathsf{GetNIZKPChallenges}(N, (\mathbf{e}, \mathbf{e}', \mathbf{c}), \tau)$ // $\mathbf{u} = (u_1, \ldots, u_N)$, see Alg. 7.5
for $i = 1, \ldots, N$ **do**
  $u'_i \leftarrow u_{j_i}$
$\mathbf{u}' \leftarrow (u'_1, \ldots, u'_N)$
$(\hat{\mathbf{c}}, \hat{\mathbf{r}}) \leftarrow \mathsf{GenCommitmentChain}(h, \mathbf{u}')$ // $\hat{\mathbf{c}} = (\hat{c}_1, \ldots, \hat{c}_N)$, see Alg. 7.45
for $i = 1, \ldots, 4$ **do**
  $\omega_i \in_R \mathbb{Z}_q$
for $i = 1, \ldots, N$ **do**
  $\hat{\omega}_i \in_R \mathbb{Z}_q$, $\omega'_i \in_R \mathbb{Z}_q$
$t_1 \leftarrow g^{\omega_1} \bmod p$
$t_2 \leftarrow g^{\omega_2} \bmod p$
$t_3 \leftarrow g^{\omega_3} \prod_{i=1}^N h_i^{\omega'_i} \bmod p$
$(t_{4,1}, t_{4,2}) \leftarrow (pk^{-\omega_4} \prod_{i=1}^N (a'_i)^{\omega'_i} \bmod p, g^{-\omega_4} \prod_{i=1}^N (b'_i)^{\omega'_i} \bmod p)$
$\hat{c}_0 \leftarrow h$
for $i = 1, \ldots, N$ **do**
  $\hat{t}_i \leftarrow g^{\hat{\omega}_i} \hat{c}_{i-1}^{\omega'_i} \bmod p$
$t \leftarrow (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), (\hat{t}_1, \ldots, \hat{t}_N))$
$y \leftarrow (\mathbf{e}, \mathbf{e}', \mathbf{c}, \hat{\mathbf{c}}, pk)$
$c \leftarrow \mathsf{GetNIZKPChallenge}(y, t, \tau)$ // see Alg. 7.4
$\bar{r} \leftarrow \sum_{i=1}^N r_i \bmod q$, $s_1 \leftarrow \omega_1 + c \cdot \bar{r} \bmod q$
$v_N \leftarrow 1$
for $i = N-1, \ldots, 1$ **do**
  $v_i \leftarrow u'_{i+1} v_{i+1} \bmod q$
$\hat{r} \leftarrow \sum_{i=1}^N \hat{r}_i v_i \bmod q$, $s_2 \leftarrow \omega_2 + c \cdot \hat{r} \bmod q$
$\tilde{r} \leftarrow \sum_{i=1}^N r_i u_i \bmod q$, $s_3 \leftarrow \omega_3 + c \cdot \tilde{r} \bmod q$
$r' \leftarrow \sum_{i=1}^N r'_i u_i \bmod q$, $s_4 \leftarrow \omega_4 + c \cdot r' \bmod q$
for $i = 1, \ldots, N$ **do**
  $\hat{s}_i \leftarrow \hat{\omega}_i + c \cdot \hat{r}_i \bmod q$, $s'_i \leftarrow \omega'_i + c \cdot u'_i \bmod q$
$s \leftarrow (s_1, s_2, s_3, s_4, (\hat{s}_1, \ldots, \hat{s}_N), (s'_1, \ldots, s'_N))$
$\pi \leftarrow (t, s, \mathbf{c}, \hat{\mathbf{c}})$
return π // $\pi \in (\mathbb{G}_q^3 \times \mathbb{G}_q^2 \times \mathbb{G}_q^N) \times (\mathbb{Z}_q^4 \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N) \times \mathbb{G}_q^N \times \mathbb{G}_q^N$

Algorithm 7.43: Generates a NIZKP of shuffle relative to ElGamal encryptions $\mathbf{e}$ and $\mathbf{e}'$, which is equivalent to proving knowledge of a permutation ψ and randomizations $\mathbf{r}'$ such that $\mathbf{e}' = \mathsf{Shuffle}_{pk}(\mathbf{e}, \mathbf{r}', \psi)$. The algorithm implements Wikström's proof of a shuffle [49, 47], except for the fact that the offline and online phases are merged. For the proof verification, see Alg. 7.47. For further background information we refer to Section 5.5.

Algorithm: $\mathsf{GenPermutationCommitment}(\psi, \mathbf{h})$

Input: Permutation $\psi = (j_1, \ldots, j_N) \in \Psi_N$
Independent generators $\mathbf{h} = (h_1, \ldots, h_N)$, $h_i \in \mathbb{G}_q \backslash \{1\}$

for $i = 1, \ldots, N$ **do**
  $r_{j_i} \in_R \mathbb{Z}_q$
  $c_{j_i} \leftarrow g^{r_{j_i}} \cdot h_i \bmod p$
$\mathbf{c} \leftarrow (c_1, \ldots, c_N)$
$\mathbf{r} \leftarrow (r_1, \ldots, r_N)$
return $(\mathbf{c}, \mathbf{r})$ // $\mathbf{c} \in \mathbb{G}_q^N$, $\mathbf{r} \in \mathbb{Z}_q^N$

Algorithm 7.44: Generates a commitment $\mathbf{c} = com(\psi, \mathbf{r})$ to a permutation ψ by committing to the columns of the corresponding permutation matrix. This algorithm is used in Alg. 7.43.

Algorithm: $\mathsf{GenCommitmentChain}(c_0, \mathbf{u})$

Input: Initial commitment $c_0 \in \mathbb{G}_q$
Public challenges $\mathbf{u} = (u_1, \ldots, u_N)$, $u_i \in \mathbb{Z}_q$

for $i = 1, \ldots, N$ **do**
  $r_i \in_R \mathbb{Z}_q$
  $c_i \leftarrow g^{r_i} \cdot c_{i-1}^{u_i} \bmod p$
$\mathbf{c} \leftarrow (c_1, \ldots, c_N)$
$\mathbf{r} \leftarrow (r_1, \ldots, r_N)$
return $(\mathbf{c}, \mathbf{r})$ // $\mathbf{c} \in \mathbb{G}_q^N$, $\mathbf{r} \in \mathbb{Z}_q^N$

Algorithm 7.45: Generates a commitment chain $c_0 \rightarrow c_1 \rightarrow \cdots \rightarrow c_N$ relative to a list of public challenges $\mathbf{u}$ and starting with a given commitment c_0. This algorithm is used in Alg. 7.43.

Algorithm: $\mathsf{CheckShuffleProofs}(\boldsymbol{\pi}, \mathbf{e}_0, \mathbf{E}, pk, i)$

Input: Shuffle proofs $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_s)$, $\pi_j \in (\mathbb{G}_q^3 \times \mathbb{G}_q^2 \times \mathbb{G}_q^N) \times (\mathbb{Z}_q^4 \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N) \times \mathbb{G}_q^N \times \mathbb{G}_q^N$
ElGamal encryptions $\mathbf{e}_0 = (e_{1,0}, \ldots, e_{N,0})$, $e_{i,0} \in \mathbb{G}_q^2$
Shuffled ElGamal encryptions $\mathbf{E} = (e_{ij})_{N \times s}$, $e_{ij} \in \mathbb{G}_q^2$
Encryption key $pk \in \mathbb{G}_q$
Authority index $i \in \{1, \ldots, s\}$

for $j = 1, \ldots, s$ **do**
  $\mathbf{e}_j \leftarrow (e_{1,j}, \ldots, e_{N,j})$
  if $i \neq j$ **then** // check proofs from others only
    if $\neg\mathsf{CheckShuffleProof}(\pi_j, \mathbf{e}_{j-1}, \mathbf{e}_j, pk)$ **then** // see Alg. 7.47
      return *false*
return *true*

Algorithm 7.46: Checks if a chain of shuffle proofs generated by s different authorities is correct.

Algorithm: $\mathsf{CheckShuffleProof}(\pi, \mathbf{e}, \mathbf{e}', pk)$

Input: Shuffle proof $\pi = (t, s, \mathbf{c}, \hat{\mathbf{c}})$, $t = (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), (\hat{t}_1, \ldots, \hat{t}_N))$,
$s = (s_1, s_2, s_3, s_4, (\hat{s}_1, \ldots, \hat{s}_N), (s'_1, \ldots, s'_N))$, $\mathbf{c} = (c_1, \ldots, c_N)$, $\hat{\mathbf{c}} = (\hat{c}_1, \ldots, \hat{c}_N)$
ElGamal encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i \in \mathbb{G}_q^2$
Shuffled ElGamal encryptions $\mathbf{e}' = (e'_1, \ldots, e'_N)$, $e'_i \in \mathbb{G}_q^2$
Encryption key $pk \in \mathbb{G}_q$

$\mathbf{h} \leftarrow \mathsf{GetGenerators}(N)$ // see Alg. 7.3
$\mathbf{u} \leftarrow \mathsf{GetNIZKPChallenges}(N, (\mathbf{e}, \mathbf{e}', \mathbf{c}), \tau)$ // $\mathbf{u} = (u_1, \ldots, u_N)$, see Alg. 7.5
$y \leftarrow (\mathbf{e}, \mathbf{e}', \mathbf{c}, \hat{\mathbf{c}}, pk)$
$c \leftarrow \mathsf{GetNIZKPChallenge}(y, t, \tau)$ // see Alg. 7.4
$\bar{c} \leftarrow \prod_{i=1}^N c_i / \prod_{i=1}^N h_i \bmod p$
$u \leftarrow \prod_{i=1}^N u_i \bmod q$
$\hat{c} \leftarrow \hat{c}_N / h^u \bmod p$
$\tilde{c} \leftarrow \prod_{i=1}^N c_i^{u_i} \bmod p$
$(a', b') \leftarrow (\prod_{i=1}^N a_i^{u_i} \bmod p, \prod_{i=1}^N b_i^{u_i} \bmod p)$
$t'_1 \leftarrow \bar{c}^{-c} \cdot g^{s_1} \bmod p$
$t'_2 \leftarrow \hat{c}^{-c} \cdot g^{s_2} \bmod p$
$t'_3 \leftarrow \tilde{c}^{-c} \cdot g^{s_3} \prod_{i=1}^N h_i^{s'_i} \bmod p$
$(t'_{4,1}, t'_{4,2}) \leftarrow ((a')^{-c} \cdot pk^{-s_4} \prod_{i=1}^N (a'_i)^{s'_i} \bmod p, (b')^{-c} \cdot g^{-s_4} \prod_{i=1}^N (b'_i)^{s'_i} \bmod p)$
for $i = 1, \ldots, N$ **do**
  $\hat{t}'_i \leftarrow \hat{c}_i^{-c} \cdot g^{\hat{s}_i} \cdot \hat{c}_{i-1}^{s'_i} \bmod p$
return $(t_1 = t'_1) \wedge (t_2 = t'_2) \wedge (t_3 = t'_3) \wedge (t_{4,1} = t'_{4,1}) \wedge (t_{4,2} = t'_{4,2}) \wedge \left[\bigwedge_{i=1}^N (\hat{t}_i = \hat{t}'_i)\right]$

Algorithm 7.47: Checks the correctness of a NIZKP of a shuffle π generated by Alg. 7.43. The public values are the ElGamal encryptions $\mathbf{e}$ and $\mathbf{e}'$ and the public encryption key pk.

Algorithm: $\mathsf{GetPartialDecryptions}(\mathbf{e}, sk)$

Input: ElGamal encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i)$, $a_i, b_i \in \mathbb{G}_q$
Decryption key $sk \in \mathbb{Z}_q$

for $i = 1, \ldots, N$ **do**
  $b'_i \leftarrow b_i^{sk} \bmod p$
$\mathbf{b}' \leftarrow (b'_1, \ldots, b'_N)$
return $\mathbf{b}'$ // $\mathbf{b}' \in \mathbb{G}_q^N$

Algorithm 7.48: Computes the partial decryptions of a given input list $\mathbf{e}$ of ElGamal encryption using a share sk of the private decryption key.

Algorithm: $\mathsf{GenDecryptionProof}(sk, pk, \mathbf{e}, \mathbf{b}')$
Input: Decryption key $sk \in \mathbb{Z}_q$
Encryption key $pk \in \mathbb{G}_q$
ElGamal encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i)$, $a_i, b_i \in \mathbb{G}_q$
Partial decryptions $\mathbf{b}' = (b'_1, \ldots, b'_N)$, $b'_i \in \mathbb{G}_q$
$\omega \in_R \mathbb{Z}_q$
$t_0 \leftarrow g^\omega \bmod p$
for $i = 1, \ldots, N$ **do**
  $t_i \leftarrow b_i^\omega \bmod p$
$t \leftarrow (t_0, (t_1, \ldots, t_N))$
$\mathbf{b} \leftarrow (b_1, \ldots, b_N)$
$y \leftarrow (pk, \mathbf{b}, \mathbf{b}')$
$c \leftarrow \mathsf{GetNIZKPChallenge}(y, t, \tau)$ // see Alg. 7.4
$s \leftarrow \omega + c \cdot sk \bmod q$
$\pi \leftarrow (t, s)$
return π // $\pi \in (\mathbb{G}_q \times \mathbb{G}_q^N) \times \mathbb{Z}_q$

Algorithm 7.49: Generates a decryption proof relative to ElGamal encryptions $\mathbf{e}$ and partial decryptions $\mathbf{b}'$. This is essentially a NIZKP of knowledge of the private key sk satisfying $b'_i = b_i^{sk}$ for all input encryptions $e_i = (a_i, b_i)$ and $pk = g^{sk}$. For the proof verification, see Alg. 7.51.

Algorithm: $\mathsf{CheckDecryptionProofs}(\boldsymbol{\pi}', \mathbf{pk}, \mathbf{e}, \mathbf{B}')$
Input: Decryption proofs $\boldsymbol{\pi}' = (\pi'_1, \ldots, \pi'_s)$, $\pi_j \in (\mathbb{G}_q \times \mathbb{G}_q^N) \times \mathbb{Z}_q$
Encryption key shares $\mathbf{pk} = (pk_1, \ldots, pk_s)$, $pk_j \in \mathbb{G}_q$
ElGamal encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i \in \mathbb{G}_q^2$
Partial decryptions $\mathbf{B}' = (b_{ij})_{N \times s}$, $b'_{ij} \in \mathbb{G}_q$
for $j = 1, \ldots, s$ **do**
  $\mathbf{b}'_j \leftarrow (b'_{1,j}, \ldots, b'_{N,j})$
  if $\neg\mathsf{CheckDecryptionProof}(\pi'_j, pk_j, \mathbf{e}, \mathbf{b}'_j)$ **then** // see Alg. 7.51
    return *false*
return *true*

Algorithm 7.50: Checks if the decryption proofs generated by s different authorities are correct.

Algorithm: $\mathsf{CheckDecryptionProof}(\pi', pk, \mathbf{e}, \mathbf{b}')$

Input: Decryption proof $\pi' = (t, s)$, $t = (t_0, (t_1, \ldots, t_N))$, $t_i \in \mathbb{G}_q$, $s \in \mathbb{Z}_q$
Encryption key share $pk \in \mathbb{G}_q$
ElGamal encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i)$, $a_i, b_i \in \mathbb{G}_q$
Partial decryptions $\mathbf{b}' = (b'_1, \ldots, b'_N)$, $b'_i \in \mathbb{G}_q$

$\mathbf{b} \leftarrow (b_1, \ldots, b_N)$
$y \leftarrow (pk, \mathbf{b}, \mathbf{b}')$
$c \leftarrow \mathsf{GetNIZKPChallenge}(y, t, \tau)$ // see Alg. 7.4
$t'_0 \leftarrow pk^{-c} \cdot g^s \bmod p$
for $i = 1, \ldots, N$ **do**
  $t'_i \leftarrow (b'_i)^{-c} \cdot b_i^s \bmod p$
return $(t_0 = t'_0) \wedge \left[\bigwedge_{i=1}^{N} (t_i = t'_i)\right]$

Algorithm 7.51: Checks the correctness of a decryption proof π generated by Alg. 7.49. The public values are the ElGamal encryptions $\mathbf{e}$, the partial decryptions $\mathbf{b}'$, and the share pk of the public encryption key.

Algorithm: $\mathsf{GetDecryptions}(\mathbf{e}, \mathbf{B}')$

Input: ElGamal encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i)$, $a_i, b_i \in \mathbb{G}_q$
Partial decryptions $\mathbf{B}' = (b'_{ii})_{N \times s}$, $b'_{ij} \in \mathbb{G}_q$

for $i = 1, \ldots, N$ **do**
  $b'_i \leftarrow \prod_{j=1}^{s} b'_{ij} \bmod p$
  $m_i \leftarrow \frac{a_i}{b'_i} \bmod p$
$\mathbf{m} \leftarrow (m_1, \ldots, m_N)$
return $\mathbf{m}$ // $\mathbf{m} \in \mathbb{G}_q^N$

Algorithm 7.52: Computes the list of decrypted plaintexts $\mathbf{m} = (m_1, \ldots, m_N)$ by assembling the partial decryptions b'_{ij} obtained from s different authorities.

Algorithm: $\mathsf{GetVotes}(\mathbf{m}, \mathbf{n}, \mathbf{w})$

Input: Encoded selections $\mathbf{m} = (m_1, \ldots, m_N)$, $m_i \in \mathbb{G}_q$
Number of candidates $\mathbf{n} = (n_1, \ldots, n_t)$, $n_j \geqslant 2$
Counting circles $\mathbf{w} = (w_1, \ldots, w_{N_E})$, $w_i \in \mathbb{N}$

$n \leftarrow \sum_{j=1}^{t} n_j$
$w \leftarrow \max_{i=1}^{N_E} w_i$
$\mathbf{p} \leftarrow \mathsf{getPrimes}(n + w)$ // $\mathbf{p} = (p_1, \ldots, p_{n+w})$, see Alg. 7.1
for $i = 1, \ldots, N$ **do**
  for $j = 1, \ldots, n$ **do**
    if $m_i \bmod p_j = 0$ **then**
      $v_{ij} \leftarrow 1$
    else
      $v_{ij} \leftarrow 0$
  for $j = 1, \ldots, w$ **do**
    if $m_i \bmod p_{n+j} = 0$ **then**
      $w_{ij} \leftarrow 1$
    else
      $w_{ij} \leftarrow 0$
$\mathbf{V} \leftarrow (v_{ij})_{N \times n}$, $\mathbf{W} \leftarrow (w_{il})_{N \times w}$
return $(\mathbf{V}, \mathbf{W})$ // $\mathbf{V} \in \mathbb{B}^{Nn}$, $\mathbf{W} \in \mathbb{B}^{Nw}$

Algorithm 7.53: Computes the election result matrix $\mathbf{V} = (v_{ij})_{n \times N}$ and corresponding counting circles $\mathbf{W} = (w_{ij})_{N \times w}$ from the products of encoded selections $\mathbf{m} = (m_1, \ldots, m_N)$ by retrieving the prime factors of each m_j. Each resulting vector $\mathbf{v}_i = (v_{i,1}, \ldots, v_{i,n})$ represents somebody's vote, and each value $v_{ij} = 1$ represents somebody's vote for a specific candidate $j \in \{1, \ldots, n\}$.

7.6. Channel Security

The additional protocol steps to achieve the necessary channel security have already been discussed in Section 6.6. Four algorithms for generating and verifying digital signatures and for encrypting and decrypting some data are required. Recall that corresponding algorithm calls are not explicitly depicted in the protocol illustrations of Section 6.5, but an exhaustive list of all necessary calls is given in Tables 6.4 and 6.5. In Table 7.5, we summarize the contents of these lists.

Nr.	Algorithm	Called by	Protocols
7.54	$\mathsf{GenSignature}(sk, m)$	Election administrator	6.1, 6.9
		Election authority	6.1, 6.2, 6.3, 6.5, 6.6, 6.7, 6.8
7.55	$\mathsf{VerifySignature}(pk, \sigma, m)$	Election administrator	6.9
		Election authority	6.1, 6.3, 6.8
		Printing authority	6.2
		Voting client	6.4, 6.5, 6.6
7.56	$\mathsf{GenCiphertext}_\phi(pk, m)$	Election authority	6.2
7.57	$\mathsf{GetPlaintext}_\phi(sk, c)$	Printing authority	6.2

Table 7.5.: Overview of algorithms used to establish channel security.

In all algorithms listed above, the message space is not further specified. In case of the signature generation and verification algorithms, which implement the Schnorr signature scheme over $\mathbb{G}_{\hat{q}}$ (see Section 5.6), we call $\mathsf{RecHash}_L(t, m)$ as a sub-routine for computing a hash value that depends on the message m, i.e., the message space supported by Alg. 4.9 determines the message space of the signature scheme. If multiple messages $m_1, \ldots, m_n$ need to be signed, we form the tuple $m = (m_1, \ldots, m_n)$ for calling the algorithms with a single message parameter.

Algorithm: $\mathsf{GenSignature}(sk, m)$
Input: Signature key $sk \in \mathbb{Z}_{\hat{q}}$
Message $m \in M$, M unspecified
repeat
  $r \in_R \mathbb{Z}_{\hat{q}}$
  $t \leftarrow \hat{q}^r \bmod \hat{p}$
  $c \leftarrow \mathsf{ToInteger}(\mathsf{RecHash}_L(t, m)) \bmod q$ // see Algs. 4.5 and 4.9
  $s \leftarrow r - c \cdot sk \bmod \hat{q}$
until $c \neq 0$ **and** $s \neq 0$
$\sigma \leftarrow (c, s)$
return σ // $\sigma \in \mathbb{Z}_{\hat{q}}^2$

Algorithm 7.54: Computes a Schnorr signature for given message m and a signature key sk. For the verification of this signature, see Alg. 7.55. By considering tuples $m = (m_1, \ldots, m_r)$, the algorithm can be used to sign multiple messages simultaneously.

Algorithm: $\mathsf{VerifySignature}(pk, \sigma, m)$
Input: Verification key $pk \in \mathbb{G}_{\hat{q}}$
Signature $\sigma = (c, s) \in \mathbb{Z}_{\hat{q}}^2$
Message $m \in M$, M unspecified
$t' \leftarrow \hat{g}^s \cdot pk^c \bmod \hat{p}$
$c' \leftarrow \mathsf{ToInteger}(\mathsf{RecHash}_L(t', m))$ // see Algs. 4.5 and 4.9
return $c = c'$

Algorithm 7.55: Verifies a Schnorr signature $\sigma = (c, s)$ generated by Alg. 7.54 using a given public verification key pk.

In case of the encryption and decryption algorithms, which implement a hybrid encryption scheme based on a key-encapsulation mechanism over $\mathbb{G}_{\hat{q}}$ (see Section 5.7), we assume that an invertible function $\phi : M \rightarrow \mathcal{B}^*$ exists for converting messages $m \in M$ into byte arrays $\phi(m) \in \mathcal{B}^*$ and vice versa. As long as $\phi^{-1}(\phi(m)) = m$ holds for all $m \in M$, any mapping that is efficiently computable in both directions is suitable. The actual choice of ϕ is therefore a technical detail of minor importance, which needs not to be specified in this document. In practice, mathematical objects such as the ones used in this document are often first serialized into a standard string format (XML, JSON, ...), before converting them into byte arrays.

Another assumption in the following two algorithms is the availability of an AES-256 block cipher implementations in combination with the CTR mode of operation (and a suitable padding mechanism).[2] For a 256-bit (32 bytes) key $k \in \mathcal{B}^{32}$, we use $B' \leftarrow \mathsf{AES\text{-}CTR}(k, B)$ to denote the encryption of a byte array $B \in \mathcal{B}^*$ of length L into a byte array $B' \in \mathcal{B}^*$ of length $L' > L$, and $B \leftarrow \mathsf{AES\text{-}CTR}^{-1}(k, B')$ for the corresponding decryption. The actual value L' depends on the padding mechanism, which we do not further specify. Note that that since AES operates on 128-bit (16 bytes) blocks, L' is always a multiple of 16.

Algorithm: $\mathsf{GenCiphertext}_\phi(pk, m)$
Input: Encryption key $pk \in \mathbb{G}_{\hat{q}}$
Message $m \in M$, M unspecified
$r \in_R \mathbb{Z}_{\hat{q}}$
$k \leftarrow \mathsf{RecHash}_{32}(pk^r \bmod \hat{p})$ // see Alg. 4.9
$c_1 \leftarrow \hat{g}^r \bmod \hat{p}$
$c_2 \leftarrow \mathsf{AES\text{-}CTR}(k, \phi(m))$
$c \leftarrow (c_1, c_2)$
return c // $c \in \mathbb{G}_{\hat{q}} \times \mathcal{B}^*$

Algorithm 7.56: Computes a hybrid encryption for a message m and a public encryption key pk. With $\phi : M \rightarrow \mathcal{B}^*$ we denote an invertible mapping from the message space M into the set of byte arrays $\mathcal{B}^*$. Alg. 7.57 is the corresponding decryption algorithm.

[2] Using the largest possible AES key length (256 bits instead of 192 or 128 bits) guarantees maximal compatibility with the security levels of Chapter 8.

Algorithm: $\mathsf{GetPlaintext}_{\phi}(sk, c)$
Input: Decryption key $sk \in \mathbb{Z}_{\hat{q}}$
Ciphertext $c = (c_1, c_2)$, $c_1 \in \mathbb{G}_{\hat{q}}$, $c_2 \in \mathcal{B}^*$
$k \leftarrow \mathsf{RecHash}_{32}(c_1^{sk} \bmod \hat{p})$ // see Alg. 4.9
$m \leftarrow \phi^{-1}(\mathsf{AES\text{-}CTR}^{-1}(k, c_2))$
return m // $m \in M$, M unspecified

Algorithm 7.57: Decrypts a ciphertext $c = (c_1, c_2)$ for a given private decryption key sk. The algorithms uses the inverse mapping $\phi^{-1} : \mathcal{B}^* \to M$ from Alg. 7.56.

Part IV.

System Specification

8. Security Levels and Parameters

In this chapter, we introduce three different security levels $\lambda \in \{1, 2, 3\}$, for which default security parameters are given. An additional security level $\lambda = 0$ with very small parameters is introduced for testing purposes. Selecting the „right“ security level is a trade-off between security, efficiency, and usability. The proposed parameters are consistent with the general constraints listed in Table 6.1 of Section 6.3.1. In Section 8.1, we define general length parameters for the hash algorithms and the mathematical groups and fields. Complete sets of recommended group and field parameters are listed in Section 8.2. We recommend that exactly these values are used in an actual implementation. In Section 8.3, we specify various alphabets and code lengths for the voting, confirmation, finalization, and verification codes.

8.1. Length Parameters

For each security level, an estimate of the achieved security strengths σ (privacy) and τ (integrity) is shown in Table 8.1. We measure security strength in the number of bits of a space, for which an exhaustive search requires at least as many basic operations as breaking the security of the system, for example by solving related mathematical problems such as DL or DDH. Except for $\lambda = 0$, the values and corresponding bit lengths given in Table 8.1 are in accordance with current NIST recommendations [10, Table 2]. Today, $\lambda = 1$ (80 bits security) is no longer considered to be sufficiently secure (DL computations for a trapdoored 1024-bit prime modulo have been reported recently [23]). Therefore, we recommend at least $\lambda = 2$ (112 bits security), which is considered to be strong enough until at least 2030. Note that a mix of security levels can be chosen for privacy and integrity, for example $\sigma = 128$ ($\lambda = 3$) for improved privacy in combination with $\tau = 112$ ($\lambda = 2$) for minimal integrity.

Security Level λ	Security Strength σ, τ	Hash Length ℓ (L)	$\mathbb{G}_q \subset \mathbb{Z}_p^*$		$\mathbb{G}_{\hat{q}} \subset \mathbb{Z}_{\hat{p}}^*$		$\mathbb{Z}_{p'}$	L_M	Crypto-period
			$\|p\|$	$\|q\|$	$\|\hat{p}\|$	$\|\hat{q}\|$	$\|p'\|$		
0	4	8 (1)	12	11	12	8	8	2	Testing
1	80	160 (20)	1024	1023	1024	160	160	40	Legacy
2	112	224 (28)	2048	2047	2048	224	224	56	$\leqslant 2030$
3	128	256 (32)	3072	3071	3072	256	256	64	> 2030

Table 8.1.: Length parameters according to current NIST recommendations. The length L_M of the OT messages follows deterministically from $\|p'\|$, see Table 6.1.

Since the minimal hash length that covers all three security levels is 256 bits (32 bytes), we propose using SHA-256 as general hash algorithm. We write $H \leftarrow \mathsf{SHA256}(B)$ for calling this algorithm with an arbitrarily long input byte array $B \in \mathcal{B}^*$ and assigning its return value to $H \in \mathcal{B}^{32}$. For $\lambda = 3$, the length of H is exactly $L = 32$ bytes. For $\lambda < 3$, we truncate the first L bytes from H to obtain the desired hash length, i.e.,

$$\mathsf{Hash}_L(B) = \mathsf{Truncate}(\mathsf{SHA256}(B), L)$$

is our general way of computing hash values for all security levels. We use it in Alg. 4.9 to compute hash values of multiple inputs.

8.2. Recommended Group and Field Parameters

In this section, we specify public parameters for $\mathbb{G}_q \subset \mathbb{Z}_p^*$, $\mathbb{G}_{\hat{q}} \subset \mathbb{Z}_{\hat{p}}^*$, and $\mathbb{Z}_{p'}$ satisfying the bit lengths of the security levels $\lambda \in \{0, 1, 2, 3\}$ of Table 8.1. To obtain parameters that are not susceptible to special-purpose attacks, and to demonstrate that no trapdoors have been put in place, we use the binary representation of Euler's number $e = 2.71828\ldots$ as a reference for selecting them. Table 8.2 shows the first 769 digits of e in hexadecimal notation, from which the necessary amount of bits (up to 3072) are taken from the fractional part. Let $e_s \in \{2^{s-1}, \ldots, 2^s - 1\}$ denote the number obtained from interpreting the s most significant bits of the fractional part of e as a non-negative integer, e.g., $e_4 = \texttt{0xB} = 11$, $e_8 = \texttt{0xB7} = 183$, $e_{10} = \lfloor \texttt{0xB7E}/4 \rfloor = 735$, $e_{12} = \texttt{0xB7E} = 2942$, etc. We use these numbers as starting points for searching suitable primes and safe primes.

```
e = 0x2.B7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190CFEF32
    4E7738926CFBE5F4BF8D8D8C31D763DA06C80ABB1185EB4F7C7B5757F5958490CFD47D
    7C19BB42158D9554F7B46BCED55C4D79FD5F24D6613C31C3839A2DDF8A9A276BCFBFA1
    C877C56284DAB79CD4C2B3293D20E9E5EAF02AC60ACC93ED874422A52ECB238FEEE5AB
    6ADD835FD1A0753D0A8F78E537D2B95BB79D8DCAEC642C1E9F23B829B5C2780BF38737
    DF8BB300D01334A0D0BD8645CBFA73A6160FFE393C48CBBBCA060F0FF8EC6D31BEB5CC
    EED7F2F0BB088017163BC60DF45A0ECB1BCD289B06CBBFEA21AD08E1847F3F7378D56C
    ED94640D6EF0D3D37BE67008E186D1BF275B9B241DEB64749A47DFDFB96632C3EB061B
    6472BBF84C26144E49C2D04C324EF10DE513D3F5114B8B5D374D93CB8879C7D52FFD72
    BA0AAE7277DA7BA1B4AF1488D8E836AF14865E6C37AB6876FE690B571121382AF341AF
    E94F77BCF06C83B8FF5675F0979074AD9A787BC5B9BD4B0C5937D3EDE4C3A79396215E
    DA
```

Table 8.2.: Hexadecimal representation of Euler's number (first 3072 bits of fractional part).[1]

««««< HEAD For each security level, we apply the following general rules. We choose the smallest safe prime $p \in \mathbb{S}$ satisfying $e_s \leqslant p < 2^s$, where $s = \|p\|$ denotes the required bit length. Similarly, for bit lengths $s = \|\hat{p}\|$ and $t = \|\hat{q}\|$, we first choose the smallest prime $\hat{q} \in \mathbb{P}$ satisfying $e_t \leqslant \hat{q} < 2^t$ and then the smallest co-factor $\hat{k} \geqslant 2$ satisfying $\hat{p} = \hat{k}\hat{q} + 1 \in \mathbb{P}$ and $e_s \leqslant \hat{p} < 2^s$. Finally, we choose the largest possible prime $p' \in \mathbb{P}$ satisfying $p' < 2^s$ for $s = \|p'\|$.[2] For every group $\mathbb{G}_q$, we use $g = 2^2 = 4$ and $h = 3^2 = 9$ as default generators

[1]Value taken from http://www.numberworld.org/constants.html.

[2]With regard to the fields $\mathbb{Z}_{p'}$, for which computational intractability assumptions are imposed, we are free to choose any prime of the given bit lengths. We choose the largest prime for reasons explained in the caption of Alg. 7.27.

(additional independent generators can be computed with Alg. 7.3). For the groups $\mathbb{G}_{\hat{q}}$, we use $\hat{g} = 2^{\hat{k}} \bmod \hat{p}$ as default generator. ======= For each security level, we apply the following general rules. We choose the smallest safe prime $p \in \mathbb{S}$ satisfying $e_s \leqslant p < 2^s$, where $s = \|p\|$ denotes the required bit length. Similarly, for bit lengths $s = \|\hat{p}\|$ and $t = \|\hat{q}\|$, we first choose the smallest prime $\hat{q} \in \mathbb{P}$ satisfying $e_t \leqslant \hat{q} < 2^t$ and then the smallest co-factor $\hat{k} \geqslant 2$ satisfying $\hat{p} = \hat{k}\hat{q} + 1 \in \mathbb{P}$ and $e_s \leqslant \hat{p} < 2^s$. Finally, we choose the largest possible prime $p' \in \mathbb{P}$ satisfying $p' < 2^s$ for $s = \|p'\|$.[3] For every group $\mathbb{G}_q$, we use $g = 2^2 = 4$ and $h = 3^2 = 9$ as default generators (additional independent generators can be computed with Alg. 7.3). For the groups $\mathbb{G}_{\hat{q}}$, we use $\hat{g} = 2^{\hat{k}} \bmod \hat{p}$ as default generators. »»»> origin/master

The following four subsections contain tables with values p, q, k, g, h, $\hat{p}$, $\hat{q}$, $\hat{k}$, $\hat{q}$, and p' for the four security levels. We also give lists $\mathbf{p} = (p_1, \ldots, p_{60})$ of the first 60 primes in $\mathbb{G}_q$, which are required to encode the selected candidates $\mathbf{s}$ as a single element $\Gamma(\mathbf{s}) \in \mathbb{G}_q$ (see Sections 5.3 and 6.5 for more details).

8.2.1. Level 0 (Testing Only)

p = `0xB93` = 2963	$\hat{p}$ = `0xEED` = 3821	p' = `0xFB` = 251
q = `0x5C9` = 1481	$\hat{q}$ = `0xBF` = 191	
$k = 2$	$\hat{k}$ = `0x14` = 20	
$g = 4$	$\hat{g}$ = `0x656` = 1622	
$h = 9$		

Table 8.3.: Groups $\mathbb{G}_q \subset \mathbb{Z}_p^*$ and $\mathbb{G}_{\hat{q}} \subset \mathbb{Z}_{\hat{p}}^*$ with default generators g, h, and $\hat{g}$, respectively, and field $\mathbb{Z}_{p'}$ for security level $\lambda = 0$ (used for testing only).

$p_1 = 3$	$p_{11} = 97$	$p_{21} = 233$	$p_{31} = 307$	$p_{41} = 409$	$p_{51} = 523$
$p_2 = 13$	$p_{12} = 107$	$p_{22} = 239$	$p_{32} = 311$	$p_{42} = 419$	$p_{52} = 547$
$p_3 = 19$	$p_{13} = 109$	$p_{23} = 251$	$p_{33} = 317$	$p_{43} = 421$	$p_{53} = 557$
$p_4 = 23$	$p_{14} = 113$	$p_{24} = 257$	$p_{34} = 331$	$p_{44} = 431$	$p_{54} = 563$
$p_5 = 29$	$p_{15} = 149$	$p_{25} = 269$	$p_{35} = 347$	$p_{45} = 433$	$p_{55} = 571$
$p_6 = 37$	$p_{16} = 163$	$p_{26} = 271$	$p_{36} = 349$	$p_{46} = 439$	$p_{56} = 593$
$p_7 = 43$	$p_{17} = 173$	$p_{27} = 277$	$p_{37} = 367$	$p_{47} = 443$	$p_{57} = 599$
$p_8 = 59$	$p_{18} = 179$	$p_{28} = 281$	$p_{38} = 373$	$p_{48} = 449$	$p_{58} = 607$
$p_9 = 71$	$p_{19} = 181$	$p_{29} = 283$	$p_{39} = 383$	$p_{49} = 499$	$p_{59} = 619$
$p_{10} = 83$	$p_{20} = 229$	$p_{30} = 293$	$p_{40} = 401$	$p_{50} = 509$	$p_{60} = 641$

Table 8.4.: The first 60 prime numbers in $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for p and q as defined in Table 8.3.

[2] Taken from http://www.numberworld.org/constants.html.

[3] With regard to the fields $\mathbb{Z}_{p'}$, for which no computational intractability assumptions are imposed, we are free to choose any prime of the given bit lengths. We choose the largest prime for reasons explained in the caption of Alg. 7.27.

8.2.2. Level 1

$p =$ 0xB7E151628AED2A6ABF7158809CF4F3 C762E7160F38B4DA56A784D9045190CF EF324E7738926CFBE5F4BF8D8D8C31D7 63DA06C80ABB1185EB4F7C7B5757F595 8490CFD47D7C19BB42158D9554F7B46B CED55C4D79FD5F24D6613C31C3839A2D DF8A9A276BCFBFA1C877C56284DAB79C D4C2B3293D20E9E5EAF02AC60ACC9425 93	$\hat{p} =$ 0xB7E151628AED2A6ABF7158809CF4F3 C762E7160F38B4DA56A784D9045190CF EF324E7738926CFBE5F4BF8D8D8C31D7 63DA06C80ABB1185EB4F7C7B5757F595 8490CFD47D7C19BB42158D9554F7B46B CED55C4D79FD5F24D6613C31C3839A2D DF8A9A276BCFBFA1C877C562C77CC8FB A599C5FBDA90A7EC659F50FB5FEA2922 09
$q =$ 0x5BF0A8B1457695355FB8AC404E7A79 E3B1738B079C5A6D2B53C26C8228C867 F799273B9C49367DF2FA5FC6C6C618EB B1ED0364055D88C2F5A7BE3DABABFACA C24867EA3EBE0CDDA10AC6CAAA7BDA35 E76AAE26BCFEAF926B309E18E1C1CD16 EFC54D13B5E7DFD0E43BE2B1426D5BCE 6A6159949E9074F2F578156305664A12 C9	$\hat{q} =$ 0xB7E151628AED2A6ABF7158809CF4F3 C762E7161D
$k = 2$	$\hat{k} =$ 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF FFFFFFFFECD143303438D0AAD939DEE6 0194B8DB990AC80D6ACFBA0AA3C285C4 ADD467AA7303859CF5F2B38A8C54CC9F 95E67E76F5C2313A29D7AC442E7EE08B 437562EFC324E7CA505E33CB314E04A5 4135A4B65F031105BE082EEBA8
$g = 4$	$\hat{g} =$ 0x4ECC560DFEB7F7C6EF0F6B74F3AE8A 01DC08FF2A41F1CADB6BFEB2396942EB 5E46D5A33EAEFD1AE25AE0C812A82815 A04431D991F56FFFD108928AC16DB496 AEED72BCCB83A7259A97093FE90991E7 89F384A478B11FDE984687156832B79C 0313BF3660C28043920B0FEBBA1CFC55 331F3DA1EFA25A732D0A510CFDA84E00 EE
$h = 9$	

Table 8.5.: Groups $\mathbb{G}_q \subset \mathbb{Z}_p^*$ and $\mathbb{G}_{\hat{q}} \subset \mathbb{Z}_{\hat{p}}^*$ for security level $\lambda = 1$ with default generators g, h, and $\hat{g}$, respectively.

$p_1 = 3$	$p_{11} = 59$	$p_{21} = 151$	$p_{31} = 263$	$p_{41} = 353$	$p_{51} = 457$
$p_2 = 5$	$p_{12} = 79$	$p_{22} = 157$	$p_{32} = 269$	$p_{42} = 367$	$p_{52} = 463$
$p_3 = 7$	$p_{13} = 83$	$p_{23} = 179$	$p_{33} = 271$	$p_{43} = 373$	$p_{53} = 467$
$p_4 = 11$	$p_{14} = 89$	$p_{24} = 181$	$p_{34} = 277$	$p_{44} = 379$	$p_{54} = 479$
$p_5 = 13$	$p_{15} = 101$	$p_{25} = 199$	$p_{35} = 281$	$p_{45} = 383$	$p_{55} = 509$
$p_6 = 23$	$p_{16} = 103$	$p_{26} = 227$	$p_{36} = 283$	$p_{46} = 409$	$p_{56} = 523$
$p_7 = 29$	$p_{17} = 109$	$p_{27} = 229$	$p_{37} = 293$	$p_{47} = 419$	$p_{57} = 547$
$p_8 = 41$	$p_{18} = 131$	$p_{28} = 239$	$p_{38} = 317$	$p_{48} = 431$	$p_{58} = 557$
$p_9 = 43$	$p_{19} = 137$	$p_{29} = 241$	$p_{39} = 337$	$p_{49} = 443$	$p_{59} = 563$
$p_{10} = 47$	$p_{20} = 149$	$p_{30} = 251$	$p_{40} = 347$	$p_{50} = 449$	$p_{60} = 569$

Table 8.6.: The first 60 prime numbers in $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for p and q as defined in Table 8.5.

p' = `0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD1`

Table 8.7.: Field $\mathbb{Z}_{p'}$ for security level $\lambda = 1$.

8.2.3. Level 2

p = `0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190CFEF324E`
`7738926CFBE5F4BF8D8D8C31D763DA06C80ABB1185EB4F7C7B5757F5958490CFD47D7C`
`19BB42158D9554F7B46BCED55C4D79FD5F24D6613C31C3839A2DDF8A9A276BCFBFA1C8`
`77C56284DAB79CD4C2B3293D20E9E5EAF02AC60ACC93ED874422A52ECB238FEEE5AB6A`
`DD835FD1A0753D0A8F78E537D2B95BB79D8DCAEC642C1E9F23B829B5C2780BF38737DF`
`8BB300D01334A0D0BD8645CBFA73A6160FFE393C48CBBBCA060F0FF8EC6D31BEB5CCEE`
`D7F2F0BB088017163BC60DF45A0ECB1BCD289B06CBBFEA21AD08E1847F3F7378D56CED`
`94640D6EF0D3D37BE69D0063`

q = `0x5BF0A8B1457695355FB8AC404E7A79E3B1738B079C5A6D2B53C26C8228C867F79927`
`3B9C49367DF2FA5FC6C6C618EBB1ED0364055D88C2F5A7BE3DABABFACAC24867EA3EBE`
`0CDDA10AC6CAAA7BDA35E76AAE26BCFEAF926B309E18E1C1CD16EFC54D13B5E7DFD0E4`
`3BE2B1426D5BCE6A6159949E9074F2F5781563056649F6C3A21152976591C7F772D5B5`
`6EC1AFE8D03A9E8547BC729BE95CADDBCEC6E57632160F4F91DC14DAE13C05F9C39BEF`
`C5D98068099A50685EC322E5FD39D30B07FF1C9E2465DDE5030787FC763698DF5AE677`
`6BF9785D84400B8B1DE306FA2D07658DE6944D8365DFF510D68470C23F9FB9BC6AB676`
`CA3206B77869E9BDF34E8031`

$k = 2$

$g = 4$

$h = 9$

Table 8.8.: Group $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for security level $\lambda = 2$ with default generators g and h.

$p_1 = 3$	$p_{11} = 53$	$p_{21} = 137$	$p_{31} = 233$	$p_{41} = 331$	$p_{51} = 433$
$p_2 = 7$	$p_{12} = 61$	$p_{22} = 139$	$p_{32} = 257$	$p_{42} = 347$	$p_{52} = 449$
$p_3 = 11$	$p_{13} = 71$	$p_{23} = 149$	$p_{33} = 263$	$p_{43} = 349$	$p_{53} = 461$
$p_4 = 17$	$p_{14} = 83$	$p_{24} = 157$	$p_{34} = 271$	$p_{44} = 353$	$p_{54} = 479$
$p_5 = 19$	$p_{15} = 97$	$p_{25} = 167$	$p_{35} = 277$	$p_{45} = 373$	$p_{55} = 487$
$p_6 = 23$	$p_{16} = 101$	$p_{26} = 179$	$p_{36} = 281$	$p_{46} = 389$	$p_{56} = 547$
$p_7 = 29$	$p_{17} = 103$	$p_{27} = 181$	$p_{37} = 283$	$p_{47} = 401$	$p_{57} = 557$
$p_8 = 37$	$p_{18} = 109$	$p_{28} = 193$	$p_{38} = 311$	$p_{48} = 419$	$p_{58} = 569$
$p_9 = 41$	$p_{19} = 127$	$p_{29} = 199$	$p_{39} = 313$	$p_{49} = 421$	$p_{59} = 571$
$p_{10} = 47$	$p_{20} = 131$	$p_{30} = 229$	$p_{40} = 317$	$p_{50} = 431$	$p_{60} = 599$

Table 8.9.: The first 60 prime numbers in $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for p and q as defined in Table 8.8.

$\hat{p}$ = 0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190CFEF324E
7738926CFBE5F4BF8D8D8C31D763DA06C80ABB1185EB4F7C7B5757F5958490CFD47D7C
19BB42158D9554F7B46BCED55C4D79FD5F24D6613C31C3839A2DDF8A9A276BCFBFA1C8
77C56284DAB79CD4C2B3293D20E9E5EAF02AC60ACC93ED874422A52ECB238FEEE5AB6A
DD835FD1A0753D0A8F78E537D2B95BB79D8DCAEC642C1E9F23B829B5C2780BF38737DF
8BB300D01334A0D0BD8645CBFA73A6160FFE393C48CBBBCA060F0FF8EC6D31BEB5CCEE
D7F2F0BB088017163BC60DF45A0ECB1BCD3548E571733F4A8C724DC97F56F0AE89897D
8A6B93C6F87D7494503A5D6D

$\hat{q}$ = 0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D991

$\hat{k}$ = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF3C244D2E2C2FD6
0A6164BC77C063F2EBBC35FD1C04CC0935158380D5FC66ECBF2D0EBBF20D83B7128970
667D9A93360EF9D99BE7F831A7C2543BDD5A111009853B48C3AA11A3FDB7F5991F05A0
316733D358632D2C05854286BD2B40A2FCF623CDA13C8029C5959399C45E01350E63D9
4F603C42EE50C5E1F254231BF6BBFB71E6C8A004EEB649A6E11D9E37AE093AB3E39CDC
D2D426CF47C3E202D9A2E4A0FAB9A54465D906A94137F8EA484202E8898A440D8BEDAC
C7C0DEAAB473927C635AC35BCACFCE88DD30AC

$\hat{g}$ = 0x7C41B5D002301514D10155BF22BA33947C96EB398837B9E6AC1A25ABFC3F9D44FB7D
943A3317771A26615814BB06E58B5531F4D81CF23B778F23A2364FFB0C28A7335AE731
761FAB304975C8DB647FCCFC1E64239373F60FAD80FE12D750B3CD753B98D548A325A9
A629B06E63A7FC2860D4EB1B885482B64D7177854104554363DFD70DAFDF529F9AFF07
2F78B7FEAA92D00DC6A7180FF49B60F84979A777919E42484A6A1C014E7F8E8CC18454
6CAE0557124F7F21FB2C16AC6EF4F122BB70966F9FBF03A7807AF8190CDF95DCDF0509
C0FA8302681130E7B60C9E9A65BDF83940F0CCC164989B558B9724D97C524E1A2810E0
BB546F83754A846000A9ADB2

Table 8.10.: Group $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for security level $\lambda = 2$ with default generator $\hat{g}$.

p' = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC1

Table 8.11.: Field $\mathbb{Z}_{p'}$ for security level $\lambda = 2$.

8.2.4. Level 3

$p =$	0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190CFEF324E 7738926CFBE5F4BF8D8D8C31D763DA06C80ABB1185EB4F7C7B5757F5958490CFD47D7C 19BB42158D9554F7B46BCED55C4D79FD5F24D6613C31C3839A2DDF8A9A276BCFBFA1C8 77C56284DAB79CD4C2B3293D20E9E5EAF02AC60ACC93ED874422A52ECB238FEEE5AB6A DD835FD1A0753D0A8F78E537D2B95BB79D8DCAEC642C1E9F23B829B5C2780BF38737DF 8BB300D01334A0D0BD8645CBFA73A6160FFE393C48CBBBCA060F0FF8EC6D31BEB5CCEE D7F2F0BB088017163BC60DF45A0ECB1BCD289B06CBBFEA21AD08E1847F3F7378D56CED 94640D6EF0D3D37BE67008E186D1BF275B9B241DEB64749A47DFDFB96632C3EB061B64 72BBF84C26144E49C2D04C324EF10DE513D3F5114B8B5D374D93CB8879C7D52FFD72BA 0AAE7277DA7BA1B4AF1488D8E836AF14865E6C37AB6876FE690B571121382AF341AFE9 4F77BCF06C83B8FF5675F0979074AD9A787BC5B9BD4B0C5937D3EDE4C3A79396419CD7
$q =$	0x5BF0A8B1457695355FB8AC404E7A79E3B1738B079C5A6D2B53C26C8228C867F79927 3B9C49367DF2FA5FC6C6C618EBB1ED0364055D88C2F5A7BE3DABABFACAC24867EA3EBE 0CDDA10AC6CAAA7BDA35E76AAE26BCFEAF926B309E18E1C1CD16EFC54D13B5E7DFD0E4 3BE2B1426D5BCE6A6159949E9074F2F5781563056649F6C3A21152976591C7F772D5B5 6EC1AFE8D03A9E8547BC729BE95CADDBCEC6E57632160F4F91DC14DAE13C05F9C39BEF C5D98068099A50685EC322E5FD39D30B07FF1C9E2465DDE5030787FC763698DF5AE677 6BF9785D84400B8B1DE306FA2D07658DE6944D8365DFF510D68470C23F9FB9BC6AB676 CA3206B77869E9BDF3380470C368DF93ADCD920EF5B23A4D23EFEFDCB31961F5830DB2 395DFC26130A2724E1682619277886F289E9FA88A5C5AE9BA6C9E5C43CE3EA97FEB95D 0557393BED3DD0DA578A446C741B578A432F361BD5B43B7F3485AB88909C1579A0D7F4 A7BBDE783641DC7FAB3AF84BC83A56CD3C3DE2DCDEA5862C9BE9F6F261D3C9CB20CE6B
$k =$	2
$g =$	4
$h =$	9

Table 8.12.: Group $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for security level $\lambda = 3$ with default generators g and h.

$p_1 = 2$	$p_{11} = 89$	$p_{21} = 167$	$p_{31} = 313$	$p_{41} = 457$	$p_{51} = 577$
$p_2 = 3$	$p_{12} = 101$	$p_{22} = 173$	$p_{32} = 317$	$p_{42} = 461$	$p_{52} = 593$
$p_3 = 7$	$p_{13} = 103$	$p_{23} = 181$	$p_{33} = 331$	$p_{43} = 467$	$p_{53} = 599$
$p_4 = 11$	$p_{14} = 109$	$p_{24} = 199$	$p_{34} = 367$	$p_{44} = 479$	$p_{54} = 607$
$p_5 = 13$	$p_{15} = 113$	$p_{25} = 211$	$p_{35} = 379$	$p_{45} = 491$	$p_{55} = 619$
$p_6 = 31$	$p_{16} = 127$	$p_{26} = 229$	$p_{36} = 383$	$p_{46} = 499$	$p_{56} = 643$
$p_7 = 61$	$p_{17} = 131$	$p_{27} = 233$	$p_{37} = 397$	$p_{47} = 503$	$p_{57} = 647$
$p_8 = 73$	$p_{18} = 139$	$p_{28} = 239$	$p_{38} = 401$	$p_{48} = 547$	$p_{58} = 659$
$p_9 = 79$	$p_{19} = 151$	$p_{29} = 251$	$p_{39} = 409$	$p_{49} = 557$	$p_{59} = 677$
$p_{10} = 83$	$p_{20} = 157$	$p_{30} = 283$	$p_{40} = 449$	$p_{50} = 563$	$p_{60} = 691$

Table 8.13.: The first 60 prime numbers in $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for p and q as defined in Table 8.12.

$\hat{p} =$ 0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190CFEF324E
7738926CFBE5F4BF8D8D8C31D763DA06C80ABB1185EB4F7C7B5757F5958490CFD47D7C
19BB42158D9554F7B46BCED55C4D79FD5F24D6613C31C3839A2DDF8A9A276BCFBFA1C8
77C56284DAB79CD4C2B3293D20E9E5EAF02AC60ACC93ED874422A52ECB238FEEE5AB6A
DD835FD1A0753D0A8F78E537D2B95BB79D8DCAEC642C1E9F23B829B5C2780BF38737DF
8BB300D01334A0D0BD8645CBFA73A6160FFE393C48CBBBCA060F0FF8EC6D31BEB5CCEE
D7F2F0BB088017163BC60DF45A0ECB1BCD289B06CBBFEA21AD08E1847F3F7378D56CED
94640D6EF0D3D37BE67008E186D1BF275B9B241DEB64749A47DFDFB96632C3EB061B64
72BBF84C26144E49C2D04C324EF10DE513D3F5114B8B5D374D93CB8879C7D52FFD72BA
0AAE7277DA7BA1B4AF1488D8E836AF14865E6C37AB6876FE690B571121382AF341AFE9
4F790F02FA1BCE9C73886B4C0ACABDC3DD14E0D8C955577C9764844038771FC25F84BB

$\hat{q} =$ 0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190D05D

$\hat{k} =$ 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF67215E
C15D7BB8A7D7B5CB2294EFCAA4C7B3C6906FC93847CD5FEFF6F1F10C1400310C2150C4
450843B67D7B0184C0A9B71708B657001B502DFAC3E8E29D3102610EB5B1D9AD470F0E
FBC232F5025A3D88C58E70D9D2097C5E4E081BBFEE2373A9B5076970B38F6865D03E16
293DBBBCA1B85E3FC5412F7262643B08A2A4CFA5EA43F5F8C9D9986B88155CEA5EC971
5322344FF714C84F18D0B19772C421923C7E2CD2A6FE1000FBFCB4BBBBACEBAAF74C38
CBC29EE75521F18B03C9816975D948F177476F6EBD8816152A0FECEA7DD6EF0AB7B6A0
99617F82337346BDFC1CA47586EADF125A9DA7C1D960DDECDE399A37D7470FEFBED940
3A4EC70A5841F41F60E3E0D40D70B1A5970EBEC446DF220714E83349462754D5C81F16
FCC5ED708EBC21C36C0F3D494E04C15E3C275C18A562BADDA0293ADE9075FAA254E965
E73402

$\hat{g} =$ 0x47DAD70733EFE399D1AFF4FE387250218BB88FD5F4040C31851AE1DF0985D0019950
A958710C6B935B6B3BB45C278381DC5883CC933C5B7052D3BC8C77D746E3D1FB2B7EF3
630C1014417D2F83BEAD0E1F4DFD986104CDF16C4AEC33BB5906C8149C83E6C5B8837E
12AB32E73A69C4ABEB0B014FFF1FBB3173EAD73A1404DAEDF52F62D605D37879001248
29751320FEDAA1F5B2D90FB846C7EB7815193E5C2460F93A3A5D16FB7A3DBAC9CE31B7
517D2F88D530E61D06B529A43A0806F6A931247C9166C32CC9BAA019823528D3F156B6
0ECE5DA9A6D60148661F59670AD98A1B8EAFEBC4A68D8A5D3F29105FD33D994751A9AD
8E0EB7367D5BFE7A2F082981869FA2F177C472D1988844E4DA58170BB3DDE9DFB2E61D
C06FA5249C3200CD3BBBF24D5C257879CB23D7931ED4AD1F9FA168B38FAA3C6DB89AA9
D89BB6DB3F47BF1BE57856C12AD2FD708A932DC4C91A48E662B37C4076A5D2BE54AC80
0EC1E6A13E1FC8EB61CA52E5D7B7608483E3BC225FBC62456AB46E39DA3CF45AB11A50

Table 8.14.: Group $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for security level $\lambda = 3$ with default generator $\hat{g}$.

$p' =$ 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF43

Table 8.15.: Field $\mathbb{Z}_{p'}$ for security level $\lambda = 3$.

8.3. Alphabets and Code Lengths

For the codes printed on the voting cards and displayed to the voters on their voting device, suitable alphabets need to be fixed. In this section, we specify several alphabets and discuss—based on their properties—their benefits and weaknesses for each type of code. The main discriminating property of the codes is the way of their usage. The voting and confirmation codes need to be entered by the voters, whereas the verification and finalization codes are displayed to the voters for comparison only. Since entering codes by users is an error-prone process, it is desirable that the chance of misspellings is as small as possible. Case-insensitive codes and codes not containing homoglyphs such as `'0'` and `'O'` are therefore preferred. We call an alphabet not containing such homoglyphs *fail-safe*.

In Table 8.16, we list some of the most common alphabets consisting of latin letters and arabic digits. Some of them are case-insensitive and some are fail-safe. The table also shows the entropy (measured in bits) of a single character in each alphabet. The alphabet A_{62}, for example, which consists of all 62 alphanumerical characters (digits 0–9, upper-case letters A–F, lower-case letters a–z), does not provide case-insensitivity or fail-safety. Each character of A_{62} corresponds to $\log 62 = 5.95$ bits entropy. Note that the Base64 alphabet A_{64} requires two non-alphanumerical characters to reach 6 bits entropy.

Another special case is the last alphabet in Table 8.16, which contains $6^5 = 7776$ different English words from the new *Diceware wordlist* of the Electronic Frontier Foundation.[4,5] The advantage of such a large alphabet is its relatively high entropy of almost 13 bits per word. Furthermore, since human users are well-trained in entering words in a natural language, entering lists of such words is less error-prone than entering codes consisting of random characters. In case of using the Diceware wordlist, the length of the codes is measured in number of words rather than number of characters. Note that analogous Diceware wordlists of equal size are available in many different languages.

Name	Alphabet	Case-insensitive	Fail-safe	Bits per character
Decimal	$A_{10} = \{\mathtt{0}, \ldots, \mathtt{9}\}$	•	•	3.32
Hexadecimal	$A_{16} = \{\mathtt{0}, \ldots, \mathtt{9}, \mathtt{A}, \ldots, \mathtt{F}\}$	•	•	4
Latin	$A_{26} = \{\mathtt{A}, \ldots, \mathtt{Z}\}$		•	4.70
Alphanumeric	$A_{32} = \{\mathtt{0}, \ldots, \mathtt{9}, \mathtt{A}, \ldots, \mathtt{Z}\} \setminus \{\mathtt{0}, \mathtt{1}, \mathtt{I}, \mathtt{O}\}$	•	•	5
	$A_{36} = \{\mathtt{0}, \ldots, \mathtt{9}, \mathtt{A}, \ldots, \mathtt{Z}\}$	•		5.17
	$A_{57} = \{\mathtt{0}, \ldots, \mathtt{9}, \mathtt{A}, \ldots, \mathtt{Z}, \mathtt{a}, \ldots, \mathtt{z}\} \setminus \{\mathtt{0}, \mathtt{1}, \mathtt{I}, \mathtt{O}, \mathtt{l}\}$		•	5.83
	$A_{62} = \{\mathtt{0}, \ldots, \mathtt{9}, \mathtt{A}, \ldots, \mathtt{Z}, \mathtt{a}, \ldots, \mathtt{z}\}$			5.95
Base64	$A_{64} = \{\mathtt{A}, \ldots, \mathtt{Z}, \mathtt{a}, \ldots, \mathtt{z}, \mathtt{0}, \ldots, \mathtt{9}, \mathtt{=}, \mathtt{/}\}$			6
Diceware	$A_{7776} = \{\texttt{"abacus"}, \ldots, \texttt{"zoom"}\}$	•	•	12.92

Table 8.16.: Common alphabets with different sizes and characteristics. Case-insensitivity and fail-safety are desirable properties to facilitate flawless user entries.

[4] See http://world.std.com/~reinhold/diceware.html.
[5] See https://www.eff.org/deeplinks/2016/07/new-wordlists-random-passphrases.

In Section 4.2, we have discussed methods for converting integers and byte arrays into strings of a given alphabet $A = \{c_1, \ldots, c_N\}$ of size $N \geqslant 2$. The conversion algorithms depend on the assumption that the characters in A are totally ordered and that a ranking function $rank_A(c_i) = i - 1$ representing this order is available. We propose to derive the ranking function from the characters as listed in Table 8.16. In the case of A_{16}, for example, this means that the ranking function looks as follows:

c_i	`0`	`1`	`2`	`3`	`4`	`5`	`6`	`7`	`8`	`9`	`A`	`B`	`C`	`D`	`E`	`F`
$rank_{A_{16}}(c_i)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

All other ranking functions are defined in exactly this way. In case of A_{32} and A_{57}, the removed homoglyphs are simply skipped in the ranking, i.e., `'2'` becomes the first character in the order. Note that the proposed order for A_{64} is consistent with the official MIME Base64 alphabet (RFC 1421, RFC 2045).

8.3.1. Voting and Confirmation Codes

For the voting and confirmation codes, which are entered by the voters during vote casting, we consider the six alphabets from Table 8.16 satisfying fail-safety. For the security levels $\lambda \in \{0, 1, 2, 3\}$ introduced in the beginning of this chapter, Table 8.17 shows the resulting code lengths for these alphabets. We propose to satisfy the constraints for corresponding upper bounds $\hat{q}_x$ and $\hat{q}_y$ by setting them to $2^{2\tau-1}$, the smallest 2τ-bit integer:

$$\hat{q}_x = \hat{q}_y = \begin{cases} 2^7, & \text{for } \lambda = 0, \\ 2^{159}, & \text{for } \lambda = 1, \\ 2^{223}, & \text{for } \lambda = 2, \\ 2^{255}, & \text{for } \lambda = 3. \end{cases}$$

By looking at the numbers in Table 8.17, we see that the necessary code lengths to achieve the desired security strength are problematical from a usability point of view. The case-insensitive Diceware alphabet A_{7776} with code lengths between 13 and 20 words, which seems to be one of the best choices, is still not very practical. We will continue the discussion of this problem in Section 9.2.

Security Level λ	Security Strength τ	$\lVert\hat{q}_x\rVert, \lVert\hat{q}_y\rVert$	ℓ_X, ℓ_Y					
			A_{10}	A_{16}	A_{26}	A_{32}	A_{57}	A_{7776}
0	4	8	3	2	2	2	2	1
1	80	160	44	40	35	32	27	13
2	112	224	61	56	48	45	38	18
3	128	256	70	64	55	52	43	20

Table 8.17.: Lengths of voting and confirmation codes for different alphabets and security levels.

8.3.2. Verification and Finalization Codes

According to the constraints of Table 6.1 in Section 6.3.1, the length of the verification and finalization codes are determined by the deterrence factor ϵ, the maximal number of candidates $n_{\max}$, and the size of the chosen alphabet. For $n_{\max} = 1678$ and security levels $\lambda \in \{0, 1, 2, 3\}$, Table 8.18 shows the resulting code lengths for different alphabets and different deterrence factors $\epsilon = 1 - 10^{-(\lambda+2)}$. This particular choice for $n_{\max}$ has two reasons. First, it satisfies the use cases described in Section 2.2 with a good margin. Second, it is the highest value for which $L_R = 3$ bytes are sufficient in security level $\lambda = 2$.

In the light of the results of Table 8.18 for the verification codes, we conclude that the alphabet A_{64} (Base64) with verification codes of length $\ell_R = 4$ in most cases seems to be a good compromise between security and usability. Since n verification codes are printed on the voting card and k verification codes are displayed to the voter, they should be as small as possible for usability reasons. On the other hand, since only one finalization code appears on every voting card, it would probably not matter much if they were slightly longer. Any of the proposed alphabets seems therefore appropriate. To make finalization codes look different from verification codes, we propose to use alphabet A_{10}, i.e., to represent finalization codes as 5-digit numbers for $\lambda \in \{1, 2\}$ or as a 8-digit numbers for $\lambda = 3$.

<table>
<tr><th rowspan="2">Security Level λ</th><th rowspan="2">Deterrence Factor ϵ</th><th rowspan="2">L_R</th><th colspan="6">ℓ_R</th><th rowspan="2">L_F</th><th colspan="6">ℓ_F</th></tr>
<tr><th>A_{10}</th><th>A_{16}</th><th>A_{26}</th><th>A_{36}</th><th>A_{62}</th><th>A_{64}</th><th>A_{10}</th><th>A_{16}</th><th>A_{26}</th><th>A_{36}</th><th>A_{62}</th><th>A_{64}</th></tr>
<tr><td>0</td><td>99%</td><td>3</td><td>8</td><td>6</td><td>6</td><td>5</td><td>5</td><td>4</td><td>1</td><td>3</td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td></tr>
<tr><td>1</td><td>99.9%</td><td>3</td><td>8</td><td>6</td><td>6</td><td>5</td><td>5</td><td>4</td><td>2</td><td>5</td><td>4</td><td>4</td><td>4</td><td>3</td><td>3</td></tr>
<tr><td>2</td><td>99.99%</td><td>3</td><td>8</td><td>6</td><td>6</td><td>5</td><td>5</td><td>4</td><td>2</td><td>5</td><td>4</td><td>4</td><td>4</td><td>3</td><td>3</td></tr>
<tr><td>3</td><td>99.999%</td><td>4</td><td>10</td><td>8</td><td>7</td><td>7</td><td>6</td><td>6</td><td>3</td><td>8</td><td>6</td><td>6</td><td>5</td><td>5</td><td>4</td></tr>
</table>

Table 8.18.: Lengths of verification and finalization codes for different alphabets and security levels. For the maximal number of candidates, we use $n_{\max} = 1678$ as default value.

Part V.

Conclusion

9. Conclusion

9.1. Recapitulation of Achievements

The system specification presented in this document provides a precise guideline for implementing the next-generation Internet voting system of the State of Geneva. It is designed to support the election use cases of Switzerland and to fulfill the requirements defined by the Federal Chancellery Ordinance on Electronic Voting (VEleS) to the extent of the full expansion stage. In Art. 2, the ordinance lists three general requirements for authorizing electronic voting. The first is about guaranteeing secure and trustworthy vote casting, the second is about providing an easy-to-use interface to voters, and the third is about documenting the details of all security-relevant technical and organizational procedures of such a system [5]. The content of this document is indented to lay the groundwork for a complete implementation of all three general requirements.

The core of the document is a new cryptographic voting protocol, which provides the following key properties based on state-of-the-art technology from the cryptographic literature:

- Votes are end-to-end encrypted from the voting client to the final tally. We use a verifiable re-encryption mix-net for breaking up the link between voters and their votes before performing the decryption.
- By comparing some codes, voters can verify that their vote has been recorded as intended. If the verification succeeds, they know with sufficiently high probability that their vote has reached the ballot box without any manipulation by malware or other types of attack. We realize this particular form of individual verifiability with an existing oblivious transfer protocol [27].
- Based on the public election data produced during the protocol execution, the correctness of the final election result can be verified by independent parties. We use digital signatures, commitments, and zero-knowledge proofs to ensure that all involved parties strictly comply with the protocol in every single step. In this way, we achieve a complete universal verification chain from the election setup all the way to the final tally.
- Every critical task of the protocol is performed in a distributed way by multiple election authorities, such that no single party involved in the protocol can manipulate the election result or break vote privacy. This way of distributing the trust involves the code generation during the election preparation, the authentication of the voters, the sharing of the encryption key, the mixing of the encrypted votes, and the final decryption.

By providing these properties, we have addressed all major security requirements of the legal ordinance (see Section 1.1). For adjusting the actual security level to current and future needs, all system parameters are derived from three principal security parameters. This way of parameterizing the protocol offers a great flexibility for trading off the desired level of security against the best possible usability. The strict parametrization is also an important prerequisite for formal security proofs.

With the protocol description given in form of precise pseudo-code algorithms, we have reached the highest possible level of details for such a document. To the best of our knowledge, today no other document in the literature on cryptographic voting protocols or in the practice of electronic voting systems offers such a detailed and complete protocol specification. With our effort of writing such a document, we hope to deliver a good example of how electronic voting systems could (or should) be documented. We believe that this is roughly the level of transparency that any electronic voting system should offer in terms of documentation. It enables software developers to link the written code precisely and systematically with corresponding parts of the specification. Such links are extremely useful for code reviewers and auditors of the resulting system.

9.2. Open Problems and Future Work

Some problems have not been directly addressed in this document or have not been solved entirely. We conclude this document by providing a list of such open problems with a short discussion of a possible solution in each case.

- *Web Browser Performance*: Due to the limited performance of interpreted JavaScript code, web browsers are relatively slow computational environments for cryptographic computations. Usually, modular exponentiations with very large numbers are the most expensive operations in cryptographic applications, but JavaScript developers have no built-in access to such a primitive. With the best JavaScript libraries available today, computing a small number of modular exponentiations is possible in a modern web browser, but computing a large number of modular exponentiations may lead to major usability problems. This is the case in our protocol when a large number of candidates must be selected. A possible solution is to outsource such computations to external servers. Many protocols with different properties exist for this purpose [15, 16, 31, 37, 50]. Their main challenge is to guarantee that no secret information is leaked to the servers. Selecting the outsourcing protocol with the best properties for our specific purpose is an open question.
- *Secure Bulletin Board*: Throughout this document, we have assumed the existence of a robust append-only bulletin board, which is available to all protocol participants at all times. However, the implementation of a secure bulletin board is a very difficult problem on its own. The main challenge is to guarantee the consistency of the messages posted to the board without creating a single point of failure. There is a considerable amount of literature on this topic, but so far no consensus about the best approach has been reached [12, 20, 28, 29, 30, 34, 36, 44]. The problem in the context of this document is a little less critical, because copies of all submitted ballots are automatically kept by all election authorities. Lost, manipulated, or added ballots are

therefore detected without any additional measures. Nevertheless, the robustness of the board is still critical for the proper functioning of the system.

- *Voting and Conformation Codes*: According to current recommendations, 112 bits is the minimal security strength for cryptographic applications. In terms of group sizes, key lengths, and output length of hash algorithms, this corresponds to 224 bits. In our protocol, this means that voters need to enter 224 bits of entropy twice, once for the voting code and once for the confirmation code. According to our calculations in Section 8.3.1, this corresponds to strings of length 38 with characters taken from a 57-character alphabet. Asking voters to enter such long strings may cause a huge usability problem. Using a word list such as Diceware may be a partial solution, but 224 bits still correspond to 18 words. Scanning a 2D barcode containing the necessary amount of bits instead of entering them over the keyboard—for example using the voter's smartphone—may be another suitable approach, but probably not if an additional device is required to perform the scanning process.
  Since voting and confirmation codes must only sustain attacks before or during the election period, reducing their lengths to 160 bits (80 bits security) or less could possibly be justified. The general problem is that such attacks can be conducted offline as soon as corresponding public credentials are published by the election authorities. In offline attacks, the workload can be distributed to a large amount of CPUs, which execute the attack in parallel. The given protocol has no way of preventing this. Changing the protocol's authentication mechanism to prevent offline attacks is an open problem. In a solution that only permits online attacks, reducing the number of security bits could be better justified.

- *Secure Printing*: The most critical component in our protocol is the printing authority (see Section 6.2). It is the only party that learns enough information to manipulate the election, for example by submitting ballots in the name of real voters. Printing sensitive information securely is known to be a difficult problem. The technical section of the VEleS ordinance accepts a solution based on organizational and procedural measures. Defining them, putting them in place, and supervising them during the printing process is a problem that needs no be addressed separately.

- *Privacy Attack on Voting Device*: The assumption that no adversary will attack the voter's privacy on the voting device is a very strong one. The problem could be solved by pure code voting [41], but this would have an enormous negative impact on the system's usability. Apparently the most viable solution to this problem is to distribute trusted hardware to voters, but this would obviously have a considerable impact on the overall costs. At the moment, however, we do not see a better solution.

- *Formal Security Proofs*: Definitions of security properties and corresponding formal proofs that these properties are satisfied by the protocol are not included in this document. The plan is to develop them in a second stage of the project by a third-party expert. As long as such proofs are missing, we can not guarantee that no attack has been overlooked. Consequently, the protocol presented in this document should be considered with care until such formal proofs are available.

Nomenclature

α	Ballot
a	Left-hand side of encrypted vote
$\mathbf{a}$	OT query
A_F	Alphabet for finalization codes
A_R	Alphabet for verification codes
A_X	Alphabet for voting codes
A_Y	Alphabet for confirmation codes
β_j	Reponse generated by authority j
$\boldsymbol{\beta}_i$	Reponses for voter i
b	Right-hand side of encrypted vote
$\mathbf{b}'_j$	Partial decryptions by authority j
B	Ballot list consisting of tuples (i, α, w) for each valid ballot (i, α)
$\mathbf{B}'$	Partial decryptions
$\mathbb{B}$	Boolean set
γ	Confirmation
$\mathbf{c}$	Vector of candidate descriptions
C	Confirmation list consisting of tuples (i, γ) for each valid confirmation (i, γ)
C_i	Candidate description
δ_j	Finalization generated by authority j
$\boldsymbol{\delta}_i$	Finalizations for voter i
d_i	Voting card data
$\hat{\mathbf{d}}_j$	Public credentials generated by authority j
$\mathbf{d}_j$	Voting card data generated by authority j
$\hat{\mathbf{D}}$	Public credentials
$\mathbf{D}$	Voting card data
ϵ	Deterrence factor
e_{ij}	Eligibility of voter i in election j
$\mathbf{E}$	Eligibility matrix
FC_i	Finalization code of voter i
g	Generator of group $\mathbb{G}_q$
$\hat{g}$	Generator of group $\mathbb{G}_{\hat{q}}$
$\mathbb{G}_q$	Multiplicative subgroup of integers modulo p (of order $q = \frac{p-1}{2}$)
$\mathbb{G}_{\hat{q}}$	Multiplicative subgroup of integers modulo $\hat{p}$ (of order $\hat{q}$)
h	Generator of group $\mathbb{G}_q$

h_i	Generator of group $\mathbb{G}_q$
i	Index over candidates $\{1,\ldots,n\}$, index over voters $\{1,\ldots,N_E\}$, index over submitted ballots $\{1,\ldots,N_B\}$, index over confirmations $\{1,\ldots,N_C\}$, index over encrypted votes $\{1,\ldots,N\}$,
j	Index over authorities $\{1,\ldots,s\}$, index over selections $\{1,\ldots,k\}$, index over elections $\{1,\ldots,t\}$
k_j	Number of selections in election j
k'_{ij}	Number of selections of voter i in each election j
k_F	String length of finalization codes
k_R	String length of verification codes
$\mathbf{k}$	Number of selections in each election
λ	Security level
l	Auxiliary index in iterations
ℓ	Output length of hash funciton (bits)
ℓ_F	Length of finalization codes (bits)
ℓ_R	Length of verification codes (bits)
ℓ_X	String length of voting code
ℓ_Y	String length of confirmation code
L	Output length of hash function (bytes)
L_F	Length of finalization codes (bytes)
L_M	Length of OT messages (bytes)
L_R	Length of verification codes (bytes)
τ	Security strength (integrity)
m	Product of selected primes
$\mathbf{m}$	Products of selected primes
n	Number of candidates
$\mathbf{n}$	Number of candidates in each election
N	Number of valid votes
N_B	Size of ballot list B
N_C	Size of confirmation list C
N_E	Number of eligible voters
$\mathbb{N}$	Natural numbers
$\mathbb{N}^+$	Positive integers
π	Ballot or confirmation NIZKP
π_j	Shuffle proof of authority j
π'_j	Decryption proof of authority j
$\boldsymbol{\pi}$	Shuffle proofs
$\boldsymbol{\pi}'$	Decryption proofs
p	Prime modulus of group $\mathbb{G}_q$
$\hat{p}$	Prime modulus of group $\mathbb{G}_{\hat{q}}$
p_{ij}	Point on polynomials of voter i

p'	Prime modulus of field $\mathbb{Z}_{p'}$
P_i	Voting page of voter i
$\mathbf{P}$	Matrix of points
$\mathbb{P}$	Primes numbers
pk	Public encryption key
pk_j	Share of public encryption key
$\mathbf{pk}$	Shares of public encryption key
q	Order of group $\mathbb{G}_q$
$\hat{q}$	Order of group $\mathbb{G}_{\hat{q}}$
$\hat{q}_x$	Upper bound for secret voting credentials
$\hat{q}_y$	Upper bound for secret confirmation credentials
$\mathbf{q}$	Selected primes
$\mathbf{rc}_i$	Verification codes of voter i
RC_{ij}	Verification code of voter i for candidate j (string)
σ	Security strength (privacy)
s	Number of authorities
s_j	Index of selected candidate
$\mathbf{s}$	Vector of indices of selected candidates
$\mathbb{S}$	Safe primes
S_i	Voting card of voter i
sk_j	Share of private decryption key
t	Number of elections in an election event
U	Unique election identifier
v	Voter index
v_{ij}	Single entry of the election result matrix
$\mathbf{v}$	Vector of voter descriptions
V_i	Voter description (first/last names, address, date of birth, etc.)
$\mathbf{V}$	Election result matrix
w	Number of counting circles
w_i	Counting circle of voter i
w_{ij}	Single entry of the counting circle matrix
$\mathbf{w}$	Vector of counting circles assigned to voters
$\mathbf{W}$	Counting circle matrix of election result
x_i	Secret voting credential of voter i
$\hat{x}_i$	Public voting credential of voter i
X_i	Voting code of voter i
y_i	Secret confirmation credential of voter i
$\hat{y}_i$	Public confirmation credential of voter i
y'_i	Secret vote validity credential of voter i
Y_i	Confirmation code of voter i
z_{ij}	Randomization used in OT response by authority j for voter i

$\mathbb{Z}_{p'}$	Field of integers modulo p'
$\mathbb{Z}_{\hat{p}}^*$	Multiplicative group of integers modulo $\hat{p}$
$\mathbb{Z}_q$	Field of integers modulo q
$\mathbb{Z}_{\hat{q}}$	Field of integers modulo $\hat{q}$

List of Tables

List of Protocols

List of Algorithms

Bibliography

[1] Elliptic curve cryptography. Technical Guideline TR-03111, Bundesamt für Sicherheit in der Informationstechnik, 2012.

[2] Digital signature standard (DSS). FIPS PUB 186-4, National Institute of Standards and Technology (NIST), 2013.

[3] *Ergänzende Dokumentation zum dritten Bericht des Bundesrates zu Vote électronique.* Die Schweizerische Bundeskanzlei (BK), 2013.

[4] *Technische und administrative Anforderungen an die elektronischen Stimmabgabe.* Die Schweizerische Bundeskanzlei (BK), 2013.

[5] *Verordnung der Bundeskanzlei über die elektronische Stimmabgabe (VEleS).* Die Schweizerische Bundeskanzlei (BK), 2013.

[6] *Verordnung über die politischen Rechte.* SR 161.11. Der Schweizerische Bundesrat, 2013.

[7] Information technology — security techniques – digital signatures with appendix – part 3: Discrete logarithm based mechanisms. ISO/IEC 14888-3:2016, International Organization for Standardization, 2016.

[8] A. Ansper, S. Heiberg, H. Lipmaa, T. A. Øverland, and F. van Laenen. Security and trust for the Norwegian e-voting pilot project E-Valg 2011. In A. Jøsang, T. Maseng, and S. J. Knapskog, editors, *NordSec'09, 14th Nordic Conference on Secure IT Systems*, LNCS 5838, pages 207–222, Oslo, Norway, 2009.

[9] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2):281–343, 2010.

[10] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Recommendation for key management. NIST Special Publication 800-57, Part 1, Rev. 3, NIST, 2012.

[11] J. Benaloh. *Verifiable Secret-Ballot Elections.* PhD thesis, Yale University, New Haven, USA, 1987.

[12] J. Beuchat. Append-only web bulletin board. Master's thesis, Bern University of Applied Sciences, Biel, Switzerland, 2012.

[13] A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Y. Desmedt, editor, *PKC'03, 6th International Workshop on Theory and Practice in Public Key Cryptography*, LNCS 2567, pages 31–46, Miami, USA, 2003.

[14] D. Chaum and T. P. Pedersen. Wallet databases with observers. In E. F. Brickell, editor, *CRYPTO'92, 12th Annual International Cryptology Conference on Advances in Cryptology*, LNCS 740, pages 89–105, Santa Barbara, USA, 1992.

[15] X. Chen, J. Li, J. Ma, Q. Tang, and W. Lou. New algorithms for secure outsourcing of modular exponentiations. *IEEE Transactions on Parallel and Distributed Systems*, 25(9):2386–2396, 2014.

[16] C. Chevalier, F. Laguillaumie, and D. Vergnaud. Privately outsourcing exponentiation to a single server: Cryptanalysis and optimal constructions. In I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, editors, *ESORICS'16, 21st European Conference on Research in Computer Security*, LNCS 9878, pages 261–278, Heraklion, Greece, 2016.

[17] C. K. Chu and W. G. Tzeng. Efficient k-out-of-n oblivious transfer schemes with adaptive and non-adaptive queries. In S. Vaudenay, editor, *PKC'05, 8th International Workshop on Theory and Practice in Public Key Cryptography*, LNCS 3386, pages 172–183, Les Diablerets, Switzerland, 2005.

[18] C. K. Chu and W. G. Tzeng. Efficient k-out-of-n oblivious transfer schemes. *Journal of Universal Computer Science*, 14(3):397–415, 2008.

[19] J.-S. Coron, J. Patarin, and Y. Seurin. The random oracle model and the ideal cipher model are equivalent. In D. Wagner, editor, *CRYPTO'08, 28th Annual International Cryptology Conference*, LNCS 5157, pages 1–20, Santa Barbara, USA, 2008.

[20] C. Culnane and S. Schneider. A peered bulletin board for robust use in verifiable voting systems. *Computing Research Repository*, arXiv:1401.4151, 2014.

[21] 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.

[22] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *CRYPTO'86, 6th Annual International Cryptology Conference on Advances in Cryptology*, LNCS 263, pages 186–194, Santa Barbara, USA, 1986.

[23] J. Fried, P. Gaudry, N. Heninger, and E. Thomé. A kilobit hidden SNFS discrete logarithm computation. *IACR Cryptology ePrint Archive*, 2016/961, 2016.

[24] I. S. Gebhardt Stenerud and C. Bull. When reality comes knocking – Norwegian experiences with verifiable electronic voting. In M. J. Kripp, M. Volkamer, and R. Grimm, editors, *EVOTE'12, 5th International Workshop on Electronic Voting*, number P-205 in Lecture Notes in Informatics, pages 21–33, Bregenz, Austria, 2012.

[25] K. Gjøsteen. The Norwegian Internet voting protocol. In A. Kiayias and H. Lipmaa, editors, *VoteID'11, 3rd International Conference on E-Voting and Identity*, LNCS 7187, pages 1–18, Tallinn, Estonia, 2011.

[26] R. Haenni, P. Locher R. E. Koenig, and E. Dubuis. Pseudo-code algorithms for verifiable re-encryption mix-nets. In *FC'17, 21st International Conference on Financial Cryptography*, Silema, Malta, 2017.

[27] R. Haenni, R. E. Koenig, and E. Dubuis. Cast-as-intended verification in electronic elections based on oblivious transfer. In J. Barrat Robert Krimmer, Melanie Volkamer, J. Benaloh, N. Goodman, P. Ryan, O. Spycher, V. Teague, and G. Wenda, editors, *E-Vote-ID'16, 1st International Joint Conference on Electronic Voting*, LNCS 10141, pages 277–296, Bregenz, Austria, 2016.

[28] S. Hauser and R. Haenni. A generic interface for the public bulletin board used in UniVote. In P. Parycek and N. Edelmann, editors, *CeDEM'16, 6th International Conference for E-Democracy and Open Government*, pages 49–56, Krems, Austria, 2016.

[29] S. Hauser and R. Haenni. Implementing broadcast channels with memory for electronic voting systems. *JeDEM – eJournal of eDemocracy and Open Government*, 8(3):61–79, 2016.

[30] J. Heather and D. Lundin. The append-only web bulletin board. In P. Degano, J. Guttman, and F. Martinelli, editors, *FAST'08, 5th International Workshop on Formal Aspects in Security and Trust*, LNCS 5491, pages 242–256, Malaga, Spain, 2008.

[31] S. Hohenberger and A. Lysyanskaya. How to securely outsource cryptographic computations. In J. Kilian, editor, *TCC'05, 2nd Theory of Cryptography Conference*, LNCS 3378, pages 264–282, Cambridge, USA, 2005.

[32] J. Katz and Y. Lindell. *Introduction to Modern Cryptography.* CRC Press, 2nd edition, 2015.

[33] Donald E. Knuth. *The Art of Computer Programming*, volume 2, Seminumerical Algorithms. Addison Wesley, 3rd edition, 1997.

[34] R. Krummenacher. Implementation of a web bulletin board for e-voting applications. Project report, Hochschule für Technik Rapperswil (HSR), Switzerland, 2010.

[35] H. Lipmaa. Verifiable homomorphic oblivious transfer and private equality test. In C. S. Laih, editor, *ASIACRYPT'03, 9th International Conference on the Theory and Application of Cryptology and Information Security*, LNCS 2894, pages 416–433, Taipei, Taiwan, 2003.

[36] D. Lundin and J. Heather. The robust append-only web bulletin board. Technical report, University of Surrey, Guildford, U.K., 2008.

[37] P. Mainini. Efficient and secure outsourcing of modular exponentiation. Bachelor thesis, Bern University of Applied Sciences, Biel, Switzerland, 2017.

[38] U. Maurer. Unifying zero-knowledge proofs of knowledge. In B. Preneel, editor, *AFRICACRYPT'09, 2nd International Conference on Cryptology in Africa*, LNCS 5580, pages 272–286, Gammarth, Tunisia, 2009.

[39] U. Maurer and C. Casanova. Bericht des Bundesrates zu Vote électronique. 3. Bericht, Schweizerischer Bundesrat, 2013.

[40] R. Oppliger. Addressing the secure platform problem for remote internet voting in Geneva. Technical report, Chancellory of the State of Geneva, 2002.

[41] R. Oppliger. How to address the secure platform problem for remote internet voting. In *SIS'02, 5th Conference on "Sicherheit in Informationssystemen"*, pages 153–173, Vienna, Austria, 2002.

[42] R. Oppliger. Traitement du problème de la sécurité des plates-formes pour le vote par internet à Genève. Technical report, ESECURITY Techologies, 2002.

[43] R. Oppliger. E-voting auf unsicheren client-plattformen. *digma – Zeitschrift für Datenrecht und Informationssicherheit*, 8(2):82–85, 2008.

[44] R. A. Peters. A secure bulletin board. Master's thesis, Department of Mathematics and Computing Science, Technische Universiteit Eindhoven, The Netherlands, 2005.

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

[46] O. Spycher, M. Volkamer, and R. E. Koenig. Transparency and technical measures to establish trust in Norwegian Internet voting. In A. Kiayias and H. Lipmaa, editors, *VoteID'11, 3rd International Conference on E-Voting and Identity*, LNCS 7187, pages 19–35, Tallinn, Estonia, 2011.

[47] B. Terelius and D. Wikström. Proofs of restricted shuffles. In D. J. Bernstein and T. Lange, editors, *AFRICACRYPT'10, 3rd International Conference on Cryptology in Africa*, LNCS 6055, pages 100–113, Stellenbosch, South Africa, 2010.

[48] T. Truderung. Cast-as-intended mechanism with return codes based on PETs. In *E-Vote-ID'17, 2nd International Joint Conference on Electronic Voting*, Bregenz, Austria, 2017 (submitted).

[49] D. Wikström. A commitment-consistent proof of a shuffle. In C. Boyd and J. González Nieto, editors, *ACISP'09, 14th Australasian Conference on Information Security and Privacy*, LNCS 5594, pages 407–421, Brisbane, Australia, 2009.

[50] J. Ye, X. Chen, and J. Ma. An improved algorithm for secure outsourcing of modular exponentiations. In L. Barolli, M. Takizawa, F. Xhafa, T. Enokido, and J. Park, editors, *AINA'15, 29th International Conference on Advanced Information Networking and Applications Workshops*, pages 73–76, Gwangju, Korea, 2015.