CHVote Protocol Specification

Version 3.3

Rolf Haenni, Reto E. Koenig, Philipp Locher, Eric Dubuis
{rolf.haenni,reto.koenig,philipp.locher,eric.dubuis}@bfh.ch

August 15th, 2022

Bern University of Applied Sciences
CH-2501 Biel, Switzerland

Revision History

Revision	Date	Description
0.1	14.07.2016	Initial Draft.
0.2	11.10.2016	Draft to present at meeting.
0.3	17.10.2016	Vote casting and confirmation algorithms finished.
0.4	24.10.2016	Update of vote casting and confirmation algorithms.
0.5	18.11.2016	Mixing process finished.
0.6	25.11.2016	String conversion introduced, tallying finished.
0.7	07.12.2016	Section 5 finished.
0.8	17.12.2016	Hashing algorithms and cryptographic parameters added.
0.9	10.01.2017	Section 8 finished.
0.10	06.02.2017	Security parameters finished.
0.11	21.02.2017	Section 6 finished, reorganization of Section 7.
0.11	14.03.2017	Section 7 finished.
0.12	21.03.2017	Section 8 finished.
0.13	30.03.2017	Minor corrections, Section 1 finished.
1.0	12.04.2017	Minor corrections, Section 2 finished.
1.1	19.04.2017	Conclusion added.
1.1.1	24.05.2017	Parameter changes.
1.2	14.07.2017	Major protocol revision, full sender privacy added to OT.
1.2.1	14.09.2017	Various minor corrections.
1.3	28.09.2017	Description and algorithms for channel security added.
1.3.1	29.11.2017	Optimization in Alg. 7.25.
1.3.2	06.12.2017	Adjusted ballot proof generation and verification.
1.4	26.03.2018	Proposals for improved usability in Section 9.2.
1.4.1	12.04.2018	Recapitulation added to 9.2.
1.4.2	29.06.2018	Two minor errors corrected.
2.0	31.08.2018	Recommendations by external reviewers implemented.
2.1	31.01.2019	Support for write-ins added.
2.2	12.04.2019	Reduced number of verification codes, introduction of inspection phase and abstention codes.
2.3	22.05.2019	Improved write-in proofs.
3.0	23.12.2019	Bulletin board delegated to election authorities, computation of finalization code by voting client, confirmation and validity credentials, separated election and voting parameters introduced, various minor corrections.

3.1	01.10.2020	Shuffle proof improved according to FC'20 publication. Chapter 12 on performance optimizations added. Section 11.4 on recovering a lost session added. Minor adjustments for improved matching with OpenCHVote.
3.2	14.12.2020	Administrator participates in generating the shared encryption key pair and in decrypting the votes in a distributed manner. Corresponding changes made to Prots. 7.1, 7.4, 7.8 and 7.9 and related algorithms and messages adjusted. Order of Prots. 7.1 to 7.3 switched to simplify information flow during election preparation. Checking multiple NIZKPs jointly replaced by checking them individually. Corresponding batch verification algorithms removed. Some other simplifications and minor corrections.
3.3	15.08.2022	Parameter generation in Alg. 10.1 improved. Alg. 8.27 simplified. Bug related to big-endian byte order corrected in Alg. 4.11. Names of type conversion algorithms modified in Chapter 4 to reduce function overloading. Algs. 8.25 and 8.35 modified to allow more efficient synchronization, Prots. 7.5 and 7.6 adjusted accordingly. Alg. 4.15 modified to avoid trivial collisions in recursive hashing. First subsection of Section 11.2 removed.

Contents

Special Thanks

Numerous people contributed to the creation of this document in different ways. In particular, we want to thank those who made the effort of looking closely at the technical details of this document and reported minor or major errors and problems. We list them here in alphabetical order:

- David Bernhard (Department of Computer Science, University of Bristol, UK)
- Guillaume Bozon (République et Canton de Genève, Switzerland)
- Timo Bürk (Bern University of Applied Sciences, Switzerland)
- Véronique Cortier (LORIA, Vandœuvre lès Nancy, France)
- Yannick Denzer (Bern University of Applied Sciences, Switzerland)
- Benjamin Fankhauser (Bern University of Applied Sciences, Switzerland)
- Pierrick Gaudry (LORIA, Vandœuvre lès Nancy, France)
- Kevin Häni (Bern University of Applied Sciences, Switzerland)
- Thomas Haines (Norwegian University of Science and Technology, Norway)
- Thomas Hofer (Objectif Sécurité SA, Gland, Switzerland)
- Pascal Junod (Snap Inc., Switzerland)
- Florian Moser (ETH Zürich, Switzerland)
- Alain Peytrignet (Bern University of Applied Sciences, Switzerland)
- Elias Schmidhalter (Bern University of Applied Sciences, Switzerland)
- Marx Stampfli (Bern University of Applied Sciences, Switzerland)
- Tomasz Truderung (Polyas GmbH, Berlin, Germany)
- Mathieu Turuani (LORIA, Vandœuvre lès Nancy, France)
- Christophe Vigouroux (République et Canton de Genève, Switzerland)
- Bogdan Warinschi (Department of Computer Science, University of Bristol, UK)
- François Weissbaum (FUB, Swiss Department of Defense)
- Christian Wenger (Bern University of Applied Sciences, Switzerland)

Part I.

Project Context

1. Introduction

Over many years, the State of Geneva has been 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 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* [52]. The concept of *verifiable elections* has been known in the scientific literature for quite some time [16], 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 [51], but so-called *code voting* approaches and other possible solutions were rejected due to usability concerns and legal problems [49].

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 [9, 29, 30, 54]. 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 e-voting projects with end-to-end verifiability.

As a response to the third report on *Vote électronique* by the Swiss Federal Council and the new requirements of the Swiss Federal Chancellery [8, 47], 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 *CHVote 2.0*, 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 was to lay the foundation for implementing a new system entirely from scratch. The main output of this collaboration is this document, which is publicly available at the *Cryptology ePrint Archive* since Version 1.0 from April 2017 [32]. In the course of the project, updated document versions have been released in regular intervals.

In November 2018, the council of the State of Geneva announced the stop of the CHVote 2.0 project due to financial reasons. It meant that with the release of Version 2.1 of this document in January 2019, the collaboration between the State of Geneva and the Bern University of Applied Sciences came to an end. In June 2019, the State of Geneva released all the public material that have been created during the CHVote 2.0 project, including the source code.[1]

To continue this project independently of the support from the State of Geneva, a new funding from *eGovernment Switzerland* has been acquired by the Bern University of Applied Sciences in August 2019. The main project goal was to implement the protocol core based on the code released by the State of Geneva, but it also included releasing a final stable version of this document. Version 3.1 of this document was the main result of this work. Corresponding source code has been released in October 2020 and is freely available at

https://gitlab.com/openchvote.

This software has recently been updated to reflect all changes made to the latest version of this document. Providing a one-to-one correspondence between future versions of this document and the released code is an important goal of the ongoing project.

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 [7, 8]. 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 [8, 5].

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://chvote2.gitlab.io

- *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 [7, 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 [7, Paragraph 4.2.2]. Other aspects of lower significance in the legal ordinance are the protection against privacy attacks by malware on the voting client or quantum-resistant measures to guarantee long-term vote privacy. 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 Switzerland. The document must 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 must 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 protocol. 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 their own implementation for scientific evaluations or to implement protocol extensions for achieving additional security properties. In any case, the target audience of this document are system designers, software developers, and cryptographic experts.

The core of this document is a complete set of algorithms in pseudo-code, which are executed by the protocol parties during the election process. The presentation of these algorithms is sufficiently detailed for implementing the protocol in a modern programming language. Cryptographic libraries are only required for standard primitives such as hash algorithms, pseudo-random generators, and computations with large integers. For one important subtask of the protocol—the mixing of the encrypted votes—two scientific publications have been published at the *International Conference on Financial Cryptography* in 2017 and 2020 [33, 35]. 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.

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 [31], 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 delegated to a separate project conducted by a group of internationally well-recognized cryptographers and e-voting researchers from the LORIA research center in Nancy (France) and from the University of Bristol (United Kingdom). The report of this sister project has been delivered to the State of Geneva in June 2018 [17]. They also conducted a review of the specification and provided some recommendations for improvements. Their recommendations have been taken into account in Version 2.0 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 8). Looking at these algorithms is not mandatory to understand the protocol and the general concepts of our approach, but for developers, they provide a useful link from the theory towards an actual implementation. The support of so-called write-ins requires some changes to the protocol and to some algorithms. This aspect of the topic is discussed separately (Chapter 9). 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 10). We also discuss some usability problems (Chapter 11) and performance optimizations (Chapter 12). Finally, in Part V, we summarize the main achievements and conclusions of this work and discuss some open problems and future work (Chapter 13).

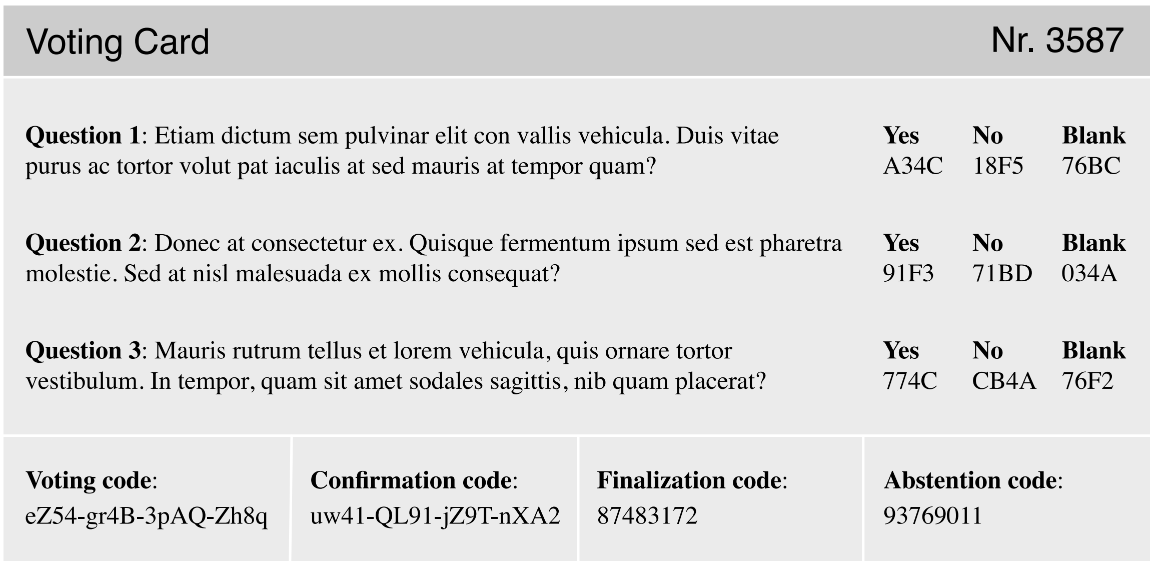
2. Election Context

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 is 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, a single *finalization code*, and a single *abstention code.* These codes are different for every voting card (except for coincidences). 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 protocol's security requirements (see Section 6.3.1).[1]



Voting Card			Nr. 3587
Question 1: Etiam dictum sem pulvinar elit con vallis vehicula. Duis vitae purus ac tortor volut pat iaculis at sed mauris at tempor quam?	**Yes** A34C	**No** 18F5	**Blank** 76BC
Question 2: Donec at consectetur ex. Quisque fermentum ipsum sed est pharetra molestie. Sed at nisl malesuada ex mollis consequat?	**Yes** 91F3	**No** 71BD	**Blank** 034A
Question 3: Mauris rutrum tellus et lorem vehicula, quis ornare tortor vestibulum. In tempor, quam sit amet sodales sagittis, nib quam placerat?	**Yes** 774C	**No** CB4A	**Blank** 76F2

Voting code:	**Confirmation code**:	**Finalization code**:	**Abstention code**:
eZ54-gr4B-3pAQ-Zh8q	uw41-QL91-jZ9T-nXA2	87483172	93769011

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, the two authentication codes are printed as alphanumeric strings, and the finalization and abstention codes are printed as an 8-digit decimal numbers.

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 [7, 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 [4, Appendix 7]. The procedure proposed in this document follows the given guideline to a considerable degree.

In addition to the verification, finalization, and abstention 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, finalization, and abstention 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).

[1] The voting and confirmation codes as shown in Figure 2.1 are possibly not sufficiently long for achieving the desired security strength. This problem will be further discussed in Section 11.2.

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[2] 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.

After the election period, when vote casting is no longer possible, abstaining voters may want to check that no vote has been cast in their name by someone else. Providing the possibility of conducting such a check is an explicit requirement in [7, Paragraph 4.4.3] of the legal ordinance. We propose to offer such a check in form of the above-mentioned abstention code. For this, the list of abstention or finalization codes of all voters will be published at the end of the voting period along with the election results. Abstaining voters can then check whether their abstention code printed on the code sheet is included in that list. Similarly, participating voters can check the inclusion of their finalization code in that list.

2.2. Election Use Cases

The voting protocol presented in this document is designed to support election events consisting of $t \geqslant 1$ simultaneous elections. Every election $j \in \{1, \ldots, t\}$ 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 of the election event. Furthermore, $\mathbf{k} = (k_1, \ldots, k_t)$ and $\mathbf{n} = (n_1, \ldots, n_t)$ denote corresponding vectors of values. Note that the constraints $0 < k_j < n_j$ and $t \geqslant 1$

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

imply $0 < k < n$, i.e., we explicitly exclude elections and election events in which voters are allowed to select zero or all candidates (at least $k_j = n_j = 1$ and therefore $k = n = t$ seems to be a plausible scenario, in which voters can only approve some given candidates, but this scenario is not relevant in the Swiss context). A single selected candidate is denoted by a value $s \in \{1, \ldots, n\}$.

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 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. The total number of voters over all counting circles is denoted by N_E. This number will correspond to the number of eligible voters in a given canton, i.e., up to approximately 1'000'000 voters in the case of the largest canton of Zürich.

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. All such values form a vector $\mathbf{w} = (w_1, \ldots, w_{N_E})$ of length N_E. By including the information about each voter's counting circle 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 11.1.2, 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. Restricted Eligibility

As stated earlier, we also have to take into account that voters may not be eligible in all t elections of an election event. Usually, voters from the same counting circle have the same voting rights, but voters from different counting circles may have very different voting rights. For example, if some of the t elections belong to a municipality, which itself defines a counting circle $j \in \{1, \ldots, w\}$, then citizens not belonging to this municipality are not eligible in those local elections. Even within a counting circle, some citizen may have restricted voting rights in some exceptional cases, for example in municipalities in which immigrants are granted the right to vote on local issues, but not on cantonal or federal issues.

To take this into account, we set $e_{ij} = 1$ if voter $i \in \{1, \ldots, N_E\}$ is eligible in election $j \in \{1, \ldots, t\}$ and $e_{ij} = 0$ otherwise. These values define a Boolean matrix with N_E rows and t columns, the so-called *eligibility matrix* $\mathbf{E} = (e_{ij})_{N_E \times t}$, 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 allowed selections over all t elections of the given election event. The vector of all such values is denoted by $\mathbf{k}' = (k'_1, \ldots, k'_{N_E})$ and their maximum $k'_{\max} = \max_{i=1}^{N_E} k'_i$ is called *maximal ballot size.*

To illustrate this general situation, consider the following example with $N_E = 7$ eligible voters, $w = 2$ counting circles, and $t = 8$ different 1-out-of-2 elections (which implies $k = 8$ and $n = 16$). Furthermore, assume that the first four voters belong to the first and the remaining three voters to the second counting circle. Election 1 and Election 2 are open for both counting circles, whereas Elections 3, 4, and 5 are restricted to the first and Elections 6, 7, and 8 to the second counting circle. There are only three exceptions to this general rule: Voter 1 and Voter 4 have no right to vote in Election 1 and Voter 6 has no voting right in Election 8. The situation in this example leads to the following values:

$$\mathbf{k} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \ \mathbf{n} = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \end{pmatrix}, \ \mathbf{w} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 2 \\ 2 \\ 2 \end{pmatrix}, \ \mathbf{E} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}, \ \mathbf{k}' = \begin{pmatrix} 4 \\ 5 \\ 5 \\ 4 \\ 5 \\ 4 \\ 5 \end{pmatrix}, \ k'_{\max} = 5.$$

In the design of the protocol, we took into account that the members of a given counting circle have almost always identical voting rights and that the voting rights differ from voting circle to voting circle. This helps making the protocol sufficiently efficient even in the case of a large number of counting circles, all with their own local elections. In a nutshell, the optimization is about exploiting the sparseness of the eligibility matrix in such cases. As an approximation, we can use $\kappa = k'_{\max}/k$ as an indicator of the sparseness of $\mathbf{E}$. This value directly determines the running times of several critical algorithms in the protocol. In the example above, it is interesting to observe that already in a simple case with only two counting circles and $k_j = 1$ in all elections, we get $\kappa = 0.675$. As a consequence, all affected algorithms will run approximately 33% faster by implementing corresponding optimizations.

In the case of a voter with restricted eligibility within a counting circle (such as Voter 1, Voter 4 and Voter 6 in the example above), a privacy problem may occur during tallying if the voter's individual pattern in the eligibility matrix remains visible in the list of decrypted votes (see recommendation in [17, Section 10.6]). In Section 6.4.2, we will present a solution that circumvents this problem based on adding the necessary amount of *default votes* to the voter's ballot (and subtracting them during tallying).

2.2.3. Type of Elections

In the elections that we consider, eligible voters must always select exactly k different candidates from a list of n candidates.[3] 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 $(k_{\text{min}}, k_{\text{max}})$-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 $k' = k_{\text{max}}$ and $n' = n + b$, where $b = k_{\text{max}} - k_{\text{min}}$ denotes the number of artificial *blank candidates*, which need to be added to the candidate list. An important special case of this augmented setting arises for $k_{\text{min}} = 0$, in which submitting a completely blank ballot is possible by selecting all $b = k_{\text{max}}$ 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 non-cumulative 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 non-cumulative k'-out-of-n' elections with $k' = k_{\text{max}}$ 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 (which implies $k' = k$). The last two entries in the list, which describe the case of party-list elections, are 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 usually restricted to $c = 2$ votes per candidate. Blank votes are allowed for both the party and the candidate election.

A special case of such a party-list election arises by prohibiting a completely blank candidate vote together with a (non-blank) party vote. This case can be handled 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, and by reducing the number of blank candidates from $b = k$ to $b = k - 1$. If an entirely blank vote is selected, candidate votes are discarded in the final tally. In this way, all possible combinations of selected parties and candidates lead to a valid vote.

An additional complication arises by allowing a distinction between vote abstention and voting for blank candidates. For $k = 1$, the problem is solved by introducing an additional *abstention candidate* to the list of candidates. For $k > 1$, we see the following two simple solutions (which degenerate into each other for $k = 1$):

[3]In this subsection, we ignore the question of counting circles and voters with restricted eligibility.

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

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

- One additional abstention candidate: if one of the selections is the abstention candidate, then the whole ballot is considered as an abstention vote, i.e., all other selections are discarded.
- k additional abstention candidates: if all abstention candidates are selected, then the whole ballot is considered as an abstention vote, otherwise votes for abstention candidates are counted as blank votes.

For keeping n' as small as possible, we generally recommend the first of the two proposals with a single additional abstention candidate. Note that in the case of party-list elections, adding a single *abstention party* to the list of parties is sufficient. If selected, votes for candidates will be discarded in the tally.

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 11.1.2).

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 $X = \{x_1, \dots, x_n\}$. 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 $S = \langle s_0, \dots, s_{n-1} \rangle \in A^*$ 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 $S[i] = s_i$ for selecting the element at index $i \in \{0, \dots, n-1\}$. $S_1 \parallel S_2$ denotes the concatenation of two sequences. For truncating a sequence S of length n to the first $m \leqslant n$ elements, and for skipping the first m elements from S, we write

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

respectively. If S denotes a sequence of pairs $s_i = (k_i, v_i) \in K \times V$, where k_i is a unique *key* in S, then $v \leftarrow \mathsf{Search}(S, k)$ denotes searching S for an entry with a matching key. The result $v \in V \cup \{\perp\}$ is either the value associated with k in S or the special symbol $\perp$ for indicating the absence of k. Similarly, $\mathsf{Contains}(S, k)$ denotes the membership test $(k, \cdot) \in S$ for checking the presence of k in S.

For vectors, we use lower-case Latin letters in bold font, for example $\mathbf{x} = (x_1, \dots, x_n) \in X^n$ for a vector of length $n = |\mathbf{x}|$. If $I = \{i_1, \dots, i_k\}$ is a set of indices $1 \leqslant i_1 < \cdots < i_k \leqslant n$ in ascending order, then we write $\mathbf{x}_I = (x_{i_1}, \dots, x_{i_k})$ for the vector of length k that results from selecting the values x_{i_j} from $\mathbf{x}$ with an index $i_j \in I$. Clearly, the size $k = |\mathbf{x}_I|$ of the resulting vector is smaller than or equal to $n = |\mathbf{x}|$.

For two-dimensional (or higher-dimensional) matrices, we use upper-case Latin letters in bold font, for example

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

for an m-by-n matrix of values $x_{ij} \in X$. We use $\mathbf{X} = (x_{ij})_{m \times n}$ as a shortcut notation and we write $\mathbf{x}_i \leftarrow \mathsf{GetRow}(\mathbf{X}, i)$ for selecting the i-th row vector $\mathbf{x}_i = (x_{i,1}, \dots, x_{i,n})$ and $\mathbf{x}_j \leftarrow \mathsf{GetCol}(\mathbf{X}, j)$ for selecting the j-th column vector $\mathbf{x}_j = (x_{1,j}, \dots, x_{m,j})$ of $\mathbf{X}$. In the rare case of a matrix $\mathbf{X} = (x_{ijk})_{m \times n \times r}$ with three dimensions, we represent it as a vector

$\vec{\mathbf{x}} = (\mathbf{X}_1, \ldots, \mathbf{X}_r)$ of two-dimensional matrices $\mathbf{X}_k = (x_{ijk})_{m \times n}$ of identical dimensions. We use an arrow symbol to accentuate this particular type of vector.

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). For two integers $x, y \in \mathbb{Z}$, the relation $x \mid y$ holds if x divides y, i.e., if $y \bmod x = 0$. 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, and some algorithms may return an error symbol $\perp$ to indicate that an exceptional state has been reached during their execution. 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 $x \in_R X$ or by $x \leftarrow \mathsf{GenRandomInteger}(q)$.

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 $\circ : G \times G \to G$, a (unary) operation $\mathrm{inv} : G \to G$, 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 $q = |G|$ is the *group order* of a finite group, then $q \cdot x = 0$ and $x^q = 1$, which implies $k \cdot x = (k \bmod q) \cdot x$ and $x^k = x^{k \bmod q}$. In other words, scalars or exponents such as k can be restricted to elements of the additive group $\mathbb{Z}_q$, in which additions are computed modulo q (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*.

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., $q|p-1$. A particular case arises when $p = 2q+1 \in \mathbb{S}$ is a safe prime. In this case, the largest possible $\mathbb{G}_q$ 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 $\mathbb{Z}_p^* \setminus \{1, p-1\}$.

Let g be a generator of either $\mathbb{G}_q$ or $\mathbb{Z}_p^*$ and x an arbitrary group element. In both cases, the problem of finding a value k such that $x = g^k$ is believed to be hard. The smallest such 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). The related *computational Diffie-Hellman problem* (CDH) consists in computing g^{ab} from given values g^a and g^b, and the *decisional Diffie-Hellman problem* (DDH) consists in distinguishing two triples (g^a, g^b, g^{ab}) and (g^a, g^b, g^c). Clearly, solving DL also solves CDH and DDH, and solving CDH also solves DDH, but not vice versa. Assuming the hardness of DDH is therefore the strongest assumption. If q is a large prime factor of $p-1$, then it is believed that both DL and CDH are hard in $\mathbb{G}_q$ and $\mathbb{Z}_p^*$. Even DDH is believed to be hard in a $\mathbb{G}_q$, but DDH is known to be easy in $\mathbb{Z}_p^*$.

3.2.2. The Field of Integers Modulo p

With $\mathbb{Z}_q = \{0, \ldots, q-1\}$ we denote the additive group of integers, in which additions are computed modulo q. This group as such is not interesting for cryptographic purposes (no hard problems are known), but for $q = p-1$, 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 $\mathbb{G}_q \subset \mathbb{Z}_p^*$.

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).

To emphasize its field structure, $\mathbb{Z}_p$ is often denoted by $\mathbb{F}_p$. For a given prime-order field $\mathbb{F}_p$, it is possible to define univariate polynomials

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

of degree $d \geqslant 0$ and with coefficients $a_i \in \mathbb{F}_p$ (degree d means $a_d \neq 0$). Clearly, such polynomials are fully determined by the vector $\mathbf{a} = (a_0, \dots, a_d)$ 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 $\mathbf{p} = (p_1, \dots, p_d)$, $p_i \in (x_i, y_i) \in \mathbb{F}_p^2$, 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. Data Types and Basic 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 $B_1 \wedge B_2$, $B_1 \vee B_2$, and $B_1 \oplus B_2$ for the results of applying respective logical operators bit-wise to B_1 and B_2. Another basic byte array operation is needed for generating unique verification codes on every voting card (see Section 6.3.1 and Algs. 8.18 and 8.30). The goal of this operation is similar to digital watermarking, which we use here for making verification codes unique on each voting card. Below we define an algorithm SetWatermark(B, m, n), which adds an integer watermark m, $0 \leqslant m < n$, to the bits of a byte array B.

```
Algorithm: SetWatermark$(B, m, n)$
Input: Byte array $B \in \mathcal{B}^*$, $b = 8 \cdot |B|$
       Watermark $m$, $0 \leqslant m < n$
       Upper bound $n$, $l = \|n-1\| \leqslant b$
for $j = 0, \ldots, l-1$ do
    $i \leftarrow \left\lfloor \frac{j \cdot b}{l} \right\rfloor$
    $B \leftarrow \mathsf{SetBit}(B, i, m \bmod 2)$                // see Alg. 4.2
    $m \leftarrow \lfloor m/2 \rfloor$
return $B$                                                // $B \in \mathcal{B}^*$
```

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

4.1.1. Converting Integers to Byte Arrays

Let $x \in \mathbb{N}$ be a non-negative integer. We use $B \leftarrow \mathsf{IntegerToByteArray}(x, n)$ 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:

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

```
Algorithm: 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
    $z \leftarrow B[j] \wedge (255 - x)$          // $\wedge$ denotes the bitwise AND operator
else
    $z \leftarrow B[j] \vee x$                    // $\vee$ denotes the bitwise OR operator
$B[j] \leftarrow z$
return $B$                                        // $B \in \mathcal{B}^*$
```

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

We use IntegerToByteArray(x) as a short-cut notation for IntegerToByteArray(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 $n \leqslant 4$.

	IntegerToByteArray(x,n)					IntegerToByteArray(x)	
x	$n=0$	$n=1$	$n=2$	$n=3$	$n=4$	n_{min}	
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	–	$\langle FF\rangle$	$\langle 00,FF\rangle$	$\langle 00,00,FF\rangle$	$\langle 00,00,00,FF\rangle$	1	$\langle FF\rangle$
256	–	–	$\langle 01,00\rangle$	$\langle 00,01,00\rangle$	$\langle 00,00,01,00\rangle$	2	$\langle 01,00\rangle$
65,535	–	–	$\langle FF,FF\rangle$	$\langle 00,FF,FF\rangle$	$\langle 00,00,FF,FF\rangle$	2	$\langle FF,FF\rangle$
65,536	–	–	–	$\langle 01,00,00\rangle$	$\langle 00,01,00,00\rangle$	3	$\langle 01,00,00\rangle$
16,777,215	–	–	–	$\langle FF,FF,FF\rangle$	$\langle 00,FF,FF,FF\rangle$	3	$\langle FF,FF,FF\rangle$
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, $B \leftarrow$ IntegerToByteArray(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 4.4).

4.1.2. Converting Byte Arrays to Integers

Since IntegerToByteArray(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 $x \in \mathbb{N}$. 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,

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

```
Algorithm: IntegerToByteArray($x$)
Input: Non-negative integer $x \in \mathbb{N}$
$n_{min} \leftarrow \lceil \frac{\|x\|}{8} \rceil$
$B \leftarrow$ IntegerToByteArray($x, n_{min}$)                    // 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: IntegerToByteArray($x, n$)
Input: Non-negative integer $x \in \mathbb{N}$
       Length of byte array $n \in \mathbb{N}$, $\lceil \frac{\|x\|}{8} \rceil \leqslant n$
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.

in which leading zeros are ignored. With $x \leftarrow$ ByteArrayToInteger(B) we denote a call to an algorithm, which computes this conversion for all $B \in \mathcal{B}^*$ (see Alg. 4.5). It will be used in non-interactive zero-knowledge proofs to generate integer challenges from Fiat-Shamir hash values (see Alg. 8.4 and Alg. 8.5). Note that $x \leftarrow$ ByteArrayToInteger(IntegerToByteArray(x)) holds for all $x \in \mathbb{N}$, but $B \leftarrow$ IntegerToByteArray(ByteArrayToInteger(B)) only holds for byte arrays without any leading zeros (i.e., only when $B[0] \neq 0$). On the other hand, $B \leftarrow$ IntegerToByteArray(ByteArrayToInteger(B), n) holds for all byte arrays $B \in \mathcal{B}^n$ of length n.

```
Algorithm: ByteArrayToInteger($B$)
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.

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'` $\in A^*_{\mathsf{ucs}}$), or `"Hello"`.

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 $B \leftarrow \mathsf{UTF8}(S)$ denote an algorithm that performs this encoding, in which characters use 1, 2, 3, or 4 bytes of space depending on the type of character. For example, $\langle \mathtt{48}, \mathtt{65}, \mathtt{6C}, \mathtt{6C}, \mathtt{6F}\rangle \leftarrow \mathsf{UTF8}(\texttt{"Hello"})$ is a byte array of length 5, because it only consists of Basic Latin characters, whereas $\langle \mathtt{56}, \mathtt{6F}, \mathtt{69}, \mathtt{6C}, \mathtt{C3}, \mathtt{A0}\rangle \leftarrow \mathsf{UTF8}(\texttt{"Voilà"})$ contains 6 bytes due to the Latin-1 Supplement character `'à'` translating into two bytes. Since UTF-8 encoders are widely available in common programming languages, we do not give an explicit algorithm in pseudo-code.

UTF-8 is the only character encoding used in this document for general UCS strings. In Section 4.4, we use it for computing cryptographic hash values of given input strings and in Section 8.5 for encrypting messages with a symmetric key. In both cases, we call the following wrapper algorithm StringToByteArray(S), which we introduce mainly for imposing our naming conventions.

```
Algorithm: StringToByteArray($S$)
Input: String $S \in A^*_{\mathsf{ucs}}$
$B \leftarrow \mathsf{UTF8}(S)$
return $B$                                        // $B \in \mathcal{B}^*$
```

Algorithm 4.6: Computes the UTF-8 encoding of an input string S.

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 $S = \langle s_0, \ldots, s_{n-1}\rangle$ of characters $s_i \in A$ of length n.

4.2.1. Converting Integers to Strings

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

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

We will use this conversion in Alg. 8.18 to print long integers in a more compact form. Note that the following algorithm Alg. 4.7 is almost identical to Alg. 4.4 given in Section 4.1.1 to obtain byte arrays from integers.

```
Algorithm: IntegerToString($x, n, A$)
Input: Integer $x \in \mathbb{N}$
       String length $n \in \mathbb{N}$, $x < N^n$
       Alphabet $A$, $N = |A|$
for $i = 1, \ldots, n$ do
    $c_{n-i} \leftarrow rank_A^{-1}(x \bmod N)$
    $x \leftarrow \lfloor \frac{x}{N} \rfloor$
$S \leftarrow \langle c_0, \ldots, c_{n-1} \rangle$
return $S$                                        // $S \in A^n$
```

Algorithm 4.7: Computes a string representation of length n in big-endian order of a given non-negative integer $x \in \mathbb{N}$ and relative to some alphabet A.

4.2.2. Converting Strings to Integers

In Algs. 8.21 and 8.32, string representations $S \leftarrow$ IntegerToString(x, n, A) of length n must be reconverted into their original integers $x \in \mathbb{N}$. In a similar way as in Section 4.1.2, we obtain the inverse of IntegerToString(x, n, A) by

$$x = \sum_{i=0}^{n-1} rank_A(S[i]) \cdot N^{n-i-1} < N^n,$$

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

```
Algorithm: StringToInteger($S, A$)
Input: String $S \in A^*$
       Alphabet $A$
$N = |A|$
$x \leftarrow 0$
for $i = 0, \ldots, |S| - 1$ do
    $x \leftarrow N \cdot x + rank_A(S[i])$
return $x$                                        // $x \in \mathbb{N}$
```

Algorithm 4.8: 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 $S \in A^m$ of length m, such that m is as small as possible. We will use this conversion in

Algs. 8.19, 8.30, 8.37 and 8.56 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 $|A^m| = N^m$ strings of length m, we derive $m = \lceil \log_N 2^{8n} \rceil = \lceil \frac{8n}{\log_2 N} \rceil$ from the inequality $2^{8n} \leqslant N^m$. To obtain an optimal string representation of B, let $x_B \leftarrow \mathsf{ByteArrayToInteger}(B) < 2^{8n}$ be the representation of B as a non-negative integer. This leads to the following length-optimal mapping from $\mathcal{B}^n$ to A^m.

```
Algorithm: ByteArrayToString($B, A$)
Input: Byte array $B \in \mathcal{B}^n$
       Alphabet $A$
$N = |A|$
$x_B \leftarrow$ ByteArrayToInteger($B$)                // see Alg. 4.5
$m \leftarrow \lceil \log_N 2^{8n} \rceil$
$S \leftarrow$ IntegerToString($x_B, m, A$)             // see Alg. 4.7
return $S$                                              // $S \in A^m$
```

Algorithm 4.9: Computes the shortest string representation of a given byte array B relative to some alphabet A.

To reconstruct UCS strings from a given UTF-8 encoded byte array B, we assume that $S \leftarrow \mathsf{UTF8}^{-1}(B)$ denotes the inverse mapping of $B \leftarrow \mathsf{UTF8}(S)$. Again, since implementations of UTF-8 decoders are widely available, we do not provide an explicit algorithm in pseudo-code. Note that a given byte array is not necessarily a valid UFT-8 encoding, i.e., the allowed inputs of the following wrapper algorithm $\mathsf{ByteArrayToString}(B)$ must be restricted to the subset $\mathcal{B}^*_{\mathsf{ucs}} \subset \mathcal{B}^*$ of valid UFT-8 encodings.

```
Algorithm: ByteArrayToString($B$)
Input: Byte array $\mathcal{B}^*_{\mathsf{ucs}}$
$S \leftarrow$ UTF8$^{-1}$($B$)
return $S$                                              // $S \in A_{\mathsf{ucs}}$
```

Algorithm 4.10: Performs the UTF-8 decoding of the given byte array B. This algorithm is the inverse of Alg. 4.6.

4.3. Generating Random Values

Generating randomness for cryptographic purposes is a very difficult problem. Many attacks against cryptographic applications are based on weak random generation methods or on weaknesses in their implementation. In the context of this document, we assume the existence of a cryptographically secure pseudo-random number generator (PRG), which we can use as a primitive. The purpose of this PRG is the generation of random byte arrays of a given length L. Therefore, we assume that calling

$$R \leftarrow \mathsf{RandomBytes}(L)$$

picks $R \in \mathcal{B}^L$ uniformly at random, i.e., that each possible return value R is selected with equal probability $P(R) = 2^{-8L}$. Furthermore, we assume that the results from calling the PRG multiple times are statistically independent, i.e., calling $R_1 \leftarrow \mathsf{RandomBytes}(L_1)$ and $R_2 \leftarrow \mathsf{RandomBytes}(L_2)$ returns every possible pair $(R_1, R_2) \in \mathcal{B}^{L_2} \times \mathcal{B}^{L_2}$ with equal probability $P(R_1, R_2) = 2^{-8(L_1+L_2)}$. If such a PRG is given as a primitive—for example as part of some cryptographic library—we can use it to derive random values such as $r \in_R \mathbb{Z}_q$, $r \in_R \mathbb{Z}_q \backslash X$, $r \in_R [a, b]$, or $r \in_R \mathbb{G}_q$. This is the purpose of the algorithms given in this section.

```
Algorithm: GenRandomInteger($q$)
Input: Upper bound $q \in \mathbb{N}^+$
$\ell \leftarrow \|q-1\|$, $L \leftarrow \lceil \frac{\ell}{8} \rceil$, $k \leftarrow 8L - \ell$          // $k$ = number of bits to cancel out
repeat
    $R \leftarrow$ RandomBytes($L$)
    for $i = 1, \dots, k$ do
        $R \leftarrow$ SetBit($R, 8-i, 0$)                                  // see Alg. 4.2
    $r \leftarrow$ ByteArrayToInteger($R$)
until $r < q$
return $r$                                                              // $r \in \mathbb{Z}_q$
```

Algorithm 4.11: Returns a uniformly distributed integer $r \in_R \mathbb{Z}_q$ between 0 (inclusive) and the specified upper bound q (exclusive).

```
Algorithm: GenRandomInteger($q, X$)
Input: Upper bound $q \in \mathbb{N}^+$
       Set of excluded values $X \subset \mathbb{Z}_q$
repeat
    $r \leftarrow$ GenRandomInteger($q$)
until $r \notin X$
return $r$                                                              // $r \in \mathbb{Z}_q \backslash X$
```

Algorithm 4.12: Returns a uniformly distributed integer $r \in_R \mathbb{Z}_q \backslash X$ between 0 (inclusive) and the specified upper bound q (exclusive), but such that values from X are never picked.

```
Algorithm: $\mathsf{GenRandomInteger}(a, b)$
Input: Lower bound $a \in \mathbb{Z}$
       Upper bound $a \in \mathbb{Z}$, $a \leqslant b$
$r' \leftarrow \mathsf{GenRandomInteger}(b - a + 1)$
$r \leftarrow a + r'$
return $r$                                                  // $r \in [a, b]$
```

Algorithm 4.13: Returns a uniformly distributed integer $r \in_R [a, b]$ between a (inclusive) and b (inclusive).

```
Algorithm: $\mathsf{GenRandomElement}()$
$r \leftarrow \mathsf{GenRandomInteger}(p, \{0\})$
$r' \leftarrow r^2 \bmod p$
return $r'$                                                 // $r' \in \mathbb{G}_q$
```

Algorithm 4.14: Returns a uniformly distributed element of the subgroup $\mathbb{G}_q \subseteq \mathbb{Z}_p^*$.

4.4. 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-3 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 10. However, it is assumed that only hash algorithms that provide a sufficient level of collision-resistance are selected.

4.4.1. Collision-Resistant Hashing of Single Values

The values that we need to hash in the protocol are often not byte arrays directly. We consider four types of atomic values, which are used frequently across the protocol: byte arrays $B \in \mathcal{B}^*$, integers $x \in \mathbb{Z}$, strings $S \in A_{\mathsf{ucs}}^*$, and the special value $\varnothing$, which is sometimes used for representing the absence of a value. In each case, the given input value $v \in \{\varnothing\} \cup \mathcal{B}^* \cup \mathbb{Z} \cup A_{\mathsf{ucs}}^*$ is first encoded as a byte array $\omega(v) \in \mathcal{B}^*$ of arbitrary length, which is then used as input for the given hash algorithm. To avoid collisions between values of different types, we prefix a type-specific byte to the encoding in each of the four cases. Here is how the encoding $\omega : \Omega \rightarrow \mathcal{B}^*$ for the input domain $\Omega = \{\varnothing\} \cup \mathcal{B}^* \cup \mathbb{Z} \cup A_{\mathsf{ucs}}^*$ is defined (exactly this encoding is included in the recursive hashing method defined below in Alg. 4.15):

$$\omega(v) = \begin{cases} \langle \mathsf{00} \rangle, & \text{if } v = \varnothing, \\ \langle \mathsf{01} \rangle \parallel v, & \text{if } v \in \mathcal{B}^*, \\ \langle \mathsf{02} \rangle \parallel \mathsf{IntegerToByteArray}(2v), & \text{if } v \in \mathbb{Z} \text{ and } v \geqslant 0, \\ \langle \mathsf{02} \rangle \parallel \mathsf{IntegerToByteArray}(-2v-1), & \text{if } v \in \mathbb{Z} \text{ and } v < 0, \\ \langle \mathsf{03} \rangle \parallel \mathsf{StringToByteArray}(v), & \text{if } v \in A^*_{\mathsf{ucs}}. \end{cases}$$

Note that since IntegerToByteArray is only defined for natural numbers, we first apply a bijective mapping $f : \mathbb{Z} \to \mathbb{N}$ from the set of integers to the set of natural numbers, which essentially appends a sign bit to the integer's binary representation (by doubling the integer's absolute value and subtracting 1 if the integer is negative). Also note that we generally apply this encoding to arbitrary subsets of integers. For subsets of non-negative integers such as $\mathbb{Z}_q$, $\mathbb{G}_q$, or $\mathbb{Z}^*_p$, this means that the appended sign bit will always be 0.

4.4.2. Collision-Resistant Hashing of Composed Values

Let $\mathbf{b} = (B_1, \ldots, B_k)$ be a vector of multiple input byte arrays $B_i \in \mathcal{B}^*$ of arbitrary length. To define a collision-resistant extension $\mathsf{Hash}_L(\mathbf{b})$ from a given collision-resistant hash algorithm $\mathsf{Hash}_L(B)$, it is important to ensure that collisions can not occur in a trivial manner. This happens for example when applying the hash algorithm directly to the concatenated byte arrays. A general approach to circumvent this type of problem is to first apply the hash algorithm individually to each B_i and then to the concatenation of the obtained hashes:

$$H \leftarrow \mathsf{Hash}_L(\mathsf{Hash}_L(B_1) \parallel \cdots \parallel \mathsf{Hash}_L(B_k)).$$

This approach can be generalized to the case where the given byte arrays are leafs of an ordered tree, similar to a *Merkle hash tree*. Even more generally, we can have an ordered tree with arbitrary values $v_i \in \Omega$ assigned to each leaf, where $\Omega = \{\varnothing\} \cup \mathcal{B}^* \cup \mathbb{Z} \cup A^*_{\mathsf{ucs}}$ is the domain of atomic values from the previous subsection. In this case, we first compute byte arrays $\omega(v_i) \in \mathcal{B}^*$ for each input value and then apply the hash algorithm at each node of the tree in a bottom-up procedure. We call this procedure *recursive hashing*. For a general input value v, recursive hashing is defined as follows:

$$\mathsf{RecHash}_L(v) = \begin{cases} \mathsf{Hash}_L(\omega(v)), & \text{if } v \in \Omega, \\ \mathsf{Hash}_L(\langle \mathsf{04} \rangle \parallel \mathsf{RecHash}_L(v_1) \parallel \cdots \parallel \mathsf{RecHash}_L(v_k)), & \text{if } v = (v_1, \ldots, v_k). \end{cases}$$

Note that in the case of hashing an internal node in the tree, an additional unique prefix byte is added for achieving second-preimage resistance. Except for the actual choice of the prefix bytes, this approach is equivalent to the tree hashing method in Google's *Certificate Transparency* implementation [43].

In our protocol, recursive hashing frequently occurs for vectors $\mathbf{v} = (v_1, \ldots, v_k)$ and tuples $v = (v_1, \ldots, v_k)$. Since vectors of size k are special cases of k-tuples with identical domains, we do not distinguish them in our recursive hashing method. Therefore, the hash value $\mathsf{Hash}_L(\langle \mathsf{04} \rangle)$ is the same for an empty vector $\mathbf{v} = ()$ and a null tuple $v = ()$, but it is different from the hash values of an empty byte array, an empty string, the special symbol $\varnothing$, or any other atomic value from the domain Ω.

Other frequently hashed objects in the protocol are two-dimensional matrices. By regarding them as a vectors of row vectors (or alternatively as a vectors of column vectors), we can directly apply the above recursive hashing method to matrices. However, to avoid trivial collisions between matrices and vectors, we prepend a different prefix byte (05 instead of 04) for nodes representing a matrix. In this way, the hashing of vectors and matrices is separated in an unambiguous manner. The corresponding extension of $\mathsf{RecHash}_L$ is included in the pseudo-code algorithm given below.

For making the algorithm signature of $\mathsf{RecHash}_L(v)$ more flexible, we permit multiple input values $v_1, \ldots, v_k$, $k \geqslant 1$, of different types. For $k = 1$, where a single input value $v = v_1$ is given, recursive hashing as defined above is applied directly on v. For $k \geqslant 2$, we consider the given inputs as a k-tuple $v = (v_1, \ldots, v_k)$ and then apply recursive hashing on v.

```
Algorithm: $\mathsf{RecHash}_L(v_1, \ldots, v_k)$
Input: Output length $0 \leqslant L \leqslant 32$
       Input values $v_1, \ldots, v_k$, $k \geqslant 1$
if $k = 1$ then
    $v \leftarrow v_1$
else
    $v \leftarrow (v_1, \ldots, v_k)$
if $v = \varnothing$ then
    return $\mathsf{Hash}_L(\langle 00 \rangle)$
if $v \in \mathcal{B}^*$ then
    return $\mathsf{Hash}_L(\langle 01 \rangle \,\|\, v)$
if $v \in \mathbb{Z}$ then
    if $v \geqslant 0$ then
        return $\mathsf{Hash}_L(\langle 02 \rangle \,\|\, \mathsf{IntegerToByteArray}(2v))$        // see Alg. 4.3
    else
        return $\mathsf{Hash}_L(\langle 02 \rangle \,\|\, \mathsf{IntegerToByteArray}(-2v-1))$     // see Alg. 4.3
if $v \in A^*_{\mathsf{ucs}}$ then
    return $\mathsf{Hash}_L(\langle 03 \rangle \,\|\, \mathsf{StringToByteArray}(v))$              // see Alg. 4.6
if $v = (v_1, \ldots, v_n)$ then
    return $\mathsf{Hash}_L(\langle 04 \rangle \,\|\, \mathsf{RecHash}_L(v_1) \,\|\, \cdots \,\|\, \mathsf{RecHash}_L(v_n))$
if $v = (v_{ij})_{n \times m}$ then
    for $i = 1, \ldots, n$ do
        $\mathbf{v}_i \leftarrow \mathsf{GetRow}(v, i)$
    return $\mathsf{Hash}_L(\langle 05 \rangle \,\|\, \mathsf{RecHash}_L(\mathbf{v}_1) \,\|\, \cdots \,\|\, \mathsf{RecHash}_L(\mathbf{v}_n))$
return $\perp$                                                            // type of $v$ not supported
```

Algorithm 4.15: Computes the hash value 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 [25]. 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,

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

is clearly an encryption of m with a fresh randomization $r + \tilde{r}$.

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 or an authorized set $S \in \Gamma$ of parties from a monotone *access structure* $\Gamma \subseteq 2^{\{1,\ldots,s\}}$ is required to decrypt a message, but these generalizations 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:

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

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

Applying this technique in a cryptographic protocol requires some additional care. It is important to ensure that all parties generate their key pairs independently of the public keys published by the others. Otherwise, a dishonest party $d \in \{1, \ldots, s\}$ could publish $pk'_d = pk_d / \prod_{j \neq d} pk_j$ instead of pk_d. This would then lead to $pk = pk_d$, which means that knowing sk_d is sufficient for decrypting messages encrypted with pk. To avoid this attack, all parties publishing a public key must prove knowledge of the corresponding private key. This can be achieved by publish a non-interactive zero-knowledge proof $\pi_j = NIZKP[(sk_j) : pk_j = g^{sk_j}]$ along with pk_j. More details of how to generate and verify such proofs are given in Section 5.4.

5.1.3. Using Multiple Key Pairs

Let $\mathbf{pk} = \{pk_1, \ldots, pk_z\}$ be the ElGamal public keys of z different parties. To encrypt individual messages $\mathbf{m} = (m_1, \ldots, m_z)$, one for each of the z parties, applying the standard ElGamal encryption scheme individually to all z messages requires z different randomizations and therefore $2z$ exponentiations. Using the so-called *multi-recipient ElGamal (MR-ElGamal) encryption scheme*, the total encryption cost can be reduced to $z + 1$ modular exponentiations by re-using the same randomization $r \in \mathbb{Z}_q$ for each message:

$$\mathsf{Enc}_{\mathbf{pk}}(\mathbf{m}, r) = ((m_1 \cdot pk_1^r, \ldots, m_z \cdot pk_z^r), g^r) \in \mathbb{G}_q^z \times \mathbb{G}_q.$$

If such an extended encryption $e = ((a_1, \ldots, a_z), b)$ is broadcast to all z parties, each of them can use its private key sk_i to decrypt (a_i, b) into m_i by calling the standard ElGamal decryption algorithm $\mathsf{Dec}_{sk_i}(a_i, b)$. It has been shown that the resulting multi-recipient encryption scheme offers IND-CPA security under the DDH assumption [14, 15, 42].

Instead of applying the MR-ElGamal encryption scheme in a context with multiple recipients, it is also possible to use it for encrypting multiple messages for a single recipient holding multiple key pairs. The benefit compared to encrypting each message individually using standard ElGamal encryption is the reduced computational costs of $z + 1$ exponentiations only. In such a context, $\mathbf{pk} = \{pk_1, \ldots, pk_z\}$ is the recipient's extended public key and $\mathbf{sk} = \{sk_1, \ldots, sk_z\}$ the recipient's extended private key. The increased key sizes and the increased complexity of the key generation algorithm define a trade-off in favor or against using this technique.

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 [3, 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 vector $\mathbf{m} = (M_1, \ldots, M_n)$ 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* [45]. 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 [22].[1] We instantiate 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 $\Gamma : \{1, \ldots, n\} \rightarrow \mathbb{G}_q$ of the possible selections into $\mathbb{G}_q$, and a collision-resistant hash function $h : \mathbb{B}^* \rightarrow \mathbb{B}^\ell$ 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.

[1]The modified protocol as presented in [23] is slightly more efficient, but fits less into the particular context of this document.

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 $a_j = \Gamma(s_j) \cdot g^{r_j}$		
	$\xrightarrow{\mathbf{a} = (a_1, \dots, a_k)}$	
		pick random $z \in_R \mathbb{Z}_q$ for $j = 1, \dots, k$ – compute $b_j = a_j^z$ for $i = 1, \dots, n$ – compute $k_i = \Gamma(i)^z$ – compute $C_i = M_i \oplus h(k_i)$ compute $d = g^z$
	$\mathbf{b} = (b_1, \dots, b_k)$, $\xleftarrow{\mathbf{c} = (C_1, \dots, C_n), d}$	
for $j = 1, \dots, k$ – 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 $\mathbb{G}_q \subset \mathbb{Z}_p^*$, $\Gamma : \{1, \dots, n\} \to \mathbb{G}_q$ 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.

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 [19].

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 6.4.3).

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 [56, Appendix B].

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 $i = 1, \ldots, n$ – compute $k_i = \Gamma(i)^{z_1}$ – for $j = 1, \ldots, k$ – compute $k_{ij} = k_i \beta_j$ – compute $C_{ij} = M_i \oplus h(k_{ij})$ compute $d = g_1^{z_1} g_2^{z_2}$
	$\xleftarrow{\mathbf{b} = (b_1, \ldots, b_k),\ \mathbf{C} = (C_{ij})_{n \times k}, d}$	
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 5.2: 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 $\mathbb{G}_q \subset \mathbb{Z}_p^*$, $\Gamma : \{1, \ldots, n\} \to \mathbb{G}_q$ 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.

5.3.3. 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 $\mathbf{n} = (n_1, \ldots, n_t)$ and $\mathbf{k} = (k_1, \ldots, k_t)$. 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 $\mathbf{m} = \mathbf{m}_1 \| \cdots \| \mathbf{m}_t = (M_1, \ldots, M_n)$ 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 $j \in \{1, \dots, k\}$ a unique index $l \in \{1, \dots, t\}$ 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. In other words, the selections in $\mathbf{s} = (s_1, \dots, s_k)$ satisfying (5.1) are partially ordered in ascending order.

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^k_n-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 $\sum_{l=1}^{t} k_l n_l$ relevant 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:

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

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, which we denote by $\varnothing$.

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 $j \in \{1, \dots, k\}$ plays the role of the counter. This is identical to applying a single concatenated hash value $h(k_i, 1) \parallel \cdots \parallel h(k_i, k)$ 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 $d = g_1^{z_1} g_2^{z_2}$
	$\xleftarrow{\mathbf{b} = (b_1, \ldots, b_k),\ \mathbf{C} = (C_{ij})_{n \times k}, d}$	
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 $y \in Y$ [46]. The most common construction of a non-interactive preimage proof results from combining the Σ-protocol with the Fiat-

Shamir heuristic [26]. 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 $(c, s) \leftarrow \mathsf{GenProof}_\phi(x, y)$ 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 $t = y^c \cdot \phi(s)$ and $c' = h(y, t)$ and checking $c = c'$. For a given proof $\pi = (c, s)$, 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 $y = \phi(x) \in Y$.

Example 1: 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 [53, 36]. 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 $\pi = (c, s)$ by computing $t = y^c \cdot g^s$ and $c' = h(y, t)$. We will use this proof to demonstrate that voters are in possession of valid voting and confirmation credentials (see Section 6.4.6).

5.4.1. AND-Compositions

Preimage proofs for n different one-way homomorphisms $\phi_i : X_i \rightarrow Y_i$, $1 \leqslant i \leqslant n$, can be reduced to a single preimage proof for a composed function $\phi : X \rightarrow Y$ for $X = X_1 \times \cdots \times X_n$ and $Y = Y_1 \times \cdots \times Y_n$, which is defined by $\mathbf{y} = \phi(\mathbf{x}) = (\phi_1(x_1), \ldots, \phi_n(x_n))$, i.e., $\mathbf{x} = (x_1, \ldots, x_n) \in X$ and $\mathbf{y} = (y_1, \ldots, y_n) \in Y$ are n-tuples. Therefore, $\mathbf{w} = (w_1, \ldots, w_n) \in X$, $\mathbf{t} = (t_1, \ldots, t_n) \in Y$, and $\mathbf{s} = (s_1, \ldots, s_n) \in X$ are also n-tuples, whereas c remains a single value. This way of combining multiple preimage proofs into a single preimage proof is sometimes called *AND-composition.* The following two notations are therefore equivalent and can be used interchangeably:

$$\mathit{NIZKP}[(x_1, \ldots, x_n) : \bigwedge_{i=1}^{n} y_i = \phi_i(x_i)] = \mathit{NIZKP}[(\mathbf{x}) : \mathbf{y} = \phi(\mathbf{x})].$$

An important special case of an AND-composition arises when all $\phi_i : X \rightarrow Y_i$ have a common domain X and when all $y_i = \phi_i(x)$ have the same preimage $x \in X$. The corresponding proof,

$$\mathit{NIZKP}[(x) : \bigwedge_{i=1}^{n} y_i = \phi_i(x)] = \mathit{NIZKP}[(x) : \mathbf{y} = \phi(x)],$$

is called *preimage equality proof.* In the special case of two exponential functions $\phi_1(x) = g^x$ and $\phi_2(x) = h^x$, this demonstrates the equality of two discrete logarithms without revealing them [21].

As shown by the following list of examples, AND-compositions in general and preimage equality proofs in particular appear frequently in many different applications. Each example will be used at some point in this document.

Example 2: Proof of Knowledge of Plaintext. An AND-composition of two preimage proofs 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

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

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 $\mathbb{G}_q$. We will use it to prove that ballots have been encrypted with a fresh randomization (see Section 7.2).

Example 3: Proof of Correct Decryption. 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 *proof of correct decryption*,

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

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

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

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 $c_j = b^{sk_j}$, where sk_j is a share of the private key sk (see Section 5.1.2).

Example 4: Proof of Encrypted Plaintext. For a given ElGamal ciphertext $e = (a, b)$, it is possible to prove that e contains a specific message $m \in \mathbb{G}_q$ without revealing the encryption randomization. The desired *proof of encrypted plaintext*

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

can be transformed into

$$\mathit{NIZKP}[(r) : (a/m, b) = (pk^r, g^r)],$$

which again corresponds to a standard proof of equality of discrete logarithms. We will use it in the context of write-ins for proving that the voter selected a write-in candidate or that the write-in encryption contains an empty string (see Chapter 9).

5.4.2. OR-Compositions

Consider n one-way homomorphisms $\phi_i : X \to Y_i$, $1 \leqslant i \leqslant n$, with a common domain X. A disjunctive proof of knowing at least one of the n preimages,

$$NIZKP[(x) : \bigvee_{i=1}^{n} y_i = \phi_i(x)],$$

of values $\mathbf{y} = (y_1, \ldots, y_n)$ can not be reduced to a single preimage proof like in the case of an AND-composition. However, by simulating transcripts for the cases where no preimage is known, an *OR*-composition can be still be established for the general case.

Suppose that $j \in \{1, \ldots, n\}$ denotes the index of the value $y_j = \phi_j(x)$, for which the preimage $x \in X$ is known, i.e., transcripts (t_i, c_i, s_i) are simulated for all $i \neq j$ and a real transcript (t_j, c_j, s_j) is generated for y_j. The simulated transcripts and the real transcript are connected over a common challenge $c = h(\mathbf{y}, \mathbf{t}) = \sum_{i=1}^{n} c_i$, where $\mathbf{t} = (t_1, \ldots, t_n)$ denotes the vector of all n commitments t_i. In the simulated transcripts, s_i and c_i are selected at random and $t_i = y_i^{c_i} \cdot \phi_i(s_i)$ is computed deterministically. In the real transcript, ω_j is selected at random, whereas $t_j = \phi_j(\omega_j)$, $c_j = c - \sum_{i \neq j} c_i$, and $s_j = \omega_j - c_j \cdot x$ are computed deterministically. The resulting non-interactive proof $\pi = (\mathbf{c}, \mathbf{s})$ of such an OR-composition consists of the vectors $\mathbf{c} = (c_1, \ldots, c_n)$ and $\mathbf{s} = (s_1, \ldots, s_n)$. Verifying π involves computing all $t_i = y^{c_i} \cdot g^{s_i}$, $\mathbf{t} = (t_1, \ldots, t_n)$, and $c' = h(\mathbf{y}, \mathbf{t})$, followed by verifying $c' = \sum_{i=1}^{n} c_i$.

Example 5: Disjunctive Proof of Encrypted Plaintext. Using an OR-composition, it is possible to prove that a given ElGamal ciphertext $e = (a, b)$ contains one of several specific messages $\{m_1, \ldots, m_n\} \subseteq \mathbb{G}_q$ without revealing the encryption randomization. The desired *disjunctive proof of encrypted plaintext*

$$\begin{aligned} &NIZKP[(r) : e = \mathsf{Enc}_{pk}(m, r) \wedge m \in \{m_1, \ldots, m_n\}] \\ &= NIZKP[(r) : \bigvee_{i=1}^{n} e = \mathsf{Enc}_{pk}(m_i, r)] = NIZKP[(r) : \bigvee_{i=1}^{n} (a/m_i, b) = (pk^r, g^r)] \end{aligned}$$

consists of n standard proofs of equality of discrete logarithms.

5.4.3. Combining AND- and OR-Compositions

In the examples of the previous subsections, we observe that $(a/m_i, b) = (pk^r, g^r)$ can be written as a conjunction $(a/m_i = pk^r) \wedge (b = g^r)$, which implies that the OR-composition is actually an OR/AND-composition of $2n$ atomic preimage proofs. By placing $b = g^r$ outside the brackets, the proof can also be transformed into an AND/OR-composition $NIZKP[(r) : (b = g^r) \wedge \bigvee_{i=1}^{n} (a/m_i = pk^r)]$ of $n + 1$ preimage proofs, which leads to a more compact transcript and therefore makes the proof generation and verification more efficient. This example shows that arbitrary combinations of AND- and OR-compositions may be of interest to cover many more applications.

The general principle when combining AND- and OR-compositions into arbitrary monotone Boolean formulae (no negations) remains the same [12]: branches of an AND-composition use the same challenge c, whereas branches in an OR-composition use different challenges, such that all of them except one can be chosen freely. As a consequence, the corresponding AND/OR-tree needs to be traversed twice. In the first traversal round, commitments are computed for branches with known preimages, whereas transcripts are simulated for branches with unknown preimages. All commitments together are then used to compute the top-level commitment, which is used in the second traversal round to compute the remaining challenges and responses. Note that a single tree traversal is sufficient to verify such proofs.

Example 6. CNF-Composition. Consider the following toy example of a combined composition of four individual preimage proofs,

$$\mathit{NIZKP}[(x_1, x_2) : (y_{11} = \phi_{11}(x_1) \vee y_{12} = \phi_{12}(x_1)) \wedge (y_{21} = \phi_{21}(x_2) \vee y_{22} = \phi_{22}(x_2))],$$

in which x_1 is the known preimage of y_{12} and x_2 is the known preimage of y_{22}. Therefore, transcripts for y_{11} and y_{21} need to be simulated in the first round of traversing the tree. For this, values c_{11}, s_{11}, c_{21}, and s_{21} are picked at random, and simulated commitments $t_{11} = y_{11}^{c_{11}} \cdot \phi_{11}(s_{11})$ and $t_{21} = y_{21}^{c_{21}} \cdot \phi_{21}(s_{21})$ are computed deterministically. The real commitments $t_{12} = \phi_{12}(\omega_1)$ and $t_{22} = \phi_{22}(\omega_2)$ are computed based on random values ω_1 and ω_2. This leads to a combined commitment $\mathbf{t} = ((t_{11}, t_{12}), (t_{21}, t_{22}))$, which has the same structure as the public input $\mathbf{y} = ((y_{11}, y_{12}), (y_{21}, y_{22}))$. By computing the top-level challenge $c = h(\mathbf{y}, \mathbf{t})$, the second traversal round can be started. This involves the computation of sub-challenges $c_{12} = c - c_{11}$ and $c_{22} = c - c_{21}$ and corresponding responses $s_{12} = \omega_1 - c_{12}x_1$ and $s_{22} = \omega_2 - c_{22}x_2$. The resulting tuples $\mathbf{c} = ((c_{11}, c_{12}), (c_{21}, c_{22}))$ and $\mathbf{s} = ((s_{11}, s_{12}), (s_{21}, s_{22}))$ form the non-interactive proof $\pi = (\mathbf{c}, \mathbf{s})$. Verifying π requires computing $t_{ij} = y_{ij}^{c_{ij}} \cdot \phi_{ij}(s_{ij})$ for all $i, j \in \{1, 2\}$, computing the top-level challenge $c' = h(\mathbf{y}, \mathbf{t})$, and verifying the consistency of the sub-challenges $c' = c_{11} + c_{12} = c_{21} + c_{22}$.

5.5. Wikström's Shuffle Proof

A *cryptographic shuffle* of a vector $\mathbf{e} = (e_1, \ldots, e_N)$ of ElGamal encryptions $e_i \leftarrow \mathsf{Enc}_{pk}(m_i, r_i)$ is another vector of ElGamal encryptions $\tilde{\mathbf{e}} = (\tilde{e}_1, \ldots, \tilde{e}_N)$, which contains the same plaintexts $m_1, \ldots, m_N$ in permuted order. Such a shuffle can be generated by selecting a random permutation $\psi : \{1, \ldots, N\} \to \{1, \ldots, N\}$ from the set Ψ_N of all such permutations (e.g., using Knuth's shuffle algorithm [39]) and by computing re-encryptions $\tilde{e}_i \leftarrow \mathsf{ReEnc}_{pk}(e_j, \tilde{r}_j)$ for $j = \psi(i)$. We write

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

for an algorithm performing this task, where $\tilde{\mathbf{r}} = (\tilde{r}_1, \ldots, \tilde{r}_N)$ denotes the randomization used to re-encrypt the input ciphertexts. Multiple parties performing a sequence of cryptographic shuffles is called *mix-net*.

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

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

Unfortunately, since $\mathsf{Shuffle}_{pk}$ does not define a group 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 $\tilde{\mathbf{e}}$ 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 $i \in \{1, \ldots, N\}$, and $\prod_{i=1}^N \sum_{j=1}^N b_{ij}x_j = \prod_{i=1}^N x_i$. 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 anything about ψ or $\mathbf{B}_\psi$.

For $\mathbf{c} = (c_1, \ldots, c_N)$, $\mathbf{r} = (r_1, \ldots, r_N)$, 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 $\mathbf{u} = (u_1, \ldots, u_N) \in \mathbb{Z}_q^N$, $\tilde{\mathbf{u}} = (\tilde{u}_1, \ldots, \tilde{u}_N) \in \mathbb{Z}_q^N$, with $\tilde{u}_i = \sum_{j=1}^N b_{ij}u_j = u_j$ for $j = \psi(i)$, and $r = \sum_{j=1}^N r_j u_j$, the second condition leads to two equalities:

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

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

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

$\mathbf{u} = (u_1, \ldots, u_N) \in \mathbb{Z}_q^N$ 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

$$u_i \leftarrow \mathsf{Hash}((\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}, pk), i).$$

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 $\tilde{\mathbf{u}}$ defined in the same way as above:

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

for $\tilde{r} = \sum_{j=1}^{N} \tilde{r}_j u_j$. By proving (5.5), it follows that every $\tilde{e}_i$ 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:

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

The last step of the preparatory work results from replacing in the above expression the equality of products, $\prod_{i=1}^{N} \tilde{u}_i = \prod_{j=1}^{N} u_j$, by an equivalent expression based on a chained vector $\hat{\mathbf{c}} = \{\hat{c}_1, \ldots, \hat{c}_N\}$ 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 $\hat{c}_i = g^{\hat{r}_i} \hat{c}_{i-1}^{\tilde{u}_i}$, which leads to $\hat{c}_N = \mathsf{Com}(u, \hat{r})$ for $u = \prod_{i=1}^{N} u_i$ and

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

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

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

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 $u_i \leftarrow \mathsf{Hash}((\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}), i)$, let $\tilde{u}_i = u_{\psi(i)}$, pick $\hat{r}_i \in_R \mathbb{Z}_q$, and compute $\hat{c}_i = g^{\hat{r}_i} \hat{c}_{i-1}^{\tilde{u}_i}$.
- Let $\hat{\mathbf{r}} = (\hat{r}_1, \ldots, \hat{r}_N)$ and $\hat{\mathbf{c}} = (\hat{c}_1, \ldots, \hat{c}_N)$.

- Compute $\bar{r} = \sum_{j=1}^{N} r_j$, $\hat{r} = \sum_{i=1}^{N} \hat{r}_i \prod_{j=i+1}^{N} \tilde{u}_j$, $r = \sum_{j=1}^{N} r_j u_j$, and $\tilde{r} = \sum_{j=1}^{N} \tilde{r}_j u_j$.

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

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

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

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

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

of size $N + 5$, where

$$\begin{aligned}X &= \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N\\ Y &= \mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q^2 \times \mathbb{G}_q^N\end{aligned}$$

denote the domain and the co-domain of ϕ, respectively. 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 $(\bar{r}, \hat{r}, r, \tilde{r}, \hat{\mathbf{r}}, \tilde{\mathbf{u}}) \in X$, we get a tuple of public values,

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

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

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

$$NIZKP\left[(\bar{r}, \hat{r}, r, \tilde{r}, \hat{\mathbf{r}}, \tilde{\mathbf{u}}) : \begin{array}{c}\bar{c} = g^{\bar{r}} \wedge \hat{c} = g^{\hat{r}} \wedge \tilde{c} = \mathsf{Com}(\tilde{\mathbf{u}}, r)\\ \wedge\ \tilde{e} = \mathsf{ReEnc}_{pk}(\prod_{i=1}^{N}(\tilde{e}_i)^{\tilde{u}_i}, -\tilde{r})\\ \wedge \left[\bigwedge_{i=1}^{N} (\hat{c}_i = g^{\hat{r}_i} \hat{c}_{i-1}^{\tilde{u}_i})\right]\end{array}\right], \qquad (5.6)$$

using the standard procedure from Section 5.4. The result of such a proof generation, $(c, s) \leftarrow \mathsf{GenProof}_{\phi}(x, y)$, consists of a single value $c = \mathsf{Hash}(y, t)$ and a tuple $s = \omega - c \cdot x \in X$ of size $2N + 4$, which we obtain from picking a tuple $w \in_R X$ of size $2N + 4$ and computing a tuple $t = \phi(w)$ of size $N + 5$. Alternatively, a different challenge $c = \mathsf{Hash}(y', t)$ could be derived directly from the public values $y' = (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}, \hat{\mathbf{c}}, pk)$, which has the advantage that $y = (\bar{c}, \hat{c}, \tilde{c}, \tilde{e}, \hat{\mathbf{c}})$ 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 $NIZKP[(\psi, \tilde{\mathbf{r}}) : \tilde{\mathbf{e}} = \mathsf{Shuffle}_{pk}(\mathbf{e}, \tilde{\mathbf{r}}, \psi)]$. We denote the generation and verification of such a proof $\pi = (c, s, \mathbf{c}, \hat{\mathbf{c}})$ by

$$\pi \leftarrow \mathsf{GenProof}_{pk}(\mathbf{e}, \tilde{\mathbf{e}}, \tilde{\mathbf{r}}, \psi)$$
$$b \leftarrow \mathsf{CheckProof}_{pk}(\pi, \mathbf{e}, \tilde{\mathbf{e}}).$$

respectively. Corresponding algorithms are depicted in Alg. 8.45 and Alg. 8.48. Note that generating the proof requires $8N+5$ and verifying the proof $9N+11$ modular exponentiations in $\mathbb{G}_q$. The proof itself consists of $4N+6$ elements ($2N+5$ elements from $\mathbb{Z}_q$ and $2N$ 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 [53]. 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 $c = h(pk, m, g^r)$ and $s = r - c \cdot sk$.[3] Using the public key pk, a given signature $\sigma = (c, s)$ of m can be verified by

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

For a 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 essentially a non-interactive zero-knowledge proof $NIZKP[(sk) : pk = g^{sk}]$, in which m is passed as an additional public input to the Fiat-Shamir hash function.

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 [2, 6]. 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.

[3]The traditional way of defining Schnorr signatures does not include the public key pk in the hash function. Mainly for reasons of consistency with the non-interactive zero-knowledge proofs of the previous subsection, we prefer here the *key-prefixing variant* of the Schnorr signature scheme [18].

5.7. Hybrid Encryption and Key-Encapsulation

For large messages $M \in \mathcal{B}^*$, public-key encryption schemes such as ElGamal are often not sufficiently efficient. This motivates the construction of *hybrid encryption schemes*, which combine the advantages of public-key encryption schemes with the advantages of 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 $(ek, C) \leftarrow \mathsf{Enc}_{pk}(M)$, which consists of the encapsulated key ek obtained from $(ek, K) \leftarrow \mathsf{Encaps}_{pk}()$ and the symmetric ciphertext $C \leftarrow \mathsf{Enc}'_K(M)$. The decryption $M \leftarrow \mathsf{Dec}_{sk}(ek, C)$ works in the opposite manner, i.e., first the symmetric key $K \leftarrow \mathsf{Decaps}_{sk}(ek)$ is reconstructed from ek and then the plaintext message $M \leftarrow \mathsf{Dec}'_K(C)$ is decrypted from C using K. Note that such a triple of algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ constructed 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 [38].

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 $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of integers modulo p, where the $p = kq + 1$ 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. A key pair in this setting consists of two values $sk \in_R \mathbb{Z}_q$ and $pk = g^{sk} \in \mathbb{G}_q$ (similar to to the Schnorr signature scheme). The key encapsulation generates a pair of values $(ek, K) = (g^r, h(pk^r))$, where $r \in_R \mathbb{Z}_q$ is picked uniformly at random. Using the private key sk, the symmetric key $K = h(e^{sk}) = h(pk^r)$ can then be reconstructed from ek. Note that both key encapsulation and decapsulation require a single exponentiation in $\mathbb{G}_q$.

A triple of algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ constructed in this way is a public-key encryption scheme, which can be proven to provide IND-CCA security provided that h is modeled as a random oracle, the gap-CDH problem[4] is hard relative to G_q, and $(\mathsf{Enc}', \mathsf{Dnc}')$ is a CCA-secure symmetric encryption scheme [38]. Later in this document (see Sections 7.4 and 8.5), we propose an instantiation of this scheme based on AES-GCM and SHA3-256. Given the significant efficiency benefits, such instantiations based on current standards are commonly accepted and widely used in practice. These standards have been designed to approximate the above preconditions to the best possible degree.

[4]Informally, the gap-CDH problem consists in solving the CDH problem when access to an oracle that solves the DDH problem is given.

Part III.
Protocol Specification

6. General Protocol Design

The goal of this chapter is to introduce the general design of the cryptographic voting protocol, which is an extension of the protocol presented in [31]. We introduce the involved parties, describe their roles, and define the communication channels over which they exchange messages during a protocol execution. We also define the adversary model and the underlying trust assumptions. The goal of the protocol design is to meet the requirements listed in Section 1.1 based on the adversary model and the trust assumptions. To define the cryptographic setting for a given election event, we give a comprehensive list of security and election parameters, which must be fixed before each protocol execution. Finally, we discuss some technical preliminaries that are necessary to understand the details of the protocol's technical concept.

6.1. Parties and Communication Channels

In our protocol, we consider five 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* (or *administrator* for short) 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 administrator determines and publishes the final election result. In the protocol extension introduced in Version 3.2 of this documents, we also let the administrator participate in generating the shared encryption key pair and in decrypting the votes in a distributed manner (together with the election authorities). The idea of extending the administrator's role in this way is to delegate the last decryption step to the party responsible for communicating the election result.[1]
- A group of *election authorities* guarantees the integrity and privacy of the votes submitted during the election period. They are numbered with indices $j \in \{1, \dots, s\}$, $s \geqslant 1$. During the pre-election phase, they establish jointly a public ElGamal encryption key pk (together with the administrator). 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

[1] In the most likely real-world setting within the Swiss context, the administrator will be controlled by a particular canton, whereas the election authorities will be controlled by one or multiple system providers. The purpose of including the canton in the decryption process is to enable a decryption ceremony on the election day, which is not under the control of the system provider.

cryptographic shuffle of the encrypted votes. Finally, they use their private key shares sk_j to decrypt the votes in a distributed manner (together with the administrator). During the protocol execution, they keep track of all their incoming and outgoing messages, which they provide as input for the universal verification process.

- The *printing authority* is responsible for printing the voting cards and delivering them to the voters. It receives the data necessary for generating the voting cards from the administrator 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.

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 election authorities in the protocol. 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 during user interaction). In Prot. 7.3, sending a message m confidentially is denoted by $[\![m]\!]$.

We assume that the administrator and the election authorities are in possession of a private signature key, which they use to sign all messages sent to other parties. Their output channels are therefore authentic. In the protocol diagrams of Chapter 7, sending a message m over an authentic channel is denoted by $[m]$. In Section 7.4, we give further details on how the presumed channel security can be achieved in practice, and in Section 8.5, 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.

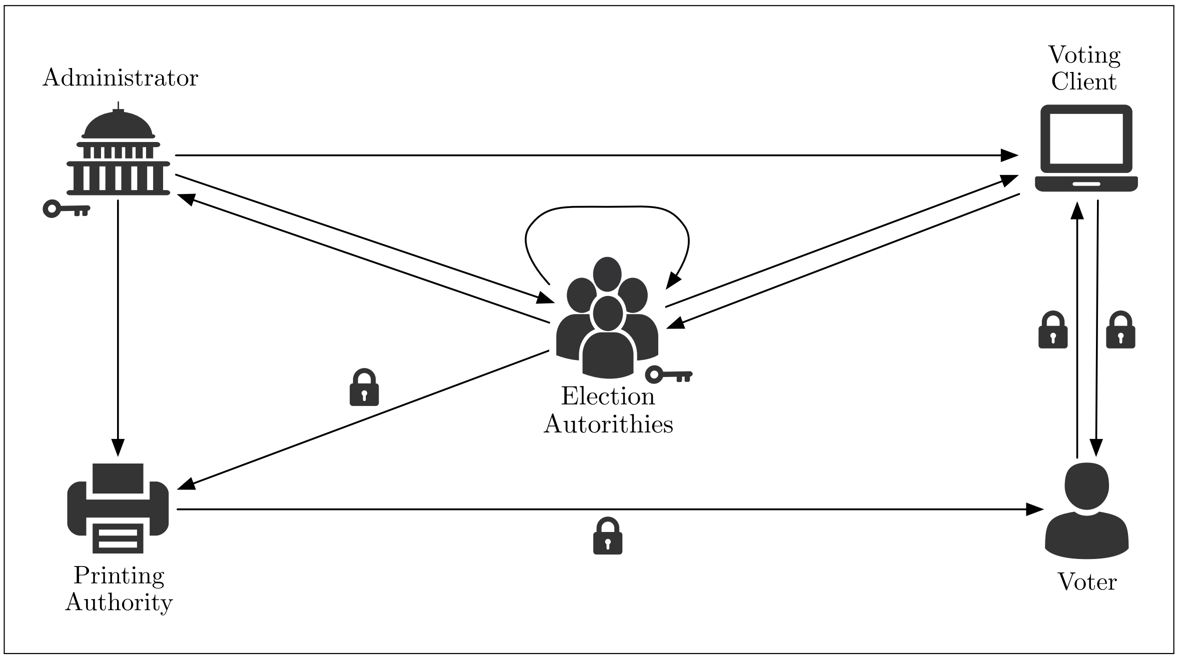


Figure 6.1.: Overview of the parties and communication channels.

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 [10]. 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 administrator and the election authorities.

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 10).

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 8.2), 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 defined and disseminated by the election administrator.

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 decreasing 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.

6.3.1.1. 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.

6.3.1.2. 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 [13, 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:

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

  Clearly, suitable prime numbers p and q providing the required number of bits can only be found efficiently using probabilistic primality tests such as the Miller-Rabin primality test. To keep the overall failure probability small enough, these tests are repeated multiple times until the desired probability is reached. In order to achieve compatibility with the selected security strength σ, the failure probabilities $P(p \notin \mathbb{P})$ and $P(q \notin \mathbb{P})$ must be $\frac{1}{2^\sigma}$ or smaller. Given that a single Miller-Rabin test results in a failure probability of at most $\frac{1}{4}$, the test must be repeated at least $\frac{\sigma}{2}$ times for both p and q. The same remark holds for the values $\hat{p}$, $\hat{q}$, and p' introduced below, for which the Miller-Rabin test must be repeated at least $\frac{\tau}{2}$ times.

  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 analogous 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. This connection requires the constraint for $\mathbb{G}_{\hat{q}}$ to be applied also to $\mathbb{Z}_{p'}$, i.e., we must consider $\|p'\| \geqslant 2\tau$ when choosing p'. For maximal simplicity, we generally set $p' = \hat{q}$, i.e., we perform the polynomial interpolation over the prime field $\mathbb{Z}_{\hat{q}}$. Then, an additional parameter that follows directly from $\hat{q}$ is the length L_M of the messages transferred by the OT-protocol. Since each of these messages represents a point in $\mathbb{Z}_{\hat{q}}^2$, we obtain $L_M = 2 \cdot \lceil \frac{\|\hat{q}\|}{8} \rceil$ bytes.

6.3.1.3. Parameters for Voting and Confirmation Codes

As we will see in Section 7.2, 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 necessary string lengths of X_i and Y_i as follows:

$$\ell_X = \lfloor \log_{|A_X|} \hat{q}_x \rfloor + 1, \qquad \ell_Y = \lfloor \log_{|A_Y|} \hat{q}_y \rfloor + 1.$$

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 two simplest cases, by setting both $\hat{q}_x$ and $\hat{q}_y$ to either $2^{2\tau-1}$ or to $\hat{q}$, these 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*[2] 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.

6.3.1.4. Parameters for Verification, Finalization, and Abstention Codes

Other elements printed on the voting card of voter i are the verification codes RC_{ij}, the finalization code FC_i, and the abstention code AC_i. Their main purpose is the detection of attacks by corrupt voting clients or election authorities. 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, F_i of length L_F, and A_i of length L_A, respectively, which are then converted into strings RC_{ij} of length ℓ_R, FC_i of length ℓ_F, and AC_i of length ℓ_A (for

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

given alphabets A_R, A_F, A_A). To provide the security defined by the deterrence factor in the covert adversary model, the following general constraints must be satisfied:

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

In our particular implementation of the inspection phase (see Prot. 7.10), we prefer to make abstention codes indistinguishable from finalization codes. This can be achieved by imposing identical code lengths $\ell_F = \ell_A$, byte lengths $L_F = L_A$, and alphabets $A_F = A_A$. We generally adopt this restriction in Table 6.1 and in the pseudo-code algorithms of Chapter 8 by replacing ℓ_F and ℓ_A by ℓ_{FA}, L_F and L_A by L_{FA}, and A_F and A_A by A_{FA}.

For $\epsilon = 0.999$ (0.001 chance of an undetected attack), for example, $L_R = L_{FA} = 2$ would be appropriate. In the case of the finalization and abstention codes, the string length ℓ_{FA} follows directly from L_{FA} and the size of the alphabet A_{FA}, respectively. 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 $j-1 \in \{0,\dots,n-1\}$ 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 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_{FA}, for example, we can calculate the lengths ℓ_R and ℓ_{FA} of corresponding strings based on respective alphabet sizes:

$$\ell_R = \left\lceil \frac{8L_R}{\log_2 |A_R|} \right\rceil, \quad \ell_{FA} = \left\lceil \frac{8L_{FA}}{\log_2 |A_{FA}|} \right\rceil.$$

For $L_R = 3$, $L_{FA} = 2$, and alphabet sizes $|A_R| = |A_{FA}| = 64$ (6 bits), $\ell_R = 4$ characters are required for the verification codes and $\ell_{FA} = 3$ characters for the finalization and abstention codes. We refer to Section 11.1.2 for further numerical examples and recommendations for practical parameters.

6.3.2. Election Parameters

A second category of parameters defines the details of a concrete election event. Defining such *election parameters* and making them accessible to every participating party is the responsibility of the election administrator. Table 6.2 at the end of this subsection summarizes the list of all election parameters and constraints to consider when selecting them. By grouping the most relevant parameters into a tuple

$$EP = (U, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{w}),$$

we obtain the most compact definition of an election event, in which all election parameters are included either explicitly or implicitly. Among the parameters included in EP, information about the voters from the electorate is included in $\mathbf{d}$, $\mathbf{E}$, and $\mathbf{w}$ (see Section 6.3.2.2).

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), $P(p \notin \mathbb{P}) \leqslant \frac{1}{2^\sigma}$
q	Order of $\mathbb{G}_q$, modulo of prime field $\mathbb{Z}_q$	$p = 2q+1$, $P(q \notin \mathbb{P}) \leqslant \frac{1}{2^\sigma}$
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), $P(\hat{p} \notin \mathbb{P}) \leqslant \frac{1}{2^\tau}$
$\hat{q}$	Order of $\mathbb{G}_{\hat{q}}$, modulo of prime field $\mathbb{Z}_{\hat{q}}$	$\|\hat{q}\| \geqslant 2\tau$, $P(\hat{q} \notin \mathbb{P}) \leqslant \frac{1}{2^\tau}$
$\hat{g}$	Generator of $\mathbb{G}_{\hat{q}}$	$g \in \mathbb{G}_{\hat{q}} \backslash 1$
L_M	Length of OT messages (bytes)	$L_M = 2 \cdot \lceil \frac{\|\hat{q}\|}{8} \rceil$
$\hat{q}_x$	Upper bound of private voting credential x	$2^{2\tau-1} \leqslant \hat{q}_x \leqslant \hat{q}$
A_X	Voting code alphabet	$\lvert A_X \rvert \geqslant 2$
ℓ_X	Length of voting codes (characters)	$\ell_X = \lfloor \log_{\lvert A_X \rvert} \hat{q}_x \rfloor + 1$
$\hat{q}_y$	Upper bound of private confirmation credential y	$2^{2\tau-1} \leqslant \hat{q}_y \leqslant \hat{q}$
A_Y	Confirmation code alphabet	$\lvert A_Y \rvert \geqslant 2$
ℓ_Y	Length of confirmation codes (characters)	$\ell_Y = \lfloor \log_{\lvert A_Y \rvert} \hat{q}_y \rfloor + 1$
$n_{\max}$	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	$\lvert A_R \rvert \geqslant 2$
ℓ_R	Length of verification codes RC_{ij} (characters)	$\ell_R = \left\lceil \frac{8L_R}{\log_2 \lvert A_R \rvert} \right\rceil$
L_{FA}	Length of finalization and abstention codes (bytes)	$8L_{FA} \geqslant \log \frac{1}{1-\epsilon}$
A_{FA}	Finalization and abstention code alphabet	$\lvert A_{FA} \rvert \geqslant 2$
ℓ_{FA}	Length of finalization and abstention codes (characters)	$\ell_{FA} = \left\lceil \frac{8L_{FA}}{\log_2 \lvert A_{FA} \rvert} \right\rceil$

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.

By selecting the relevant information for voter i,

$$VP_i = (U, \mathbf{c}, D_i, \mathbf{e}, \mathbf{n}, \mathbf{k}, \mathbf{e}_i, w_i),$$

we obtain the voter-specific *voting parameters* required for presenting the voting page to the voter and conducting the vote casting process. Therefore, VP_i (instead of EP) is sent to the voting client at the beginning of the vote casting process.

6.3.2.1. Election Event

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 particularly 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 including it in every digital signature issued during the protocol execution (see Section 7.4), 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 or a unique election event number) to allow participating parties to judge whether U is a fresh value or not.

6.3.2.2. 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.[3] For each voter i, a *voter description* $D_i \in A^*_{\mathsf{ucs}}$ and a counting circle $w_i \in \{1, \ldots, w\}$ must be specified. The electorate is therefore specified by two vectors $\mathbf{d} = (D_1, \ldots, D_{N_E})$ and $\mathbf{w} = (w_1, \ldots, w_{N_E})$ of length N_E. By assuming that voter descriptions are given as arbitrary UCS strings, we do not further define the type and format of the given information. Note that in the given election use cases of Section 2.2, voters are 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. The matrix $\mathbf{E} = (e_{ij})_{N_E \times t}$ of all such values is called *eligibility matrix* and its rows $\mathbf{e}_i = (e_{i,1}, \ldots, e_{i,t})$ are called *eligibility vectors*. Note that eligibility vectors $\mathbf{e}_i = (0, \ldots, 0)$ representing completely ineligible voters are not expected to exist in any of the considered use cases, but we do not exclude them explicitly.

6.3.2.3. Elections and Candidates

For each of the t elections, we assume that an *election description* $E_j \in A^*_{\mathsf{ucs}}$ is provided. While we do not further specify the type and format of the information given for each election, we assume $\mathbf{e} = \{e_1, \ldots, e_t\}$ to contain enough information for printing the voting cards and for displaying the voting page to the voter in the most meaningful way. In addition to $\mathbf{e}$, let $\mathbf{n} = (n_1, \ldots, n_t)$ specify the number of candidates in each election. The sum of these values, $n = \sum_{j=1}^{t} n_j$, represents the total number of candidates in the election event. For each such candidate $i \in \{1, \ldots, n\}$, a *candidate description* $C_i \in A^*_{\mathsf{ucs}}$ must be provided. Again, we do not further specify the type and format of the information given for each candidate.

[3] 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 $N \leqslant N_C$.

Other important parameters of an election event are the numbers of candidates k_j, $0 < k_j < n_j$, which voters are allowed to select in each election j. We exclude the two limiting cases of $k_j = 0$ and $k_j = n_j$, for which no use cases exist in the Swiss election context.[4] For a given eligibility matrix, we can derive the *maximal ballot size* $k'_{\max} = \max_{i=1}^{N_E} k'_i$, where $k'_i = \sum_{j=1}^{t} e_{ij} k_j$ denotes the total number of allowed selections of voter i. Note that $k'_{\max}$ can be considerably smaller than $k = \sum_{j=1}^{t} k_j$, depending on the sparseness of the eligibility matrix (see example in Section 2.2.2). It is limited by a constraint that follows from our particular vote encoding method (see Section 6.4.1).

Parameters		Constraints
U	Unique election event identifier	$U \in A^*_{\mathsf{ucs}}$
t	Number of elections	$t \in \mathbb{N}$
$\mathbf{e} = (E_1, \ldots, E_t)$	Election descriptions	$E_i \in A^*_{\mathsf{ucs}}$
$\mathbf{n} = (n_1, \ldots, n_t)$	Number of candidates	$n_j \in \mathbb{N}^+$
$\mathbf{k} = (k_1, \ldots, k_t)$	Number of selections	$k_j \in \mathbb{N}^+$, $k_j < n_j$
n	Total number of candidates	$n = \sum_{j=1}^{t} n_j$
$\mathbf{c} = (C_1, \ldots, C_n)$	Candidate descriptions	$C_i \in A^*_{\mathsf{ucs}}$
N_E	Number of eligible voters	$N_E \geqslant 0$
$\mathbf{d} = (D_1, \ldots, D_{N_E})$	Voter descriptions	$D_i \in A^*_{\mathsf{ucs}}$
$\mathbf{w} = (w_1, \ldots, w_{N_E})$	Assigned counting circles	$w_i \in \mathbb{N}^+$
w	Number of counting circles	$w = \max_{i=1}^{N_E} w_i$
$\mathbf{E} = (e_{ij})_{N_E \times t}$	Eligibility matrix	$e_{ij} \in \mathbb{B}$
$k'_{\max} = \max_{i=1}^{N_E} \sum_{j=1}^{t} e_{ij} k_j$	Maximum ballot size	$p_{n+w} \prod_{j=1}^{k'_{\max}} p_{n-j+1} < p$

Table 6.2.: List of election parameters and constraints.

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 and the algorithms in Chapters 7 and 8, 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

[4] In the current protocol, allowing $k_j = n_j$ and thus $k = n$ would reveal the finalization code to the voting client together with the verification codes. This could be exploited by a malicious voting client, which could then display the correct finalization code without submitting the ballot confirmation to the authorities. Therefore, the restriction $0 < k_j < n_j$ is also important to avoid this attack.

$\mathbb{G}_q$ as message space, we must define an invertible mapping Γ from the set of all possible 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 $\Gamma(\mathbf{s}) < p$ and efficient when n is small [30]. For optimal capacity, we choose the n second smallest prime numbers $p_1, \ldots, p_n \in \mathbb{P} \cap \mathbb{G}_q$, $p_i < p_{i+1}$, and define $\Gamma(i) = p_i$ for $i \in \{1, \ldots, n\}$.[5]

Since each encrypted votes is attributed to a counting circle, we extend the above invertible mapping $\Gamma : \{1, \ldots, n\}^k \rightarrow \mathbb{G}_q$ into $\Gamma' : \{1, \ldots, n\}^k \times \{1, \ldots, w\} \rightarrow \mathbb{G}_q$ 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 $c \in \{1, \ldots, w\}$ can then be encoded together as $\Gamma'(\mathbf{s}, c) = p_{n+c} \cdot \Gamma(\mathbf{s})$. 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 p, i.e., if $p_{n+w} \prod_{j=1}^{k} p_{n-j+1} < p$. 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 c 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

$$(p_{n+c} \cdot a \bmod p, b) = \mathsf{Enc}_{pk}(p_{n+c}, 0) \cdot \mathsf{Enc}_{pk}(\Gamma(\mathbf{s}), r) = \mathsf{Enc}_{pk}(\Gamma'(\mathbf{s}, c), r)$$

is an ElGamal encryption of $\Gamma'(\mathbf{s}, c)$. 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. Protecting Vote Privacy of Voters with Restricted Eligibility

To diminish the magnitude of the vote privacy problem mentioned in Section 2.2.2, we propose to extend the ballots from voters with restricted eligibility within their counting circle to the size of the ballots from voters with unrestricted voting rights. For this, we infer from $\mathbf{w}$ and $\mathbf{E}$ the *default eligibility matrix* $\mathbf{E}^* = (e^*_{cj})_{w \times t}$. Each of its values $e^*_{cj} \in \mathbb{B}$ represents the default eligibility of the members of counting circle $c \in \{1, \ldots, w\}$ in election $j \in \{1, \ldots, t\}$. These value can be computed by

$$e^*_{cj} = \max_{i \in I_c} e_{ij},$$

where $I_c = \{i \in \{1, \ldots, N_E\} : w_i = c\}$ denotes the set of indices of voters from the same counting circle.

From $\mathbf{E}^*$ we can infer the *default ballot size* $k^*_c = \sum_{j=1}^{t} e^*_{cj} k_j$ of each counting circle c, which defines a vector $\mathbf{k}^* = (k^*_1, \ldots, k^*_w)$ of size w. In the example of Section 2.2.2 with election parameters $w = 2$, $t = 8$, and $k_1 = \cdots = k_8 = 1$, we get

$$\mathbf{E}^* = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \text{ and } \mathbf{k}^* = \begin{pmatrix} 5 \\ 5 \end{pmatrix}.$$

[5]The smallest prime number in $p_0 \in \mathbb{G}_q$ is reserved for a different purpose.

For voter $i \in \{1, \ldots, N_E\}$ with restricted eligibility in counting circle $c = w_i$, the default ballot size of the counting circle can be reached by adding *default votes* for

$$k^*_c - k'_i = \sum_{j=1}^{t} (e^*_{cj} - e_{ij})k_j$$

candidates to the voter's ballot. This is the proposed way of dealing with the aforementioned vote privacy problem, which arises for example if no voter other than i has the same restricted eligibility in counting circle c. Therefore, computing the matrix $\mathbf{E}^*$ will be necessary at some point in the protocol to determine the elections for which a given voter's eligibility deviates from the default eligibility of the voter's counting circle. When this happens, we get $e_{ij} = 0$ and $e^*_{cj} = 1$, i.e., such cases can be detected by checking $e_{ij} < e^*_{cj}$ over all election indices $j \in \{1, \ldots, t\}$. This is the test that we use for toggling the inclusion of default votes.

A default vote in election j is a vote for k_j candidates from the set $I_j = \{n'_j + 1, \ldots, n'_j + n_j\}$, $n'_j = \sum_{i=1}^{j-1} n_i$. For solving the above-mentioned privacy problem, candidates included in a default vote should be unlikely to receive zero votes from regular voters. Because predicting the amount of votes is impossible without any further information about the candidates and their prospects in the election, we can not offer a technical solution to guarantee the best possible selection. Therefore, we simply propose to select the k_j first candidates from I_j, i.e., all candidates from $I'_j = \{n'_j + 1, \ldots, n'_j + k_j\} \subset I_j$.

Default votes are added to the ballots in a verifiable manner by the election authorities, i.e., before mixing them in the re-encryption mix-net. For example, if $(a, b) = \mathsf{Enc}_{pk}(\Gamma'(\mathbf{s}, c), r)$ denotes an encrypted vote from a voter in counting circle c with restricted eligibility in election j, then adding a default vote for I'_j can be achieved by replacing the encryption (a, b) obtained from the ballot by

$$(a', b') = (a \cdot \prod_{i \in I'_j} p_i \bmod p, b).$$

In other words, default votes are added by multiplying their prime number representations to the left-hand side of the ElGamal encryption obtained from the ballot. As explained above, the total amount of candidates to be added in this way is $k^*_c - k'_i$.

To guarantee the correct election outcome at tallying, every added default vote needs to be subtracted from the final result in an unambiguous way. For this purpose, we propose to create additional encryptions containing votes for exactly the same added candidates. These encryptions are marked using the smallest prime number $p_0 \in \mathbb{P} \cap \mathbb{G}_q$ and added to the list of encrypted votes for further processing. This step is performed by each election authority in a deterministic way using the encryption randomization $r = 0$. Therefore, if a default vote for I'_j has been added to an encrypted vote in counting circle c, then

$$(a^*, b^*) = (p_0 \cdot p_{n+c} \cdot \prod_{i \in I'_j} p_i \bmod p, 1)$$

is the corresponding marked encryption to be created. At tallying, when $p_0 \,|\, m^*$ indicates the presence of the mark p_0 for a given decrypted vote $m^* = p_0 \cdot p_{n+c} \cdot \prod_{i \in I'_j} p_i \bmod p$, the subtraction of corresponding default votes is triggered. In Section 6.5, we will give further details on how this process works.

6.4.3. Linking OT Queries to ElGamal Encryptions

If the same encoding $\Gamma : \{1, \ldots, n\} \rightarrow \mathbb{G}_q$ 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 [31]. 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.

6.4.4. Verification Codes

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. But 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. Assuming that a given voter is eligible in all t elections, let $RC_1, \ldots, RC_n \in A_R^{\ell_R}$ be the voter's verification codes for the $n = \sum_{j=1}^{t} n_j$ candidates. These codes are constructed as follows (the same procedure is repeated for every voter):

- Authority $j \in \{1, \ldots, s\}$ picks a random polynomial $A_j(X) \in_R \mathbb{Z}_{\hat{q}}[X]$ of degree $k - 1$, where $k = \sum_{j=1}^{t} k_j$ denotes the total number of allowed selections. From this polynomial, the authority selects n random points $p_{ij} = (x_{ij}, y_{ij})$ by picking n distinct random values $x_{ij} \in_R \mathbb{Z}_{\hat{q}} \backslash \{0\}$ and computing $y_{ij} = A_j(x_{ij})$. The result is a vector $\mathbf{p}_j = (p_{1,j}, \ldots, p_{n,j})$ of length n. Over all s authorities, this defines a matrix $\mathbf{P} = (p_{ij})_{n \times s}$. Different such matrices are generated for all voters.
- The verification codes RC_i are derived from the columns $\mathbf{p}_i = (p_{i,1}, \ldots, p_{i,s})$ of $\mathbf{P}$ by first computing
$$R_{ij} = \mathsf{Truncate}(\mathsf{RecHash}(p_{ij}), L_R)$$
for each point. The length L_R of the resulting byte arrays matches with the desired code length ℓ_R and the selected alphabet A_R (see Table 6.1). The values R_{ij} are then combined into a single value $R_i = \oplus_{j=1}^{s} R_{ij}$. To avoid identical codes on a single code sheet, the candidate index i is added to R_i as a watermark. By converting the resulting watermarked byte array R'_i into a string, we finally obtain the verification code $RC_i \in A_R^{\ell_R}$.

To prepare the voting cards prior to an election, the printing authority receives from the election authorities one such matrix $\mathbf{P}$ for every voter. The verification codes can then be derived as explained above.

During vote casting, every authority transfers only $k < n$ points from $\mathbf{p}_j$ obliviously to the voter's voting client, i.e., the voting client receives a sub-matrix $\mathbf{P_s} = (p_{ij})_{k \times s}$ of such points, which depends on the voter's selection $\mathbf{s} = (s_1, \ldots, s_k)$. The verification code RC_{s_i} for the selected candidate s_i is derived from the points of row $\mathbf{p}_{s_i} = (p_{i,1}, \ldots, p_{i,s})$ of $\mathbf{P_s}$ in the same way as explained above for the full matrix $\mathbf{P}$. Repeating this procedure for all of the voter's k selections leads to the desired vector $\mathbf{rc_s} = (RC_{s_1}, \ldots, RC_{s_k})$ of verification codes for all selected candidates.

By obtaining k points from election authority j, the voting client can reconstruct the polynomial $A_j(X)$ of degree $k-1$, if at least k distinct points from $A_j(X)$ are available (see Section 3.2.2). If this is the case, the simultaneous $\mathrm{OT}^{\mathbf{k}}_{\mathbf{n}}$ query must have been formed properly under the constraints given by $\mathbf{n}$ and $\mathbf{k}$. 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 the value $z_j = A_j(0)$, which can not be guessed efficiently without knowing the polynomial. In this way, the voting client obtains s such values, one from every authority. Their integer sum $z_i = \sum_{j=1}^{s} z_{ij} \bmod \hat{q}$ is called *vote validity credential*. It is used in the vote confirmation process to prove the well-formedness of the encrypted vote (see following subsection).

For voters with restricted eligibility, the above procedure needs to be slightly adjusted. For voter i, if $k'_i = \sum_{j=1}^{t} e_{ij} k_j$ denotes the restricted number of allowed selections and $n'_i = \sum_{j=1}^{t} e_{ij} n_j$ the corresponding number of available candidates, then the degree of the randomly selected polynomial is set to $k'_i - 1$ and the amount of randomly selected points on the polynomial is reduced to n'_i. For all $n - n'_i$ other candidates, default points $(0,0)$ not lying on the polynomial are added to the vector $\mathbf{p}_j$, i.e., the resulting matrix $\mathbf{P}$ contains $n - n'_i$ columns consisting of such default points only. We introduce this step mainly for obtaining matrices $\mathbf{P}$ of consistent height n. During vote casting, by fixing $\mathbf{n}$ over all voters, this greatly simplifies the implementation of the simultaneous OT protocol.

6.4.5. Finalization and Abstention Codes

The finalization code FC is derived from the same matrix $\mathbf{P}$ of randomly selected points on the polynomials $A_j(X)$, but by taking into account all its points (including the default points $(0,0)$ added to compensate for the voter's restricted eligibility). First, values

$$F_j = \mathsf{Truncate}(\mathsf{RecHash}(\mathbf{p}_j), L_{FA})$$

are computed for each row $\mathbf{p}_j = (p_{1,j}, \ldots, p_{n,j})$ of $\mathbf{P}$. As above, the length L_{FA} of the resulting byte arrays is consistent with the desired code length ℓ_{FA} and the selected alphabet A_{FA}. By combining the values F_j into a single value $F = \bigoplus_{j=1}^{s} F_j$ and converting it into a string $FC \in A_{FA}^{\ell_{FA}}$, we obtain the voter's finalization code.[6] This is the procedure conducted by the printing authority during the election preparation after receiving $\mathbf{P}$ from the election authorities.

For the voting client, this procedure for obtaining the full matrix $\mathbf{P}$ is slightly more complicated. After confirming the submitted votes, each of the authorities responds with the two randomizations used to conduct the OT protocol. Let us just consider the case of a single

[6]Note that this particular way of generating the finalization codes requires the number of selections k to be strictly smaller than the number of candidates n. Otherwise, submitting a valid ballot would not only reveal all $k = n$ verification codes to the voting client, but also the finalization code. A malicious voting client could then suppress the ballot confirmation, but still display the correct finalization code.

election authority, which send the values $(z_1, z_2) \in \mathbb{Z}_q^2$ to the voting client. By computing $d' = g^{z_1}h^{z_2}$ and checking $d = d'$, the voting client first verifies the correctness of the received values. The k values β_j can then be derived from $\mathbf{b}$ and the n values $k_i = \Gamma(i)^{z_1}$ from the prime number representations $\Gamma(i) = p_i$ (see Prot. 5.3 in Section 5.3.3). The resulting keys $k_{ij} = k_i\beta_j$ can then be used to open all transmitted messages from $\mathbf{C}$ and thus to derive all points from $\mathbf{p}_j$. This step requires $n + k$ many modular exponentiations.

The abstention code $AC \in A_{FA}^{\ell_{FA}}$ of a given voting card is also generated in a distributed manner, but the procedure is much simpler. Each authority j simply picks a random byte array $A_j \in \mathcal{B}^{L_{FA}}$ of length L_{FA}, which are then combined with an exclusive-or into a single value $A = \oplus_{j=1}^{s} A_j$. This value is then converted into a string of length ℓ_A. Exactly the same procedure is conducted by the printing authority and the voting client.

6.4.6. 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). Without entering these codes, or by entering incorrect codes, the identification fails and the process stops.

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

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

For a given pair $(x, \hat{x}) \in \mathbb{Z}_{\hat{q}} \times \mathbb{G}_{\hat{q}}$ of private and public voting credentials, executing the Schnorr identification protocol corresponds to computing a non-interactive zero-knowledge proof $NIZKP[(x) : \hat{x} = \hat{g}^x \bmod \hat{p}]$. In our protocol, we combine this proof with a proof of knowledge of the plaintext vote contained in the submitted ballot (see Section 5.4.1). Similarly, for a given pair $(y, \hat{y}) \in \mathbb{Z}_{\hat{q}} \times \mathbb{G}_{\hat{q}}$, a non-interactive zero-knowledge proof $NIZKP[(y) : \hat{y} = \hat{g}^y \bmod \hat{p}]$ is computed to prove knowledge of y. This proof is combined with a proof of knowing the vote validity credential $z = \sum_{j=1}^{s} z_j \bmod \hat{q}$, which is derived from the values $z_j = A_j(0)$ obtained during vote casting from successfully conducting the OT protocol with every election authority (see Section 6.4.4).

6.5. Election Result

After successfully mixing and decrypting the votes, the *raw election result* $ER = (\mathbf{u}, \mathbf{V}, \mathbf{W})$ can be derived from the data available to the election administrator. This tuple consists of one vector $\mathbf{u} = (u_1, \ldots, u_N)$ and two matrices $\mathbf{V} = (v_{ik})_{N \times n}$ and $\mathbf{W} = (w_{ic})_{N \times w}$, where N denotes the number of decrypted votes. For $i \in \{1, \ldots, N\}$, $k \in \{1, \ldots, n\}$, and $c \in \{1, \ldots, w\}$, the meaning of the values contained in this vector and the two matrices is as follows:

- $u_i \in \{-1, 1\}$ is set to 1, if plaintext vote i represents a regular vote from an eligible voter, and to -1, if the vote has been added by the election authorities for canceling out added default votes;
- $v_{ik} \in \mathbb{B}$ is set to 1, if plaintext vote i contains a vote for candidate k, and to 0, if this is not the case;
- $w_{ic} \in \mathbb{B}$ is set to 1, if plaintext vote i contains a vote for counting circle c, and to 0, if this is not the case.

To illustrate the idea behind these values, consider the example from Section 2.2.2 with parameters $t = 9$, $n = 16$, $w = 2$, and $N_E = 7$. Furthermore, suppose that all $N_E = 7$ voters have submitted a vote. Since two voters from the first counting circle have restricted eligibility in Election 1 and one voter from the second counting circle has restricted eligibility in Election 8, two default votes for Candidate 1 and one default vote for Candidate 15 have been added to their ballots. This gives a total of $N = 10$ decrypted votes. The election result could therefore look as follows:

$$\mathbf{u} = \begin{pmatrix} 1 \\ -1 \\ 1 \\ 1 \\ -1 \\ 1 \\ 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}, \quad \mathbf{V} = \begin{pmatrix} 1&0&0&1&0&0&0&0&0&0&0&1&0&1&1&0 \\ 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 1&0&0&1&0&1&0&1&0&1&0&0&0&0&0&0 \\ 1&0&1&0&1&0&1&0&0&1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&1&1&0&0&1&1&0&1&0&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0&0&0&1&0&1&0&1&0 \\ 0&1&0&1&0&0&0&0&0&0&1&0&0&1&1&0 \\ 1&0&0&1&1&0&0&1&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0 \end{pmatrix}, \quad \mathbf{W} = \begin{pmatrix} 0&1 \\ 1&0 \\ 1&0 \\ 1&0 \\ 1&0 \\ 1&0 \\ 0&1 \\ 0&1 \\ 1&0 \\ 0&1 \end{pmatrix}.$$

The raw election result can be used to sum up the votes for all candidates. If necessary, local results can computed for each counting circle, and the turnout of each counting circle can be determined:

- Number of votes for candidate $k \in \{1, \ldots, n\}$ in counting circle $c \in \{1, \ldots, w\}$:

$$V(k, c) = \sum_{i=1}^{N} u_i v_{ik} w_{ic}.$$

- Total number of votes for candidate $k \in \{1, \ldots, n\}$ over all counting circles:

$$V(k) = \sum_{c=1}^{w} V(k, c) = \sum_{i=1}^{N} u_i v_{ik}.$$

- Total number of submitted votes in counting circle $c \in \{1, \dots, w\}$:

$$W(c) = \frac{1}{2} \sum_{i=1}^{N} (u_i + 1) \cdot w_{ic}.$$

- Total number of submitted votes over all counting circles:

$$W = \sum_{c=1}^{w} W(c) = \frac{1}{2} \sum_{i=1}^{N} (u_i + 1).$$

The results obtained for the above example are shown in the following table. Note that two votes for Candidate 1 and one vote for Candidate 15 have been subtracted to compensate for corresponding default votes added to the ballots of Voters 1, 4, and 7. The turnouts of the two counting circles are $W(1) = 4$ and $W(2) = 3$, and the total turnout is $W = 7$.

Election	1		2		3		4		5		6		7		8	
Candidate k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$V(k,1)$	1	1	2	2	2	2	2	2	1	3	0	0	0	0	0	0
$V(k,2)$	2	1	1	2	0	0	0	0	0	0	2	1	1	2	2	0
$V(k)$	3	2	3	4	2	2	2	2	1	3	2	1	1	2	2	0

7. 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 sections with sufficient technical details to understand 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 7.1. This overview illustrates the central role and strong involvement of the election authorities in almost every step of the whole process. Their common view of the public election data assembled during the election process defines the main input for the verification process.[1]

Phase	Administrator	Election Authority	Printing Authority	Voter	Voting Client	Protocol Nr.
1. Pre-Election	•	•	•	•		
1.1 Key Generation	•	•				7.1
1.2 Election Preparation		•				7.2
1.3 Printing of Voting Cards	•	•	•	•		7.3
2. Election	•	•		•	•	
2.1 Candidate Selection	•			•	•	7.4
2.2 Vote Casting		•			•	7.5
2.3 Vote Confirmation		•		•	•	7.6
3. Post-Election	•	•		•	•	
3.1 Mixing		•				7.7
3.2 Decryption	•	•				7.8
3.3 Tallying	•					7.9
3.4 Inspection		•		•	•	7.10

Table 7.1.: Overview of the protocol phases and sub-phases with the involved parties.

In each of the following sections, we provide comprehensive illustrations of corresponding protocol sub-phases. The illustrations are numbered from Prot. 7.1 to Prot. 7.10. 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 protocol sub-phase, 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 8. Note that the illustrations

[1]The verification process is currently not discussed in this document. It is planned to be discussed in a separate section to this document in a future release.

also show the signatures that are generated by the administrator and the election authorities. These signatures are important to provide authenticity, i.e., they must be generated whenever a message m is sent and verified whenever m is received. We use $[m]$ to indicate that a signature is attached to m by the sender. Further details of the signature generation are discussed in Section 7.4 and corresponding algorithms are given in Section 8.5.

7.1. Pre-Election Phase

The pre-election phase of the protocol involves all necessary tasks to setup an election event. Besides generating a shared public encryption key, 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 tuple

$$VC_i = (i, X_i, Y_i, \mathbf{rc}_i, FC_i, AC_i)$$

of values, which the voter needs to successfully submit a vote. This tuple contains the voter index i, the voting code X_i, the confirmation code Y_i, the verification codes $\mathbf{rc}_i$, the finalization code FC_i, and the abstention code AC_i. Other information required for printing the voting cards are included in the election parameters EP.

7.1.1. Key Generation

In the first step of the election preparation, a public ElGamal encryption key $pk \in \mathbb{G}_q$ is generated jointly by the administrator and the election authorities. As shown in Prot. 7.1, pk is known to every authority at the end of the protocol step, and each of them holds a share $sk_j \in \mathbb{Z}_q$ of the corresponding private key. The sub-protocol involves calls to two algorithms $\mathsf{GenKeyPair}()$ and $\mathsf{GenKeyPairProof}(sk_j, pk_j)$ for generating the key shares and corresponding cryptographic proofs. If all keys and proofs are valid, pk results from calling $\mathsf{GetPublicKey}(\mathbf{pk})$. For details of these algorithms, we refer to Section 5.1.2 and Algs. 8.6 to 8.9.

7.1.2. Election Preparation

The voting cards are generated by the s election authorities in a distributed manner (see Sections 6.4.4 to 6.4.6 for technical background). For this, each election authority j calls an algorithm $\mathsf{GenElectorateData}(\mathbf{n}, \mathbf{k}, \mathbf{E}, s)$ with the election parameters $\mathbf{n}$, $\mathbf{k}$, and $\mathbf{E}$ obtained beforehand from the election administrator as part of EP. The result obtained from calling this algorithm consists of the voting card data $\mathbf{d}_j$ to be sent to the printing authority, the shares of the private and public credentials $\mathbf{x}_j$, $\mathbf{y}_j$, $\hat{\mathbf{x}}_j$, and $\hat{\mathbf{y}}_j$, and the matrix of random points $\mathbf{P}_j$. For proving knowledge of the shares of the private credentials, a non-interactive proof $\hat{\pi}_j$ is generated and published along with $\hat{\mathbf{x}}_j$ and $\hat{\mathbf{y}}_j$. These first steps are depicted in the upper part of Prot. 7.2.

At the end of the above process, every election authority knows all the shares $\hat{\mathbf{X}} = (\hat{\mathbf{x}}_1, \dots, \hat{\mathbf{x}}_s)$ and $\hat{\mathbf{Y}} = (\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_s)$ of the public credentials of the whole electorate. If the attached cryptographic proofs $\hat{\boldsymbol{\pi}} = (\hat{\pi}_1, \dots, \hat{\pi}_s)$ are all valid, the authorities call $\mathsf{GetPublicCredentials}(\hat{\mathbf{X}}, \hat{\mathbf{Y}}, \hat{\mathbf{Z}})$

Administrator		Election Authority $j \in \{1,\dots,s\}$		Election Authority $k \in \{1,\dots,s\}\backslash\{j\}$
knows EP				
$(sk_0, pk_0) \leftarrow \mathsf{GenKeyPair}()$				
$\pi_0 \leftarrow \mathsf{GenKeyPairProof}(sk_0, pk_0)$				
	$\xrightarrow{[EP, pk_0, \pi_0]_{\varnothing}}$			
		if $\neg\mathsf{CheckKeyPairProof}(\pi_0, pk_0)$ **abort**		
		$(sk_j, pk_j) \leftarrow \mathsf{GenKeyPair}()$		
		$\pi_j \leftarrow \mathsf{GenKeyPairProof}(sk_j, pk_j)$		
			$\xrightarrow{[pk_j, \pi_j]_{EP}}$	
			$\xleftarrow{[pk_k, \pi_k]_{EP}}$	
		if $\neg\mathsf{CheckKeyPairProof}(\pi_k, pk_k)$ **abort**		
		$\mathbf{pk} \leftarrow (pk_0, pk_1, \dots, pk_s)$		
		$pk \leftarrow \mathsf{GetPublicKey}(\mathbf{pk})$		

Protocol 7.1: Key Generation

to obtain the two lists $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ of aggregated public credentials. They are used to identify the voters during vote casting and confirmation (see Section 6.4.6 and Alg. 8.17 for further details).

7.1.3. Printing of Code Sheets

The voting card data $\mathbf{d}_j$ generated by authority j contains for every voting card the authority's shares of both the private voting and confirmation credentials and the verification, finalization, and abstention codes. This information is very sensitive and can only be shared with the trusted printing authority. The process of sending $\mathbf{d}_j$ to the printing authority is depicted in Prot. 7.3. 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 use double brackets $[\![\dots]\!]$ to indicate the added encryption layer. Further details on this are given in Section 7.4.

The voting cards can be generated from the collected voting card data $\mathbf{D} = (\mathbf{d}_1, \dots, \mathbf{d}_s)$. The printing authority uses it as inputs for the algorithm $\mathsf{GetVotingCards}(\mathbf{D})$, which produces corresponding voting cards $\mathbf{vc} = (VC_1, \dots, VC_{N_E})$. Confidential printouts of these voting cards are sent to the voters, for example using a trusted postal service. Again, we use double brackets $[\![\dots]\!]$ to show the confidentiality of the transmitted information.

Election Authority $j \in \{1, \dots, s\}$		Election Authority $k \in \{1, \dots, s\} \backslash \{j\}$
knows EP		
$(\cdot, \cdot, \cdot, \cdot, \mathbf{n}, \mathbf{k}, \mathbf{E}, \cdot) \leftarrow EP$		
$(\mathbf{a}_j, \mathbf{d}_j, \mathbf{x}_j, \mathbf{y}_j, \mathbf{z}_j, \hat{\mathbf{x}}_j, \hat{\mathbf{y}}_j, \hat{\mathbf{z}}_j, \mathbf{P}_j)$ $\leftarrow \mathsf{GenElectorateData}(\mathbf{n}, \mathbf{k}, \mathbf{E}, s)$		
$\hat{\pi}_j \leftarrow \mathsf{GenCredentialProof}(\mathbf{x}_j, \mathbf{y}_j, \mathbf{z}_j, \hat{\mathbf{x}}_j, \hat{\mathbf{y}}_j, \hat{\mathbf{z}}_j)$		
	$\xrightarrow{[\hat{\mathbf{x}}_j, \hat{\mathbf{y}}_j, \hat{\mathbf{z}}_j, \hat{\pi}_j]_{EP}}$	
	$\xleftarrow{[\hat{\mathbf{x}}_k, \hat{\mathbf{y}}_k, \hat{\mathbf{z}}_k, \hat{\pi}_k]_{EP}}$	
if $\neg\mathsf{CheckCredentialProof}(\hat{\pi}_k, \hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}})$ **abort**		
$\hat{\mathbf{X}} \leftarrow (\hat{\mathbf{x}}_1, \dots, \hat{\mathbf{x}}_s), \hat{\mathbf{Y}} \leftarrow (\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_s), \hat{\mathbf{Z}} \leftarrow (\hat{\mathbf{z}}_1, \dots, \hat{\mathbf{z}}_s)$		
$(\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}) \leftarrow \mathsf{GetPublicCredentials}(\hat{\mathbf{X}}, \hat{\mathbf{Y}}, \hat{\mathbf{Z}})$		

Protocol 7.2: Election Preparation.

Administrator		Printing Authority		Election Authority $j \in \{1, \dots, s\}$
knows EP				knows $\mathbf{d}_j$
	$\xrightarrow{[EP]}$			
			$\xleftarrow{[[\![\mathbf{d}_j]\!]]_{EP}}$	
		$\mathbf{D} \leftarrow (\mathbf{d}_1, \dots, \mathbf{d}_s)$		
		$\mathbf{vc} \leftarrow \mathsf{GetVotingCards}(\mathbf{D})$		
				Voter $v \in \{1, \dots, N_E\}$
			$\xrightarrow{[\![VC_v]\!]}$	

Protocol 7.3: Printing of Voting Cards.

7.2. Election Phase

The election phase is the core of the cryptographic voting protocol. The start and end of this phase is 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 multiple ways of dealing with this problem, we do not propose a solution in this document. We only assume that the election authorities will always agree on whether a particular ballot or confirmation has been submitted within the election period, and only accept it if this is the case.

The main actors of the election phase are the voters and the election authorities. 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.

7.2.1. Candidate Selection

The first step for the voter is the selection of the candidates. In an election event with t simultaneous elections, voter $v \in \{1, \ldots, N_E\}$ must select exactly $e_{vj}k_j$ candidates for each election $j \in \{1, \ldots, t\}$ 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 $\mathbf{e}_v$, which the voting client receives from the administrator as part of voting parameters VP_v. This preparatory step is shown in the upper part of Prot. 7.4, where $\mathsf{GetVotingParameters}(v, EP)$ is called by the voting client. The voter's selection is a set $S = \{s_1, \ldots, s_{k'_v}\}$ of distinct values $s_j \in \{1, \ldots, n\}$ satisfying the constraints of the current election event and the voter's eligibility. The voter enters these values together with the voting code X_v from the voting card. To disambiguate permutations of the same selections and to guarantee the constraint (5.1) from Section 5.3.3, we assume voters to sort this set in ascending order by $\mathbf{s} \leftarrow \mathsf{Sort}_{\leqslant}(S)$.

7.2.2. Vote Casting

Based on the voter's selection $\mathbf{s} = (s_1, \ldots, s_{k'_v})$, the voting client generates a ballot $\alpha = (\hat{x}_v, \mathbf{a}, \pi)$ by calling an algorithm $\mathsf{GenBallot}(X_v, \mathbf{s}, pk, \mathbf{n}, w_v)$ using the authorities' common public encryption key pk. The ballot contains an OT query $\mathbf{a} = (a_1, \ldots, a_{k'_v}) \in (\mathbb{G}_q^2)^{k'_v}$ for corresponding verification codes. By using the public encryption key pk in the oblivious transfer as a generator of the group $\mathbb{G}_q$ (see Section 6.4.3), 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

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

that links the OT query to the voting credentials. This proof includes all elements of a Schnorr identification relative to $\hat{x}_v$ (see Section 6.4.6).

The ballot α is submitted to the election authorities. Each authority checks its validity by calling $\mathsf{CheckBallot}(v, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}})$. This algorithm verifies that the size of $\mathbf{a}$ is exactly

Voter $v \in \{1,\ldots,N_E\}$		Voting Client		Administrator
knows VC_v				knows EP, pk_0, π_0
$(v,\cdot,\cdot,\cdot,\cdot,\cdot) \leftarrow VC_v$				
	$\xrightarrow{v}$			
			$\xrightarrow{v}$	
				$VP_v \leftarrow$ $\mathsf{GetVotingParameters}(v, EP)$
			$\xleftarrow{[VP_v, pk_0, \pi_0]_\varnothing}$	
		if $\neg\mathsf{CheckKeyPairProof}(\pi_0, pk_0)$ **abort**		
	$\xleftarrow{VP_v}$			
selects S $\mathbf{s} \leftarrow \mathsf{Sort}_{\leqslant}(S)$ $(\cdot, X_v,\cdot,\cdot,\cdot,\cdot) \leftarrow VC_v$				
	$\xrightarrow{X_v, \mathbf{s}}$			

Protocol 7.4: Candidate Selection

$k'_v = \sum_{j=1}^{t} e_{vj} k_j$, that the public voting credential $\hat{x}_v$ is included in $\hat{\mathbf{x}}$, and that the zero-knowledge proof π is valid (which implies that the voter is in possession of a valid voting code X_v). An additional test is necessary for checking 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 (see Section 6.4.4). 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'_v matrix $\mathbf{P}$ of selected points from every β_j. Finally, by calling $\mathsf{GetVerificationCodes}(\mathbf{s}, \mathbf{P})$, it computes the verification codes $\mathbf{rc} = (RC_{s_1}, \ldots, RC_{s_{k'_v}})$ for the selected candidates. This whole procedure is depicted in Prot. 7.5.

7.2.3. Vote Confirmation

The voting client displays the verification codes $\mathbf{rc} = (RC_{s_1}, \ldots, RC_{s_{k'_v}})$ for the selected candidates to the voter for comparing them with the codes $\mathbf{rc}_v$ printed on the voter's voting card. We describe this process by an algorithm call $\mathsf{CheckVerificationCodes}(\mathbf{rc}_v, \mathbf{rc}, \mathbf{s})$, which

Voting Client		Election Authority $j \in \{1,\dots,s\}$
knows $VC_v, \mathbf{s}, VP_v, pk_0, \pi_0$		knows $EP, pk, \hat{\mathbf{x}}, pk_j, \pi_j, \mathbf{P}_j, B_j, F_j$
$(v,\cdot,\cdot,\cdot,\cdot,\cdot) \leftarrow VC_v$		
	$\xrightarrow{v}$	
		$VP_v \leftarrow \mathsf{GetVotingParameters}(v, EP)$
	$\xleftarrow{[pk_j, \pi_j]_{VP_v}}$	
if $\neg\mathsf{CheckKeyPairProof}(\pi_j, pk_j)$ **abort**		
$\mathbf{pk} \leftarrow (pk_0, pk_1, \dots, pk_s)$ $pk \leftarrow \mathsf{GetPublicKey}(\mathbf{pk})$ $(\cdot, X_v, \cdot,\cdot,\cdot,\cdot) \leftarrow VC_v$ $(\cdot,\cdot,\cdot,\mathbf{n},\cdot,\cdot,w_v) \leftarrow VP_v$ $(\alpha, \mathbf{r}) \leftarrow \mathsf{GenBallot}(X_v, \mathbf{s}, pk, \mathbf{n}, w_v)$		
	$\xrightarrow{v, \alpha}$	
		$(\cdot,\cdot,\cdot,\cdot,\mathbf{n},\mathbf{k},\mathbf{E},\cdot) \leftarrow EP$ **if** $\neg\mathsf{CheckBallot}(v, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}})$ **abort** $(\cdot, \mathbf{a}, \cdot) \leftarrow \alpha$ $(\beta_j, \delta_j) \leftarrow \mathsf{GenResponse}(v, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}_j)$ $VP_v \leftarrow \mathsf{GetVotingParameters}(v, EP)$ **if** $\mathsf{Contains}(B_j, v)$ **abort** $B_j \leftarrow B_j \,\|\, \langle(v,\alpha)\rangle$, $F_j \leftarrow F_j \,\|\, \langle(v,\delta_j)\rangle$
	$\xleftarrow{[\beta_j]_{VP_v}}$	
$\boldsymbol{\beta} \leftarrow (\beta_1, \dots, \beta_s)$ $\mathbf{P} \leftarrow \mathsf{GetPointMatrix}(\boldsymbol{\beta}, \mathbf{s}, \mathbf{r})$ $\mathbf{rc} \leftarrow \mathsf{GetVerificationCodes}(\mathbf{s}, \mathbf{P})$		

Protocol 7.5: Vote Casting

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* $\gamma = (\hat{y}_v, \hat{z}_v, \pi)$ consisting of the voter's public confirmation credential $\hat{y}_v$, vote validity credential $\hat{z}_v$, and a non-interactive zero-knowledge proof

$$\pi = NIZKP[(y,z) : \hat{y}_v = \hat{g}^y \bmod \hat{p} \;\wedge\; \hat{z}_v = \hat{g}^z \bmod \hat{p}].$$

In this way, the voting client proves knowledge of values y_v (derived from Y_v) and z_v (derived from $\mathbf{P}$). The motivation and details of this particular construction have been discussed in Section 6.4.6.

Voter $v \in \{1,\dots,N_E\}$		Voting Client		Election Authority $j \in \{1,\dots,s\}$
knows $VC_v, \mathbf{s}$		knows $v, \mathbf{s}, VP_v, pk, \mathbf{r}, \boldsymbol{\beta}, \mathbf{P}, \mathbf{rc}$		knows $EP, \hat{\mathbf{y}}, \hat{\mathbf{z}}, C_j, F_j$
	$\xleftarrow{\mathbf{rc}}$			
$(\cdot,\cdot,Y_v,\mathbf{rc}_v,\cdot,\cdot) \leftarrow VC_v$ **if** $\neg$CheckVerificationCodes$(\mathbf{rc}_v,\mathbf{rc},\mathbf{s})$ **abort**				
	$\xrightarrow{Y_v}$			
		$\gamma \leftarrow$ GenConfirmation$(Y_v, \mathbf{P})$		
			$\xrightarrow{v,\gamma}$	

				if $\neg$CheckConfirmation$(v,\gamma,\hat{\mathbf{y}},\hat{\mathbf{z}})$ **abort**
				$VP_v \leftarrow$ GetVotingParameters(v, EP)
				if $\neg$Contains(F_j, v) **abort**
				$\delta_j \leftarrow$ Search(F_j, v)
				if Contains(C_j, v) **abort**
				$C_j \leftarrow C_j \parallel \langle(v,\gamma)\rangle$

			$\xleftarrow{[\delta_j]_{VP_v}}$	
		$\boldsymbol{\delta} \leftarrow (\delta_1,\dots,\delta_s)$ $(\cdot,\cdot,\cdot,\mathbf{n},\mathbf{k},\mathbf{e}_v,\cdot) \leftarrow VP_v$ $FC \leftarrow$ GetFinalizationCode$(\boldsymbol{\beta},\boldsymbol{\delta},\mathbf{s},\mathbf{r},\mathbf{n},\mathbf{k},\mathbf{e}_v,pk)$		
	$\xleftarrow{FC}$			
$(\cdot,\cdot,\cdot,\cdot,FC_v,\cdot) \leftarrow VC_v$ **if** $\neg$CheckFinalizationCode(FC_v, FC) **abort**				

Protocol 7.6: Vote Confirmation

After submitting γ to every authority, they check the validity of the zero-knowledge proof included. In the success case, they respond with their *finalization* δ_j, which consists of the two randomizations from the OT response. These values have been stored in the *finalization list* F_j after responding to the OT query. The voting client uses $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_s)$ to retrieve the finalization code FC from the OT responses $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_s)$ by calling $\mathsf{GetFinalizationCode}(\boldsymbol{\beta}, \boldsymbol{\delta}, \mathbf{s}, \mathbf{r}, \mathbf{n}, \mathbf{k}, \mathbf{e}_v, pk)$. The voting client then display FC to the voter for comparison. As above, we describe this process by letting the human voter execute the algorithm $\mathsf{CheckFinalizationCode}(FC_v, FC)$. The whole process is depicted in Prot. 7.6.

7.3. Post-Election Phase

In the post-election phase, all $N_C \leqslant N_E$ 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 decryption key, which they have generated during the pre-election phase. Before applying their key shares to the output of the mixing, they verify the correctness of the shuffle proofs. In addition to performing the decryption, they demonstrate its correctness with a non-interactive zero-knowledge proof. The very last step of the entire election process is the final decryption step by the election administrator and the computation and announcement of the election result.

7.3.1. 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 first authority, the second mix by the second authority, and so on. The process is the same for everyone, except for the first authority, which starts from the list of encrypted votes extracted from the submitted ballots. Recall that during vote casting, each authority keeps track of all submitted ballots and confirmations. In case of the first authority, corresponding lists are denoted by B_1 and C_1, respectively. By calling $\mathsf{GetEncryptions}(B_1, C_1, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{w})$, the first authority retrieves the vector $\tilde{\mathbf{e}}_0$ of encrypted votes, and by calling $\mathsf{GenShuffle}(\tilde{\mathbf{e}}_0, pk)$, this vector is shuffled into $\tilde{\mathbf{e}}_1 = \mathsf{Shuffle}_{pk}(\tilde{\mathbf{e}}_0, \tilde{\mathbf{r}}_1, \psi_1)$, where $\tilde{\mathbf{r}}_1$ denotes the re-encryption randomizations and ψ_1 the permutation selected uniformly at random. These values are the secret inputs for generating a non-interactive proof

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

which proves the correctness of the first shuffle. This proof results from calling the algorithm $\mathsf{GenShuffleProof}(U, \tilde{\mathbf{e}}_0, \tilde{\mathbf{e}}_1, \tilde{\mathbf{r}}_1, \psi_1, pk)$. The result from conducting the first shuffle—the shuffled vector of encryptions $\tilde{\mathbf{e}}_1$ and the zero-knowledge proof $\tilde{\pi}_1$—is sent to every other election authority. Upon receiving $\tilde{\mathbf{e}}_1$ and $\tilde{\pi}_1$, $\mathsf{CheckShuffleProof}(U, \tilde{\pi}_1, \tilde{\mathbf{e}}_0, \tilde{\mathbf{e}}_1, pk)$ is called by all the others to verify the correctness of the first shuffle. As shown in Prot. 7.7, the process aborts if one or more than one check fails.

The same shuffling procedure is repeated s times in ascending index ordering, where the output $\tilde{\mathbf{e}}_{j-1}$ of the shuffle performed by authority $j-1$ becomes the input for the next shuffle $\tilde{\mathbf{e}}_j = \mathsf{Shuffle}_{pk}(\tilde{\mathbf{e}}_{j-1}, \tilde{\mathbf{r}}_j, \psi_j)$ 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 the vector $\tilde{\mathbf{e}}_s$ generated by authority s.

Election Authority $j \in \{1,\dots,s\}$		Election Authority $k \in \{1,\dots,s\}\backslash\{j\}$
knows j, EP, pk, B_j, C_j		
$(\cdot,\cdot,\cdot,\cdot,\mathbf{n},\mathbf{k},\mathbf{E},\mathbf{w}) \leftarrow EP$		
$\tilde{\mathbf{e}}_0 \leftarrow \mathsf{GetEncryptions}(B_j, C_j, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{w})$		
$(U,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot) \leftarrow EP$		
$(\tilde{\mathbf{e}}_j, \tilde{\mathbf{r}}_j, \psi_j) \leftarrow \mathsf{GenShuffle}(\tilde{\mathbf{e}}_{j-1}, pk)$		
$\tilde{\pi}_j \leftarrow \mathsf{GenShuffleProof}(U, \tilde{\mathbf{e}}_{j-1}, \tilde{\mathbf{e}}_j, \tilde{\mathbf{r}}_j, \psi_j, pk)$		
	$\xrightarrow{[\tilde{\mathbf{e}}_j, \tilde{\pi}_j]_{EP}}$	
	$\xleftarrow{[\tilde{\mathbf{e}}_k, \tilde{\pi}_k]_{EP}}$	
if $\neg\mathsf{CheckShuffleProof}(U, \tilde{\pi}_k, \tilde{\mathbf{e}}_{k-1}, \tilde{\mathbf{e}}_k, pk)$ **abort**		

Protocol 7.7: Mixing

7.3.2. Decryption

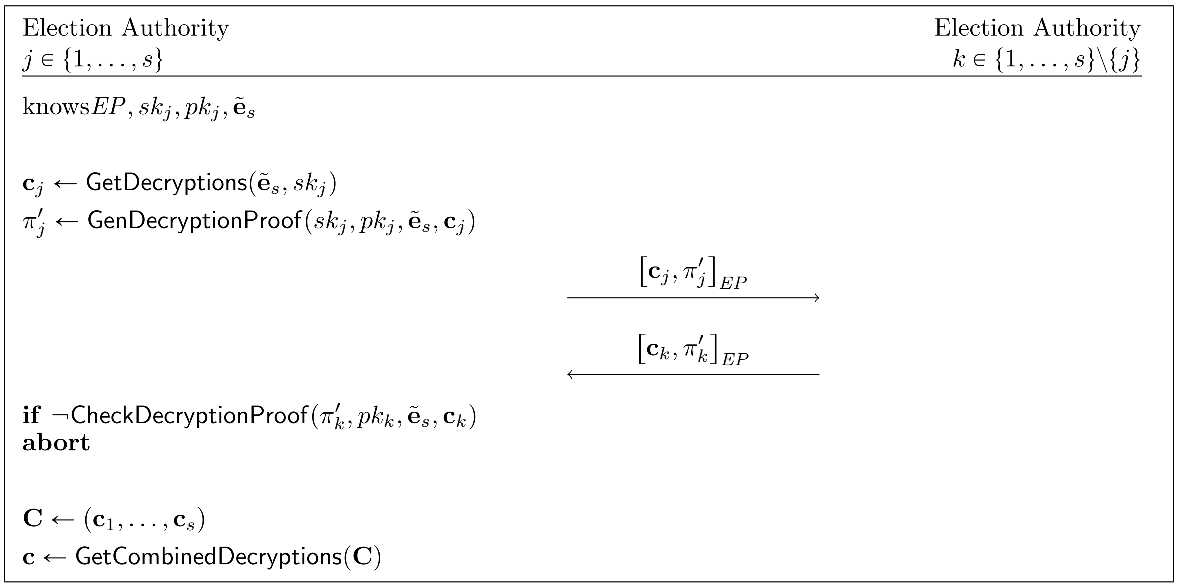
After successful mixing the list of encrypted votes, every authority knows the same mix-net output $\tilde{\mathbf{e}}_s = ((a_1, b_1), \dots (a_N, b_N))$. Each of them uses their share sk_j of the encryption private key to partially decrypt $\tilde{\mathbf{e}}_s$. Calling $\mathsf{GetDecryptions}(\tilde{\mathbf{e}}_s, sk_j)$ returns a vector $\mathbf{c}_j = (c_{1,j}, \dots, c_{N,j})$ of *partial decryptions* $c_{ij} = b_i^{sk_j}$. To guarantee the correctness of the decryption, a non-interactive decryption proof

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

is computed by calling $\mathsf{GenDecryptionProof}(sk_j, pk_j, \tilde{\mathbf{e}}_s, \mathbf{c}_j)$. Note that this is a proof of equality of multiple discrete logarithms (see Section 5.4.1). At the end of this process, the partial decryptions and the decryption proofs are sent to the other election authorities, which mutually check the validity of all proofs. The process aborts if one or more than one check fails. In the success case, all partial decryptions are combined by calling $\mathsf{GetCombinedDecryptions}(\mathbf{C})$, where $\mathbf{C} = (c_{ij})$ denotes the matrix of all $N \times s$ partial decryptions. The resulting vector $\mathbf{c}$ is the same for every election authority.

7.3.3. Final Decryption and Tallying

Before starting the tallying process, the administrator performs the last decryption step using its share sk_0 of the private decryption key. For this, the administrator is supposed to receive from every authority the same list of encryptions $\tilde{\mathbf{e}}_s$ and the same list of combined



Election Authority $j \in \{1,\dots,s\}$ | Election Authority $k \in \{1,\dots,s\}\backslash\{j\}$

knows$EP, sk_j, pk_j, \tilde{\mathbf{e}}_s$

$\mathbf{c}_j \leftarrow \mathsf{GetDecryptions}(\tilde{\mathbf{e}}_s, sk_j)$
$\pi'_j \leftarrow \mathsf{GenDecryptionProof}(sk_j, pk_j, \tilde{\mathbf{e}}_s, \mathbf{c}_j)$

$\xrightarrow{[\mathbf{c}_j, \pi'_j]_{EP}}$

$\xleftarrow{[\mathbf{c}_k, \pi'_k]_{EP}}$

if $\neg\mathsf{CheckDecryptionProof}(\pi'_k, pk_k, \tilde{\mathbf{e}}_s, \mathbf{c}_k)$
abort

$\mathbf{C} \leftarrow (\mathbf{c}_1, \dots, \mathbf{c}_s)$
$\mathbf{c} \leftarrow \mathsf{GetCombinedDecryptions}(\mathbf{C})$

Protocol 7.8: Decryption

partial decryptions $\mathbf{c}$. If the messages from all authorities are identical, it means for the administrator that the authorities agree about everything. If this is not the case, the last decryption step is aborted and the tallying procedure is stopped.

Otherwise, the administrator concludes the election process by computing the election result and presenting it to the general public. By calling $\mathsf{GetVotes}(\tilde{\mathbf{e}}_s, \mathbf{c}, \mathbf{c}')$, the partial decryptions are assembled and the plaintext votes are determined. Recall from Sections 6.4.2 and 6.4.3 that every such plaintext vote represents some voter's selection of candidates and the voter's counting circle or a default vote to be subtracted from the tally. The votes for individual candidates and the counting circle can be retrieved by factorizing this number. By calling $\mathsf{GetElectionResult}(\mathbf{m}, \mathbf{n}, \mathbf{w})$, this process is performed for all plaintext votes and the election result ER is published. The whole tallying process is depicted in Prot. 7.9.

7.3.4. Inspection

To offer abstaining voters the possibility to check that their voting right has not been abused by someone else (see discussion at the bottom of Section 2.1), the election authorities publish their shares of the abstention codes of all abstaining voters together with the shares of the finalization codes of all participating voters. This can be done immediately after closing the voting period or after conducting the tallying. For this, they call $\mathsf{GetInspection}(v, \mathbf{P}_j, \mathbf{a}_j, C_j)$ to determine their share $I_j \in \{A_j, F_j\}$ of some voter's abstention or finalization code. From the resulting vector $\mathbf{i} = (I_1, \dots, I_s)$, the *inspection code* $IC_v \in A_{FA}^{\ell_{FA}}$ can be computed by the voting client.

From the perspective of the election authority, this is the last protocol step. It can be seen as a confirmation that the whole protocol run was a success, at least from the authority's point of view. Both abstaining and participating voters can then simply check whether their abstention or finalization code printed on the code sheet corresponds to the code displayed by the voting client. We call this supplementary protocol step *inspection phase.*

Election Authority $j \in \{1, \dots, s\}$		Administrator	General Public
knows $EP, \mathbf{c}, \tilde{\mathbf{e}}_s$		knows EP, sk_0, pk_0	
	$\xrightarrow{\left[\mathbf{c}^{(j)}, \tilde{\mathbf{e}}_s^{(j)}\right]_{EP}}$		
		if $\neg\left[\mathbf{c}^{(1)} = \cdots = \mathbf{c}^{(s)} \wedge \tilde{\mathbf{e}}_s^{(1)} = \cdots = \tilde{\mathbf{e}}_s^{(s)}\right]$ **abort**	
		$\mathbf{c}_0 \leftarrow \mathsf{GetDecryptions}(\tilde{\mathbf{e}}_s, sk_0)$	
		$\pi'_0 \leftarrow \mathsf{GenDecryptionProof}(sk_0, pk_0, \tilde{\mathbf{e}}_s, \mathbf{c}_0)$	
		$\mathbf{m} \leftarrow \mathsf{GetVotes}(\tilde{\mathbf{e}}_s, \mathbf{c}, \mathbf{c}_0)$	
		$(\cdot, \cdot, \cdot, \cdot, \mathbf{n}, \cdot, \cdot, \mathbf{w}) \leftarrow EP$	
		$ER \leftarrow \mathsf{GetElectionResult}(\mathbf{m}, \mathbf{n}, \mathbf{w})$	
		$\xrightarrow{[ER]_{\varnothing}}$	

Protocol 7.9: Tallying

Note that for the security of the protocol, it is not mandatory that voters actually conduct the inspection. In Prot. 7.10, the *participation bit* $pb \in \mathbb{B}$ indicates whether the voter has successfully submitted a vote during the election period.

Voter $v \in \{1,\dots,N_E\}$		Voting Client		Election Authority $j \in \{1,\dots,s\}$
knows VC_v, pb				knows $\mathbf{P}_j, \mathbf{a}_j, C_j$
$(v,\cdot,\cdot,\cdot,\cdot,\cdot) \leftarrow VC_v$				
	$\xrightarrow{v}$			
			$\xrightarrow{v}$	
				$I_j \leftarrow \mathsf{GetInspection}(v, \mathbf{P}_j, \mathbf{a}_j, C_j)$
			$\xleftarrow{[I_j]_\varnothing}$	
		$\mathbf{i} \leftarrow (I_1,\dots,I_s)$ $IC \leftarrow \mathsf{GetInspectionCode}(\mathbf{i})$		
	$\xleftarrow{IC}$			
$(\cdot,\cdot,\cdot,\cdot,FC_v,AC_v) \leftarrow VC_v$ **if** $\neg\mathsf{CheckInspectionCode}(pb, FC_v, AC_v, IC)$ **abort**				

Protocol 7.10: Inspection

7.4. Channel Security

In Section 6.1, we have already identified the channels that need to be secured by cryptographic means. Most importantly, we require every messages m sent by either the administrator or the election authorities to be digitally signed. In the protocol diagrams of Chapter 7, signed messages are denoted by $[m]$. To generate $[m]$, we assume each of these parties to possess a Schnorr signature key pair (sk_X, pk_X) and a certificate C_X that binds the public key pk_X to party $X \in \{\mathsf{Admin}, \mathsf{Auth}_1, \dots, \mathsf{Auth}_s\}$. We assume that checking the validity of certificates is part of checking a signature, but 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, corresponding software libraries, and best practices.

Table 7.2 gives an overview of all signatures generated during the protocol execution. Generally, for generating a signature for an arbitrary message m, we make an algorithm call to $\sigma \leftarrow \mathsf{GenSignature}(sk_X, m, aux)$ using the party's private signature key sk_X. This algorithm implements Schnorr's signature scheme as described in Section 5.6 and as implemented in Alg. 8.58. For signing $r \geqslant 0$ messages simultaneously, we generate the signature for the corresponding tuple $m = (m_1, \dots, m_r)$.

The first part of the additional algorithm parameter aux is used to link the transmitted message to the current election context. Creating such links is a common countermeasure against all sorts of replay attacks. Messages sent by the election authorities are either linked to EP or VP_v, depending on which information is available to the receiver. These links guarantee that all protocol participants have a common view of the current election event.

Sender	Nr.	Protocol	Receiver	Message m_i	aux
Administrator	7.1	Key Generation	EA j	EP, pk_0, π_0	`"MAE1"`
	7.3	Printing	PA	EP	`"MAP1"`
	7.4	Candidate Selection	VC v	VP_v, pk_0, π_0	`"MAC1"`
	7.9	Tallying	Public	ER	`"MAT1"`
Election authority $j \in \{1, \ldots, s\}$	7.1	Key generation	EA k	pk_j, π_j	EP, `"MEE1"`
	7.2	Election preparation	EA k	$\hat{\mathbf{x}}_j, \hat{\mathbf{y}}_j, \hat{\mathbf{z}}_j, \hat{\pi}_j$	EP, `"MEE2"`
	7.3	Printing	PA	$[\![\mathbf{d}_j]\!]$	EP, `"MEP1"`
	7.5	Vote casting	VC v	pk_j, π_j	VP_v, `"MEC1"`
			VC v	β_j	VP_v, `"MEC2"`
	7.6	Vote confirmation	VC v	δ_j	VP_v, `"MEC3"`
	7.7	Mixing	EA k	$\tilde{\mathbf{e}}_j, \tilde{\pi}_j$	EP, `"MEE3"`
	7.8	Decryption	EA k	$\mathbf{c}_j, \pi'_j$	EP, `"MEE4"`
	7.9	Tallying	Admin	$\mathbf{c}, \tilde{\mathbf{e}}_s$	EP, `"MEA1"`
	7.10	Inspection	VC v	I_j	`"MEC4"`

Table 7.2.: Overview of the signatures generated during the protocol execution. In the column "Receiver", EA stands for *election authority*, PA for *printing authority*, VC for *voting client*, and Admin for *administrator*.

They also prevent that transferring election data across multiple election events remains undetected. This solves the problem encountered in [17, Section 10.9], which exploits the chosen-ciphertext attack from [38, Section 12.9].

The second part of the additional parameter aux links the transmitted message to its particular type and role within the protocol. We use strings such as `"MAE1"` for that purpose, which can be interpreted as "Message 1 from Administrator to Election Authority". This clarifies the sender's intention with respect to a given message sent during the protocol execution of a given election event. Again, the purpose of this measure is to prevent replay attacks, which are possible whenever two messages from the same sender are structurally identical. Although such cases do not exist in the current version of the protocol, we introduce this measure as a precaution for future protocol versions.

A special case in the list of signatures shown in Table 7.2 is the election authority's entry for Prot. 7.3. Recall that the voting card data $\mathbf{d}_j$ 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 the key-encapsulation mechanism from Section 5.7. We follow the *encrypt-then-sign* approach as described in [38], in which the hybrid ciphertext $c_j = \mathsf{Enc}_{pk_{\mathsf{Print}}}(D_j)$ is signed instead of $\mathbf{d}_j$ itself. For creating such a ciphertext, we assume that a byte array representation $D_j \in \mathcal{B}^*$ of $\mathbf{d}_j$ is available, i.e., the ciphertext is obtained from calling an algorithm

$$c_j \leftarrow \mathsf{GenCiphertext}(pk_{\mathsf{Print}}, D_j),$$

where 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. Upon receiving this message, the printing authority decrypts c_j into

$$D_j \leftarrow \mathsf{GetPlaintext}(sk_{\mathsf{Print}}, c_j)$$

using the private key sk_{Print}, from which $\mathbf{d}_j$ can be derived deterministically. The generation and verification of the signature attached to this message is identical to all other signatures, except that the signature and verification algorithms are applied to c_j instead of $\mathbf{d}_j$. For the hybrid encryption scheme, we use the same group parameters $\hat{p}$, $\hat{q}$, and $\hat{g}$ as for the Schnorr signatures, which again are the same for the voting and confirmation credentials. The printing authority's encryption key pair $(sk_{\mathsf{Print}}, pk_{\mathsf{Print}})$ is therefore an element of $\mathbb{Z}_{\hat{q}} \times \mathbb{G}_{\hat{q}}$.

8. 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 8.1, we start with some general algorithms for specific tasks, which are needed at multiple places. In Sections 8.2 to 8.4, we specify the algorithms of the respective protocol phases.

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 (PRG) and access to a high-entropy seed. We require such a PRG in Section 4.3 for generating random byte arrays $R \in_R \mathcal{B}^L$ of length L, from which random values $r \in_R \mathbb{Z}_q$, $r \in_R \mathbb{G}_q$, $r \in_R \mathbb{Z}_{\hat{q}}$, or $r \in_R [a, b]$ can be derived. Since implementing a PRG is a difficult problem on its own, it cannot be addressed in this document.

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, $\hat{p}$, or $\hat{q}$. 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 required 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.

8.1. 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 8.1 we give an overview.

Nr.	Algorithm	Called by	Protocol
8.1	$\mathsf{GetPrimes}(n)$	Algs. 8.21, 8.27, 8.40, 8.54 and 9.8	7.5, 7.7, 7.8, 7.9
8.2	$\hookrightarrow$ $\mathsf{IsPrime}(x)$		
8.3	$\mathsf{GetGenerators}(n, U)$	Algs. 8.45 and 8.48	7.7, 7.8
8.4	$\mathsf{GetChallenge}(y, t)$	Algs. 8.7, 8.8, 8.15, 8.16, 8.24, 8.26, 8.34, 8.36, 8.45, 8.48, 8.50, 8.51, 9.4, 9.5, 9.13 and 9.16	7.1, 7.2, 7.5, 7.6, 7.7, 7.8, 7.9
8.5	$\mathsf{GetChallenges}(n, y)$	Algs. 8.45 and 8.48	7.7, 7.8

Table 8.1.: Overview of general algorithms for specific tasks.

Other general algorithms have been introduced in the Chapter 4 for converting integers, strings, and byte arrays, for hash value computations, and for picking numbers uniformly at random. We do not repeat them here. There are two algorithms in total, for which we give no explicit pseudo-code: $\mathsf{Sort}_{\preceq}(\mathbf{s})$ for sorting a vector $\mathbf{s}$ according to some total order $\preceq$ 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.

Standard implementations for efficient sorting algorithms 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 arithmetics, provides an implementation of the Kronecker symbol, which includes the Jacobi symbol as a special case. If no off-the-shelf implementation is available, we refer to existing pseudo-code algorithms such as given in [3, pp. 76–77], [48, p. 73], or [11, pp.1̇13].

[1]See https://gmplib.org

```
**Algorithm:** GetPrimes($n$)
**Input:** Number of primes $n \in \mathbb{N}$
$x \leftarrow 1$
for $i = 0, \ldots, n$ do
    repeat
        if $x = 1$ or $x = 2$ then
            $x \leftarrow x + 1$
        else
            $x \leftarrow x + 2$
        if $x \geqslant p$ then
            return $\perp$                                        // $n$ is incompatible with $p$
    until IsPrime($x$) and JacobiSymbol($x, p$) $= 1$             // see Alg. 8.2
    $p_i \leftarrow x$
$\mathbf{p} \leftarrow (p_0, \ldots, p_n)$
return $\mathbf{p}$                                               // $\mathbf{p} \in (\mathbb{G}_q \cap \mathbb{P})^n$
```

Algorithm 8.1: Computes the first $n{+}1$ prime numbers from $\mathbb{G}_q \subset \mathbb{Z}_p^*$. The computation possibly fails if n is too large or p is too small. In a more efficient implementation of this algorithm, the vector of resulting primes is accumulated in a cache or precomputed for the largest expected value.

```
**Algorithm:** IsPrime($x$)
**Input:** Number to test $x \in \mathbb{N}$
if $x \leqslant 1$ then
    return *false*
if $x \leqslant 3$ then
    return *true*
if $2 \mid x$ or $3 \mid x$ then
    return *false*
$i \leftarrow 5$
while $i^2 \leqslant x$ do
    if $i \mid x$ or $(i + 2) \mid x$ then
        return *false*
    $i \leftarrow i + 6$
return *true*
```

Algorithm 8.2: Checks if a positive integer $x \in \mathbb{N}$ is prime. The algorithm is deterministic and known as the "Optimized School Method". The worst-case running time of the algorithm is $O(\sqrt{x})$, i.e., only relatively small integers can be checked efficiently.

```
**Algorithm:** GetGenerators$(n, U)$
**Input:** Number of independent geneators $n \in \mathbb{N}$
Election event identifier $U \in A^*_{\mathsf{ucs}}$
for $i = 1, \dots, n$ do
    $x \leftarrow 0$
    repeat
        $x \leftarrow x + 1$
        $h_i \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecHash}_L(U, \texttt{"ggen"}, i, x)) \bmod p$    // see Algs. 4.5 and 4.15
        $h_i \leftarrow h_i^2 \bmod p$
    until $h_i \notin \{0, 1\}$    // these cases are very unlikely
$\mathbf{h} \leftarrow (h_1, \dots, h_n)$
return $\mathbf{h}$    // $\mathbf{h} \in (\mathbb{G}_q \backslash \{1\})^n$
```

Algorithm 8.3: 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 [3, Appendix A.2.3]. Making the generators dependent on U guarantees that the resulting values are specific to the current election. 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.

```
**Algorithm:** GetChallenge$(y, t)$
**Input:** Public value $y \in Y$, $Y$ unspecified
           Commitment $t \in T$, $T$ unspecified
$c \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecHash}_L(y, t)) \bmod 2^\tau$    // see Algs. 4.5 and 4.15
return $c$    // $c \in \mathbb{Z}_{2^\tau}$
```

Algorithm 8.4: Computes a NIZKP challenge $0 \leqslant c < 2^\tau$ for a given public value y and a public commitment t. The domains Y and T of the input values are unspecified.

```
**Algorithm:** GetChallenges$(n, y)$
**Input:** Number of challenges $n \in \mathbb{N}$
           Public value $y \in Y$, $Y$ unspecified
$H \leftarrow \mathsf{RecHash}_L(y)$    // see Alg. 4.15
for $i = 1, \dots, n$ do
    $I \leftarrow \mathsf{RecHash}_L(i)$    // see Alg. 4.15
    $c_i \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{RecHash}_L(H, I)) \bmod 2^\tau$    // see Alg. 4.5
$\mathbf{c} \leftarrow (c_1, \dots, c_n)$
return $\mathbf{c}$    // $\mathbf{c} \in \mathbb{Z}^n_{2^\tau}$
```

Algorithm 8.5: Computes n challenges $0 \leqslant c_i < 2^\tau$ for a given of public value y. The domain Y of the input value is unspecified. Note that the resulting challenges are identical to $c_i = \mathsf{ByteArrayToInteger}(\mathsf{RecHash}_L(y, i)) \bmod 2^\tau$, but precomputing H makes the algorithm more efficient, especially if y is a complex mathematical object.

8.2. Pre-Election Phase

The main actors in the pre-election phase are the election authorities. For the given election parameters *EP* defining the values $\mathbf{n}$, $\mathbf{k}$, and $\mathbf{E}$, each election authority generates a share of the electorate data by calling Alg. 8.10. This is the main algorithm of the election preparation, which invokes several sub-algorithms for more specific tasks. Table 8.2 gives an overview of all algorithms of the pre-election phase. The shares of the public credentials of every authority are assembled by every election authority using Alg. 8.17. The shares of the voting card data, which are sent to the printing authority over a confidential channel, are assembled to create the voting cards by calling Alg. 8.18. Some sub-tasks for creating a single voting card are delegated to Alg. 8.19. Some other algorithms are required for generating shares of the encryption key pair and for assembling the resulting shares of the public key. For a more detailed description of the pre-election phase, we refer to Section 7.1.

Nr.	Algorithm	Called by	Protocol
8.6	$\mathsf{GenKeyPair}()$	Administrator,	7.1
8.7	$\mathsf{GenKeyPairProof}(sk, pk)$	election authority	
8.8	$\mathsf{CheckKeyPairProof}(\pi, pk)$	Election authority	
8.9	$\mathsf{GetPublicKey}(\mathbf{pk})$		
8.10	$\mathsf{GenElectorateData}(\mathbf{n}, \mathbf{k}, \mathbf{E}, s)$	Election authority	7.2
8.11	↳ $\mathsf{GenPoints}(\mathbf{n}, \mathbf{k}, \mathbf{e})$		
8.12	↳ $\mathsf{GenPolynomial}(d)$		
8.13	↳ $\mathsf{GetYValue}(x, \mathbf{a})$		
8.14	↳ $\mathsf{GenCredentials}(z, s)$		
8.15	$\mathsf{GenCredentialProof}(\mathbf{x}, \mathbf{y}, \mathbf{z}, \hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}})$		
8.16	$\mathsf{CheckCredentialProof}(\pi, \hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}})$		
8.17	$\mathsf{GetPublicCredentials}(\hat{\mathbf{X}}, \hat{\mathbf{Y}}, \hat{\mathbf{Z}})$		
8.18	$\mathsf{GetVotingCards}(\mathbf{D})$	Printing authority	7.3
8.19	↳ $\mathsf{GetVotingCard}(v, \mathbf{d})$		

Table 8.2.: Overview of algorithms and sub-algorithms of the pre-election phase.

```
Algorithm: GenKeyPair()
$sk \leftarrow \mathsf{GenRandomInteger}(q)$                    // see Alg. 4.11
$pk \leftarrow g^{sk} \bmod p$
return $(sk, pk)$                                            // $(sk, pk) \in \mathbb{Z}_q \times \mathbb{G}_q$
```

Algorithm 8.6: Generates a random ElGamal encryption key pair $(sk, pk) \in \mathbb{Z}_q \times \mathbb{G}_q$ or shares of such a key pair. This algorithm is used in Prot. 7.1 by the authorities to generate private shares of a common public encryption key.

```
Algorithm: GenKeyPairProof($sk, pk$)
Input: Private decryption key $sk \in \mathbb{Z}_q$
       Public encryption key $pk \in \mathbb{G}_q$
$\omega \leftarrow$ GenRandomInteger($q$)                    // see Alg. 4.11
$t \leftarrow g^{\omega} \bmod p$
$c \leftarrow$ GetChallenge($pk, t$)                           // see Alg. 8.4
$s \leftarrow \omega - c \cdot sk \bmod q$
$\pi \leftarrow (c, s)$
return $\pi$                                               // $\pi \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_q$
```

Algorithm 8.7: Generates a key pair proof, i.e., a proof of knowing the private decryption key sk satisfying $pk = g^{sk}$. For the proof verification, see Alg. 8.8.

```
Algorithm: CheckKeyPairProof($\pi, pk$)
Input: Key pair proof $\pi = (c, s) \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_q$
       Encryption key share $pk \in \mathbb{G}_q$
$t \leftarrow pk^c \cdot g^s \bmod p$
$c' \leftarrow$ GetChallenge($pk, t$)                          // see Alg. 8.4
return $c = c'$
```

Algorithm 8.8: Checks the correctness of a key pair proof π generated by Alg. 8.7.

```
Algorithm: GetPublicKey($\mathbf{pk}$)
Input: Public keys $\mathbf{pk} = (pk_0, pk_1, \ldots, pk_s) \in \mathbb{G}_q^{s+1}$
$pk \leftarrow \prod_{j=0}^{s} pk_j \bmod p$
return $pk$                                                // $pk \in \mathbb{G}_q$
```

Algorithm 8.9: Computes a public ElGamal encryption key $pk \in \mathbb{G}_q$ from given shares $pk_j \in \mathbb{G}_q$.

```
Algorithm: $\mathsf{GenElectorateData}(\mathbf{n}, \mathbf{k}, \mathbf{E}, s)$
Input: Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$
       Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Eligibility matrix $\mathbf{E} \in \mathbb{B}^{N_E \times t}$
       Number of election authorities $s \in \mathbb{N}^+$
$n \leftarrow \sum_{j=1}^{t} n_j$
for $i = 1, \ldots, N_E$ do
    $\mathbf{e}_i \leftarrow \mathsf{GetRow}(\mathbf{E}, i)$
    $(\mathbf{p}_i, z_i) \leftarrow \mathsf{GenPoints}(\mathbf{n}, \mathbf{k}, \mathbf{e}_i)$                // see Alg. 8.11
    $(x_i, y_i, \hat{x}_i, \hat{y}_i, \hat{z}_i) \leftarrow \mathsf{GenCredentials}(z_i, s)$          // see Alg. 8.14
    $A_i \leftarrow \mathsf{RandomBytes}(L_{FA})$
    $d_i \leftarrow (x_i, y_i, A_i, \mathbf{p}_i)$
$\mathbf{a} \leftarrow (A_1, \ldots, A_{N_E})$, $\mathbf{d} \leftarrow (d_1, \ldots, d_{N_E})$
$\mathbf{x} \leftarrow (x_1, \ldots, x_{N_E})$, $\mathbf{y} \leftarrow (y_1, \ldots, y_{N_E})$, $\mathbf{z} \leftarrow (z_1, \ldots, z_{N_E})$
$\hat{\mathbf{x}} \leftarrow (\hat{x}_1, \ldots, \hat{x}_{N_E})$, $\hat{\mathbf{y}} \leftarrow (\hat{y}_1, \ldots, \hat{y}_{N_E})$, $\hat{\mathbf{z}} \leftarrow (\hat{z}_1, \ldots, \hat{z}_{N_E})$
$\mathbf{P} \leftarrow (\mathbf{p}_1, \ldots, \mathbf{p}_{N_E})$
return $(\mathbf{a}, \mathbf{d}, \mathbf{x}, \mathbf{y}, \mathbf{z}, \hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}, \mathbf{P})$     // $\mathbf{a} \in (\mathcal{B}^{L_{FA}})^{N_E}$, $\mathbf{d} \in (\mathbb{Z}_{\hat{q}_x} \times \mathbb{Z}_{\hat{q}_y} \times \mathcal{B}^{L_{FA}} \times (\mathbb{Z}_{\hat{q}}^2)^n)^{N_E}$,
                                          // $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{Z}_{\hat{q}_x}^{N_E}$, $\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}} \in \mathbb{G}_{\hat{q}}^{N_E}$, $\mathbf{P} \in (\mathbb{Z}_{\hat{q}}^2)^{N_E \times n}$
```

Algorithm 8.10: Generates the authority's share of the voting card data and credentials for the whole electorate. For this, Algs. 8.11 and 8.14 are called to generate the voting card data and the credentials for each single voter. The responses of these calls are then grouped into corresponding tuples and matrices.

```
Algorithm: GenPoints($\mathbf{n}, \mathbf{k}, \mathbf{e}$)
Input: Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$, $n = \sum_{j=1}^{t} n_j$
       Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Eligibility vector $\mathbf{e} = (e_1, \ldots, e_t) \in \mathbb{B}^t$
$k \leftarrow \sum_{j=1}^{t} e_j k_j$
$\mathbf{a} \leftarrow$ GenPolynomial($k-1$)                                  // see Alg. 8.12
$X \leftarrow \{0\}$                                                  // set of $x$-values to be excluded
$n' \leftarrow 0$
for $j = 1, \ldots, t$ do
    for $i = n'+1, \ldots, n'+n_j$ do                                  // loop for $i = 1, \ldots, n$
        if $e_j = 1$ then
            $x \leftarrow$ GenRandomInteger($\hat{q}, X$)                  // see Alg. 4.12
            $X \leftarrow X \cup \{x\}$                            // avoid picking the same $x$-value twice
            $y \leftarrow$ GetYValue($x, \mathbf{a}$)                        // see Alg. 8.13
            $p_i \leftarrow (x, y)$
        else
            $p_i \leftarrow (0, 0)$                                // default point for non-eligibility
    $n' \leftarrow n' + n_j$
$y_0 \leftarrow$ GetYValue($0, \mathbf{a}$)                                   // see Alg. 8.13
$\mathbf{p} \leftarrow (p_1, \ldots, p_n)$
return $(\mathbf{p}, y_0)$                                          // $\mathbf{p} \in (\mathbb{Z}_{\hat{q}}^2)^n$, $y_0 \in \mathbb{Z}_{\hat{q}}$
```

Algorithm 8.11: Generates a list of random points picked from a random polynomial $A(X) \in_R \mathbb{Z}_{\hat{q}}[X]$ of degree $k-1$. The random polynomial is obtained from calling Alg. 8.12. Additionally, using Alg. 8.13, the value $y_0 = A(0)$ 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
        $a_i \leftarrow$ GenRandomInteger($\hat{q}$)                           // see Alg. 4.11
    $a_d \leftarrow$ GenRandomInteger($\hat{q}, \{0\}$)                          // see Alg. 4.12
    $\mathbf{a} \leftarrow (a_0, \ldots, a_d)$
return $\mathbf{a}$                                          // $\mathbf{a} \in \mathbb{Z}_{\hat{q}}^{d'}$, $d' = \max(1, d+1)$
```

Algorithm 8.12: Generates the coefficients $a_0, \ldots, a_d$ of a random polynomial $A(X) = \sum_{i=0}^{d} a_i X^i \bmod \hat{q}$ 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 vector $\mathbf{a} = (0)$.

```
Algorithm: GetYValue($x, \mathbf{a}$)
Input: Value $x \in \mathbb{Z}_{\hat{q}}$
       Coefficients $\mathbf{a} = (a_0, \ldots, a_d) \in \mathbb{Z}_{\hat{q}}^{d+1}$, $d \geqslant 0$
if $x = 0$ then
    $y \leftarrow a_0$
else
    $y \leftarrow 0$
    for $i = d, \ldots, 0$ do
        $y \leftarrow a_i + x \cdot y \bmod \hat{q}$
return $y$                                                    // $y \in \mathbb{Z}_{\hat{q}}$
```

Algorithm 8.13: Computes the value $y = A(x) \in \mathbb{Z}_{\hat{q}}$ obtained from evaluating the polynomial $A(X) = \sum_{i=0}^{d} a_i X^i \bmod \hat{q}$ at position x. The algorithm is an implementation of Horner's method.

```
Algorithm: GenCredentials($z, s$)
Input: Vote validitiy credential $z \in \mathbb{Z}_{\hat{q}}$
       Number of election authorities $s \in \mathbb{N}^+$
$x \leftarrow$ GenRandomInteger($\lfloor \hat{q}_x / s \rfloor$)                  // see Alg. 4.11
$y \leftarrow$ GenRandomInteger($\lfloor \hat{q}_y / s \rfloor$)                  // see Alg. 4.11
$\hat{x} \leftarrow \hat{g}^x \bmod \hat{p}$
$\hat{y} \leftarrow \hat{g}^y \bmod \hat{p}$
$\hat{z} \leftarrow \hat{g}^z \bmod \hat{p}$
return $(x, y, \hat{x}, \hat{y}, \hat{z})$                    // $x, y \in \mathbb{Z}_{\hat{q}}$, $\hat{x}, \hat{y}, \hat{z} \in \mathbb{G}_{\hat{q}}$
```

Algorithm 8.14: Generates an authority's shares of the voting, confirmation, and vote validity credentials for a single voter.

```
Algorithm: GenCredentialProof($\mathbf{x}, \mathbf{y}, \mathbf{z}, \hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}$)
Input: Private voting credentials $\mathbf{x} = (x_1, \ldots, x_{N_E}) \in \mathbb{Z}_{\hat{q}}^{N_E}$
       Private confirmation credentials $\mathbf{y} = (y_1, \ldots, y_{N_E}) \in \mathbb{Z}_{\hat{q}}^{N_E}$
       Private vote validity credentials $\mathbf{z} = (z_1, \ldots, z_{N_E}) \in \mathbb{Z}_{\hat{q}}^{N_E}$
       Public voting credentials $\hat{\mathbf{x}} = (\hat{x}_1, \ldots, \hat{x}_{N_E}) \in \mathbb{G}_{\hat{q}}^{N_E}$
       Public confirmation credentials $\hat{\mathbf{y}} = (\hat{y}_1, \ldots, \hat{y}_{N_E}) \in \mathbb{G}_{\hat{q}}^{N_E}$
       Public vote validity credentials $\hat{\mathbf{z}} = (\hat{z}_1, \ldots, \hat{z}_{N_E}) \in \mathbb{G}_{\hat{q}}^{N_E}$
for $i = 1, \ldots, N_E$ do
    $\omega_{i,1} \leftarrow$ GenRandomInteger($\hat{q}$)                          // see Alg. 4.11
    $\omega_{i,2} \leftarrow$ GenRandomInteger($\hat{q}$)                          // see Alg. 4.11
    $\omega_{i,3} \leftarrow$ GenRandomInteger($\hat{q}$)                          // see Alg. 4.11
    $t_{i,1} \leftarrow \hat{g}^{\omega_{i,1}} \bmod \hat{p}$, $t_{i,2} \leftarrow \hat{g}^{\omega_{i,2}} \bmod \hat{p}$, $t_{i,3} \leftarrow \hat{g}^{\omega_{i,3}} \bmod \hat{p}$
    $t_i \leftarrow (t_{i,1}, t_{i,2}, t_{i,3})$
$\mathbf{t} \leftarrow (t_1, \ldots, t_{N_E})$
$y \leftarrow (\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}})$
$c \leftarrow$ GetChallenge($y, \mathbf{t}$)                                      // see Alg. 8.4
for $i = 1, \ldots, N_E$ do
    $s_{i,1} \leftarrow \omega_{i,1} - c \cdot x_i \bmod \hat{q}$, $s_{i,2} \leftarrow \omega_{i,2} - c \cdot y_i \bmod \hat{q}$, $s_{i,3} \leftarrow \omega_{i,3} - c \cdot z_i \bmod \hat{q}$
    $s_i \leftarrow (s_{i,1}, s_{i,2}, s_{i,3})$
$\mathbf{s} \leftarrow (s_1, \ldots, s_{N_E})$
$\pi \leftarrow (c, \mathbf{s})$
return $\pi$                                                      // $\pi \in \mathbb{Z}_{2^\tau} \times (\mathbb{Z}_q^3)^{N_E}$
```

Algorithm 8.15: Generates a proof of knowing all private credentials **x**, **y**, and **z**. For the proof verification, see Alg. 8.16.

```
Algorithm: CheckCredentialProof($\pi, \hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}$)
Input: Credential proof $\pi = (c, \mathbf{s}) \in \mathbb{Z}_{2^\tau} \times (\mathbb{Z}_q^3)^{N_E}$, $\mathbf{s} = (s_1, \ldots, s_{N_E})$, $s_i = (s_{i,1}, s_{i,2}, s_{i,3})$
       Public voting credentials $\hat{\mathbf{x}} = (\hat{x}_1, \ldots, \hat{x}_{N_E}) \in \mathbb{G}_{\hat{q}}^{N_E}$
       Public confirmation credentials $\hat{\mathbf{y}} = (\hat{y}_1, \ldots, \hat{y}_{N_E}) \in \mathbb{G}_{\hat{q}}^{N_E}$
       Public vote validity credentials $\hat{\mathbf{z}} = (\hat{z}_1, \ldots, \hat{z}_{N_E}) \in \mathbb{G}_{\hat{q}}^{N_E}$
for $i = 1, \ldots, N_E$ do
    $t_{i,1} \leftarrow \hat{x}_i^c \cdot \hat{g}^{s_{i,1}} \bmod \hat{p}$, $t_{i,2} \leftarrow \hat{y}_i^c \cdot \hat{g}^{s_{i,2}} \bmod \hat{p}$, $t_{i,3} \leftarrow \hat{z}_i^c \cdot \hat{g}^{s_{i,3}} \bmod \hat{p}$
    $t_i \leftarrow (t_{i,1}, t_{i,2}, t_{i,3})$
$\mathbf{t} \leftarrow (t_1, \ldots, t_{N_E})$
$y \leftarrow (\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}})$
$c' \leftarrow$ GetChallenge($y, \mathbf{t}$)                                     // see Alg. 8.4
return $c = c'$
```

Algorithm 8.16: Checks the correctness of a credential proof π generated by Alg. 8.15.

```
Algorithm: GetPublicCredentials($\hat{\mathbf{X}}, \hat{\mathbf{Y}}, \hat{\mathbf{Z}}$)
Input: Public voting credentials $\hat{\mathbf{X}} = (\hat{x}_{ij}) \in \mathbb{G}_{\hat{q}}^{N_E \times s}$
       Public confirmation credentials $\hat{\mathbf{Y}} = (\hat{y}_{ij}) \in \mathbb{G}_{\hat{q}}^{N_E \times s}$
       Public vote validity credentials $\hat{\mathbf{Z}} = (\hat{z}_{ij}) \in \mathbb{G}_{\hat{q}}^{N_E \times s}$
for $i = 1, \dots, N_E$ do
    $\hat{x}_i \leftarrow \prod_{j=1}^{s} \hat{x}_{ij} \bmod \hat{p}$
    $\hat{y}_i \leftarrow \prod_{j=1}^{s} \hat{y}_{ij} \bmod \hat{p}$
    $\hat{z}_i \leftarrow \prod_{j=1}^{s} \hat{z}_{ij} \bmod \hat{p}$
$\hat{\mathbf{x}} \leftarrow (\hat{x}_1, \dots, \hat{x}_{N_E})$
$\hat{\mathbf{y}} \leftarrow (\hat{y}_1, \dots, \hat{y}_{N_E})$
$\hat{\mathbf{z}} \leftarrow (\hat{z}_1, \dots, \hat{z}_{N_E})$
return $(\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}})$            // $\hat{\mathbf{x}} \in \mathbb{G}_{\hat{q}}^{N_E}$, $\hat{\mathbf{y}} \in \mathbb{G}_{\hat{q}}^{N_E}$, $\hat{\mathbf{z}} \in \mathbb{G}_{\hat{q}}^{N_E}$
```

Algorithm 8.17: Computes the vectors $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ of public credentials, which are obtained by multiplying corresponding shares obtained from the election authorities.

```
Algorithm: GetVotingCards($\mathbf{D}$)
Input: Voting card data $\mathbf{D} = (d_{ij}) \in (\mathbb{Z}_{\hat{q}_x} \times \mathbb{Z}_{\hat{q}_y} \times \mathcal{B}^{L_{FA}} \times (\mathbb{Z}_{\hat{q}}^2)^n)^{N_E \times s}$
for $i = 1, \dots, N_E$ do
    $\mathbf{d}_i \leftarrow$ GetRow($\mathbf{D}, i$)
    $VC_i \leftarrow$ GetVotingCard($i, \mathbf{d}_i$)                    // see Alg. 8.19
$\mathbf{vc} \leftarrow (VC_1, \dots, VC_{N_E})$
return $\mathbf{vc}$        // $\mathbf{vc} \in (\mathbb{N}^+ \times A_X^{\ell_X} \times A_Y^{\ell_Y} \times (A_R^{\ell_R})^n \times A_{FA}^{\ell_{FA}} \times A_{FA}^{\ell_{FA}})^{N_E}$
```

Algorithm 8.18: Computes the vector $\mathbf{vc} = (VC_1, \dots, VC_{N_E})$ of voting cards for every voter. A single voting card is tuple containing all the necessary information for creating corresponding printouts, which are then sent to the voter using the postal channel.

```
Algorithm: GetVotingCard($v, \mathbf{d}$)
Input: Voter index $v \in \mathbb{N}^+$
       Voting card data $\mathbf{d} = (d_1, \ldots, d_s)$, $d_j = (x_j, y_j, A_j, \mathbf{p}_j)$, $x_j \in \mathbb{Z}_{\hat{q}_x}$, $y_j \in \mathbb{Z}_{\hat{q}_y}$,
       $A_j \in \mathcal{B}^{L_{FA}}$, $\mathbf{p}_j = (p_{ij}) \in (\mathbb{Z}^2_{\hat{q}})^n$, $\sum_{j=1}^s x_j \in \mathbb{Z}_{\hat{q}_x}$, $\sum_{j=1}^s y_j \in \mathbb{Z}_{\hat{q}_y}$
$X \leftarrow \mathsf{IntegerToString}(\sum_{j=1}^s x_j, \ell_X, A_X)$
$Y \leftarrow \mathsf{IntegerToString}(\sum_{j=1}^s y_j, \ell_Y, A_Y)$                    // see Alg. 4.7
for $i = 1, \ldots, n$ do
    for $j = 1, \ldots, s$ do
        $R_{ij} \leftarrow \mathsf{Truncate}(\mathsf{RecHash}_L(p_{ij}), L_{FA})$               // see Alg. 4.15
    $R_i \leftarrow \bigoplus_{j=1}^s R_{ij}$
    $R_i \leftarrow \mathsf{SetWatermark}(R_i, i-1, n_{\max})$                      // see Alg. 4.1
    $RC_i \leftarrow \mathsf{ByteArrayToString}(R_i, A_R)$                         // see Alg. 4.9
$\mathbf{rc} \leftarrow (RC_1, \ldots, RC_n)$
for $j = 1, \ldots, s$ do
    $F_j \leftarrow \mathsf{Truncate}(\mathsf{RecHash}_L(\mathbf{p}_j), L_{FA})$                   // see Alg. 4.15
$FC \leftarrow \mathsf{ByteArrayToString}(\bigoplus_{j=1}^s F_j, A_{FA})$
$AC \leftarrow \mathsf{ByteArrayToString}(\bigoplus_{j=1}^s A_j, A_{FA})$                   // see Alg. 4.9
$VC \leftarrow (v, X, Y, \mathbf{rc}, FC, AC)$
return $VC$                 // $VC \in \mathbb{N}^+ \times A_X^{\ell_X} \times A_Y^{\ell_Y} \times (A_R^{\ell_R})^n \times A_{FA}^{\ell_{FA}} \times A_{FA}^{\ell_{FA}}$
```

Algorithm 8.19: Computes string representations of the voting and confirmation credentials and generates the abstention, finalization, and verification codes. The resulting strings will be printed on the respective voting cards.

8.3. 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 8.3. To submit a ballot containing the voter's selection $\mathbf{s}$, the voting client receives from the administrator the voter-specific voting parameters required for presenting the voting page to the voter. Based on the voter's inputs X and $\mathbf{s}$, the ballot is constructed by calling Alg. 8.21, which internally invokes several sub-algorithms. The authorities call Alg. 8.25 to check the validity of the ballot and Alg. 8.27 to generate the response to the OT query included in the ballot. The voting client unpacks the responses by calling Alg. 8.28 and assembles the resulting point matrix into the verification codes of the selected candidates by calling Alg. 8.30. The voter then compares the displayed

Nr.	Algorithm	Called by	Protocol
8.8	CheckKeyPairProof(π, pk)	Voting Client	7.4
8.20	GetVotingParameters(v, EP)	Administrator	
8.8	CheckKeyPairProof(π, pk)	Voting Client	7.5
8.9	GetPublicKey$(\mathbf{pk})$		
8.21	GenBallot$(X, \mathbf{s}, pk, \mathbf{n}, w)$		
8.22	↳ GetEncodedSelections$(\mathbf{s}, \mathbf{p})$		
8.23	↳ GenQuery$(\mathbf{m}, pk)$		
8.24	↳ GenBallotProof$(x, m, r, \hat{x}, \mathbf{a}, pk)$		
8.28	GetPointMatrix$(\boldsymbol{\beta}, \mathbf{s}, \mathbf{r})$		
8.29	↳ GetPoints$(\beta, \mathbf{s}, \mathbf{r})$		
8.30	GetVerificationCodes$(\mathbf{s}, \mathbf{P})$		
8.20	GetVotingParameters(v, EP)	Election authority	
8.25	CheckBallot$(v, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}})$		
8.26	↳ CheckBallotProof$(\pi, \hat{x}, \mathbf{a}, pk)$		
8.27	GenResponse$(v, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P})$		
8.31	CheckVerificationCodes$(\mathbf{rc}, \mathbf{rc}', \mathbf{s})$	Voter	7.6
8.39	CheckFinalizationCode(FC, FC')		
8.32	GenConfirmation$(Y, \mathbf{P})$	Voting client	
8.33	↳ GetValidityCredential$(\mathbf{p})$		
8.34	↳ GenConfirmationProof$(y, z, \hat{y}, \hat{z})$		
8.37	GetFinalizationCode$(\boldsymbol{\beta}, \boldsymbol{\delta}, \mathbf{s}, \mathbf{r}, \mathbf{n}, \mathbf{k}, \mathbf{e}, pk)$		
8.38	↳ GetAllPoints$(\beta, \delta, \mathbf{s}, \mathbf{r}, \mathbf{n}, \mathbf{k}, \mathbf{e}, pk)$		
8.20	GetVotingParameters(v, EP)	Election authority	
8.35	CheckConfirmation$(v, \gamma, \hat{\mathbf{y}}, \hat{\mathbf{z}})$		
8.36	↳ CheckConfirmationProof$(\pi, \hat{y}, \hat{z})$		

Table 8.3.: Overview of algorithms and sub-algorithms of the election phase.

verification codes with the ones on the voting card and enters the confirmation code Y. We describe the (human) execution of this task by a call to Alg. 8.31. The voting client then generates the confirmation message using Alg. 8.32, which invokes several sub-algorithms. By calling Alg. 8.35, the authorities check the confirmation and return their finalization. Using 8.37, the voting client assembles the finalization code and displays it to the voter, which finally executes Alg. 8.39 to compare it with the finalization code printed on the voting card. Section 7.2 describes the election phase in more details.

```
Algorithm: $\mathsf{GetVotingParameters}(v, EP)$
Input: Voter index $v \in \{1, \ldots, N_E\}$
       Election parameters $EP = (U, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{w})$, $U \in A^*_{\mathsf{ucs}}$, $\mathbf{c} \in (A^*_{\mathsf{ucs}})^n$,
       $\mathbf{d} \in (A^*_{\mathsf{ucs}})^{N_E}$, $\mathbf{e} \in (A^*_{\mathsf{ucs}})^t$, $\mathbf{n} \in (\mathbb{N}^+)^t$, $\mathbf{k} \in (\mathbb{N}^+)^t$, $\mathbf{E} \in \mathbb{B}^{N_E \times t}$, $\mathbf{w} \in (\mathbb{N}^+)^{N_E}$
$\mathbf{e}_v \leftarrow \mathsf{GetRow}(\mathbf{E}, v)$
$VP_v \leftarrow (U, \mathbf{c}, D_v, \mathbf{e}, \mathbf{n}, \mathbf{k}, \mathbf{e}_v, w_v)$
return $VP_v$    // $VP_v \in A^*_{\mathsf{ucs}} \times (A^*_{\mathsf{ucs}})^n \times A^*_{\mathsf{ucs}} \times (A^*_{\mathsf{ucs}})^t \times (\mathbb{N}^+)^t \times (\mathbb{N}^+)^t \times \mathbb{B}^t \times \mathbb{N}^+$
```

Algorithm 8.20: Collects the voter-specific election parameters required to display the voting page to the voter. Specifying the details of presenting the information on the voting page is beyond the scope of this document.

```
Algorithm: $\mathsf{GenBallot}(X, \mathbf{s}, pk, \mathbf{n}, w)$
Input: Voting code $X \in A_X^{\ell_X}$
       Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$
       Encryption key $pk \in \mathbb{G}_q$
       Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$, $n = \sum_{j=1}^t n_j$
       Counting circle $w \in \mathbb{N}^+$
$x \leftarrow \mathsf{StringToInteger}(X, A_x)$                       // see Alg. 4.8
$\hat{x} \leftarrow \hat{g}^x \bmod \hat{p}$
$\mathbf{p} \leftarrow \mathsf{GetPrimes}(n + w)$                  // $\mathbf{p} = (p_0, \ldots, p_{n+w})$, see Alg. 8.1
$\mathbf{m} \leftarrow \mathsf{GetEncodedSelections}(\mathbf{s}, \mathbf{p})$           // $\mathbf{m} = (m_1, \ldots, m_k)$, see Alg. 8.22
$m \leftarrow \prod_{j=1}^k m_j$
if $p_{n+w} \cdot m \geqslant p$ then
    return $\perp$                        // $\mathbf{s}$, $\mathbf{n}$, and $w$ are incompatible with $p$
$(\mathbf{a}, \mathbf{r}) \leftarrow \mathsf{GenQuery}(\mathbf{m}, pk)$        // $\mathbf{a} = (a_1, \ldots, a_k)$, $\mathbf{r} = (r_1, \ldots, r_k)$, see Alg. 8.23
$r \leftarrow \sum_{j=1}^k r_j \bmod q$
$\pi \leftarrow \mathsf{GenBallotProof}(x, m, r, \hat{x}, \mathbf{a}, pk)$                 // see Alg. 8.24
$\alpha \leftarrow (\hat{x}, \mathbf{a}, \pi)$
return $(\alpha, \mathbf{r})$          // $\alpha \in \mathbb{G}_{\hat{q}} \times (\mathbb{G}_q^2)^k \times (\mathbb{Z}_{2^\tau} \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q))$, $\mathbf{r} \in \mathbb{Z}_q^k$
```

Algorithm 8.21: 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. 8.29 to derive the transferred messages from the OT response.

```
Algorithm: GetEncodedSelections($\mathbf{s}, \mathbf{p}$)
Input: Selections $\mathbf{s} = (s_1, \dots, s_k)$, $1 \leqslant s_1 < \cdots < s_k$
       Primes $\mathbf{p} = (p_0, \dots, p_{s_k}) \in (\mathbb{P} \cap \mathbb{G}_q)^{s_k}$
for $j = 1, \dots, k$ do
    $m_j \leftarrow p_{s_j}$
$\mathbf{m} \leftarrow (m_1, \dots, m_k)$
return $\mathbf{m}$                                              // $\mathbf{m} \in (\mathbb{G}_q \cap \mathbb{P})^k$
```

Algorithm 8.22: Selects k prime numbers from $\mathbf{p}$ corresponding to the given indices $\mathbf{s} = (s_1, \dots, s_k)$. For example, $\mathbf{s} = (1, 3, 7)$ means selecting the first, the third, and the seventh prime from $\mathbf{p}$.

```
Algorithm: GenQuery($\mathbf{m}, pk$)
Input: Selected primes $\mathbf{m} = (m_1, \dots, m_k) \in (\mathbb{P} \cap \mathbb{G}_q)^k$
       Encryption key $pk \in \mathbb{G}_q$
for $j = 1, \dots, k$ do
    $r_j \leftarrow$ GenRandomInteger($q$)                        // see Alg. 4.11
    $a_j \leftarrow (m_j \cdot pk^{r_j} \bmod p, g^{r_j} \bmod p)$
$\mathbf{a} \leftarrow (a_1, \dots, a_k)$
$\mathbf{r} \leftarrow (r_1, \dots, r_k)$
return $(\mathbf{a}, \mathbf{r})$                  // $\mathbf{a} \in (\mathbb{G}_q \times \mathbb{G}_q)^k$, $\mathbf{r} \in \mathbb{Z}_q^k$
```

Algorithm 8.23: Generates an OT query $\mathbf{a}$ from the prime numbers $m_j \in \mathbb{P} \cap \mathbb{G}_q$ representing the voter's selections and for a given public encryption public key (which serves as a generator of $\mathbb{G}_q$).

```
**Algorithm:** GenBallotProof$(x, m, r, \hat{x}, \mathbf{a}, pk)$
**Input:** Private voting credentials $x \in \mathbb{Z}_{\hat{q}}$
           Product of selected primes $m \in \mathbb{G}_q$
           Randomization $r \in \mathbb{Z}_q$
           Public voting credential $\hat{x} \in \mathbb{G}_{\hat{q}}$
           Encrypted selections $\mathbf{a} \in (\mathbb{G}_q^2)^k$
           Encryption key $pk \in \mathbb{G}_q$
$\omega_1 \leftarrow$ GenRandomInteger$(\hat{q})$                       // see Alg. 4.11
$\omega_2 \leftarrow$ GenRandomElement$()$                        // see Alg. 4.14
$\omega_3 \leftarrow$ GenRandomInteger$(q)$                       // see Alg. 4.11
$t_1 \leftarrow \hat{g}^{\omega_1} \bmod \hat{p}$, $t_2 \leftarrow \omega_2 \cdot pk^{\omega_3} \bmod p$, $t_3 \leftarrow g^{\omega_3} \bmod p$
$t \leftarrow (t_1, t_2, t_3)$
$y \leftarrow (\hat{x}, \mathbf{a}, pk)$
$c \leftarrow$ GetChallenge$(y, t)$                                // see Alg. 8.4
$s_1 \leftarrow \omega_1 - c \cdot x \bmod \hat{q}$, $s_2 \leftarrow \omega_2 \cdot m^{-c} \bmod p$, $s_3 \leftarrow \omega_3 - c \cdot r \bmod q$
$s \leftarrow (s_1, s_2, s_3)$
$\pi \leftarrow (c, s)$
**return** $\pi$                         // $\pi \in \mathbb{Z}_{2^\tau} \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q)$
```

Algorithm 8.24: Generates a NIZKP, which proves that the ballot has been formed properly. This proof includes a proof of knowledge of the private voting credential x that matches with the public voting credential $\hat{x}$. Note that this is equivalent to a Schnorr identification proof [53]. For the verification of this proof, see Alg. 8.26.

```
**Algorithm:** CheckBallot$(v, \alpha, pk, \mathbf{k}, \mathbf{E}, \hat{\mathbf{x}})$
**Input:** Voter index $v \in \{1, \ldots N_E\}$
           Ballot $\alpha = (\hat{x}, \mathbf{a}, \pi) \in \mathbb{G}_{\hat{q}} \times (\mathbb{G}_q^2)^k \times (\mathbb{Z}_{2^\tau} \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q))$
           Encryption key $pk \in \mathbb{G}_q$
           Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$
           Eligibility matrix $\mathbf{E} = (e_{ij}) \in \mathbb{B}^{N_E \times t}$
           Public voting credentials $\hat{\mathbf{x}} = (\hat{x}_1, \ldots, \hat{x}_{N_E}) \in \mathbb{G}_{\hat{q}}^{N_E}$
$k' \leftarrow \sum_{j=1}^{t} e_{vj} k_j$
**if** $\hat{x} = \hat{x}_v$ **and** $k = k'$ **then**
    **return** CheckBallotProof$(\pi, \hat{x}, \mathbf{a}, pk)$            // see Alg. 8.26
**return** *false*
```

Algorithm 8.25: Checks if a ballot α obtained from voter v is valid. For this, $\hat{x}$ must be the public voting credential of voter v, the length $k = |\mathbf{a}|$ must be equal to $k' = \sum_{j=1}^{t} e_{vj} k_j$, and π must be valid.

```
Algorithm: CheckBallotProof$(\pi, \hat{x}, \mathbf{a}, pk)$
Input: Ballot proof $\pi = (c, s) \in \mathbb{Z}_{2^\tau} \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q)$, $s = (s_1, s_2, s_3)$
       Public voting credential $\hat{x} \in \mathbb{G}_{\hat{q}}$
       Encrypted selections $\mathbf{a} = (a_1, \ldots, a_k)$, $a_j = (a_{j,1}, a_{j,2}) \in \mathbb{G}_q^2$
       Encryption key $pk \in \mathbb{G}_q$
$a_1 \leftarrow \prod_{j=1}^{k} a_{j,1} \bmod p$, $a_2 \leftarrow \prod_{j=1}^{k} a_{j,2} \bmod p$
$t_1 \leftarrow \hat{x}^c \cdot \hat{g}^{s_1} \bmod \hat{p}$
$t_2 \leftarrow a_1^c \cdot s_2 \cdot pk^{s_3} \bmod p$
$t_3 \leftarrow a_2^c \cdot g^{s_3} \bmod p$
$t \leftarrow (t_1, t_2, t_3)$
$y \leftarrow (\hat{x}, \mathbf{a}, pk)$
$c' \leftarrow$ GetChallenge$(y, t)$                                   // see Alg. 8.4
return $c = c'$
```

Algorithm 8.26: Checks the correctness of a NIZKP π generated by Alg. 8.24. The public values of this proof are the public voting credential $\hat{x}$ and the OT query $\mathbf{a} = (a_1, \ldots, a_k)$.

```
**Algorithm:** GenResponse($v, \mathbf{a}, pk, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{P}$)
**Input:** Voter index $v \in \{1, \ldots, N_E\}$
       Query $\mathbf{a} = (a_1, \ldots, a_k)$, $a_j = (a_{j,1}, a_{j,2}) \in \mathbb{G}_q^2$
       Encryption key $pk \in \mathbb{G}_q$
       Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$, $n = \sum_{j=1}^t n_j$
       Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Eligibility matrix $\mathbf{E} = (e_{ij}) \in \mathbb{B}^{N_E \times t}$, $k = \sum_{j=1}^t e_{vj} k_j$
       Points $\mathbf{P} = (p_{ij}) \in (\mathbb{Z}_{\hat{q}}^2)^{N_E \times n}$, $p_{ij} = (x_{ij}, y_{ij})$
$(e_1, \ldots, e_t) \leftarrow$ GetRow($\mathbf{E}, v$)
$z_1 \leftarrow$ GenRandomInteger($q$), $z_2 \leftarrow$ GenRandomInteger($q$)        // see Alg. 4.11
**for** $j = 1, \ldots, k$ **do**
    $\beta_j \leftarrow$ GenRandomElement()        // see Alg. 4.14
    $b_j \leftarrow a_{j,1}^{z_1} a_{j,2}^{z_2} \beta_j \bmod p$
**for** $i = 1, \ldots, n$ **do**
    **for** $j = 1, \ldots, k$ **do**
        $C_{ij} \leftarrow \varnothing$
$\ell_M \leftarrow \lceil L_M / L \rceil$
$\mathbf{p} \leftarrow$ GetPrimes($n$)        // $\mathbf{p} = (p_0, \ldots, p_n)$, see Alg. 8.1
$n' \leftarrow 0$, $k' \leftarrow 0$
**for** $l = 1, \ldots, t$ **do**
    **if** $e_l = 1$ **then**
        **for** $i = n' + 1, \ldots, n' + n_l$ **do**        // loop for $i = 1, \ldots, n$
            $p'_i \leftarrow p_i^{z_1} \bmod p$
            $M_i \leftarrow$ IntegerToByteArray($x_{vi}, \frac{L_M}{2}$) $\|$ IntegerToByteArray($y_{vi}, \frac{L_M}{2}$)        // see Alg. 4.4
            **for** $j = k' + 1, \ldots, k' + k_l$ **do**        // loop for $j = 1, \ldots, k$
                $k_{ij} \leftarrow p'_i \beta_j \bmod p$
                $K_{ij} \leftarrow$ Truncate($\|_{c=1}^{\ell_M}$ RecHash$_L(k_{ij}, c), L_M$)        // see Alg. 4.15
                $C_{ij} \leftarrow M_i \oplus K_{ij}$
    $n' \leftarrow n' + n_l$, $k' \leftarrow k' + e_l k_l$
$\mathbf{b} \leftarrow (b_1, \ldots, b_k)$, $\mathbf{C} \leftarrow (C_{ij})_{n \times k}$, $d \leftarrow pk^{z_1} g^{z_2} \bmod p$
$\beta \leftarrow (\mathbf{b}, \mathbf{C}, d)$
$\delta = (z_1, z_2)$
**return** $(\beta, \delta)$        // $\beta \in \mathbb{G}_q^k \times (\mathcal{B}^{L_M} \cup \{\varnothing\})^{n \times k} \times \mathbb{G}_q$, $\delta \in \mathbb{Z}_q^2$
```

Algorithm 8.27: 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 $\delta = (z_1, z_2)$ used to generate the response.

```
Algorithm: GetPointMatrix($\boldsymbol{\beta}, \mathbf{s}, \mathbf{r}$)
Input: Responses $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_s)$, $\beta_j \in \mathbb{G}_q^k \times (\mathcal{B}^{L_M} \cup \{\varnothing\})^{n \times k} \times \mathbb{G}_q$
       Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$
       Randomizations $\mathbf{r} = (r_1, \ldots, r_k) \in \mathbb{Z}_q^k$
for $j = 1, \ldots, s$ do
    $\mathbf{p}_j \leftarrow$ GetPoints($\beta_j, \mathbf{s}, \mathbf{r}$)        // $\mathbf{p}_j = (p_{1,j}, \ldots, p_{k,j})$, see Alg. 8.29
$\mathbf{P} \leftarrow (p_{ij})_{k \times s}$
return $\mathbf{P}$                                                  // $\mathbf{P} \in (\mathbb{Z}_p^2)^{k \times s}$
```

Algorithm 8.28: Computes the s-by-k matrix $\mathbf{P} = (p_{ij})_{s \times k}$ 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: GetPoints($\beta, \mathbf{s}, \mathbf{r}$)
Input: Response $\beta = (\mathbf{b}, \mathbf{C}, d)$, $\mathbf{b} = (b_1, \ldots, b_k) \in \mathbb{G}_q^k$, $\mathbf{C} = (C_{ij}) \in (\mathcal{B}^{L_M} \cup \{\varnothing\})^{n \times k}$,
       $d \in \mathbb{G}_q$
       Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$
       Randomizations $\mathbf{r} = (r_1, \ldots, r_k) \in \mathbb{Z}_q^k$
$\ell_M \leftarrow \lceil L_M / L \rceil$
for $j = 1, \ldots, k$ do
    $k_j \leftarrow b_j \cdot d^{-r_j} \bmod p$
    $K_j \leftarrow$ Truncate($\|_{c=1}^{\ell_M}$RecHash$_L(k_j, c), L_M$)            // see Alg. 4.15
    if $C_{s_j,j} = \varnothing$ then
        return $\perp$                                               // invalid matrix entry
    $M_j \leftarrow C_{s_j,j} \oplus K_j$
    $x_j \leftarrow$ ByteArrayToInteger(Truncate($M_j, \frac{L_M}{2}$))          // see Alg. 4.5
    $y_j \leftarrow$ ByteArrayToInteger(Skip($M_j, \frac{L_M}{2}$))              // see Alg. 4.5
    if $x_j \geqslant \hat{q}$ or $y_j \geqslant \hat{q}$ then
        return $\perp$                                               // point not in $\mathbb{Z}_{\hat{q}}^2$
    $p_j \leftarrow (x_j, y_j)$
$\mathbf{p} \leftarrow (p_1, \ldots, p_k)$
return $\mathbf{p}$                                                  // $\mathbf{p} \in (\mathbb{Z}_{\hat{q}}^2)^k$
```

Algorithm 8.29: 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 transferred points lie outside $\mathbb{Z}_{\hat{q}}^2$.

```
Algorithm: $\mathsf{GetVerificationCodes}(\mathbf{s}, \mathbf{P})$
Input: Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n_{\max}$
       Points $\mathbf{P} = (p_{ij}) \in (\mathbb{Z}_{\hat{q}}^2)^{k \times s}$
for $i = 1, \ldots, k$ do
    for $j = 1, \ldots, s$ do
        $R_{ij} \leftarrow \mathsf{Truncate}(\mathsf{RecHash}_L(p_{ij}), L_R)$          // see Alg. 4.15
    $R_i \leftarrow \bigoplus_{j=1}^{s} R_{ij}$
    $R_i \leftarrow \mathsf{SetWatermark}(R_i, s_i - 1, n_{\max})$          // see Alg. 4.1
    $RC_i \leftarrow \mathsf{ByteArrayToString}(R_i, A_R)$          // see Alg. 4.9
$\mathbf{rc} \leftarrow (RC_1, \ldots, RC_k)$
return $\mathbf{rc}$          // $\mathbf{rc} \in (A_R^{\ell_R})^k$
```

Algorithm 8.30: Computes the k verification codes $\mathbf{rc} = (RC_{s_1}, \ldots, RC_{s_k})$ for the selected candidates by combining the hash values of the transferred points $p_{ij} \in \mathbf{P}$ from different authorities. This algorithm is the counterpart of Alg. 8.19.

```
Algorithm: $\mathsf{CheckVerificationCodes}(\mathbf{rc}, \mathbf{rc}', \mathbf{s})$
Input: Printed verification codes $\mathbf{rc} = (RC_1, \ldots, RC_n) \in (A_R^{\ell_R})^n$
       Displayed verification codes $\mathbf{rc}' = (RC'_1, \ldots, RC'_k) \in (A_R^{\ell_R})^k$
       Selections $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$
for $j = 1, \ldots, k$ do
    if $RC_{s_j} \neq RC'_j$ then
        return $false$
return $true$
```

Algorithm 8.31: Checks if every displayed verification code RC'_ij matches with the verification code RC_{s_j} of the selected candidate s_j as printed on the voting card. Note that this algorithm is executed by humans.

```
Algorithm: GenConfirmation($Y, \mathbf{P}$)
Input: Confirmation code $Y \in A_Y^{\ell_Y}$
       Points $\mathbf{P} = (p_{ij}) \in (\mathbb{Z}_{\hat{q}}^2)^{k \times s}$
$y \leftarrow$ StringToInteger($Y, A_Y$)                              // see Alg. 4.8
$\hat{y} \leftarrow \hat{g}^y \bmod \hat{p}$                                         // see Alg. 4.8
for $j = 1, \dots, s$ do
    $\mathbf{p}_j \leftarrow$ GetCol($\mathbf{P}, j$)
    $z_j \leftarrow$ GetValidityCredential($\mathbf{p}_j$)                  // see Alg. 8.33
$z \leftarrow \sum_{j=1}^{s} z_j \bmod \hat{q}$, $\hat{z} \leftarrow \hat{g}^z \bmod \hat{p}$
$\pi \leftarrow$ GenConfirmationProof($y, z, \hat{y}, \hat{z}$)              // see Alg. 8.34
$\gamma \leftarrow (\hat{y}, \hat{z}, \pi)$
return $\gamma$                                // $\gamma \in \mathbb{G}_{\hat{q}} \times \mathbb{G}_{\hat{q}} \times (\mathbb{Z}_{2^\tau} \times \mathbb{Z}_{\hat{q}})$
```

Algorithm 8.32: Generates the confirmation γ, which consists of the public confirmation and vote validity credentials $\hat{y}$ and $\hat{z}$, respectively, and a NIZKP of knowledge π of corresponding private credentials y and z.

```
Algorithm: GetValidityCredential($\mathbf{p}$)
Input: Points $\mathbf{p} = (p_1, \dots, p_k)$, $p_j = (x_j, y_j) \in \mathbb{Z}_{\hat{q}}^2$, $x_j \neq x_k$ for $j \neq k$
$z \leftarrow 0$
for $i = 1, \dots, k$ do
    $n \leftarrow 1$, $d \leftarrow 1$
    for $j = 1, \dots, k$ do
        if $i \neq j$ then
            $n \leftarrow n \cdot x_j \bmod \hat{q}$
            $d \leftarrow d \cdot (x_j - x_i) \bmod \hat{q}$
    $z \leftarrow z + y_i \cdot \frac{n}{d} \bmod \hat{q}$
return $z$                                                    // $z \in \mathbb{Z}_{\hat{q}}$
```

Algorithm 8.33: Computes a polynomial $A(X)$ from given points $\mathbf{p} = (p_1, \dots, p_k)$ using Lagrange's interpolation method and returns the value $z = A(0)$.

```
Algorithm: GenConfirmationProof$(y, z, \hat{y}, \hat{z})$
Input: Private confirmation credential $y \in \mathbb{Z}_{\hat{q}}$
       Private vote validity credential $z \in \mathbb{Z}_{\hat{q}}$
       Public confirmation credential $\hat{y} \in \mathbb{G}_{\hat{q}}$
       Public confirmation credential $\hat{z} \in \mathbb{G}_{\hat{q}}$
$\omega_1 \leftarrow$ GenRandomInteger$(\hat{q})$, $\omega_2 \leftarrow$ GenRandomInteger$(\hat{q})$        // see Alg. 4.11
$t_1 \leftarrow \hat{g}^{\omega_1} \bmod \hat{p}$, $t_2 \leftarrow \hat{g}^{\omega_2} \bmod \hat{p}$
$t \leftarrow (t_1, t_2)$
$c \leftarrow$ GetChallenge$((\hat{y}, \hat{z}), t)$        // see Alg. 8.4
$s_1 \leftarrow \omega_1 - c \cdot y \bmod \hat{q}$, $s_2 \leftarrow \omega_2 - c \cdot z \bmod \hat{q}$
$s \leftarrow (s_1, s_2)$
$\pi \leftarrow (c, s)$
return $\pi$        // $\pi \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_{\hat{q}}^2$
```

Algorithm 8.34: Generates a NIZKP of knowledge of the private confirmation and vote validity credentials that match with corresponding public credential. For the verification of π, see Alg. 8.36.

```
Algorithm: CheckConfirmation$(v, \gamma, \hat{\mathbf{y}}, \hat{\mathbf{z}})$
Input: Voter index $v \in \{1, \ldots, N_E\}$
       Confirmation $\gamma = (\hat{y}, \hat{z}, \pi) \in \mathbb{G}_{\hat{q}} \times \mathbb{G}_{\hat{q}} \times (\mathbb{Z}_{2^\tau} \times \mathbb{Z}_{\hat{q}}^2)$
       Public confirmation credentials $\hat{\mathbf{y}} = (\hat{y}_1, \ldots, \hat{y}_{N_E}) \in \mathbb{G}_{\hat{q}}^{N_E}$
       Public vote validity credentials $\hat{\mathbf{z}} = (\hat{z}_1, \ldots, \hat{z}_{N_E}) \in \mathbb{G}_{\hat{q}}^{N_E}$
if $\hat{y} = \hat{y}_v$ and $\hat{z} = \hat{z}_v$ then
    return CheckConfirmationProof$(\pi, \hat{y}, \hat{z})$        // see Alg. 8.36
return false
```

Algorithm 8.35: Checks if a confirmation γ obtained from voter v is valid. The check succeeds if π is valid and if $\hat{y}$ and $\hat{z}$ are the public confirmation and vote validity credentials of voter v.

```
Algorithm: CheckConfirmationProof$(\pi, \hat{y}, \hat{z})$
Input: Confirmation proof $\pi = (c, s) \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_{\hat{q}}^2$, $s = (s_1, s_2)$
       Public confirmation credential $\hat{y} \in \mathbb{G}_{\hat{q}}$
       Public vote validity credential $\hat{z} \in \mathbb{G}_{\hat{q}}$
$y \leftarrow (\hat{y}, \hat{z})$
$t_1 \leftarrow \hat{y}^c \cdot \hat{g}^{s_1} \bmod \hat{p}$
$t_2 \leftarrow \hat{z}^c \cdot \hat{g}^{s_2} \bmod \hat{p}$
$t \leftarrow (t_1, t_2)$
$c' \leftarrow$ GetChallenge$(y, t)$        // see Alg. 8.4
return $c = c'$
```

Algorithm 8.36: Checks the correctness of a NIZKP π generated by Alg. 8.34. The public values of this proof are the public confirmation and vote validity credentials $\hat{y}$ and $\hat{z}$, respectively.

```
Algorithm: GetFinalizationCode($\boldsymbol{\beta}, \boldsymbol{\delta}, \mathbf{s}, \mathbf{r}, \mathbf{n}, \mathbf{k}, \mathbf{e}, pk$)
Input: Responses $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_s) \in (\mathbb{G}_q^k \times (\mathcal{B}^{L_M} \cup \{\varnothing\})^{n \times k} \times \mathbb{G}_q)^s$
       Finalizations $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_s) \in (\mathbb{Z}_q^2)^s$
       Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$
       Randomizations $\mathbf{r} = (r_1, \ldots, r_k) \in \mathbb{Z}_q^k$
       Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$, $n = \sum_{j=1}^t n_j$
       Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Eligibility vector $\mathbf{e} = (e_1, \ldots, e_t) \in \mathbb{B}^t$, $k = \sum_{j=1}^t e_j k_j$
       Encryption key $pk \in \mathbb{G}_q$
for $j = 1, \ldots, s$ do
    $\mathbf{p}_j \leftarrow$ GetAllPoints($\beta_j, \delta_j, \mathbf{s}, \mathbf{r}, \mathbf{n}, \mathbf{k}, \mathbf{e}, pk$)      // see Alg. 8.38
    $F_j \leftarrow$ Truncate(RecHash$_L(\mathbf{p}_j), L_{FA}$)      // see Alg. 4.15
$FC \leftarrow$ ByteArrayToString($\oplus_{j=1}^s F_j, A_{FA}$)      // see Alg. 4.9
return $FC$      // $FC \in A_{FA}^{\ell_{FA}}$
```

Algorithm 8.37: Computes a finalization code FC by combining the values F_j received from the authorities.

```
Algorithm: $\mathsf{GetAllPoints}(\beta, \delta, \mathbf{s}, \mathbf{r}, \mathbf{n}, \mathbf{k}, \mathbf{e}, pk)$
Input: Response $\beta = (\mathbf{b}, \mathbf{C}, d)$, $\mathbf{b} = (b_1, \ldots, b_k) \in \mathbb{G}_q^k$, $\mathbf{C} = (C_{ij}) \in (\mathcal{B}^{L_M} \cup \{\varnothing\})^{n \times k}$,
       $d \in \mathbb{G}_q$
       Randomizations $\delta = (z_1, z_2) \in \mathbb{Z}_q^2$
       Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$
       Randomizations $\mathbf{r} = (r_1, \ldots, r_k) \in \mathbb{Z}_q^k$
       Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$, $n = \sum_{j=1}^t n_j$
       Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Eligibility vector $\mathbf{e} = (e_1, \ldots, e_t) \in \mathbb{B}^t$, $k = \sum_{j=1}^t e_j k_j$
       Encryption key $pk \in \mathbb{G}_q$
$d' \leftarrow pk^{z_1} g^{z_2} \bmod p$
if $d \neq d'$ then
    return $\perp$                                   // values $(z_1, z_2)$ incompatible with $d$
$(p_0, \ldots, p_n) \leftarrow \mathsf{GetPrimes}(n)$                          // see Alg. 8.1
$\ell_M \leftarrow \lceil L_M / L \rceil$
for $i = 1, \ldots, n$ do
    $p_i \leftarrow (0, 0)$                              // default value for non-eligibility
$n' \leftarrow 0$, $k' \leftarrow 0$
for $l = 1, \ldots, t$ do
    if $e_l = 1$ then
        for $i = n' + 1, \ldots, n' + n_l$ do
            $p'_i \leftarrow p_i^{z_1} \bmod p$
        for $j = k' + 1, \ldots, k' + k_l$ do
            $\beta_j \leftarrow b_j \cdot d^{-r_j} \cdot (p'_{s_j})^{-1} \bmod p$
        for $i = n' + 1, \ldots, n' + n_l$ do                 // loop for $i = 1, \ldots, n$
            for $j = k' + 1, \ldots, k' + k_l$ do
                $k_{ij} \leftarrow p'_i \beta_j \bmod p$
                $K_{ij} \leftarrow \mathsf{Truncate}(\|_{c=1}^{\ell_M} \mathsf{RecHash}_L(k_{ij}, c), L_M)$       // see Alg. 4.15
                $M_{ij} \leftarrow C_{ij} \oplus K_{ij}$
            if $\neg(M_{i,k'+1} = \cdots = M_{i,k'+k_l})$ then
                return $\perp$                           // incompatible messages
            $x_i \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{Truncate}(M_{i,k'+1}, \frac{L_M}{2}))$      // see Alg. 4.5
            $y_i \leftarrow \mathsf{ByteArrayToInteger}(\mathsf{Skip}(M_{i,k'+1}, \frac{L_M}{2}))$          // see Alg. 4.5
            if $x_i \geqslant \hat{q}$ or $y_i \geqslant \hat{q}$ then
                return $\perp$                               // point not in $\mathbb{Z}_{\hat{q}}^2$
            $p_i \leftarrow (x_i, y_i)$
    $n' \leftarrow n' + n_l$, $k' \leftarrow k' + e_l k_l$
$\mathbf{p} \leftarrow (p_1, \ldots, p_n)$
return $\mathbf{p}$                                                  // $\mathbf{p} \in (\mathbb{Z}_{\hat{q}}^2)^n$
```

Algorithm 8.38: Computes all n points $\mathbf{p} = (p_1, \ldots, p_n)$ from the OT response β using the authority's random values $z = (z_1, z_2)$. The algorithm returns $\perp$, if some points lie outside $\mathbb{Z}_{\hat{q}}^2$.

```
Algorithm: CheckFinalizationCode$(FC, FC')$
Input: Printed finalization code $FC \in A_{FA}^{\ell_{FA}}$
       Displayed finalization code $FC' \in A_{FA}^{\ell_{FA}}$
return $FC = FC'$
```

Algorithm 8.39: 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.

8.4. 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 8.4. To initiate the mixing process, the first election authority calls Alg. 8.40 to cleanse the list of submitted ballots and to extract a sorted list of encrypted votes to shuffle. By calling Algs. 8.42 and 8.45, this list is shuffled according to a random permutation and a non-interactive shuffle proof 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. 8.48 to check the validity of every shuffle proof received from the others. The partial decryptions are then computed using Alg. 8.49 and corresponding decryption proofs are generated using Alg. 8.50. After terminating all tasks, the process is handed over from the election authorities to the administrator, who calls Alg. 8.49 to perform the final decryption, Alg. 8.53 to obtain the plaintext votes, and Alg. 8.54 to derive the election result. We refer to Section 7.3 for a more detailed description of this process.

Nr.	Algorithm	Called by	Protocol
8.40	GetEncryptions$(B, C, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{w})$	Election authority	7.7
8.41	↳ GetDefaultEligibility$(\mathbf{w}, \mathbf{E})$		
8.42	GenShuffle$(\mathbf{e}, pk)$		
8.43	↳ GenPermutation(N)		
8.44	↳ GenReEncryption(e, pk)		
8.45	GenShuffleProof$(U, \mathbf{e}, \tilde{\mathbf{e}}, \tilde{\mathbf{r}}, \psi, pk)$		
8.46	↳ GenPermutationCommitment$(\psi, \mathbf{h})$		
8.47	↳ GenCommitmentChain$(\tilde{\mathbf{u}})$		
8.48	CheckShuffleProof$(U, \pi, \mathbf{e}, \tilde{\mathbf{e}}, pk)$		
8.49	GetDecryptions$(\mathbf{e}, sk)$	Election authority	7.8
8.50	GenDecryptionProof$(sk, pk, \mathbf{e}, \mathbf{c})$		
8.51	CheckDecryptionProof$(\pi, pk, \mathbf{e}, \mathbf{c})$		
8.52	GetCombinedDecryptions$(\mathbf{C})$		
8.49	GetDecryptions$(\mathbf{e}, sk)$	Administrator	7.9
8.50	GenDecryptionProof$(sk, pk, \mathbf{e}, \mathbf{c})$		
8.53	GetVotes$(\mathbf{e}, \mathbf{c}, \mathbf{c}')$		
8.54	GetElectionResult$(\mathbf{m}, \mathbf{n}, \mathbf{w})$		
8.55	GetInspection$(v, \mathbf{P}, \mathbf{a}, C)$	Election authority	7.10
8.56	GetInspectionCode$(\mathbf{i})$	Voting Client	
8.57	CheckInspectionCode(pb, FC, AC, IC)	Voter	

Table 8.4.: Overview of algorithms and sub-algorithms of the post-election phase.

```
Algorithm: GetEncryptions$(B, C, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{w})$
Input: Ballot list $B = \langle(v_i, \alpha_i)\rangle_{i=0}^{N_B-1}$, $v_i \in \{1, \ldots, N_E\}$, $\alpha_i = (\hat{x}_i, \mathbf{a}_i, \pi_i)$, $\mathbf{a}_i \in (\mathbb{G}_q^2)^{k'_i}$
       Confirmation list $C = \langle(v_i, \gamma_i)\rangle_{i=0}^{N_C-1}$, $v_i \in \{1, \ldots, N_E\}$
       Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$
       Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Eligibility matrix $\mathbf{E} = (e_{ij}) \in \mathbb{B}^{N_E \times t}$
       Counting circles $\mathbf{w} = (w_1, \ldots, w_{N_E}) \in (\mathbb{N}^+)^{N_E}$
$n = \sum_{j=1}^{t} n_j$
$w \leftarrow \max_{i=1}^{N_E} w_i$
$\mathbf{E}^* \leftarrow$ GetDefaultEligibility$(\mathbf{w}, \mathbf{E})$          // $\mathbf{E}^* = (e^*_{cj})_{w \times t}$, see Alg. 8.41
$\mathbf{p} \leftarrow$ GetPrimes$(n + w)$          // $\mathbf{p} = (p_0, \ldots, p_{n+w})$, see Alg. 8.1
$i \leftarrow 1$
foreach $(v, \alpha) \in B$ do          // $\alpha = (\hat{x}, (a_1, \ldots, a_k), \pi)$, $a_j = (a_{j,1}, a_{j,2}) \in \mathbb{G}_q^2$
    if $(v, \cdot) \in C$ then
        $c \leftarrow w_v$
        $a \leftarrow p_{n+c}$          // add counting circle
        $n' \leftarrow 0$
        for $l = 1, \ldots, t$ do
            if $e_{vl} < e^*_{cl}$ then          // check for restricted eligibility
                for $j = n' + 1, \ldots, n' + k_l$ do          // add default candidates
                    $a \leftarrow a \cdot p_j \bmod p$
            $n' \leftarrow n' + n_l$
        $i \leftarrow i + 1$
        $e_i \leftarrow (a \cdot \prod_{j=1}^{k} a_{j,1} \bmod p, \prod_{j=1}^{k} a_{j,2} \bmod p)$
        if $a \neq p_{n+c}$ then
            $i \leftarrow i + 1$
            $e_i \leftarrow (p_0 \cdot a \bmod p, 1)$          // mark encryption with $p_0$
$N \leftarrow i$
$\mathbf{e} \leftarrow (e_1, \ldots, e_N)$
$\mathbf{e}' \leftarrow$ Sort$_{\preceq}(\mathbf{e}')$
return $\mathbf{e}'$          // $\mathbf{e} \in (\mathbb{G}_q^2)^N$
```

Algorithm 8.40: Computes a sorted list of ElGamal encryptions from the list of submitted ballots, for which a valid confirmation is available. The counting circles are added to the encryptions. Default candidates are added for voters with restricted eligibility and corresponding marked encryptions are generated and added to the resulting list. Sorting the resulting vector $\mathbf{e}$ of encrypted votes into $\mathbf{e}'$ is necessary to guarantee a unique result. For this, we define a total order over $\mathbb{G}_q^2$ by $e_i \preceq e_j \Leftrightarrow (a_i, b_i) \preceq (a_j, b_j) \Leftrightarrow (a_i < a_j) \vee (a_i = a_j \wedge b_i \leqslant b_j)$.

```
Algorithm: GetDefaultEligibility($\mathbf{w}, \mathbf{E}$)
Input: Counting circles $\mathbf{w} = (w_1, \ldots, w_{N_E}) \in (\mathbb{N}^+)^{N_E}$
       Eligibility matrix $\mathbf{E} = (e_{ij}) \in \mathbb{B}^{N_E \times t}$
$w \leftarrow \max_{i=1}^{N_E} w_i$
for $j = 1, \ldots, t$ do
    for $c = 1, \ldots, w$ do
        $e^*_{cj} \leftarrow 0$
    for $i = 1, \ldots, N_E$ do
        if $e_{ij} = 1$ then
            $e^*_{w_i,j} \leftarrow 1$
$\mathbf{E}^* \leftarrow (e^*_{cj})_{w \times t}$
return $\mathbf{E}^*$                                                // $\mathbf{E}^* \in \mathbb{B}^{w \times t}$
```

Algorithm 8.41: Computes the default eligibility of all counting circles.

```
Algorithm: GenShuffle($\mathbf{e}, pk$)
Input: Encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i \in \mathbb{G}_q^2$
       Encryption key $pk \in \mathbb{G}_q$
$\psi \leftarrow$ GenPermutation($N$)                 // $\psi = (j_1, \ldots, j_N) \in \Psi_N$, see Alg. 8.43
for $i = 1, \ldots, N$ do
    $(\tilde{e}_i, \tilde{r}_{j_i}) \leftarrow$ GenReEncryption($e_{j_i}, pk$)              // see Alg. 8.44
$\tilde{\mathbf{e}} \leftarrow (\tilde{e}_1, \ldots, \tilde{e}_N)$
$\tilde{\mathbf{r}} \leftarrow (\tilde{r}_1, \ldots, \tilde{r}_N)$
return $(\tilde{\mathbf{e}}, \tilde{\mathbf{r}}, \psi)$                  // $\tilde{\mathbf{e}} \in (\mathbb{G}_q^2)^N$, $\tilde{\mathbf{r}} \in \mathbb{Z}_q^N$, $\psi \in \Psi_N$
```

Algorithm 8.42: Generates a random permutation $\psi \in \Psi_N$ and uses it to shuffle a given vector $\mathbf{e} = (e_1, \ldots, e_N)$ of 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: GenPermutation($N$)
Input: Permutation size $N \in \mathbb{N}$
$I \leftarrow \langle 1, \ldots, N \rangle$
for $i = 0, \ldots, N-1$ do
    $k \leftarrow$ GenRandomInteger($i, N-1$)                      // see Alg. 4.13
    $j_{i+1} \leftarrow I[k]$
    $I[k] \leftarrow I[i]$
$\psi \leftarrow (j_1, \ldots, j_N)$
return $\psi$                                                    // $\psi \in \Psi_N$
```

Algorithm 8.43: Generates a random permutation $\psi \in \Psi_N$ following Knuth's shuffle algorithm [39, pp. 139–140].

```
**Algorithm:** GenReEncryption($e, pk$)
**Input:** Encryption $e = (a, b) \in \mathbb{G}_q^2$
           Encryption key $pk \in \mathbb{G}_q$
$\tilde{r} \leftarrow$ GenRandomInteger($q$)                                  // see Alg. 4.11
$\tilde{a} \leftarrow a \cdot pk^{\tilde{r}} \bmod p$
$\tilde{b} \leftarrow b \cdot g^{\tilde{r}} \bmod p$
$\tilde{e} \leftarrow (\tilde{a}, \tilde{b})$
**return** $(\tilde{e}, \tilde{r})$                                        // $\tilde{e} \in \mathbb{G}_q^2$, $\tilde{r} \in \mathbb{Z}_q$
```

Algorithm 8.44: Generates a re-encryption $e' = (a \cdot pk^{r'}, b \cdot g^{r'})$ of the given encryption $e = (a, b) \in \mathbb{G}_q^2$. The re-encryption $\tilde{e}$ is returned together with the randomization $\tilde{r} \in \mathbb{Z}_q$.

```
**Algorithm:** $\mathsf{GenShuffleProof}(U, \mathbf{e}, \tilde{\mathbf{e}}, \tilde{\mathbf{r}}, \psi, pk)$
**Input:** Election event identifier $U \in A^*_{\mathsf{ucs}}$
           Encryptions $\mathbf{e} \in (\mathbb{G}_q^2)^N$
           Shuffled encryptions $\tilde{\mathbf{e}} = (\tilde{e}_1, \ldots, \tilde{e}_N)$, $\tilde{e}_i = (\tilde{a}_i, \tilde{b}_i) \in \mathbb{G}_q^2$
           Re-encryption randomizations $\tilde{\mathbf{r}} = (\tilde{r}_1, \ldots, \tilde{r}_N) \in \mathbb{Z}_q^N$
           Permutation $\psi = (j_1, \ldots, j_N) \in \Psi_N$
           Encryption key $pk \in \mathbb{G}_q$
$\mathbf{h} \leftarrow \mathsf{GetGenerators}(N, U)$                                   // see Alg. 8.3
$(\mathbf{c}, \mathbf{r}) \leftarrow \mathsf{GenPermutationCommitment}(\psi, \mathbf{h})$     // $\mathbf{c} = (c_1, \ldots, c_N)$, $\mathbf{r} = (r_1, \ldots, r_N)$
$\mathbf{u} \leftarrow \mathsf{GetChallenges}(N, (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}, pk))$              // $\mathbf{u} = (u_1, \ldots, u_N)$, see Alg. 8.5
**for** $i = 1, \ldots, N$ **do**
    $\tilde{u}_i \leftarrow u_{j_i}$
$\tilde{\mathbf{u}} \leftarrow (\tilde{u}_1, \ldots, \tilde{u}_N)$
$(\hat{\mathbf{c}}, \hat{\mathbf{r}}) \leftarrow \mathsf{GenCommitmentChain}(\tilde{\mathbf{u}})$                // $\hat{\mathbf{c}} = (\hat{c}_1, \ldots, \hat{c}_N)$, see Alg. 8.47
$R_0 \leftarrow 0$, $U_0 \leftarrow 1$
**for** $i = 1, \ldots, N$ **do**
    $\hat{\omega}_i \leftarrow \mathsf{GenRandomInteger}(q)$, $\tilde{\omega}_i \leftarrow \mathsf{GenRandomInteger}(q)$          // see Alg. 4.11
    $R_i \leftarrow \hat{r}_i + \tilde{u}_i R_{i-1} \bmod q$, $R'_i \leftarrow \hat{\omega}_i + \tilde{\omega}_i R_{i-1} \bmod q$
    $U_i \leftarrow \tilde{u}_i U_{i-1} \bmod q$, $U'_i \leftarrow \tilde{\omega}_i U_{i-1} \bmod q$
    $\hat{t}_i \leftarrow g^{R'_i} h^{U'_i} \bmod p$
$\omega_1 \leftarrow \mathsf{GenRandomInteger}(q)$, $t_1 \leftarrow g^{\omega_1} \bmod p$                      // see Alg. 4.11
$\omega_2 \leftarrow \mathsf{GenRandomInteger}(q)$, $t_2 \leftarrow g^{\omega_2} \bmod p$                      // see Alg. 4.11
$\omega_3 \leftarrow \mathsf{GenRandomInteger}(q)$, $t_3 \leftarrow g^{\omega_3} \prod_{i=1}^N h_i^{\tilde{\omega}_i} \bmod p$            // see Alg. 4.11
$\omega_4 \leftarrow \mathsf{GenRandomInteger}(q)$                                    // see Alg. 4.11
$(t_{4,1}, t_{4,2}) \leftarrow (pk^{-\omega_4} \prod_{i=1}^N \tilde{a}_i^{\tilde{\omega}_i} \bmod p, g^{-\omega_4} \prod_{i=1}^N \tilde{b}_i^{\tilde{\omega}_i} \bmod p)$
$t \leftarrow (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), (\hat{t}_1, \ldots, \hat{t}_N))$
$y \leftarrow (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}, \hat{\mathbf{c}}, pk)$
$c \leftarrow \mathsf{GetChallenge}(y, t)$                                          // see Alg. 8.4
$\bar{r} \leftarrow \sum_{i=1}^N r_i \bmod q$, $s_1 \leftarrow \omega_1 - c \cdot \bar{r} \bmod q$
$v_N \leftarrow 1$
**for** $i = N, \ldots, 1$ **do**
    $v_{i-1} \leftarrow \tilde{u}_i v_i \bmod q$
$\hat{r} \leftarrow \sum_{i=1}^N \hat{r}_i v_i \bmod q$, $s_2 \leftarrow \omega_2 - c \cdot \hat{r} \bmod q$
$r \leftarrow \sum_{i=1}^N r_i u_i \bmod q$, $s_3 \leftarrow \omega_3 - c \cdot r \bmod q$
$\tilde{r} \leftarrow \sum_{i=1}^N \tilde{r}_i u_i \bmod q$, $s_4 \leftarrow \omega_4 - c \cdot \tilde{r} \bmod q$
**for** $i = 1, \ldots, N$ **do**
    $\hat{s}_i \leftarrow \hat{\omega}_i - c \cdot \hat{r}_i \bmod q$, $\tilde{s}_i \leftarrow \tilde{\omega}_i - c \cdot \tilde{u}_i \bmod q$
$s \leftarrow (s_1, s_2, s_3, s_4, (\hat{s}_1, \ldots, \hat{s}_N), (\tilde{s}_1, \ldots, \tilde{s}_N))$
$\pi \leftarrow (c, s, \mathbf{c}, \hat{\mathbf{c}})$
**return** $\pi$        // $\pi \in \mathbb{Z}_{2^\tau} \times (\mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N) \times \mathbb{G}_q^N \times \mathbb{G}_q^N$
```

Algorithm 8.45: Generates a shuffle proof π relative to encryptions $\mathbf{e}$ and $\tilde{\mathbf{e}}$, which is equivalent to proving knowledge of a permutation ψ and randomizations $\tilde{\mathbf{r}}$ such that $\tilde{\mathbf{e}} = \mathsf{Shuffle}_{pk}(\mathbf{e}, \tilde{\mathbf{r}}, \psi)$. The algorithm implements Wikström's proof of a shuffle [55, 58], except for the fact that the offline and online phases are merged. For the proof verification, see Alg. 8.48. For further background information we refer to Section 5.5.

```
Algorithm: GenPermutationCommitment($\psi, \mathbf{h}$)
Input: Permutation $\psi = (j_1, \ldots, j_N) \in \Psi_N$
       Independent generators $\mathbf{h} = (h_1, \ldots, h_N) \in (\mathbb{G}_q \backslash \{1\})^N$
for $i = 1, \ldots, N$ do
    $r_{j_i} \leftarrow$ GenRandomInteger($q$)                          // see Alg. 4.11
    $c_{j_i} \leftarrow g^{r_{j_i}} h_i \bmod p$
$\mathbf{c} \leftarrow (c_1, \ldots, c_N)$
$\mathbf{r} \leftarrow (r_1, \ldots, r_N)$
return $(\mathbf{c}, \mathbf{r})$                                       // $\mathbf{c} \in \mathbb{G}_q^N$, $\mathbf{r} \in \mathbb{Z}_q^N$
```

Algorithm 8.46: 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. 8.45.

```
Algorithm: GenCommitmentChain($\tilde{\mathbf{u}}$)
Input: Permuted public challenges $\tilde{\mathbf{u}} = (\tilde{u}_1, \ldots, \tilde{u}_N) \in \mathbb{Z}_q^N$
$R_0 \leftarrow 0$, $U_0 \leftarrow 1$
for $i = 1, \ldots, N$ do
    $\hat{r}_i \leftarrow$ GenRandomInteger($q$)                        // see Alg. 4.11
    $R_i \leftarrow \hat{r}_i + \tilde{u}_i R_{i-1} \bmod q$, $U_i \leftarrow \tilde{u}_i U_{i-1} \bmod q$
    $\hat{c}_i \leftarrow g^{R_i} h^{U_i} \bmod p$
$\hat{\mathbf{c}} \leftarrow (\hat{c}_1, \ldots, \hat{c}_N)$
$\hat{\mathbf{r}} \leftarrow (\hat{r}_1, \ldots, \hat{r}_N)$
return $(\hat{\mathbf{c}}, \hat{\mathbf{r}})$                           // $\hat{\mathbf{c}} \in \mathbb{G}_q$, $\hat{\mathbf{r}} \in \mathbb{Z}_q^N$
```

Algorithm 8.47: Generates a commitment chain $\hat{c}_1 \rightarrow \cdots \rightarrow \hat{c}_N$ relative to a vector of public challenges $\tilde{\mathbf{u}}$ and the second public generator $h \in \mathbb{G}_q$. This algorithm is used in Alg. 8.45.

```
Algorithm: CheckShuffleProof$(U, \pi, \mathbf{e}, \tilde{\mathbf{e}}, pk)$
Input: Election event identifier $U \in A_{\mathsf{ucs}}^*$
       Shuffle proof $\pi = (c, s, \mathbf{c}, \hat{\mathbf{c}}) \in \mathbb{Z}_{2^\tau} \times (\mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N) \times \mathbb{G}_q^N \times \mathbb{G}_q^N$,
       $s = (s_1, s_2, s_3, s_4, (\hat{s}_1, \ldots, \hat{s}_N), (\tilde{s}_1, \ldots, \tilde{s}_N))$, $\mathbf{c} = (c_1, \ldots, c_N)$, $\hat{\mathbf{c}} = (\hat{c}_1, \ldots, \hat{c}_N)$
       Encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i) \in \mathbb{G}_q^2$
       Shuffled encryptions $\tilde{\mathbf{e}} = (\tilde{e}_1, \ldots, \tilde{e}_N)$, $\tilde{e}_i = (\tilde{a}_i, \tilde{b}_i) \in \mathbb{G}_q^2$
       Encryption key $pk \in \mathbb{G}_q$
$\mathbf{h} \leftarrow$ GetGenerators$(N, U)$                         // $\mathbf{h} = (h_1, \ldots, h_N)$, see Alg. 8.3
$\mathbf{u} \leftarrow$ GetChallenges$(N, (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}, pk))$              // $\mathbf{u} = (u_1, \ldots, u_N)$, see Alg. 8.5
$\hat{c}_0 \leftarrow h$
$\bar{c} \leftarrow \prod_{i=1}^N c_i / \prod_{i=1}^N h_i \bmod p$
$u \leftarrow \prod_{i=1}^N u_i \bmod q$
$\hat{c} \leftarrow \hat{c}_N / h^u \bmod p$
$\tilde{c} \leftarrow \prod_{i=1}^N c_i^{u_i} \bmod p$
$(\tilde{a}, \tilde{b}) \leftarrow (\prod_{i=1}^N a_i^{u_i} \bmod p, \prod_{i=1}^N b_i^{u_i} \bmod p)$
for $i = 1, \ldots, N$ do
    $\hat{t}_i \leftarrow \hat{c}_i^c \cdot g^{\hat{s}_i} \cdot \hat{c}_{i-1}^{\tilde{s}_i} \bmod p$
$t_1 \leftarrow \bar{c}^c \cdot g^{s_1} \bmod p$
$t_2 \leftarrow \hat{c}^c \cdot g^{s_2} \bmod p$
$t_3 \leftarrow \tilde{c}^c \cdot g^{s_3} \prod_{i=1}^N h_i^{\tilde{s}_i} \bmod p$
$(t_{4,1}, t_{4,2}) \leftarrow (\tilde{a}^c \cdot pk^{-s_4} \prod_{i=1}^N \tilde{a}_i^{\tilde{s}_i} \bmod p, \tilde{b}^c \cdot g^{-s_4} \prod_{i=1}^N \tilde{b}_i^{\tilde{s}_i} \bmod p)$
$t \leftarrow (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), (\hat{t}_1, \ldots, \hat{t}_N))$
$y \leftarrow (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}, \hat{\mathbf{c}}, pk)$
$c' \leftarrow$ GetChallenge$(y, t)$                                        // see Alg. 8.4
return $c = c'$
```

Algorithm 8.48: Checks the correctness of a shuffle proof π generated by Alg. 8.45. The public values are the ElGamal encryptions $\mathbf{e}$ and $\tilde{\mathbf{e}}$ and the public encryption key pk.

```
Algorithm: GetDecryptions$(\mathbf{e}, sk)$
Input: Encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i) \in \mathbb{G}_q^2$
       Decryption key share $sk \in \mathbb{Z}_q$
for $i = 1, \ldots, N$ do
    $c_i \leftarrow b_i^{sk} \bmod p$
$\mathbf{c} \leftarrow (c_1, \ldots, c_N)$
return $\mathbf{c}$                                                  // $\mathbf{c} \in \mathbb{G}_q^N$
```

Algorithm 8.49: Computes the partial decryptions of a given input vector $\mathbf{e}$ of encryptions using a share sk of the private decryption key.

```
Algorithm: GenDecryptionProof$(sk, pk, \mathbf{e}, \mathbf{c})$
Input: Decryption key share $sk \in \mathbb{Z}_q$
       Encryption key share $pk \in \mathbb{G}_q$
       Encryptions $\mathbf{e} = (e_1, \dots, e_N)$, $e_i = (a_i, b_i) \in \mathbb{G}_q^2$
       Partial decryptions $\mathbf{c} = (c_1, \dots, c_N) \in \mathbb{G}_q^N$
$\omega \leftarrow$ GenRandomInteger$(q)$                         // see Alg. 4.11
$t_0 \leftarrow g^\omega \bmod p$
for $i = 1, \dots, N$ do
    $t_i \leftarrow b_i^\omega \bmod p$
$\mathbf{t} \leftarrow (t_0, t_1, \dots, t_N)$
$y \leftarrow (pk, \mathbf{e}, \mathbf{c})$
$c \leftarrow$ GetChallenge$(y, \mathbf{t})$                           // see Alg. 8.4
$s \leftarrow \omega - c \cdot sk \bmod q$
$\pi \leftarrow (c, s)$
return $\pi$                                              // $\pi \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_q$
```

Algorithm 8.50: Generates a decryption proof relative to encryptions $\mathbf{e}$ and partial decryptions $\mathbf{c}$. This is essentially a NIZKP of knowledge of the private key sk satisfying $c_i = b_i^{sk}$ for all input encryptions $e_i = (a_i, b_i)$ and $pk = g^{sk}$. For the proof verification, see Alg. 8.51.

```
Algorithm: CheckDecryptionProof$(\pi, pk, \mathbf{e}, \mathbf{c})$
Input: Decryption proof $\pi = (c, s) \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_q$
       Encryption key share $pk \in \mathbb{G}_q$
       Encryptions $\mathbf{e} = (e_1, \dots, e_N)$, $e_i = (a_i, b_i) \in \mathbb{G}_q^2$
       Partial decryptions $\mathbf{c} = (c_1, \dots, c_N) \in \mathbb{G}_q^N$
$t_0 \leftarrow pk^c \cdot g^s \bmod p$
for $i = 1, \dots, N$ do
    $t_i \leftarrow c_i^c \cdot b_i^s \bmod p$
$\mathbf{t} \leftarrow (t_0, t_1, \dots, t_N)$
$y \leftarrow (pk, \mathbf{e}, \mathbf{c})$
$c' \leftarrow$ GetChallenge$(y, \mathbf{t})$                           // see Alg. 8.4
return $c = c'$
```

Algorithm 8.51: Checks the correctness of a decryption proof π generated by Alg. 8.50. The public values are the encryptions $\mathbf{e}$, the partial decryptions $\mathbf{c}$, and the share pk of the public encryption key.

```
Algorithm: GetCombinedDecryptions($\mathbf{C}$)
Input: Partial decryptions $\mathbf{C} = (c_{ij}) \in \mathbb{G}_q^{N \times s}$
for $i = 1, \ldots, N$ do
    $c_i \leftarrow \prod_{j=1}^{s} c_{ij} \bmod p$
$\mathbf{c} \leftarrow (c_1, \ldots, c_N)$
return $\mathbf{c}$                                              // $\mathbf{c} \in \mathbb{G}_q^N$
```

Algorithm 8.52: Computes the vector $\mathbf{c} = (c_1, \ldots, c_N)$ of combined partial decryptions.

```
Algorithm: GetVotes($\mathbf{e}, \mathbf{c}, \mathbf{c}'$)
Input: Encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i) \in \mathbb{G}_q^2$
       First partial decryptions $\mathbf{c} = (c_1, \ldots, c_N) \in \mathbb{G}_q^N$
       Second partial decryptions $\mathbf{c}' = (c'_1, \ldots, c'_N) \in \mathbb{G}_q^N$
for $i = 1, \ldots, N$ do
    $m_i \leftarrow a_i \cdot (c_i \, c'_i)^{-1} \bmod p$
$\mathbf{m} \leftarrow (m_1, \ldots, m_N)$
return $\mathbf{m}$                                              // $\mathbf{m} \in \mathbb{G}_q^N$
```

Algorithm 8.53: Computes the vector of decrypted plaintext votes $\mathbf{m} = (m_1, \ldots, m_N)$ by deducting the partial decryptions c_i and c'_i from the left-hand sides a_i of the ElGamal encryptions $e_i = (a_i, b_i)$.

```
Algorithm: GetElectionResult($\mathbf{m}, \mathbf{n}, \mathbf{w}$)
Input: Encoded selections $\mathbf{m} = (m_1, \ldots, m_N) \in \mathbb{G}_q^N$
       Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$
       Counting circles $\mathbf{w} = (w_1, \ldots, w_{N_E}) \in (\mathbb{N}^+)^{N_E}$
$n \leftarrow \sum_{j=1}^{t} n_j$, $w \leftarrow \max_{i=1}^{N_E} w_i$
$\mathbf{p} \leftarrow$ GetPrimes($n + w$)                    // $\mathbf{p} = (p_0, \ldots, p_{n+w})$, see Alg. 8.1
for $i = 1, \ldots, N$ do
    for $k = 1, \ldots, n$ do
        if $p_k \mid m_i$ then
            $v_{ik} \leftarrow 1$
        else
            $v_{ik} \leftarrow 0$
    for $j = 1, \ldots, w$ do
        if $p_{n+j} \mid m_i$ then
            $w_{ij} \leftarrow 1$
        else
            $w_{ij} \leftarrow 0$
    if $p_0 \mid m_i$ then
        $u_i \leftarrow -1$
    else
        $u_i \leftarrow 1$
$\mathbf{u} = (u_1, \ldots, u_N)$, $\mathbf{V} \leftarrow (v_{ik})_{N \times n}$, $\mathbf{W} \leftarrow (w_{ij})_{N \times w}$
$ER \leftarrow (\mathbf{u}, \mathbf{V}, \mathbf{W})$
return $ER$                    // $ER \in \{-1, 1\}^N \times \mathbb{B}^{N \times n} \in \mathbb{B}^{N \times w}$
```

Algorithm 8.54: Computes the election result from the products of encoded selections $\mathbf{m} = (m_1, \ldots, m_N)$ by retrieving the prime factors of each m_i. Each value $v_{ik} = 1$ represents somebody's vote for a specific candidate $k \in \{1, \ldots, n\}$.

```
Algorithm: GetInspection($v, \mathbf{P}, \mathbf{a}, C$)
Input: Voter index $v \in \{1, \ldots, N_E\}$
       Points $\mathbf{P} = (p_{ij}) \in (\mathbb{Z}_{\hat{q}}^2)^{N_E \times n}$
       Abstention codes $\mathbf{a} = (A_1, \ldots, A_{N_E}) \in (\mathcal{B}^{L_{FA}})^{N_E}$
       Confirmation list $C = \langle (v_i, \gamma_i) \rangle_{i=0}^{N_C - 1}$, $v_i \in \{1, \ldots, N_E\}$
if $(v, \cdot) \in C$ then
    $\mathbf{p}_v \leftarrow$ GetRow($\mathbf{P}, v$)
    $I_v \leftarrow$ Truncate(RecHash$_L$($\mathbf{p}_v$), $L_{FA}$)                    // see Alg. 4.15
else
    $I_v \leftarrow A_v$
return $I_v$                    // $I_v \in \mathcal{B}^{L_{FA}}$
```

Algorithm 8.55: Selects and returns the share of the inspection code.

```
Algorithm: GetInspectionCode($\mathbf{i}$)
Input: Inspections $\mathbf{i} = (I_1, \ldots, I_s) \in (\mathcal{B}^{L_{FA}})^s$
$I \leftarrow \bigoplus_{j=1}^{s} I_j$
$IC \leftarrow \mathsf{ByteArrayToString}(I, A_{FA})$                    // see Alg. 4.9
return $IC$                                                              // $IC \in A_{FA}^{\ell_{FA}}$
```

Algorithm 8.56: Computes a inspection code of a given voter by combining corresponding values I_j received from the authorities.

```
Algorithm: CheckInspectionCode($pb, FC, AC, IC$)
Input: Participation bit $pb \in \mathbb{B}$
       Printed abstention code $FC \in A_{FA}^{\ell_{FA}}$
       Printed finalization code $AC \in A_{FA}^{\ell_{FA}}$
       Displayed inspection code $IC \in A_{FA}^{\ell_{FA}}$
if $pb$ then
    return $IC = FC$
else
    return $IC = AC$
```

Algorithm 8.57: Checks if the displayed inspection code IC matches with the finalization or abstention codes from the voting card. The participation bit pb indicates whether the voter has submitted a ballot and therefore participated in the election. Note that this algorithm is executed by humans.

8.5. Channel Security

The additional protocol steps to achieve the necessary channel security have already been discussed in Section 7.4. 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 shown in the protocol illustrations of Chapter 7, but messages to be signed or encrypted are depicted as $[m]$ and $[\![m]\!]$, respectively, and the list of all necessary signatures is given in Table 7.2. In Table 8.5, we summarize the contents of this list.

Nr.	Algorithm	Called by	Protocols
8.58	$\mathsf{GenSignature}(sk, m, aux)$	Election administrator	7.2, 7.9
		Election authority	7.1, 7.2, 7.3, 7.5, 7.6, 7.7, 7.8, 7.10
8.59	$\mathsf{CheckSignature}(\sigma, pk, m, aux)$	Administrator	7.9
		7.1, Election authority	7.2, 7.8
		Printing authority	7.3
		Voting client	7.4, 7.5, 7.6, 7.10
8.60	$\mathsf{GenCiphertext}(pk, M)$	Election authority	7.3
8.61	$\mathsf{GetPlaintext}(sk, e)$	Printing authority	7.3
8.62	$\mathsf{GenSchnorrKeyPair}()$	All	PKI setup

Table 8.5.: Overview of algorithms used to establish channel security.

In the signature generation and verification algorithms listed above, which implement the Schnorr signature scheme over $\mathbb{G}_{\hat{q}}$ from Section 5.6, the message space is not further specified. We call $\mathsf{GetChallenge}$ (Alg. 8.4) and therefore $\mathsf{RecHash}_L$ (Alg. 4.15) as a sub-routine for computing a hash value that depends on the message m. Therefore, the message space supported by Alg. 4.15 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.

In case of the encryption and decryption algorithms, which implement the hybrid encryption scheme based on the key-encapsulation mechanism of Section 5.7, we assume the availability of an AES-128 block cipher implementation in combination with the GCM mode of operation.[2] For a 128-bit key $K \in \mathcal{B}^{16}$ (16 bytes) and a standard 96-bit GCM initialization vector $IV \in \mathcal{B}^{12}$ (12 bytes), we use $C \leftarrow \mathsf{AESEncrypt}(K, IV, M)$ to denote the encryption of a plaintext $M \in \mathcal{B}^*$ into a ciphertext $C \in \mathcal{B}^*$ of equal length, and $M \leftarrow \mathsf{AESDecrypt}(K, IV, C)$ for the corresponding decryption.

As proposed in Section 7.4, we implement the encrypt-then-sign approach. For encrypting a message m of an arbitrary type, we assume the existence of algorithms $\mathsf{encode}(m)$

[2]Using 128-bit AES keys is consistent with all security levels of Chapter 10. In GCM mode, the block cipher is transformed into a stream cipher, and therefore no padding is needed. When using AES-GCM, a 96-bit random IV is generally recommended. By adding an authentication tag to the ciphertext, AES-GCM implements authenticated encryption, which provides security against chosen-ciphertext attacks. Both AES and GCM are well-established NIST standards [1, 24].

```
Algorithm: GenSignature($sk, m, aux$)
Input: Signature key $sk \in \mathbb{Z}_{\hat{q}}$
       Message $m \in M$, $M$ unspecified
       Auxiliary input $aux \in X$, $X$ unspecified
$pk \leftarrow \hat{g}^{sk} \bmod \hat{p}$
$\omega \leftarrow$ GenRandomInteger($\hat{q}$)                    // see Alg. 4.11
$t \leftarrow \hat{g}^{\omega} \bmod \hat{p}$
if $aux = \varnothing$ then
    $y \leftarrow (pk, m)$
else
    $y \leftarrow (pk, m, aux)$
$c \leftarrow$ GetChallenge($y, t$)                              // see Alg. 8.4
$s \leftarrow \omega - c \cdot sk \bmod \hat{q}$
$\sigma \leftarrow (c, s)$
return $\sigma$                                                 // $\sigma \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_{\hat{q}}$
```

Algorithm 8.58: Computes a Schnorr signature for a given message m and signature key sk. For the verification of this signature, see Alg. 8.59. Using tuples $m = (m_1, \dots, m_r)$ as input, the algorithm can be used for signing multiple messages simultaneously.

```
Algorithm: CheckSignature($\sigma, pk, m, aux$)
Input: Signature $\sigma = (c, s) \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_{\hat{q}}$
       Verification key $pk \in \mathbb{G}_{\hat{q}}$
       Message $m \in M$, $M$ unspecified
       Auxiliary input $aux \in X$, $X$ unspecified
$t \leftarrow pk^c \cdot \hat{g}^s \bmod \hat{p}$
if $aux = \varnothing$ then
    $y \leftarrow (pk, m)$
else
    $y \leftarrow (pk, m, aux)$
$c' \leftarrow$ GetChallenge($y, t$)                    // see Algs. 4.5 and 4.15
return $c = c'$
```

Algorithm 8.59: Verifies a Schnorr signature $\sigma = (c, s)$ generated by Alg. 8.58 using a given public verification key pk.

and decode(M) for converting such messages into byte arrays $M \in \mathcal{B}^*$ and back, i.e., decode(encode(m)) $= m$ must hold for all possible m. The exact shape of these conversion algorithms has no impact on the security of the encryption scheme, as long as they satisfy the above condition.

In Algs. 8.60 and 8.62, the key-encapsulation mechanism is defined over $\mathbb{G}_{\hat{q}}$, i.e., we use the same modular group for signing and encrypting. This implies that the same key generation algorithm can be used for generating signature and encryption keys. This simplifies the PKI setup of the whole system. Note that Alg. 8.62 is identical to Alg. 8.6, except for the group parameters.

```
Algorithm: GenCiphertext($pk, M$)
Input: Encryption key $pk \in \mathbb{G}_{\hat{q}}$
       Message $M \in \mathcal{B}^*$
$r \leftarrow$ GenRandomInteger($\hat{q}$)                        // see Alg. 4.11
$ek \leftarrow \hat{g}^r \bmod \hat{p}$
$K \leftarrow$ RecHash$_{16}$($pk^r \bmod \hat{p}$)                   // see Alg. 4.15
$IV \leftarrow$ RandomBytes(12)
$C \leftarrow$ AESEncrypt($K, IV, M$)
$e \leftarrow (ek, IV, C)$
return $e$                                       // $e \in \mathbb{G}_{\hat{q}} \times \mathcal{B}^{12} \times \mathcal{B}^*$
```

Algorithm 8.60: Computes the hybrid encryption of given string M using a public encryption key pk. Alg. 8.61 is the corresponding decryption algorithm.

```
Algorithm: GetPlaintext($sk, e$)
Input: Decryption key $sk \in \mathbb{Z}_{\hat{q}}$
       Ciphertext $e = (ek, IV, C)$, $ek \in \mathbb{G}_{\hat{q}}$, $IV \in \mathcal{B}^{12}$, $C \in \mathcal{B}^*$
$K \leftarrow$ RecHash$_{16}$($ek^{sk} \bmod \hat{p}$)                  // see Alg. 4.15
$M \leftarrow$ AESDecrypt($K, IV, C$)
return $M$                                             // $M \in \mathcal{B}^*$
```

Algorithm 8.61: Decrypts a ciphertext into a string M using a private key sk.

```
Algorithm: GenSchnorrKeyPair()
$sk \leftarrow$ GenRandomInteger($\hat{q}$)                       // see Alg. 4.11
$pk \leftarrow \hat{g}^{sk} \bmod \hat{p}$
return $(sk, pk)$                                // $(sk, pk) \in \mathbb{Z}_{\hat{q}} \times \mathbb{G}_{\hat{q}}$
```

Algorithm 8.62: Generates a random key pair $(sk, pk) \in \mathbb{Z}_{\hat{q}} \times \mathbb{G}_{\hat{q}}$, which can be used for both Schnorr signatures (Algs. 8.58 and 8.59) and hybrid encryptions (Algs. 8.60 and 8.61).

9. Write-Ins

In some cantons, voters are allowed to vote for arbitrary candidates, i.e., even for candidates not included in the official candidate list. They can do so by writing the first and last names of the chosen candidates (together with other identifying information) onto the ballot. Such votes for arbitrary candidates are commonly called *write-ins* and corresponding elections are called *write-in elections*. Practical experience from paper-based write-in elections in Switzerland shows that write-ins are not submitted very frequently. Usually, they are irrelevant for determining the winners of an election. Nevertheless, to comply with electoral laws, write-ins must be supported equally by all voting channels.

The challenge of implementing write-ins into the CHVote protocol—or into cryptographic voting protocols in general—is to provide the same level of cast-as-intended verifiability as for regular votes. At the moment, we are not aware of any technique that would offer cast-as-intended verifiability for write-ins as required by VEleS in combination with reasonable usability. Given that the number of write-ins are usually insignificantly low in real elections, the Swiss Federal Chancellery has approved a restricted level of cast-as-intended verifiability along the following procedure. If a voter submits one or multiple write-ins, then verification codes are displayed to the voter for checking that the correct number of write-ins have been submitted. Therefore, if a malware-infected computer is used for vote casting, then changing the voter's intention to submit a certain amount of write-ins would lead to mismatched verification codes, which could be detected by the voter by the standard verification procedure. This measure, however, does not prevent the malware from changing the actual content of a write-in, for example replacing the selected candidate name by any other name. At this point, cast-as-intended verifiability is limited, but only for the small percentage of voters submitting write-ins. This limits the scalability of corresponding attacks to a degree that seems to be compliant with the given legal framework and the requirements of the Federal Chancellery.

9.1. General Protocol Design

Due to the fact that write-ins are only allowed in some cantons, we decided to describe their inclusion as an optional protocol add-on. In this chapter, we provide all the necessary information for implementing write-ins on top of an existing implementation of the basic protocol. To maximize the compatibility between the basic and the extended protocol, we implemented some minor modifications to the basic protocol and to some algorithms. Unfortunately, one incompatibility between default votes and write-in candidates has resisted our attempts of providing a perfectly smooth integration. We circumvent this problem by switching off the mechanism for handling default votes in elections with write-ins. Our solution to the privacy problem encountered in [17, Recommendation 10.6] is therefore ineffective in election with write-ins. Further information on this issue is given at other places in this chapter.

9.1.1. System Parameters

In the given context, a single write-in consists of two text fields of ℓ_W characters from an alphabet A_W. We expect A_W to contain mainly lower-case and upper-case Latin letters with optional accents and special letters from common Western European languages. The size of such an alphabet size is approximately $|A_W| = 130$ characters, i.e., slightly more than 7 bits are necessary for representing a single character $c \in A_W$ (without applying compression algorithms). This means that for $\ell_W = 100$, approximately 1400 bits are necessary for representing the two text fields of a write-in. As we will see below, we will encrypt write-ins individually using the multi-recipient ElGamal encryption scheme. In security level $\lambda = 2$, the group size of 2047 bits will therefore be sufficiently large for this purpose (see Section 10.1).

For encoding a write-in as a group element of $\mathbb{G}_q$, a particular *padding character* $c_W \notin A_W$ will be used for extending the two strings as entered by the voter to the maximal length of ℓ_W characters. To ensure that the encoding is injective, it is important that c_W is not a regular character from A_W. Therefore, the general constraint to consider when choosing ℓ_W and A_W is $(|A_W| + 1)^{2\ell_W} \leqslant q$. We will see in Alg. 9.1 that if the rank of c_W in the extended alphabet $A_W \cup \{c_W\}$ is 0, then encoding an empty write-in $S = ("","")$ will lead to the identity element $1 \in \mathbb{G}_q$. We refrain from introducing $rank_{A_W \cup \{c_W\}}(c_W) = 0$ as a general constraint, but it may help to simplify some algorithms in an actual implementation.

Cantons allowing write-ins do so on all political levels. It is therefore possible, that multiple write-in elections are conducted simultaneously within a single election event. Therefore, let $\mathbf{z} = (z_1, \ldots, z_t) \in \mathbb{B}^t$ denote Boolean vector of length t, where $z_j = 1$ means that write-in candidates are allowed in election $j \in \{1, \ldots, t\}$. In all practical use cases, the allowed number of write-ins in a single k_j-out-of-n_j write-in election corresponds to the number of allowed selections k_j, i.e., voters may choose up to k_j many write-ins in election j. We propose to handle these cases by adding k_j special *write-in candidates* to the candidate list of every election, which gives a total of $z = \sum_{j=1}^{t} k_j z_i$ special write-in candidates. Write-in candidates are treated in the same way as regular candidates, including the generation of corresponding verification codes. Recall that blank votes are handled similarly (see Section 2.2.3). Write-in candidates are specified by a Boolean vector $\mathbf{v} = \{v_1, \ldots, v_n\} \in \mathbb{B}^n$ of length n, where $v_i = 1$ means that candidate i is a write-in candidate. As in Section 6.3.2, let

$$I_j = \{\sum_{i=1}^{j-1} n_i + 1, \ldots, \sum_{i=1}^{j-1} n_i + n_j\}$$

denote the set of indices of all candidates of a given election j. To ensure that the right amount of write-in candidates is available in every election, $k_j z_j = \sum_{i \in I_j} v_i$ must hold for every election $j \in \{1, \ldots, t\}$. By adding $\mathbf{z}$ and $\mathbf{v}$ to the election parameters from Section 6.3.2, we obtain the following extended tuple:

$$EP = (U, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{w}, \mathbf{z}, \mathbf{v}),$$

For an election event of size t with election parameters $\mathbf{k}$, $\mathbf{E}$, and $\mathbf{z}$, we can compute the total number

$$z'_i = \sum_{j=1}^{t} e_{ij} k_j z_j \leqslant k'_i$$

of allowed write-in candidates of voter $i \in \{1, \ldots, N_E\}$ and the maximum number

$$z_{\max} = \max_{i=1}^{N_E} z'_i = \max_{i=1}^{N_E} \sum_{j=1}^{t} e_{ij} k_j z_j \leqslant z$$

of allowed write-in candidates over the whole electorate. Since this value depends on $\mathbf{E}$, it can not be determined by the voting client receiving only the relevant row $\mathbf{e}_i$ of $\mathbf{E}$. We therefore add $z_{\max}$ together with $\mathbf{z}$ and $\mathbf{v}$ to the voter-specific voting parameters from Section 6.3.2:

$$VP_i = (U, \mathbf{c}, D_i, \mathbf{e}, \mathbf{n}, \mathbf{k}, \mathbf{e}_i, w_i, \mathbf{z}, \mathbf{v}, z_{\max}).$$

In our approach, every voter will add a multi-recipient ElGamal encryption e' of size $z'_i \leqslant z_{\max}$ to the ballot (see Section 5.1.3), even if the actual number of chosen write-ins is smaller than z'_i. Some of the encrypted write-ins will therefore be empty, i.e., the voter needs to generate a NIZKP to ensure that e' contains the right amount of empty write-ins. For establishing a consistent input to the mix-net over all voters, $z_{\max} - z'_i$ additional trivial encryptions are later added to e', thus enlarging the size of e' from z'_i to $z_{\max}$. For this reason, $z_{\max}$ is a very important additional election parameter of a given election event. Keeping it as small as possible is crucial for the efficiency of the mix-net. In the election use cases that we have to consider, we expect upper bounds $z_{\max} = 14$ for the whole election event and $k_j = 7$ for an individual write-in election. In an election event without write-ins, $\mathbf{z} = (0, \ldots, 0)$ implies $z = 0$ and therefore $z_{\max} = 0$.

Parameters		Constraints
A_W	Alphabet of permitted characters	$\lvert A_W \rvert \geqslant 2$
c_W	Padding character	$c_W \notin A_W$
ℓ_W	Length of write-in text fields	$(\lvert A_W \rvert + 1)^{2\ell_W} \leqslant q$
$\mathbf{z} = (z_1, \ldots, z_t)$	Write-in elections	$z_j \in \mathbb{B}$
$\mathbf{v} = (v_1, \ldots, v_n)$	Write-in candidates	$v_i \in \mathbb{B}$, $k_j z_j = \sum_{i \in I_j} v_i$
$z_{\max}$	Maximum number of write-ins	$z_{\max} = \max_{i=1}^{N_E} \sum_{j=1}^{t} e_{ij} k_j z_j$

Table 9.1.: List of additional system and election parameters for handling write-ins. We assume that the first three parameters are fixed and publicly known to every party participating in the protocol in addition to the security parameters given in Table 6.1. The other two parameters are additional election parameters, which are defined by the election administrator along with the parameters from Table 6.2.

9.1.2. Technical Preliminaries

Implementing the extended protocol requires a few additional cryptographic techniques to achieve the desired security. In order to facilitate the exposition of the extended protocol and corresponding pseudo-code algorithms, we introduce them beforehand.

9.1.2.1. Encoding and Encrypting Write-Ins

For improved performance, we use the multi-recipient ElGamal encryption scheme from Section 5.1.3 for encrypting write-ins. Let $A_W^L = \bigcup_{\ell \in L} A_W^\ell$ denote the set of all strings of length $\ell \in L$, for example $A_W^{\{0,\ldots,\ell_W\}}$ for strings of length ℓ_W or less, i.e., including the empty string `""` $\in A_W^0$. Furthermore, let $\Delta : \mathcal{W} \rightarrow \mathbb{G}_q$ denote an injective mapping from the set

$$\mathcal{W} = A_W^{\{0,\ldots,\ell_W\}} \times A_W^{\{0,\ldots,\ell_W\}}$$

of pairs of strings of length ℓ_W or less into $\mathbb{G}_q$. If $\mathbf{s} = (S_1, \ldots, S_z)$ denotes a vector of write-ins $S_j = (S_{j,1}, S_{j,2}) \in \mathcal{W}$, then a single random value $r \in \mathbb{Z}_q$ is sufficient for encrypting $\mathbf{s}$ into

$$e = \mathsf{Enc}_{\mathbf{pk}}((\Delta(S_1), \ldots, \Delta(S_z)), r) = ((\Delta(S_1) \cdot pk_1^r, \ldots, \Delta(S_z) \cdot pk_z^r), g^r) \in \mathbb{G}_q^z \times \mathbb{G}_q.$$

In this context, $\mathbf{pk} = (pk_1, \ldots, pk_z)$ denotes a vector of z different ElGamal public keys. The resulting encryption $e = (\mathbf{a}, b)$ consists of a vector $\mathbf{a} = (a_1, \ldots, a_z) \in \mathbb{G}_q^z$ and a single value $b \in \mathbb{G}_q$. Note that every pair (a_j, b) defines an ordinary ElGamal encryption for the corresponding public key pk_j, which can be decrypted in the usual way using sk_j.

The injective mapping Δ is constructed as a composition of the following three injective mappings:

$$\begin{aligned}
&\Delta_1 : \mathcal{W} \rightarrow (A_W \cup \{c_W\})^{2\ell_W}, \\
&\Delta_2 : (A_W \cup \{c_W\})^{2\ell_W} \rightarrow \mathbb{Z}_q, \\
&\Delta_3 : \mathbb{Z}_q \rightarrow \mathbb{G}_q.
\end{aligned}$$

For Δ_1, we extend the two input strings to the maximal length ℓ_W using the padding character c_W. The resulting padded strings are concatenated to a single string of length $2\ell_W$. For Δ_2, we use the string-to-integer conversion method from Section 4.2.2, and for Δ_3, we use the property that $\mathbb{G}_q$ corresponds to the set $\{1, 4, 9, \ldots, q^2 \bmod p\}$ of quadratic residues modulo p. If $y = \Delta(S) = \Delta_3(\Delta_2(\Delta_1(S)))$ denotes the group element obtained from applying the composed mapping to a single write-in S, then $S = \Delta^{-1}(y) = \Delta_1^{-1}(\Delta_2^{-1}(\Delta_3^{-1}(y)))$ defines the reverse process from y to S. Further details are given in Algs. 9.1 and 9.2.

9.1.2.2. Proving Vote Validity

In an election event of size t, let all voters have unrestricted voting rights.[1] This implies that every participating voter $i \in \{1, \ldots, N_E\}$ submits exactly $k = k'_i = \sum_{j=1}^t k_j$ votes and up to $z = z'_i = \sum_{j=1}^t k_j z_j$ write-ins. Furthermore, assuming that write-ins are allowed in all elections, i.e., $\mathbf{z} = (1, \ldots, 1)$, then the vector $\mathbf{s} = (s_1, \ldots, s_k)$ of selected candidates $s_j \in \{1, \ldots, n\}$ and the vector $\mathbf{s}' = (S_1, \ldots, S_z)$ of write-ins $S_j \in \mathcal{W}$ are of equal length $z = k$. This allows us to write $\mathbf{s} = (s_1, \ldots, s_z)$ as a vector of length z, and similarly

$$\begin{aligned}
\mathbf{e} &= (\mathsf{Enc}_{pk}(\Gamma(s_1), r_1), \ldots, \mathsf{Enc}_{pk}(\Gamma(s_z), r_z)) = ((a_1, b_1), \ldots, (a_z, b_z)), \\
e' &= \mathsf{Enc}_{\mathbf{pk}'}((\Delta(S_1), \ldots, \Delta(S_z)), r') = ((a'_1, \ldots, a'_z), b').
\end{aligned}$$

[1]In the general case, when some values in the eligibility matrix are set to 0, the actual numbers of selections and allowed write-ins will be slightly smaller for some voters. However, this does not affect the principle behind the technique presented in this subsection. We hide this additional technical complication here for making the presentation more comprehensible.

Note that in the context of the OT protocol from Section 5.3 and in Algs. 8.21 and 8.23, the vector $\mathbf{e} = (e_1, \dots, e_z)$ of pairs $e_j = (a_j, b_j)$ is denoted as a vector $\mathbf{a} = (a_1, \dots, a_z)$ of pairs $a_j = (a_{j,1}, a_{j,2})$. For the purpose of disambiguation and improved clarity, the preferred notation in the current context is $\mathbf{e} = ((a_1, b_1), \dots, (a_z, b_z))$.

By submitting $\mathbf{e}$ as part of a ballot $\alpha = (\hat{x}, \mathbf{e}, \pi)$ according to Alg. 8.21, the basic protocol guarantees that exactly k_j different valid selections have been chosen for every election $j \in \{1, \dots, t\}$ and thus that $\mathbf{e}$ contains a valid vote. Clearly, this is not necessarily true if e' is submitted along with α in an extended ballot $\alpha' = (\hat{x}, \mathbf{e}, \pi, e')$, because e' may contain non-empty write-ins even if no write-in candidates have been selected in $\mathbf{s}$.[2] To avoid that such invalid ballots are accepted by the election authorities, we propose to add a second NIZKP π' to the ballot, thus to submit a quintuple $\alpha' = (\hat{x}, \mathbf{e}, \pi, e', \pi')$. The exact purpose of π' will be discussed below. Note that in principle, π and π' could be merged into a single composed proof, but in the spirit of implementing write-ins as an add-on to the current protocol, we prefer to keep them separate.

To start with, consider the simplest case of an election event with only $t = 1$ election. Without introducing any restrictions, each of the voter's selections $\mathbf{s} = (s_1, \dots, s_z)$ may be one of the z write-in candidates. If this is the case for selection s_j, then S_j can take any of the admitted pairs of strings from $\mathcal{W}$. Otherwise, if s_j is a selection for a regular candidate, we require that S_j is set to the pair $\mathcal{E} = (\texttt{""}, \texttt{""})$ of empty strings. The idea of this particular choice is that not submitting a write-in means to leave the two write-in text fields empty. Note that $v_{s_j} = 1$ means that a write-in candidate has been selected and $v_{s_j} = 0$ that a regular candidate has been selected.

Let $\mathbf{p} = (p_1, \dots, p_z)$ be the vector of encoded write-in candidates $p_j = \Gamma(i)$ for all $i \in \{1, \dots, n\}$ satisfying $v_i = 1$, such that the elements appear in $\mathbf{p}$ in ascending order. Furthermore, let $\varepsilon = \Delta(\mathcal{E}) \in \mathbb{G}_q$ denote the encoded pair of empty strings. The general rule that guarantees that $e_j = (a_j, b_j)$ together with $e'_j = (a'_j, b')$ is a valid vote can then be expressed as follows:

$$\begin{aligned}
&\left[e'_j \neq \mathsf{Enc}_{pk'_j}(\varepsilon, r') \Rightarrow e_j \in \{\mathsf{Enc}_{pk}(p_1, r_j), \cdots, \mathsf{Enc}_{pk}(p_z, r_j)\}\right] \\
&\quad\equiv \left[e_j = \mathsf{Enc}_{pk}(p_1, r_j) \vee \cdots \vee e_j = \mathsf{Enc}_{pk}(p_z, r_j) \vee e'_j = \mathsf{Enc}_{pk'_j}(\varepsilon, r')\right].
\end{aligned}$$

The intuition of this rule is to exclude the case where a regular candidate is selected together with a non-empty write-in. As this rule must hold for all encryptions, vote validity relative to $\mathbf{e}$ and e' can be proven by the following non-interactive CNF-proof:

$$NIZKP\left[(\mathbf{r}, r') : \bigwedge_{j=1}^{z} e_j = \mathsf{Enc}_{pk}(p_1, r_j) \vee \cdots \vee e_j = \mathsf{Enc}_{pk}(p_z, r_j) \vee e'_j = \mathsf{Enc}_{pk'_j}(\varepsilon, r')\right].$$

The problem with this proof is its quadratic size relative to z. Therefore, generating and verifying such proofs gets very inefficient for large z. We therefore propose a more efficient proof of linear size, which slightly diminishes the flexibility of submitting any possible combination of write-in candidates, but without restricting the voter's right to submit an arbitrary amount—any number between 0 and z—of write-ins. The general idea is to enforce that if e'_j contains a non-empty write-in, then e_j must be an encryption of the j-th write-in

[2]To the best of our knowledge, ballots containing such invalid combinations of selections and write-ins are invalid, see Federal Act on Political Rights, Art.12.1.

candidate p_j, i.e., write-in candidates other than p_j are not allowed for e_j in this case. This translates into the following rule relative to e_i and e'_j,

$$\big[e'_j \neq \mathsf{Enc}_{pk'_j}(\varepsilon, r') \Rightarrow e_j = \mathsf{Enc}_{pk}(p_j, r_j)\big] \;\equiv\; \big[e_j = \mathsf{Enc}_{pk}(p_j, r_j) \vee e'_j = \mathsf{Enc}_{pk'_j}(\varepsilon, r')\big],$$

which is obviously less complex but slightly more restrictive than the rule given above. It leads to the CNF-proof

$$\mathit{NIZKP}\left[(\mathbf{r}, r') : \bigwedge_{j=1}^{z} e_j = \mathsf{Enc}_{pk}(p_j, r_j) \vee e'_j = \mathsf{Enc}_{pk'_j}(\varepsilon, r')\right],$$

which contains a total of $2z$ atomic proofs of encrypted plaintexts (see Section 5.4.1). This is the proof π' that we will add to the ballot to guarantee its validity. Note that for $m = z$ and $n = 2$, it can be seen as a special case of the general CNF-proof

$$\begin{aligned} &\mathit{NIZKP}\left[(\mathbf{r}^*) : \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n} e^*_{ij} = \mathsf{Enc}_{pk^*_{ij}}(m^*_{ij}, r^*_i)\right] \\ &= \mathit{NIZKP}\left[(\mathbf{r}^*) : \bigwedge_{i=1}^{m} \bigvee_{j=1}^{n} \left(\frac{a^*_{ij}}{m^*_{ij}} = (pk^*_{ij})^{r^*_i} \wedge b^*_{ij} = g^{r^*_i}\right)\right] \end{aligned}$$

consisting of arbitrary public keys $\mathbf{PK}^* = (pk^*_{ij}) \in \mathbb{G}_q^{m\times n}$, messages $\mathbf{M}^* = (m^*_{ij}) \in \mathbb{G}_q^{m\times n}$, and encryptions $\mathbf{E}^* = (e^*_{ij}) \in (\mathbb{G}_q^2)^{m\times n}$ with $e^*_{ij} = (a^*_{ij}, b^*_{ij})$. The secret input to this proof is a vector of randomizations $\mathbf{r}^* = (r^*_1, \ldots, r^*_n) \in \mathbb{Z}_q^n$, which we construct by selecting $r^*_j = r_j$ if s_j is a write-in candidate and $r^*_j = r'$ if s_j is a regular candidate. For example for $z = 3$, we get 3-by-2 matrices

$$\mathbf{PK}^* = \begin{pmatrix} pk & pk'_1 \\ pk & pk'_2 \\ pk & pk'_3 \end{pmatrix},\; \mathbf{M}^* = \begin{pmatrix} p_1 & \varepsilon \\ p_2 & \varepsilon \\ p_3 & \varepsilon \end{pmatrix},\; \mathbf{E}^* = \begin{pmatrix} e_1 & e'_1 \\ e_2 & e'_2 \\ e_3 & e'_3 \end{pmatrix},$$

and a vector $\mathbf{r}^* = (r^*_1, r^*_2, r^*_3)$ of randomizations $r^*_j \in \{r_j, r'\}$, for example $\mathbf{r}^* = (r', r_2, r_3)$ if two write-ins have been selected for $j = 2$ and $j = 3$.

General CNF-proofs of this kind can be generated and verified using the techniques discussed in Section 5.4.2. This leads to the methods presented in Algs. 9.13 and 9.16. Note that exactly the same type of proof can be used in general election events with $t \geqslant 1$ elections. For $t = 3$, $\mathbf{k} = (2, 3, 2)$, and $\mathbf{z} = (1, 1, 1)$, for example, we get $z = 7$ and corresponding 7-by-2 input matrices $\mathbf{PK}^*$, $\mathbf{M}^*$, and $\mathbf{E}^*$, and a randomization vector $\mathbf{r}^*$ of length 7.

To map the ballot generation in the most general case into this particular proof pattern, some preparatory work is needed. For example, encryptions of elections not permitting write-ins must be sorted out and the input matrices must be composed. This preparatory work is the main purpose of Algs. 9.12 and 9.15, which then call Algs. 9.13 and 9.16 as respective sub-routines.

9.1.2.3. Cryptographic Shuffle

In the presence of write-ins, the cryptographic shuffle performed by the mix-net must be extended to a new type of encryption. Recall that two ElGamal encryptions are included in each ballot, a regular ElGamal encryption (a, b) of the selected candidates, which is derived from the OT query $\mathbf{a}$ included in the ballot, and a multi-recipient ElGamal encryption $(\mathbf{a}', b')$ of the chosen write-ins. For making the processing through the mix-net as simple as possible, we first normalize the size of $\mathbf{a}'$ from $z'_v = |\mathbf{a}'|$ to the maximal size $z_{\max}$ by appending $z_{\max} - z'_v$ identity elements $1 \in \mathbb{G}_q$ to it.[3] In the following discussion, we assume that $\mathbf{a}'$ is already normalized and that $z = z_{\max}$ denotes its size. This allows us to merge the two encryptions into a single tuple $e = (a, b, \mathbf{a}', b') \in \mathbb{E}_z$, where $\mathbb{E}_z = \mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q^z \times \mathbb{G}_q$ denotes the combined ciphertext space. The new input to the mix-net is therefore a vector $\mathbf{e} = (e_1, \ldots, e_N)$ of such normalized and combined ElGamal encryptions, all of them consisting of exactly $z + 3$ group elements. We call them *augmented ElGamal encryptions.*

To shuffle a vector of augmented ElGamal encryptions, the first technical problem to look at is re-encryption. Note that $e = (a, b, (a'_1, \ldots, a'_z), b')$ contains commitments to two randomizations $r \in \mathbb{Z}_q$ (in $b = g^r$) and $r' \in \mathbb{Z}_q$ (in $b' = g^{r'}$).[4] Therefore, re-encrypting e requires picking two fresh randomizations $\tilde{r}$ and $\tilde{r}'$ from $\mathbb{Z}_q$, which can then be used to encrypt the identity element $1_z = (1, (1, \ldots, 1)) \in \mathbb{G}_q \times \mathbb{G}_q^z$ into

$$\mathsf{Enc}_{pk,\mathbf{pk}'}(1_z, \tilde{r}, \tilde{r}') = (pk^{\tilde{r}}, g^{\tilde{r}}, ((pk'_1)^{\tilde{r}'}, \ldots, (pk'_z)^{\tilde{r}'}), g^{\tilde{r}'}).$$

The re-encryption of an augmented ElGamal encryption $e \in \mathbb{E}_z$ can then be defined as follows:

$$\begin{aligned}
\tilde{e} &= \mathsf{ReEnc}_{pk,\mathbf{pk}'}(e, \tilde{r}, \tilde{r}') = e \cdot \mathsf{Enc}_{pk,\mathbf{pk}'}(1_z, \tilde{r}, \tilde{r}') \\
&= (a, b, (a'_1, \ldots, a'_z), b') \cdot (pk^{\tilde{r}}, g^{\tilde{r}}, ((pk'_1)^{\tilde{r}'}, \ldots, (pk'_z)^{\tilde{r}'}), g^{\tilde{r}'}) \\
&= (a \cdot pk^{\tilde{r}}, b \cdot g^{\tilde{r}}, (a'_1 \cdot (pk'_1)^{\tilde{r}'}, \ldots, a'_z \cdot (pk'_z)^{\tilde{r}'}), b' \cdot g^{\tilde{r}'}) \\
&= (\tilde{a}, \tilde{b}, (\tilde{a}'_1, \ldots, \tilde{a}'_z), \tilde{b}').
\end{aligned}$$

Using this extended re-encryption method, shuffling a vector $\mathbf{e} = (e_1, \ldots, e_N)$ of augmented ElGamal encryptions works in the same way as described in Section 5.5 for regular ElGamal encryptions, except that two randomization lists $\tilde{\mathbf{r}} = (\tilde{r}_1, \ldots, \tilde{r}_N)$ and $\tilde{\mathbf{r}}' = (\tilde{r}'_1, \ldots, \tilde{r}'_N)$ must be provided:

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

Extending Wikström's shuffle proof for this extended situation is not very difficult, because only the online part of the proof is affected, i.e., the part that deals with the shuffled re-encryptions. This follows from the proof description of (5.6), which shows exactly the part that is responsible for handling the re-encryptions. In the adjusted expression

$$NIZKP\left[(\bar{r}, \hat{r}, r, (\tilde{r}, \tilde{r}'), \hat{\mathbf{r}}, \tilde{\mathbf{u}}) : \begin{array}{c} \bar{c} = g^{\bar{r}} \wedge \hat{c} = g^{\hat{r}} \wedge \tilde{c} = \mathsf{Com}(\tilde{\mathbf{u}}, r) \\ \wedge\ \tilde{e} = \mathsf{ReEnc}_{pk,\mathbf{pk}}(\prod_{i=1}^N (\tilde{e}_i)^{\tilde{u}_i}, -\tilde{r}, -\tilde{r}') \\ \wedge \left[\bigwedge_{i=1}^N (\hat{c}_i = g^{\hat{r}_i} \hat{c}_{i-1}^{\tilde{u}_i})\right] \end{array}\right], \tag{9.1}$$

[3]Appending identity elements to $\mathbf{a}'$ is a somewhat arbitrary choice. Decrypting such an extended encryption e' will then lead to randomly looking plaintexts $(b')^{-sk_i}$, from which nothing meaningful can be inferred. This is not a problem, because they will be skipped anyway during tallying.

[4]Under the condition that $pk \neq pk'_j$ holds for all $j \in \{1, \ldots, z\}$, a single fresh randomization would be sufficient for conducting the re-encryption of an augmented ElGamal encryption. In the extended key generation method presented in Alg. 9.3, this condition is actually satisfied. However, since conducting the re-encryption shuffle and proving its correctness is only slightly more expensive with two fresh randomizations, we prefer not to introduce additional assumptions and dependencies, which at some point could easily get forgotten and lead to unwanted side-effects.

we obtain $\tilde{e} = \prod_{j=1}^{N} e_j^{u_j}$ from the augmented encryptions $\mathbf{e}$ and $\tilde{r}' = \sum_{j=1}^{N} \tilde{r}_j' u_j$ from the additional re-encryption randomizations $\tilde{\mathbf{r}}'$. The core changes from Algs. 8.45 and 8.48 to the extended Algs. 9.20 and 9.21 are therefore mainly related to the commitment pair $(t_{4,1}, t_{4,2})$, which has to be extended into an element $(t_{4,1}, t_{4,2}, \mathbf{t}_{4,3}, t_{4,4}) \in \mathbb{E}_z$ of size $z + 3$, and to the value $s_4 \in \mathbb{Z}_q$, which becomes a pair $(s_4, s_4') \in \mathbb{Z}_q^2$.

9.1.2.4. Decryption Proof

The partial decryption of an augmented ElGamal encryption $e = (a, b, \mathbf{a}', b') \in \mathbb{E}_z$ with private keys sk and $\mathbf{sk}' = (sk_1', \ldots, sk_z')$ is a simple composition of the partial decryption $c = b^{sk}$ of the regular ElGamal ciphertext (a, b) using sk and of the partial decryption $(c_1', \ldots, c_z') = ((b')^{sk_1'}, \ldots, (b')^{sk_z'})$ of the multi-recipient ElGamal ciphertext $(\mathbf{a}', b')$ using $\mathbf{sk}'$. Decrypting multiple augmented ElGamal encryptions $\mathbf{e} = (e_1, \ldots, e_N)$ using the same keys sk and $\mathbf{sk}'$ therefore produces a vector $\mathbf{c} = (c_1, \ldots, c_n)$ and a matrix $\mathbf{D} = (d_{ij})_{N \times z}$ of partial decryptions $c_i = b_i^{sk}$ and $d_{ij} = (b_i')^{sk_j'}$, respectively. For proving the correctness of this operation, we generate the following cryptographic proof:

$$NIZKP\left[(sk, \mathbf{sk}') : pk = g^{sk} \wedge (\bigwedge_{j=1}^{z} pk_j' = g^{sk_j'}) \wedge (\bigwedge_{i=1}^{N} c_i = b_i^{sk}) \wedge (\bigwedge_{i=1}^{N} \bigwedge_{j=1}^{z} d_{ij} = (b_i')^{sk_j'})\right].$$

Note that the above proof is a conjunction of $(N+1)(z+1)$ single Schnorr proofs. Therefore, we can arrange the above proof as a $(N+1)$-by-$(z+1)$ matrix,

$$NIZKP\left[(sk, \mathbf{sk}') : \begin{pmatrix} pk & pk_1' & \cdots & pk_z' \\ c_1 & d_{1,1} & \cdots & d_{1,z} \\ \vdots & \vdots & \ddots & \vdots \\ c_N & d_{N,1} & \cdots & d_{N,z} \end{pmatrix} = \begin{pmatrix} g^{sk} & g^{sk_1'} & \cdots & g^{sk_z'} \\ b_1^{sk} & (b_1')^{sk_1'} & \cdots & (b_1')^{sk_z'} \\ \vdots & \vdots & \ddots & \vdots \\ b_N^{sk} & (b_N')^{sk_1'} & \cdots & (b_N')^{sk_z'} \end{pmatrix}\right],$$

which can be processes accordingly as a double loop that iterates over $(N+1)(z+1)$ steps. This is the general idea behind the decryption proof generation and verification in Algs. 9.23 and 9.24.

9.1.3. Election Result

The main additional election result of an election event with write-ins is the matrix $\mathbf{S} = (S_{ik})_{N \times z}$, which contains all submitted write-ins S_{ik} consisting of two strings of length smaller or equal to ℓ_W. Each row of the matrix $\mathbf{S}$ belongs to the corresponding row in the election result matrix $\mathbf{V} = (v_{ik})_{N \times n}$. Let $\mathbf{s}_i = (S_{i,1}, \ldots, S_{i,z})$ and $\mathbf{v}_i = (v_{i,1}, \ldots, v_{i,n})$ denote a pair of such connected rows, which together represent some voter's intention. The structure of $\mathbf{s}_i$ is as follows: the first z_i' values are regular pairs of strings and the last $z - z_i'$ values are equal to $\varnothing$:

$$\mathbf{s}_i = (S_{i,1}, \ldots, S_{i,z_i'}, \underbrace{\varnothing, \ldots, \varnothing}_{z-z_i'}), \text{ for } 0 \leqslant z_i' \leqslant z.$$

The value z_i' represents therefore the number of write-in string pairs submitted by the voter. Furthermore, recall from Section 9.1.2.2 that the pair $\mathcal{E} = ("", "")$ of empty strings is submitted whenever a regular candidate has been selected. Many values S_{ik} will therefore be equal to $\mathcal{E}$. If $\mathbf{v}_i$ only contains votes for regular candidates, then $\mathbf{s}_i$ will look as follows:

$$\mathbf{s}_i = (\underbrace{\mathcal{E},\dots,\mathcal{E}}_{z'_i}, \underbrace{\varnothing,\dots,\varnothing}_{z-z'_i}), \text{ for } 0 \leqslant z'_i \leqslant z.$$

Given the observation that write-ins are only submitted rarely in elections with write-ins, this seemingly exceptional case will actually be the most common one. In other words, we expect $\mathbf{S}$ to be a sparse matrix with only a few values different from $\mathcal{E}$ and $\varnothing$.

The second additional election result $\mathbf{T} = (t_{ik})_{N\times z}$ assigns the submitted write-ins $S_{ik} \neq \varnothing$ to the write-in elections. The assignments $t_{ik} \in \{1,\dots,t,\varnothing\}$ contained in $\mathbf{t}_i$ (the i-th row of $\mathbf{T}$) can be derived from the $\mathbf{v}_i$, $\mathbf{n}$, and $\mathbf{z}$ based on the increasing candidate ordering over all t elections (see Alg. 9.28).[5] Note that exactly the same values remain unassigned in $\mathbf{S}$ and $\mathbf{T}$, i.e., $S_{ik} = \varnothing$ whenever $t_{ik} = \varnothing$.

In the presence of write-ins, the two matrices $\mathbf{S}$ and $\mathbf{T}$ extend the raw election result to the following tuple:

$$ER = (\mathbf{u}, \mathbf{V}, \mathbf{W}, \mathbf{S}, \mathbf{T}).$$

The extended election result can be used to obtain the following aggregated results for each write-in election:

- Set of write-ins submitted to election $j \in \{1,\dots,t\}$:

$$S(j) = \{S_{ik} \in \mathbf{S} : t_{ik} = j\} \setminus \{\mathcal{E}, \varnothing\}.$$

- Number of votes for write-in $S \in S(j)$ in election $j \in \{1,\dots,t\}$ and counting circle $c \in \{1,\dots,w\}$:

$$V(S,j,c) = \sum_{i=1}^{N}\sum_{k=1}^{z} b_{ik}(S,j,c), \text{ for } b_{ik}(S,j,c) = \begin{cases} 1, & \text{if } S_{ik} = S, t_{ik} = j, w_{ic} = 1, \\ 0, & \text{otherwise.} \end{cases}$$

- Total number of votes for write-in $S \in S(j)$ in election $j \in \{1,\dots,t\}$:

$$V(S,j) = \sum_{c=1}^{w} V(S,j,c).$$

Note that some voters may have chosen the option of submitting a write-in, but without entering text into the two write-in text fields, i.e., the submitted write-in is a pair ("","") of empty strings. To the best of our understanding, such empty write-ins must be interpreted as blank votes. The problem with such *blank write-ins* is that they cannot be located unambiguously in the matrix $\mathbf{S}$, because the same pair of empty strings is submitted by

[5] Adding default candidates to ballots from voters with restricted eligibility by Alg. 8.40 may break up the matching order between candidates and write-ins. An unambiguous assignment of write-ins to elections is then no longer possible. As discussed in the beginning of Section 9.1, we have no better solution other than switching off the default vote mechanism for write-in elections (see Alg. 9.17). Note that this may restrict vote privacy in some rare cases. For further details on this issue, we refer to the discussion of Recommendation 10.6 in [17].

default when no write-in candidates are selected.[6] However, we can deduce the number of submitted blank write-ins by subtracting the number of submitted write-ins different from $\mathcal{E} = ("", "")$ from the total number of submitted write-in candidates. For this, let

$$I'_J = \{i \in I_j : v_i = 1\} \subseteq I_j$$

denote that set of indices of the write-in candidates of election j. The number of blank write-ins can then be computed as follows:

- Number of blank write-ins in election $j \in \{1, \dots, t\}$ and counting circle $c \in \{1, \dots, w\}$:

$$B(j,c) = \sum_{i \in I'_j} V(j,c) - \sum_{S \in S(j)} V(S,j,c).$$

- Total number of blank write-ins in election $j \in \{1, \dots, t\}$:

$$B(j) = \sum_{c=1}^{w} B(j,c) = \sum_{i \in I'_j} V(j) - \sum_{S \in S(j)} V(S,j).$$

9.2. Protocol Description

The general structure of the protocol is not affected by the write-in extension, i.e., we still have three top-level protocol phases (pre-election, election, post-election), and each of them has the same three or four sub-phases. Therefore, the protocol overview from Table 7.1 and the parties involved in each phase remain exactly the same.

The main difference is the extended information flow. At certain protocol steps, some parties need to provide or compute additional information and include it in the messages sent to to other participating parties. Table 9.2 summarizes the changes to the general information flow and Table 9.3 gives an overview of the additional (or modified) computations. The protocol diagrams from Chapter 7 need to be updated accordingly (which should be straightforward based on the information from Tables 9.2 and 9.3).

Note that the extended protocol can also be used when no write-ins are allowed at all. In this case, the election administrator defines $\mathbf{z} = (0, \dots, 0)$ and therefore $\mathbf{v} = (0, \dots, 0)$. This implies $z'_i = 0$ for all voters $i \in \{1, \dots, N_E\}$ and therefore $z_{\max} = 0$. In the information computed and exchanged during the extended protocol, this leads to empty vectors $\mathbf{pk}'_j$, $\mathbf{s}'$, and $\tilde{\mathbf{r}}'_j$, and empty matrices $\mathbf{D}_j$, $\mathbf{D}$, $\mathbf{M}$, $\mathbf{S}$, and $\mathbf{T}$. Empty vectors and empty matrices also arise internally in some other values, for example in each augmented encryption or in the proofs π_j, $\tilde{\pi}_j$, and π'_j. The protocol and the algorithms are designed to deal with such empty vectors and matrices appropriately as special cases. The same holds from a performance points of view. The necessary computational steps in the extended protocol degenerate into exactly the computational steps of the basic protocol when no write-ins are allowed.

[6] One could think that using two different default values would solve this problem, but this would not prevent a malicious voting client from generating this conflict on purpose.

Protocol	Sending Party	Receiving Party	Original Message	Extended Message
7.1	Administrator	Authority j	EP, pk_0, π_0	$EP', pk_0, \mathbf{pk}'_0, \pi_0$
	Authority j	Authority k	pk_j, π_j	$pk_j, \mathbf{pk}'_j, \pi_j$
7.3	Administrator	Printing authority	EP	EP'
7.4	Administrator	Voting client v	VP_v, pk_0, π_0	$VP'_v, pk_0, \mathbf{pk}'_0, \pi_0$
	Voting client v	Voter v	VP_v	VP'_v
	Voter v	Voting client v	$X_v, \mathbf{s}$	$X_v, \mathbf{s}, \mathbf{s}'$
7.5	Authority j	Voting client v	pk_j, π_j	$pk_j, \mathbf{pk}'_j, \pi_j$
	Voting client v	Authority j	v, α	v, α'
7.7	Authority j	Authority k	$\tilde{\mathbf{e}}_j, \tilde{\pi}_j$	$\tilde{\mathbf{e}}'_j, \tilde{\pi}'_j$
7.8	Authority j	Authority k	$\mathbf{c}_j, \pi'_j$	$\mathbf{c}_j, \mathbf{D}_j, \pi'_j$
7.9	Authority j	Administrator	$\mathbf{c}, \tilde{\mathbf{e}}_s$	$\mathbf{c}, \mathbf{D}, \tilde{\mathbf{e}}'_s$
	Administrator	General public	ER	ER'

Table 9.2.: Changes to the information flow in the extended protocol.

Protocol	Party	Computations in Extended Protocol
7.1	Administrator	$(sk_0, pk_0, \mathbf{sk}'_0, \mathbf{pk}'_0) \leftarrow \mathsf{GenKeyPairs}(\mathbf{k}, \mathbf{E}, \mathbf{z})$
		$\pi_0 \leftarrow \mathsf{GenKeyPairProof}(sk_0, pk_0, \mathbf{sk}'_0, \mathbf{pk}'_0)$
	Authority j	$\mathsf{CheckKeyPairProof}(\pi_0, pk_0, \mathbf{pk}'_0)$
		$(sk_j, pk_j, \mathbf{sk}'_j, \mathbf{pk}'_j) \leftarrow \mathsf{GenKeyPairs}(\mathbf{k}, \mathbf{E}, \mathbf{z})$
		$\pi_j \leftarrow \mathsf{GenKeyPairProof}(sk_j, pk_j, \mathbf{sk}'_j, \mathbf{pk}'_j)$
		$\mathsf{CheckKeyPairProof}(\pi_k, pk_k, \mathbf{pk}'_k)$
		$(pk, \mathbf{pk}') \leftarrow \mathsf{GetPublicKeys}(\mathbf{pk}, \mathbf{PK}')$
7.4	Administrator	$VP_v \leftarrow \mathsf{GetVotingParameters}(v, EP)$
	Voting client	$\mathsf{CheckKeyPairProof}(\pi_0, pk_0, \mathbf{pk}'_0)$
7.5	Voting client	$\mathsf{CheckKeyPairProof}(\pi_j, pk_j, \mathbf{pk}'_j)$
		$(pk, \mathbf{pk}') \leftarrow \mathsf{GetPublicKeys}(\mathbf{pk}, \mathbf{PK}')$
		$(\alpha, \mathbf{r}) \leftarrow \mathsf{GenBallot}(X, \mathbf{s}, \mathbf{s}', pk, \mathbf{pk}', \mathbf{n}, \mathbf{k}, \mathbf{e}, w, \mathbf{z}, \mathbf{v})$
	Authority j	$VP_v \leftarrow \mathsf{GetVotingParameters}(v, EP)$
		$\mathsf{CheckBallot}(v, \alpha, pk, \mathbf{pk}', \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{z}, \mathbf{v}, \hat{\mathbf{x}})$
7.7	Authority j	$\tilde{\mathbf{e}}_0 \leftarrow \mathsf{GetEncryptions}(B_1, C_1, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{w}, \mathbf{z})$
		$(\tilde{\mathbf{e}}_j, \tilde{\mathbf{r}}_j, \tilde{\mathbf{r}}'_j, \psi_j) \leftarrow \mathsf{GenShuffle}(\tilde{\mathbf{e}}_{j-1}, pk, \mathbf{pk}')$
		$\tilde{\pi}_j \leftarrow \mathsf{GenShuffleProof}(U, \tilde{\mathbf{e}}_{j-1}, \tilde{\mathbf{e}}_j, \tilde{\mathbf{r}}_j, \tilde{\mathbf{r}}'_j, \psi_j, pk, \mathbf{pk}')$
		$\mathsf{CheckShuffleProof}(U, \tilde{\pi}, \mathbf{e}, \tilde{\mathbf{e}}, pk, \mathbf{pk}')$
7.8	Authority j	$(\mathbf{c}_j, \mathbf{D}_j) \leftarrow \mathsf{GetDecryptions}(\tilde{\mathbf{e}}_s, sk_j, \mathbf{sk}'_j)$
		$\pi'_j \leftarrow \mathsf{GenDecryptionProof}(sk_j, pk_j, \mathbf{sk}'_j, \mathbf{pk}'_j, \tilde{\mathbf{e}}_s, \mathbf{c}_j, \mathbf{D}_j)$
		$\mathsf{CheckDecryptionProof}(\pi, pk, \mathbf{pk}', \mathbf{e}, \mathbf{c}, \mathbf{D})$
		$(\mathbf{c}, \mathbf{D}) \leftarrow \mathsf{GetCombinedDecryptions}(\mathbf{C}, \overrightarrow{\mathbf{d}})$
7.9	Administrator	$(\mathbf{c}_0, \mathbf{D}_0) \leftarrow \mathsf{GetDecryptions}(\mathbf{e}_1, sk_0, \mathbf{sk}'_0)$
		$\pi'_0 \leftarrow \mathsf{GenDecryptionProof}(sk_0, pk_0, \mathbf{sk}'_0, \mathbf{pk}'_0, \mathbf{e}_1, \mathbf{c}_0, \mathbf{C}'_0)$
		$(\mathbf{m}, \mathbf{M}) \leftarrow \mathsf{GetVotes}(\tilde{\mathbf{e}}_s, \mathbf{c}, \mathbf{c}', \mathbf{D}, \mathbf{D}')$
		$(\mathbf{u}, \mathbf{V}, \mathbf{W}, \mathbf{S}, \mathbf{T}) \leftarrow \mathsf{GetElectionResult}(\mathbf{m}, \mathbf{n}, \mathbf{w}, \mathbf{M}, \mathbf{k}, \mathbf{z})$

Table 9.3.: Modified computations in the extended protocol.

9.3. Pseudo-Code Algorithms

In this section, we give all the algorithmic details of the extended protocol that supports write-ins. We give updates of some algorithms from Chapter 8 to reflect the changes listed in Table 9.3. Some of them require calls to additional sub-algorithms to deal with the particularities of processing the write-ins. The section is structured according to the three phases of the protocol. At the beginning of corresponding subsections, we give an overview of all algorithms and sub-algorithms presented. The overview also links the algorithms of the extended protocol to their counterparts in the basic protocol. If the extended protocol is implemented on top of an existing implementation of the basic protocol, then these algorithms need to be adjusted accordingly.

9.3.1. General Algorithms

We propose two new general algorithms for encoding pairs of strings into elements of the group $\mathbb{G}_q$ and vice versa. They implement the injective mapping from Section 9.1.2.1, which consists of three nested encoding steps.

```
Algorithm: GetEncodedStrings($S, A, \ell, c$)
Input: Strings $S = (S_1, S_2) \in A^{\{0,\dots,\ell\}} \times A^{\{0,\dots,\ell\}}$
       Alphabet $A$, $|A| \geqslant 2$
       Maximal string length $\ell \geqslant 0$, $(|A|+1)^{2\ell} < q$
       Padding symbol $c \notin A$
while $|S_1| < \ell$ do
    $S_1 \leftarrow \langle c \rangle \,\|\, S_1$
while $|S_2| < \ell$ do
    $S_2 \leftarrow \langle c \rangle \,\|\, S_2$
$S_{12} \leftarrow S_1 \,\|\, S_2$
$x \leftarrow$ StringToInteger($S_{12}, A \cup \{c\}$) + 1          // $x \in \mathbb{Z}_p^*$, see Alg. 4.8
$y \leftarrow x^2 \bmod p$
return $y$                                                      // $y \in \mathbb{G}_q$
```

Algorithm 9.1: Encodes a pair of strings S from an alphabet A into a group element $y \in \mathbb{G}_q$.

```
**Algorithm:** GetDecodedStrings($y, A, \ell, c$)
**Input:** Group element $y \in \mathbb{G}_q$
           Alphabet $A$, $|A| \geqslant 2$
           Maximal string length $\ell \geqslant 0$, $(|A|+1)^{2\ell} < q$
           Padding symbol $c \notin A$
$x \leftarrow y^{\frac{q+1}{2}} \bmod p$
$x \leftarrow \min(x, p-x)$                                         // $x \in \mathbb{Z}*_p$
$S_{12} \leftarrow$ IntegerToString($x-1, 2\ell, A \cup \{c\}$)     // $S \in (A \cup \{c\})^{2\ell}$, see Alg. 4.7
$S_1 \leftarrow$ Truncate($S_{12}, \ell$)
**while** $|S_1| > 0$ **and** $S_1[0] = c$ **do**
    $S_1 \leftarrow$ Skip($S_1, 1$)
$S_2 \leftarrow$ Skip($S_{12}, \ell$)
**while** $|S_2| > 0$ **and** $S_2[0] = c$ **do**
    $S_2 \leftarrow$ Skip($S_2, 1$)
$S \leftarrow (S_1, S_2)$
**return** $S$                                                      // $S \in A^{\{0,\ldots,\ell\}} \times A^{\{0,\ldots,\ell\}}$
```

Algorithm 9.2: Decodes a given group element $y \in \mathbb{G}_q$ into a pair of strings $S = (S_1, S_2)$ of length $|S_i| \leqslant \ell$.

9.3.2. Pre-Election Phase

In the pre-election phase, the main extension to the basic protocol is the computation of additional key pairs. As discussed in Section 9.1.2.1, we use the multi-recipient ElGamal encryption scheme to encrypt the write-ins. The maximal number of write-ins is determined by the value $z_{\max}$, which can be derived from $\mathbf{k}$, $\mathbf{E}$, and $\mathbf{z}$. At the end of the key generation process, $z_{\max}$ many additional public keys $\mathbf{pk}' = (pk'_1, \ldots, pk'_{z_{\max}})$ are known to each election authority, and each of them holds shares $\mathbf{sk}'_j$ of corresponding private keys. Moving from a single key to $z_{\max}+1$ keys also means to increase the proofs π_j accordingly. Clearly, this affects both the generation and the verification of the proofs. The full set of new algorithms for dealing with this particular issue is depicted in Table 9.4.

Nr.	Algorithm	Called by	Protocol
8.6 ⇒ 9.3	$\mathsf{GenKeyPairs}(\mathbf{k}, \mathbf{E}, \mathbf{z})$	Administrator	7.1
8.7 ⇒ 9.4	$\mathsf{GenKeyPairProof}(sk, pk, \mathbf{sk}', \mathbf{pk}')$		
8.6 ⇒ 9.3	$\mathsf{GenKeyPairs}(\mathbf{k}, \mathbf{E}, \mathbf{z})$	Election authority	
8.7 ⇒ 9.4	$\mathsf{GenKeyPairProof}(sk, pk, \mathbf{sk}', \mathbf{pk}')$		
8.8 ⇒ 9.5	$\mathsf{CheckKeyPairProof}(\pi, pk, \mathbf{pk}')$		
8.9 ⇒ 9.6	$\mathsf{GetPublicKeys}(\mathbf{pk}, \mathbf{PK}')$		

Table 9.4.: Overview of pre-election phase algorithms used to implement write-ins.

```
Algorithm: $\mathsf{GenKeyPairs}(\mathbf{k}, \mathbf{E}, \mathbf{z})$
Input: Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$
       Eligibility matrix $\mathbf{E} = (e_{ij}) \in \mathbb{B}^{N_E \times t}$
       Write-in elections $\mathbf{z} = (z_1, \ldots, z_t) \in \mathbb{B}^t$
$sk \leftarrow \mathsf{GenRandomInteger}(q)$                                  // see Alg. 4.11
$pk \leftarrow g^{sk} \bmod p$
$z_{\max} \leftarrow \max_{i=1}^{N_E} \sum_{j=1}^{t} e_{ij} k_j z_j$
for $i = 1, \ldots, z_{\max}$ do
    $sk'_i \leftarrow \mathsf{GenRandomInteger}(q)$                           // see Alg. 4.11
    $pk'_i \leftarrow g^{sk'_i} \bmod p$
$\mathbf{sk}' \leftarrow (sk'_1, \ldots, sk'_{z_{\max}})$
$\mathbf{pk}' \leftarrow (pk'_1, \ldots, pk'_{z_{\max}})$
return $(sk, pk, \mathbf{sk}', \mathbf{pk}')$      // $sk \in \mathbb{Z}_q$, $pk \in \mathbb{G}_q$, $\mathbf{sk}' \in \mathbb{Z}_q^{z_{\max}}$, $\mathbf{pk}' \in \mathbb{G}_q^{z_{\max}}$
```

Algorithm 9.3: Generates the necessary amount of ElGamal key pairs or shares of such key pairs. This algorithm is an extension of Alg. 8.6 from Section 8.2.

```
**Algorithm:** GenKeyPairProof($sk, pk, \mathbf{sk}', \mathbf{pk}'$)
**Input:** Decryption key $sk \in \mathbb{Z}_q$
           Encryption key $pk \in \mathbb{G}_q$
           Write-in decryption keys $\mathbf{sk}' = (sk'_1, \ldots, sk'_z) \in \mathbb{Z}_q^z$
           Write-in encryption keys $\mathbf{pk}' \in \mathbb{Z}_q^z$
for $i = 0, \ldots, z$ do
    $\omega_i \leftarrow$ GenRandomInteger($q$)                        // see Alg. 4.11
    $t_i \leftarrow g^{\omega_i} \bmod p$
$\mathbf{t} \leftarrow (t_0, t_1, \ldots, t_z)$
$y \leftarrow (pk, \mathbf{pk}')$
$c \leftarrow$ GetChallenge($y, \mathbf{t}$)                              // see Alg. 8.4
$s_0 \leftarrow \omega_0 - c \cdot sk \bmod q$
for $i = 1, \ldots, z$ do
    $s_i \leftarrow \omega_i - c \cdot sk'_i \bmod q$
$\mathbf{s} \leftarrow (s_0, s_1, \ldots, s_z)$
$\pi \leftarrow (c, \mathbf{s})$
return $\pi$                                                   // $\pi \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_q^{z+1}$
```

Algorithm 9.4: Generates a proof of knowing the decryption keys. For the proof verification, see Alg. 9.5. This algorithm is an extension of Alg. 8.7 from Section 8.2.

```
**Algorithm:** CheckKeyPairProof($\pi, pk, \mathbf{pk}'$)
**Input:** Key pair proof $\pi = (c, \mathbf{s}) \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_q^{z+1}$, $\mathbf{s} = (s_0, s_1, \ldots, s_z)$
           Encryption key $pk \in \mathbb{G}_q$
           Write-in encryption keys $\mathbf{pk}' = (pk'_1, \ldots, pk'_z) \in \mathbb{G}_q^z$
$t_0 \leftarrow pk^c \cdot g^{s_0} \bmod p$
for $j = 1, \ldots, z$ do
    $t_i \leftarrow (pk'_i)^c \cdot g^{s_i} \bmod p$
$\mathbf{t} \leftarrow (t_0, t_1, \ldots, t_z)$
$y \leftarrow (pk, \mathbf{pk}')$
$c' \leftarrow$ GetChallenge($y, \mathbf{t}$)                             // see Alg. 8.4
return $c = c'$
```

Algorithm 9.5: Checks the correctness of a key pair proof π generated by Alg. 9.4. This algorithm is an extension of Alg. 8.8 from Section 8.2

```
**Algorithm:** GetPublicKeys($\mathbf{pk}, \mathbf{PK}'$)
**Input:** Encryption key shares $\mathbf{pk} = (pk_0, pk_1, \ldots, pk_s) \in \mathbb{G}_q^{s+1}$
           Write-in encryption key shares $\mathbf{PK}' = (pk'_{ij}) \in \mathbb{G}_q^{z \times (s+1)}$
$pk \leftarrow \prod_{j=0}^{s} pk_j \bmod p$
**for** $i = 1, \ldots, z$ **do**
    $pk'_i \leftarrow \prod_{j=0}^{s} pk'_{ij} \bmod p$
$\mathbf{pk}' \leftarrow (pk'_1, \ldots, pk'_z)$
**return** $(pk, \mathbf{pk}')$                                  // $pk \in \mathbb{G}_q$, $\mathbf{pk}' \in \mathbb{G}_q^z$
```

Algorithm 9.6: Computes public encryption keys from given shares. This algorithm is an extension of Alg. 8.9 from Section 8.2

9.3.3. Election Phase

In the election phase, the main extension compared to the basic protocol deals with the encryption of the chosen write-ins $\mathbf{s}'$. For this, the two main algorithms GenBallot and CheckBallot obtain some additional arguments, for example the write-in public keys $\mathbf{pk}'$ and the additional election parameters $\mathbf{z}$ and $\mathbf{v}$. From a technical point of view, the most complex new algorithms deal with the generation and verification of the additional zero-knowledge proof π', which is added to the ballot α along with the multi-recipient ElGamal encryption e'. Corresponding algorithms GenWriteInProof and CheckWriteInProof implement the method described in Section 9.1.2.2. The full set of modified and new algorithms of the election phase is depicted in Table 9.5.

Nr.	Algorithm	Called by	Protocol
8.8 $\Rightarrow$ 9.5	CheckKeyPairProof$(\pi, pk, \mathbf{pk}')$	Voting client	7.4
8.20 $\Rightarrow$ 9.7	GetVotingParameters(v, EP)	Administrator	
8.8 $\Rightarrow$ 9.5	CheckKeyPairProof$(\pi, pk, \mathbf{pk}')$	Voting client	7.5
8.9 $\Rightarrow$ 9.6	GetPublicKeys$(\mathbf{pk}, \mathbf{PK}')$		
8.21 $\Rightarrow$ 9.8	GenBallot$(X, \mathbf{s}, \mathbf{s}', pk, \mathbf{pk}', \mathbf{n}, \mathbf{k}, \mathbf{e}, w, \mathbf{z}, \mathbf{v})$		
8.22	↳ GetEncodedSelections$(\mathbf{s}, \mathbf{p})$		
8.23	↳ GenQuery$(\mathbf{m}, pk)$		
8.24	↳ GenBallotProof$(x, m, r, \hat{x}, \mathbf{a}, pk)$		
9.9	↳ GetEncodedWriteIns$(\mathbf{s})$		
9.1	↳ GetEncodedStrings(S, A, ℓ, c)		
9.10	↳ GenWriteInEncryption$(\mathbf{pk}, \mathbf{s})$		
9.11	↳ GetWriteInIndices$(\mathbf{n}, \mathbf{k}, \mathbf{e}, \mathbf{z}, \mathbf{v})$		
9.12	↳ GenWriteInProof$(pk, \mathbf{m}, \mathbf{a}, \mathbf{r}, \mathbf{pk}', \mathbf{m}', e', r', \mathbf{p})$		
9.1	↳ GetEncodedStrings(S, A, ℓ, c)		
9.13	↳ GenCNFProof$(\mathbf{Y}, \mathbf{r}, \mathbf{w})$		
8.20 $\Rightarrow$ 9.7	GetVotingParameters(v, EP)	Election authority	
8.25 $\Rightarrow$ 9.14	CheckBallot$(v, \alpha, pk, \mathbf{pk}', \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{z}, \mathbf{v}, \hat{\mathbf{x}})$		
8.26	↳ CheckBallotProof$(\pi, \hat{x}, \mathbf{a}, pk)$		
9.11	↳ GetWriteInIndices$(\mathbf{n}, \mathbf{k}, \mathbf{e}, \mathbf{z}, \mathbf{v})$		
9.15	↳ CheckWriteInProof$(\pi, pk, \mathbf{a}, \mathbf{pk}', e', \mathbf{p})$		
9.1	↳ GetEncodedStrings(S, A, ℓ, c)		
9.16	↳ CheckCNFProof$(\pi, \mathbf{Y})$		

Table 9.5.: Overview of election phase algorithms used to implement write-ins.

```
Algorithm: GetVotingParameters($v, EP$)
Input: Voter index $v \in \{1, \ldots, N_E\}$
       Election parameters $EP = (U, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{w}, \mathbf{z}, \mathbf{v})$, $U \in A^*_{\mathsf{ucs}}$, $\mathbf{c} \in (A^*_{\mathsf{ucs}})^n$,
       $\mathbf{d} \in (A^*_{\mathsf{ucs}})^{N_E}$, $\mathbf{e} \in (A^*_{\mathsf{ucs}})^t$, $\mathbf{n} \in (\mathbb{N}^+)^t$, $\mathbf{k} \in (\mathbb{N}^+)^t$, $\mathbf{E} \in \mathbb{B}^{N_E \times t}$, $\mathbf{w} \in (\mathbb{N}^+)^{N_E}$,
       $\mathbf{z} \in \mathbb{B}^t, \mathbf{v} \in \mathbb{B}^n$
$\mathbf{e}_v \leftarrow \mathsf{GetRow}(\mathbf{E}, v)$
$z_{\max} \leftarrow \max_{i=1}^{N_E} \sum_{j=1}^{t} e_{ij} k_j z_j$
$VP_v \leftarrow (U, \mathbf{c}, D_v, \mathbf{e}, \mathbf{n}, \mathbf{k}, \mathbf{e}_v, w_v, \mathbf{z}, \mathbf{v}, z_{\max})$
return $VP_v$
    // $VP_v \in A^*_{\mathsf{ucs}} \times (A^*_{\mathsf{ucs}})^n \times A^*_{\mathsf{ucs}} \times (A^*_{\mathsf{ucs}})^t \times (\mathbb{N}^+)^t \times (\mathbb{N}^+)^t \times \mathbb{B}^t \times \mathbb{N}^+ \times \mathbb{B}^t \times \mathbb{B}^n \times \mathbb{N}$
```

Algorithm 9.7: Collects the information to be displayed to the voter v on the voting page. This algorithm is an extension of Alg. 8.20 from Section 8.3 with an additional parameter $\mathbf{z}$. The information from $\mathbf{z}$ is important to let the voter know about the elections that support write-ins. Note that the value $z_{\max}$ is the same for every voter, i.e., it could possibly be cached by an efficient implementation.

```
Algorithm: $\mathsf{GenBallot}(X, \mathbf{s}, \mathbf{s}', pk, \mathbf{pk}', \mathbf{n}, \mathbf{k}, \mathbf{e}, w, \mathbf{z}, \mathbf{v}, z_{\max})$
Input: Voting code $X \in A_X^{\ell_X}$
       Selection $\mathbf{s} = (s_1, \ldots, s_k)$, $1 \leqslant s_1 < \cdots < s_k \leqslant n$
       Write-ins $\mathbf{s}' \in \mathcal{W}^{z'}$
       Encryption key $pk \in \mathbb{G}_q$
       Write-in encryption keys $\mathbf{pk}' \in \mathbb{G}_q^{z_{\max}}$
       Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$, $n = \sum_{j=1}^t n_j$
       Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Eligibility vector $\mathbf{e} = (e_1, \ldots, e_t) \in \mathbb{B}^t$, $k = \sum_{j=1}^t e_j k_j$
       Counting circle $w \in \mathbb{N}^+$
       Write-in elections $\mathbf{z} = (z_1, \ldots, z_t) \in \mathbb{B}^t$, $z' = \sum_{j=1}^t e_j k_j z_j$, $z = \sum_{j=1}^t k_j z_j$
       Write-in candidates $\mathbf{v} \in \mathbb{B}^n$
       Maximum number of write-ins $z_{\max} \in \mathbb{N}$, $z' \leqslant z_{\max} \leqslant z$
$x \leftarrow \mathsf{StringToInteger}(X)$, $\hat{x} \leftarrow \hat{g}^x \bmod \hat{p}$                    // see Alg. 4.8
$\mathbf{p} \leftarrow \mathsf{GetPrimes}(n + w)$                              // $\mathbf{p} = (p_0, \ldots, p_{n+w})$ see Alg. 8.1
$\mathbf{m} \leftarrow \mathsf{GetEncodedSelections}(\mathbf{s}, \mathbf{p})$                 // $\mathbf{m} = (m_1, \ldots, m_k)$, see Alg. 8.22
$m \leftarrow \prod_{j=1}^k m_j$
if $p_{n+w} \cdot m \geqslant p$ then
    return $\perp$                                   // $\mathbf{s}$, $\mathbf{n}$, and $w$ are incompatible with $p$
$(\mathbf{a}, \mathbf{r}) \leftarrow \mathsf{GenQuery}(\mathbf{m}, pk)$            // $\mathbf{a} = (a_1, \ldots, a_k)$, $\mathbf{r} = (r_1, \ldots, r_k)$, see Alg. 8.23
$r \leftarrow \sum_{j=1}^k r_j \bmod q$
$\pi \leftarrow \mathsf{GenBallotProof}(x, m, r, \hat{x}, \mathbf{a}, pk)$                          // see Alg. 8.24
$\mathbf{m}' \leftarrow \mathsf{GetEncodedWriteIns}(\mathbf{s}')$                          // $\mathbf{m}' \in \mathbb{G}_q^{z'}$, see Alg. 9.9
$(e', r') \leftarrow \mathsf{GenWriteInEncryption}(\mathbf{pk}', \mathbf{m}')$                     // see Alg. 9.10
$(I, J) \leftarrow \mathsf{GetWriteInIndices}(\mathbf{n}, \mathbf{k}, \mathbf{e}, \mathbf{z}, \mathbf{v})$             // $z' = |I| = |J|$, see Alg. 9.11
$\pi' \leftarrow \mathsf{GenWriteInProof}(pk, \mathbf{m}_I, \mathbf{a}_I, \mathbf{r}_I, \mathbf{pk}', \mathbf{m}', e', r', \mathbf{p}_J)$          // see Alg. 9.12
$\alpha \leftarrow (\hat{x}, \mathbf{a}, \pi, e', \pi')$
return $(\alpha, \mathbf{r})$             // $\alpha \in \mathbb{Z}_{\hat{q}} \times (\mathbb{G}_q^2)^k \times ((\mathbb{G}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{G}_q) \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q))$
                                 // $\times(\mathbb{G}_q^{z'} \times \mathbb{G}_q) \times ((\mathbb{G}_q^2)^{z' \times 2} \times \mathbb{Z}_{2^\tau}^{z' \times 2} \times \mathbb{Z}_q^{z' \times 2})$, $\mathbf{r} \in \mathbb{Z}_q^k$
```

Algorithm 9.8: Generates a ballot based on the selection of candidates $\mathbf{s}$, the write-ins $\mathbf{s}'$, and the voting code X. This algorithm is an extension of Alg. 8.21 with additional code for generating the multi-recipient encryption e' and the write-in proof π'.

```
Algorithm: $\mathsf{GetEncodedWriteIns}(\mathbf{s})$
Input: Write-ins $\mathbf{s} = (S_1, \ldots, S_z) \in \mathcal{W}^z$
for $i = 1, \ldots, z$ do
    $m_i \leftarrow \mathsf{GetEncodedStrings}(S_i, A_W, \ell_W, c_W)$                          // see Alg. 9.1
$\mathbf{m} \leftarrow (m_1, \ldots, m_z)$
return $\mathbf{m}$                                                     // $\mathbf{m} \in \mathbb{G}_q^z$
```

Algorithm 9.9: Encodes the given write-ins as elements of $\mathbb{G}_q$.

```
Algorithm: GenWriteInEncryption($\mathbf{pk}, \mathbf{m}$)
Input: Write-in encryption keys $\mathbf{pk} = \{pk_1, \ldots, pk_{z_{\max}}\} \in \mathbb{G}_q^{z_{\max}}$
       Encoded write-ins $\mathbf{m} = \{m_1, \ldots, m_z\} \in \mathbb{G}_q^z$, $z \leqslant z_{\max}$
$r \leftarrow$ GenRandomInteger($q$)                                   // see Alg. 4.11
for $i = 1, \ldots, z$ do
    $a_i \leftarrow m_i \cdot pk_i^r \bmod p$
$\mathbf{a} \leftarrow (a_1, \ldots, a_z)$, $b \leftarrow g^r \bmod p$
$e \leftarrow (\mathbf{a}, b)$
return $(e, r)$                                          // $\mathbf{e} \in \mathbb{G}_q^z \times \mathbb{G}_q$, $r \in \mathbb{Z}_q$
```

Algorithm 9.10: Creates a multi-recipient ElGamal encryption from a given vector **m** of encoded write-ins and public keys **pk**.

```
Algorithm: GetWriteInIndices($\mathbf{n}, \mathbf{k}, \mathbf{e}, \mathbf{z}, \mathbf{v}$)
Input: Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$, $n = \sum_{j=1}^t n_j$
       Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Eligibility vector $\mathbf{e} = (e_1, \ldots, e_t) \in \mathbb{B}^t$, $k = \sum_{i=j}^t e_{vj} k_j$
       Write-in elections $\mathbf{z} = (z_1, \ldots, z_t) \in \mathbb{B}^t$
       Write-in candidates $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbb{B}^n$
$k' \leftarrow 0$, $n' \leftarrow 0$
$I \leftarrow \varnothing$, $J \leftarrow \varnothing$
for $l = 1, \ldots, t$ do
    if $e_l = 1$ then
        if $z_l = 1$ then
            for $i = k' + 1, \ldots, k' + k_l$ do                 // loop for $i = 1, \ldots, k$
                $I \leftarrow I \cup \{i\}$
            for $j = n' + 1, \ldots, n' + n_l$ do                 // loop for $j = 1, \ldots, n$
                if $v_j = 1$ then
                    $J \leftarrow J \cup \{j\}$
        $k' \leftarrow k' + k_l$
    $n' \leftarrow n' + n_l$
return $(I, J)$                              // $I \subseteq \{1, \ldots, k\}$, $J \subseteq \{1, \ldots, n\}$
```

Algorithm 9.11: Computes two sets of indices I and J, which are needed in Alg. 9.8 for selecting in some vectors the entries that are relevant for generating and verifying the write-in proof. The set I contains the indices of the voter's selection **s** belonging to a write-in election, and J contains the indices of all write-in candidates in elections, in which the voter participates.

```
**Algorithm:** GenWriteInProof$(pk, \mathbf{m}, \mathbf{a}, \mathbf{r}, \mathbf{pk}', \mathbf{m}', e', r', \mathbf{p})$
**Input:** Public key $pk \in \mathbb{G}_q$
           Encoded selections $\mathbf{m} = (m_1, \dots, m_z) \in \mathbb{G}_q^z$
           Encrypted selections $\mathbf{a} = (a_1, \dots, a_z) \in (\mathbb{G}_q^2)^z$
           Randomizations $\mathbf{r} = (r_1, \dots, r_z) \in \mathbb{Z}_q^z$
           Write-in encryption keys $\mathbf{pk}' = (pk'_1, \dots, pk'_{z_{\max}}) \in \mathbb{G}_q^{z_{\max}}$, $z \leqslant z_{\max}$
           Encoded write-ins $\mathbf{m}' = (m'_1, \dots, m'_z) \in \mathbb{G}_q^z$
           Encrypted write-ins $e' = ((a'_1, \dots, a'_z), b') \in \mathbb{G}_q^z \times \mathbb{G}_q$
           Randomization $r' \in \mathbb{Z}_q$
           Encoded write-in candidates $\mathbf{p} = (p_1, \dots, p_z) \in (\mathbb{G}_q \cap \mathbb{P})^z$
$\varepsilon \leftarrow$ GetEncodedStrings$(("", ""), A_W, \ell_W, c_W)$                // see Alg. 9.1
**for** $i = 1, \dots, z$ **do**
    $y^*_{i,1} \leftarrow (pk, p_i, a_i)$
    $y^*_{i,2} \leftarrow (pk'_i, \varepsilon, (a'_i, b'))$
    **if** $m_i = p_i$ **then**
        $r^*_i \leftarrow r_i$, $j_i \leftarrow 1$
    **else if** $m'_i = \varepsilon$ **then**
        $r^*_i \leftarrow r'$, $j_i \leftarrow 2$
    **else**
        **return** $\perp$                                          // invalid input
$\mathbf{Y}^* \leftarrow (y^*_{ij})_{z \times 2}$, $\mathbf{r}^* \leftarrow (r^*_1, \dots, r^*_z)$, $\mathbf{j} \leftarrow (j_1, \dots, j_z)$
$\pi \leftarrow$ GenCNFProof$(\mathbf{Y}^*, \mathbf{r}^*, \mathbf{j})$                                // see Alg. 9.13
**return** $\pi \in \mathbb{Z}_{2^\tau}^{z \times 2} \times \mathbb{Z}_q^{z \times 2}$
```

Algorithm 9.12: Generates a NIZKP, which proves that the write-in candidates and the write-ins have been chosen properly. It normalizes the given private and public values into the particular form of the CNF-proof presented in Section 9.1.2.2. Calling Alg. 9.13 as a sub-routine then generates the CNF-proof.

```
**Algorithm:** GenCNFProof($\mathbf{Y}, \mathbf{r}, \mathbf{j}$)
**Input:** Public inputs $\mathbf{Y} = (y_{ij}) \in (\mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q^2)^{m \times n}$, $y_{ij} = (pk_{ij}, m_{ij}, e_{ij})$
       Known randomizations $\mathbf{r} = (r_1, \ldots, r_m) \in \mathbb{Z}_q^m$
       Indices of known randomizations $\mathbf{j} = (j_1, \ldots, j_m) \in \{1, \ldots, n\}^m$
**for** $i = 1, \ldots, m$ **do**
    $c_i \leftarrow 0$
    **for** $j = 1, \ldots, n$ **do**
        **if** $j = j_i$ **then**
            $\omega_i \leftarrow$ GenRandomInteger($q$)                                   // see Alg. 4.11
            $t_{ij} \leftarrow (pk_{ij}^{\omega_i} \bmod p, g^{\omega_i} \bmod p)$
        **else**
            $c_{ij} \leftarrow$ GenRandomInteger($2^\tau$)                                // see Alg. 4.11
            $s_{ij} \leftarrow$ GenRandomInteger($q$)                                     // see Alg. 4.11
            $t_{ij} \leftarrow (pk_{ij}^{s_{ij}} \cdot (\frac{a_{ij}}{m_{ij}})^{c_{ij}} \bmod p, g^{s_{ij}} \cdot b_{ij}^{c_{ij}} \bmod p)$
            $c_i \leftarrow c_i + c_{ij} \bmod 2^\tau$
$\mathbf{T} \leftarrow (t_{ij})_{m \times n}$
$c \leftarrow$ GetChallenge($\mathbf{Y}, \mathbf{T}$)                                     // $c' \in \mathbb{Z}_{2^\tau}$, see Alg. 8.4
**for** $i = 1, \ldots, m$ **do**
    $j \leftarrow j_i$
    $c_{ij} \leftarrow c - c_i \bmod 2^\tau$
    $s_{ij} \leftarrow \omega_i - c_{ij} \cdot r_i \bmod q$
$\mathbf{C} \leftarrow (c_{ij})_{m \times n}$, $\mathbf{S} \leftarrow (s_{ij})_{m \times n}$
$\pi \leftarrow (\mathbf{C}, \mathbf{S})$
**return** $\pi$                                                                   // $\pi \in \mathbb{Z}_{2^\tau}^{m \times n} \times \mathbb{Z}_q^{m \times n}$
```

Algorithm 9.13: Generates a CNF proof of knowing at least one randomization in each row of an m-by-n matrix of ElGamal encryptions.

```
Algorithm: $\mathsf{CheckBallot}(v, \alpha, pk, \mathbf{pk}', \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{z}, \mathbf{v}, \hat{\mathbf{x}})$
Input: Voter index $v \in \{1, \dots N_E\}$
       Ballot
       $\alpha = (\hat{x}, \mathbf{a}, \pi, e', \pi') \in \mathbb{Z}_{\hat{q}} \times (\mathbb{G}_q^2)^k \times (\mathbb{Z}_{2^\tau} \times (\mathbb{Z}_{\hat{q}} \times \mathbb{G}_q \times \mathbb{Z}_q)) \times (\mathbb{G}_q^z \times \mathbb{G}_q) \times (\mathbb{Z}_{2^\tau}^{z \times 2} \times \mathbb{Z}_q^{z \times 2})$
       Encryption key $pk \in \mathbb{G}_q$
       Write-in encryption keys $\mathbf{pk}' \in \mathbb{G}_q^{z_{\max}}$
       Number of candidates $\mathbf{n} = (n_1, \dots, n_t) \in (\mathbb{N}^+)^t$, $n = \sum_{j=1}^t n_j$
       Number of selections $\mathbf{k} = (k_1, \dots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Eligibility matrix $\mathbf{E} = (e_{ij}) \in \mathbb{B}^{N_E \times t}$
       Write-in elections $\mathbf{z} = (z_1, \dots, z_t) \in \mathbb{B}^t$, $z_{\max} = \max_{i=1}^{N_E} \sum_{j=1}^t e_{ij} k_j z_j$
       Write-in candidates $\mathbf{v} \in \mathbb{B}^n$
       Public voting credentials $\hat{\mathbf{x}} = (\hat{x}_1, \dots, \hat{x}_{N_E}) \in \mathbb{G}_{\hat{q}}^{N_E}$
$k' \leftarrow \sum_{j=1}^t e_{vj} k_j$, $z' = \sum_{j=1}^t e_{vj} k_j z_j$
if $\hat{x} = \hat{x}_v$ and $k = k'$ and $z = z'$ then
    if $\mathsf{CheckBallotProof}(\pi, \hat{x}, \mathbf{a}, pk)$ then                     // see Alg. 8.26
        $\mathbf{p} \leftarrow \mathsf{GetPrimes}(n)$                                       // see Alg. 8.1
        $\mathbf{pk}' \leftarrow \mathbf{pk}'_{\{1,\dots,z\}}$
        $\mathbf{e}_v \leftarrow \mathsf{GetRow}(\mathbf{E}, v)$
        $(I, J) \leftarrow \mathsf{GetWriteInIndices}(\mathbf{n}, \mathbf{k}, \mathbf{e}_v, \mathbf{z}, \mathbf{v})$           // see Alg. 9.11
        return $\mathsf{CheckWriteInProof}(\pi', pk, \mathbf{a}_I, \mathbf{pk}', e', \mathbf{p}_J)$      // see Alg. 9.15
return $false$
```

Algorithm 9.14: Checks if a ballot α obtained from voter v is valid. This algorithm is an extension of Alg. 8.25 with an additional test for checking the validity of π'.

```
Algorithm: $\mathsf{CheckWriteInProof}(\pi, pk, \mathbf{a}, \mathbf{pk}', e', \mathbf{p})$
Input: Write-in proof $\pi \in \mathbb{Z}_{2^\tau}^{z \times 2} \times \mathbb{Z}_q^{z \times 2}$
       Encryption key $pk \in \mathbb{G}_q$
       Encrypted selections $\mathbf{a} = (a_1, \dots, a_z) \in (\mathbb{G}_q^2)^z$
       Write-in encryption keys $\mathbf{pk}' = (pk'_1, \dots, pk'_{z_{\max}}) \in \mathbb{G}_q^{z_{\max}}$, $z \leqslant z_{\max}$
       Encrypted write-ins $e' = ((a'_1, \dots, a'_z), b') \in \mathbb{G}_q^z \times \mathbb{G}_q$
       Encoded write-in candidates $\mathbf{p} = (p_1, \dots, p_z) \in (\mathbb{G}_q \cap \mathbb{P})^z$
$\varepsilon \leftarrow \mathsf{GetEncodedStrings}((\texttt{""}, \texttt{""}), A_W, \ell_W, c_W)$                  // see Alg. 9.1
for $i = 1, \dots, z$ do
    $y^*_{i,1} \leftarrow (pk, p_i, a_i)$
    $y^*_{i,2} \leftarrow (pk'_i, \varepsilon, (a'_i, b'))$
$\mathbf{Y}^* \leftarrow (y^*_{ij})_{z \times 2}$
$b \leftarrow \mathsf{CheckCNFProof}(\pi, \mathbf{Y}^*)$                                    // see Alg. 9.16
return $b$
```

Algorithm 9.15: Checks the correctness of a NIZKP π generated by Alg. 9.12. Essentially the same preparatory steps are conducted to normalize the input into the particular form of the CNF-proof of Section 9.1.2.2, which can then be verified by calling Alg. 9.16 as a sub-routine.

```
Algorithm: CheckCNFProof($\pi, \mathbf{Y}$)
Input: CNF proof $\pi = (\mathbf{C}, \mathbf{S}) \in \mathbb{Z}_{2^\tau}^{m \times n} \times \mathbb{Z}_q^{m \times n}$, $\mathbf{C} = (c_{ij})$, $\mathbf{S} = (s_{ij})$
       Public values $\mathbf{Y} = (y_{ij}) \in (\mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q^2)^{m \times n}$, $y_{ij} = (pk_{ij}, m_{ij}, e_{ij})$, $e_{ij} = (a_{ij}, b_{ij})$
for $i = 1, \ldots, m$ do
    $c_i \leftarrow \sum_{j=1}^{n} c_{ij} \bmod 2^\tau$
    for $j = 1, \ldots, n$ do
        $t_{ij} \leftarrow (pk_{ij}^{s_{ij}} \cdot (\frac{a_{ij}}{m_{ij}})^{c_{ij}} \bmod p, g^{s_{ij}} \cdot b_{ij}^{c_{ij}} \bmod p)$
$\mathbf{T} \leftarrow (t_{ij})_{m \times n}$
$c' \leftarrow$ GetChallenge($\mathbf{Y}, \mathbf{T}$)                         // $c' \in \mathbb{Z}_{2^\tau}$, see Alg. 8.4
return $c_1 = \cdots = c_m = c'$
```

Algorithm 9.16: Checks the correctness of a NIZKP π generated by Alg. 9.13.

9.3.4. Post-Election Phase

Most protocol changes in the post-election phase are necessary to enable the processing of the augmented ElGamal encryptions $e_i \in \mathbb{E}_z$ contained in the ballots α_i. This affects the algorithm GetEncryptions, which is responsible for normalizing the size of the submitted augmented encryptions from z to the maximal number of write-ins $z_{\max}$ (see explanations given at the beginning of Section 9.1.2.3). It also affects the shuffling algorithm GenShuffle, which performs the re-encryptions, and the shuffle proof algorithms GenShuffleProof and CheckShuffleProof, which have to be modified according to the discussion in Section 9.1.2.3. Similar changes are necessary in the algorithms GetDecryptions and GetVotes, which perform the (partial) decryption of the votes and the write-ins. Clearly, this also affects the algorithms GenDecryptionProof and CheckDecryptionProof, which generate and verify corresponding decryption proofs. A new algorithm GetWriteIns is added to decode and select the submitted write-ins from a bare matrix $\mathbf{M}$ of decrypted plaintext write-ins and to assign them to respective elections.

Nr.	Algorithm	Called by	Protocol
8.40 $\Rightarrow$ 9.17	GetEncryptions$(B, C, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{w}, \mathbf{z})$	Election authority	7.7
8.41	↳ GetDefaultEligibility$(\mathbf{w}, \mathbf{E})$		
8.42 $\Rightarrow$ 9.18	GenShuffle$(\mathbf{e}, pk, \mathbf{pk}')$		
8.43	↳ GenPermutation(N)		
8.44 $\Rightarrow$ 9.19	↳ GenReEncryption$(e, pk, \mathbf{pk}')$		
8.45 $\Rightarrow$ 9.20	GenShuffleProof$(U, \mathbf{e}, \tilde{\mathbf{e}}, \tilde{\mathbf{r}}, \tilde{\mathbf{r}}', \psi, pk, \mathbf{pk}')$		
8.46	↳ GenPermutationCommitment$(\psi, \mathbf{h})$		
8.47	↳ GenCommitmentChain$(\tilde{\mathbf{u}})$		
8.48 $\Rightarrow$ 9.21	CheckShuffleProof$(U, \pi, \mathbf{e}, \tilde{\mathbf{e}}, pk, \mathbf{pk}')$		
8.49 $\Rightarrow$ 9.22	GetDecryptions$(\mathbf{e}, sk, \mathbf{sk}')$	Election authority	7.8
8.50 $\Rightarrow$ 9.23	GenDecryptionProof$(sk, pk, \mathbf{sk}', \mathbf{pk}', \mathbf{e}, \mathbf{c}, \mathbf{D})$		
8.51 $\Rightarrow$ 9.24	CheckDecryptionProof$(\pi, pk, \mathbf{pk}', \mathbf{e}, \mathbf{c}, \mathbf{D})$		
8.52 $\Rightarrow$ 9.25	GetCombinedDecryptions$(\mathbf{C}, \vec{\mathbf{d}})$		
8.49 $\Rightarrow$ 9.22	GetDecryptions$(\mathbf{e}, sk, \mathbf{sk}')$	Administrator	7.9
8.50 $\Rightarrow$ 9.23	GenDecryptionProof$(sk, pk, \mathbf{sk}', \mathbf{pk}', \mathbf{e}, \mathbf{c}, \mathbf{D})$		
8.53 $\Rightarrow$ 9.26	GetVotes$(\mathbf{e}, \mathbf{c}, \mathbf{c}', \mathbf{D}, \mathbf{D}')$		
8.54 $\Rightarrow$ 9.27	GetElectionResult$(\mathbf{m}, \mathbf{n}, \mathbf{w}, \mathbf{M}, \mathbf{k}, \mathbf{z})$		
9.28	↳ GetWriteIns$(\mathbf{M}, \mathbf{V}, \mathbf{n}, \mathbf{k}, \mathbf{z})$		
9.29	↳ GetEligibility$(\mathbf{v}, \mathbf{n})$		
9.2	↳ GetDecodedStrings(y, A, ℓ, c)		

Table 9.6.: Overview of post-election phase algorithms used to implement write-ins.

```
Algorithm: GetEncryptions$(B, C, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{w}, \mathbf{z})$
Input: Ballot list $B = \langle(v_i, \alpha_i)\rangle_{i=0}^{N_B-1}$, $v_i \in \{1, \ldots, N_E\}$
       Confirmation list $C = \langle(v_i, \gamma_i)\rangle_{i=0}^{N_C-1}$, $v_i \in \{1, \ldots, N_E\}$
       Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$
       Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Eligibility matrix $\mathbf{E} = (e_{ij}) \in \mathbb{B}^{N_E \times t}$
       Counting circles $\mathbf{w} = (w_1, \ldots, w_{N_E}) \in (\mathbb{N}^+)^{N_E}$
       Write-in elections $\mathbf{z} = (z_1, \ldots, z_t) \in \mathbb{B}^t$
$n = \sum_{j=1}^{t} n_j$
$w \leftarrow \max_{i=1}^{N_W} w_i$
$z_{\max} \leftarrow \max_{i=1}^{N_E} \sum_{j=1}^{t} e_{ij} k_j z_j$
$\mathbf{E}^* \leftarrow$ GetDefaultEligibility$(\mathbf{w}, \mathbf{E})$          // $\mathbf{E}^* = (e^*_{cj})_{w \times t}$, see Alg. 8.41
$\mathbf{p} \leftarrow$ GetPrimes$(n + w)$          // $\mathbf{p} = (p_0, \ldots, p_{n+w})$, see Alg. 8.1
$i \leftarrow 0$
foreach $(v, \alpha) \in B$ do  // $\alpha = (\hat{x}, (a_1, \ldots, a_k), \pi, ((a'_1, \ldots, a'_{z'}), b'), \pi')$, $a_j = (a_{j,1}, a_{j,2})$
    if $(v, \cdot) \in C$ then
        $c \leftarrow w_v$
        $a \leftarrow p_{n+c}$          // add counting circle
        $n' \leftarrow 0$
        for $l = 1, \ldots, t$ do
            if $z_l = 0$ and $e_{vl} < e^*_{cl}$ then          // check for restricted eligibility
                for $j = n' + 1, \ldots, n' + k_l$ do          // add default candidates
                    $a \leftarrow a \cdot p_j \bmod p$
            $n' \leftarrow n' + n_l$
        $\mathbf{a}' = (a'_1, \ldots, a'_{z'}, 1, \ldots, 1)$          // normalize length to $z_{\max} = |\mathbf{a}'|$
        $i \leftarrow i + 1$
        $e_i \leftarrow (a \cdot \prod_{j=1}^{k} a_{j,1} \bmod p, \prod_{j=1}^{k} a_{j,2} \bmod p, \mathbf{a}', b')$
        if $a \neq p_{n+c}$ then
            $i \leftarrow i + 1$
            $e_i \leftarrow (p_0 \cdot a \bmod p, 1, (1, \ldots, 1), 1)$          // mark encryption with $p_0$
$N \leftarrow i$
$\mathbf{e} = (e_1, \ldots, e_N)$
$\mathbf{e}' \leftarrow$ Sort$_{\leq}(\mathbf{e})$
return $\mathbf{e}'$          // $\mathbf{e}' \in (\mathbb{E}_{z_{\max}})^N$
```

Algorithm 9.17: Computes a sorted vector of encryptions from the list of submitted ballots. This algorithm is an extension of Alg. 8.40 from Section 8.4 with additional code for handling the augmented ElGamal encryptions. It differs from Alg. 8.40 by restricting the attachment of default candidates to elections, in which write-ins are not accepted (see Footnote 5 on Page 137).

```
Algorithm: $\mathsf{GenShuffle}(\mathbf{e}, pk, \mathbf{pk}')$
Input: Augmented encryptions $\mathbf{e} = (e_1, \ldots, e_N) \in \mathbb{E}_z^N$
       Encryption key $pk \in \mathbb{G}_q$
       Write-in encryption keys $\mathbf{pk}' \in \mathbb{G}_q^z$
$\psi \leftarrow \mathsf{GenPermutation}(N)$                // $\psi = (j_1, \ldots, j_N) \in \Psi_N$, see Alg. 8.43
for $i = 1, \ldots, N$ do
    $(\tilde{e}_i, \tilde{r}_{j_i}, \tilde{r}'_{j_i}) \leftarrow \mathsf{GenReEncryption}(e_{j_i}, pk, \mathbf{pk}')$       // see Alg. 9.19
$\tilde{\mathbf{e}} \leftarrow (\tilde{e}_1, \ldots, \tilde{e}_N)$
$\tilde{\mathbf{r}} \leftarrow (\tilde{r}_1, \ldots, \tilde{r}_N)$
$\tilde{\mathbf{r}}' \leftarrow (\tilde{r}'_1, \ldots, \tilde{r}'_N)$
return $(\tilde{\mathbf{e}}, \tilde{\mathbf{r}}, \tilde{\mathbf{r}}', \psi)$      // $\tilde{\mathbf{e}} \in (\mathbb{E}_z)^N$, $\tilde{\mathbf{r}} \in \mathbb{Z}_q^N$, $\tilde{\mathbf{r}}' \in \mathbb{Z}_q^N$, $\psi \in \Psi_N$
```

Algorithm 9.18: Generates a random permutation $\psi \in \Psi_N$ and uses it to shuffle a given vector $\mathbf{e} = (e_1, \ldots, e_N)$ of augmented ElGamal encryptions $e_i \in \mathbb{E}_z$. This algorithm is an extension of Alg. 8.42 from Section 8.4.

```
Algorithm: $\mathsf{GenReEncryption}(e, pk, \mathbf{pk}')$
Input: Augmented encryption $e = (a, b, (a'_1, \ldots, a'_z), b') \in \mathbb{E}_z$
       Encryption key $pk \in \mathbb{G}_q$
       Write-in encryption keys $\mathbf{pk}' = (pk'_1, \ldots, pk'_z) \in \mathbb{G}_q^z$
$\tilde{r} \leftarrow \mathsf{GenRandomInteger}(q)$, $\tilde{r}' \leftarrow \mathsf{GenRandomInteger}(q)$      // see Alg. 4.11
$\tilde{a} \leftarrow a \cdot pk^{\tilde{r}} \bmod p$
$\tilde{b} \leftarrow b \cdot g^{\tilde{r}} \bmod p$
for $i = 1 \ldots, z$ do
    $\tilde{a}'_i \leftarrow a'_i \cdot (pk'_i)^{\tilde{r}'} \bmod p$
$\tilde{b}' \leftarrow b' \cdot g^{\tilde{r}'} \bmod p$
$\tilde{e} \leftarrow (\tilde{a}, \tilde{b}, (\tilde{a}'_1, \ldots, \tilde{a}'_z), \tilde{b}')$
return $(\tilde{e}, \tilde{r}, \tilde{r}')$                // $\tilde{e} \in \mathbb{E}_z$, $\tilde{r} \in \mathbb{Z}_q$, $\tilde{r}' \in \mathbb{Z}_q$
```

Algorithm 9.19: Generates a re-encryption of the given augmented ElGamal encryption $e \in \mathbb{E}_z$. This algorithm is an extension of Alg. 8.44 of Section 8.4.

```
Algorithm: $\mathsf{GenShuffleProof}(U, \mathbf{e}, \tilde{\mathbf{e}}, \tilde{\mathbf{r}}, \tilde{\mathbf{r}}', \psi, pk, \mathbf{pk}')$
Input: Election event identifier $U \in A^*_{\mathsf{ucs}}$
       Augmented encryptions $\mathbf{e} \in \mathbb{E}_z^N$
       Shuffled encryptions $\tilde{\mathbf{e}} = (\tilde{e}_1, \ldots, \tilde{e}_N)$, $\tilde{e}_i = (\tilde{a}_i, \tilde{b}_i, (\tilde{a}'_{i,1}, \ldots, \tilde{a}'_{i,z}), \tilde{b}'_i) \in \mathbb{E}_z$
       Re-encryption randomizations $\tilde{\mathbf{r}} = (\tilde{r}_1, \ldots, \tilde{r}_N) \in \mathbb{Z}_q^N$
       Re-encryption randomizations $\tilde{\mathbf{r}}' = (\tilde{r}'_1, \ldots, \tilde{r}'_N) \in \mathbb{Z}_q^N$
       Permutation $\psi = (j_1, \ldots, j_N) \in \Psi_N$
       Encryption key $pk \in \mathbb{G}_q$
       Write-in encryption keys $\mathbf{pk}' = (pk'_1, \ldots, pk'_z) \in \mathbb{G}_q^z$
 ⋮                                                          // same code as in Alg. 8.45
$\omega_1 \leftarrow \mathsf{GenRandomInteger}(q)$, $\omega_2 \leftarrow \mathsf{GenRandomInteger}(q)$, $\omega_3 \leftarrow \mathsf{GenRandomInteger}(q)$
$\omega_4 \leftarrow \mathsf{GenRandomInteger}(q)$, $\omega'_4 \leftarrow \mathsf{GenRandomInteger}(q)$        // see Alg. 4.11
 ⋮                                                          // same code as in Alg. 8.45
for $j = 1, \ldots, z$ do
    $t_{4,3,j} \leftarrow (pk'_j)^{-\omega'_4} \prod_{i=1}^N (\tilde{a}'_{ij})^{\tilde{\omega}_i} \bmod p$
$\mathbf{t}_{4,3} \leftarrow (t_{4,3,1}, \ldots, t_{4,3,z})$
$t_{4,4} \leftarrow g^{-\omega'_4} \prod_{i=1}^N (\tilde{b}'_i)^{\tilde{\omega}_i} \bmod p$
 ⋮                                                          // same code as in Alg. 8.45
$t \leftarrow (t_1, t_2, t_3, (t_{4,1}, t_{4,2}, \mathbf{t}_{4,3}, t_{4,4}), (\hat{t}_1, \ldots, \hat{t}_N))$
$y \leftarrow (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}, \hat{\mathbf{c}}, pk, \mathbf{pk}')$
 ⋮                                                          // same code as in Alg. 8.45
$\tilde{r}' \leftarrow \sum_{i=1}^N \tilde{r}'_i u_i \bmod q$, $s'_4 \leftarrow \omega'_4 - c \cdot \tilde{r}' \bmod q$
 ⋮                                                          // same code as in Alg. 8.45
$s \leftarrow (s_1, s_2, s_3, (s_4, s'_4), (\hat{s}_1, \ldots, \hat{s}_N), (\tilde{s}_1, \ldots, \tilde{s}_N))$
$\pi \leftarrow (c, s, \mathbf{c}, \hat{\mathbf{c}})$
return $\pi$                      // $\pi \in \mathbb{Z}_{2^\tau} \times (\mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q^2 \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N) \times \mathbb{G}_q^N \times \mathbb{G}_q^N$
```

Algorithm 9.20: Generates a shuffle proof π relative to augmented ElGamal encryptions $\mathbf{e}$ and $\tilde{\mathbf{e}}$, which is equivalent to proving knowledge of a permutation ψ and randomizations $\tilde{\mathbf{r}}$ and $\tilde{\mathbf{r}}'$ such that $\tilde{\mathbf{e}} = \mathsf{Shuffle}_{pk,\mathbf{pk}'}(\mathbf{e}, \tilde{\mathbf{r}}, \tilde{\mathbf{r}}', \psi)$. This algorithm is an extension of Alg. 8.45 from Section 8.4. Here we only show the necessary code lines for extending the proof generation from regular to augmented ElGamal encryptions. For the proof verification, see Alg. 9.21.

```
Algorithm: CheckShuffleProof$(U, \pi, \mathbf{e}, \tilde{\mathbf{e}}, pk, \mathbf{pk}')$
Input: Election event identifier $U \in A^*_{\mathsf{ucs}}$
       Shuffle proof $\pi = (c, s, \mathbf{c}, \hat{\mathbf{c}}) \in \mathbb{Z}_{2^\tau} \times (\mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q \times \mathbb{Z}_q^2 \times \mathbb{Z}_q^N \times \mathbb{Z}_q^N) \times \mathbb{G}_q^N \times \mathbb{G}_q^N$,
       $s = (s_1, s_2, s_3, (s_4, s'_4), (\hat{s}_1, \ldots, \hat{s}_N), (\tilde{s}_1, \ldots, \tilde{s}_N))$, $\mathbf{c} = (c_1, \ldots, c_N)$,
       $\hat{\mathbf{c}} = (\hat{c}_1, \ldots, \hat{c}_N)$
       Augmented encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i, (a'_{i,1}, \ldots, a'_{i,z}), b'_i) \in \mathbb{E}_z$
       Shuffled encryptions $\tilde{\mathbf{e}} = (\tilde{e}_1, \ldots, \tilde{e}_N)$, $\tilde{e}_i = (\tilde{a}_i, \tilde{b}_i, (\tilde{a}'_{i,1}, \ldots, \tilde{a}'_{i,z}), \tilde{b}'_i) \in \mathbb{E}_z$
       Encryption key $pk \in \mathbb{G}_q$
       Write-in encryption keys $\mathbf{pk}' = (pk'_1, \ldots, pk'_z) \in \mathbb{G}_q^z$
  ⋮                                                        // same code as in Alg. 8.48
for $j = 1, \ldots, z$ do
    $a'_j \leftarrow \prod_{i=1}^N (a'_{ij})^{u_i} \bmod p$
    $t_{4,3,j} \leftarrow (a'_j)^c \cdot (pk'_j)^{-s'_4} \prod_{i=1}^N (\tilde{a}'_{ij})^{\tilde{s}_i} \bmod p$
$\mathbf{t}_{4,3} \leftarrow (t_{4,3,1}, \ldots, t_{4,3,z})$
$b' \leftarrow \prod_{i=1}^N (b'_i)^{u_i} \bmod p$
$t_{4,4} \leftarrow (b')^c \cdot g^{-s'_4} \prod_{i=1}^N (\tilde{b}'_i)^{\tilde{s}_i} \bmod p$
  ⋮                                                        // same code as in Alg. 8.48
$t \leftarrow (t_1, t_2, t_3, (t_{4,1}, t_{4,2}, \mathbf{t}_{4,3}, t_{4,4}), (\hat{t}_1, \ldots, \hat{t}_N))$
$y \leftarrow (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}, \hat{\mathbf{c}}, pk, \mathbf{pk}')$
$c' \leftarrow$ GetChallenge$(y, t)$                                  // see Alg. 8.4
return $c = c'$
```

Algorithm 9.21: Checks the correctness of a shuffle proof π generated by Alg. 9.20. This algorithm is an extension of Alg. 8.48 from Section 8.4. Here we only show the necessary code lines for extending the proof verification from regular to augmented ElGamal encryptions.

```
Algorithm: GetDecryptions$(\mathbf{e}, sk, \mathbf{sk}')$
Input: Augmented encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i, \mathbf{a}'_i, b'_i) \in \mathbb{E}_z$
       Decryption key share $sk \in \mathbb{Z}_q$
       Write-in decryption key shares $\mathbf{sk}' = (sk'_1, \ldots, sk'_z) \in \mathbb{Z}_q^z$
for $i = 1, \ldots, N$ do
    $c_i \leftarrow b_i^{sk} \bmod p$
    for $j = 1, \ldots, z$ do
        $d_{ij} \leftarrow (b'_i)^{sk'_j} \bmod p$
$\mathbf{c} \leftarrow (c_1, \ldots, c_N)$, $\mathbf{D} \leftarrow (d_{ij})_{N \times z}$
return $(\mathbf{c}, \mathbf{D})$                              // $\mathbf{c} \in \mathbb{G}_q^N$, $\mathbf{D} \in \mathbb{G}_q^{N \times z}$
```

Algorithm 9.22: Computes the partial decryptions of a given input vector $\mathbf{e}$ of augmented encryptions using the shares sk and $\mathbf{sk}'$ of the private decryption keys. This algorithm is an extension of Alg. 8.49 from Section 8.4.

```
Algorithm: GenDecryptionProof($sk, pk, \mathbf{sk}', \mathbf{pk}', \mathbf{e}, \mathbf{c}, \mathbf{D}$)
Input: Decryption key share $sk \in \mathbb{Z}_q$
       Encryption key share $pk \in \mathbb{G}_q$
       Write-in decryption key share $\mathbf{sk}' = (sk'_1, \ldots, sk'_z) \in \mathbb{Z}_q^z$
       Write-in encryption key shares $\mathbf{pk}' \in \mathbb{Z}_q^z$
       Augmented encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i, \mathbf{a}'_i, b'_i) \in \mathbb{E}_z$
       Partial decryptions $\mathbf{c} \in \mathbb{G}_q^N$
       Partial write-in decryptions $\mathbf{D} \in \mathbb{G}_q^{N \times z}$
for $j = 0, \ldots, z$ do
    $\omega_j \leftarrow$ GenRandomInteger($q$)                    // see Alg. 4.11
    for $i = 0, \ldots, N$ do
        if $i = 0$ then
            $t_{ij} \leftarrow g^{\omega_j} \bmod p$
        else if $j = 0$ then
            $t_{ij} \leftarrow b_i^{\omega_j} \bmod p$
        else
            $t_{ij} \leftarrow (b'_i)^{\omega_j} \bmod p$
$\mathbf{T} \leftarrow (t_{ij})_{(N+1) \times (z+1)}$
$y \leftarrow (pk, \mathbf{pk}', \mathbf{e}, \mathbf{c}, \mathbf{D})$
$c \leftarrow$ GetChallenge($y, \mathbf{T}$)                       // see Alg. 8.4
for $j = 0, \ldots, z$ do
    if $j = 0$ then
        $s_j \leftarrow \omega_j - c \cdot sk \bmod q$
    else
        $s_j \leftarrow \omega_j - c \cdot sk'_j \bmod q$
$\mathbf{s} \leftarrow (s_0, \ldots, s_z)$
$\pi \leftarrow (c, \mathbf{s})$
return $\pi$                                          // $\pi \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_q^{z+1}$
```

Algorithm 9.23: Generates a decryption proof relative to augmented encryptions **e** and partial decryptions **c** and **D**. This algorithm is an extension of Alg. 8.50 from Section 8.4. For the proof verification, see Alg. 9.24.

```
Algorithm: CheckDecryptionProof($\pi, pk, \mathbf{pk}', \mathbf{e}, \mathbf{c}, \mathbf{D}$)
Input: Decryption proof $\pi = (c, \mathbf{s}) \in \mathbb{Z}_{2^\tau} \times \mathbb{Z}_q^{z+1}$, $\mathbf{s} = (s_0, \ldots, s_z)$
       Encryption key share $pk \in \mathbb{G}_q$
       Write-in encryption keys $\mathbf{pk}' = (pk'_1, \ldots, pk'_z) \in \mathbb{Z}_q^z$
       Augmented encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i, \mathbf{a}'_i, b'_i) \in \mathbb{E}_z$
       Partial decryptions $\mathbf{c} = (c_1, \ldots, c_N) \in \mathbb{G}_q^N$
       Partial write-in decryptions $\mathbf{D} = (d_{ij}) \in \mathbb{G}_q^{N \times z}$
for $j = 0, \ldots, z$ do
    for $i = 0, \ldots, N$ do
        if $i = j = 0$ then
            $t_{ij} \leftarrow pk^c \cdot g^{s_j} \bmod p$
        else if $i = 0$ then
            $t_{ij} \leftarrow (pk'_j)^c \cdot g^{s_j} \bmod p$
        else if $j = 0$ then
            $t_{ij} \leftarrow c_i^c \cdot b_i^{s_j}$
        else
            $t_{ij} \leftarrow (d_{ij})^c \cdot (b'_i)^{s_j} \bmod p$
$\mathbf{T} \leftarrow (t_{ij})_{(N+1) \times (z+1)}$
$y \leftarrow (pk, \mathbf{pk}', \mathbf{e}, \mathbf{c}, \mathbf{D})$
$c' \leftarrow$ GetChallenge($y, \mathbf{T}$)                                  // see Alg. 8.4
return $c = c'$
```

Algorithm 9.24: Checks the correctness of a decryption proof π generated by Alg. 9.23. This algorithm is an extension of Alg. 8.51 from Section 8.4.

```
Algorithm: GetCombinedDecryptions($\mathbf{C}, \vec{\mathbf{d}}$)
Input: Partial decryptions $\mathbf{C} = (c_{ij}) \in \mathbb{G}_q^{N \times s}$
       Partial write-in decryptions $\vec{\mathbf{d}} = (\mathbf{D}_1, \ldots, \mathbf{D}_s)$, $\mathbf{D}_k = (d_{ijk}) \in \mathbb{G}_q^{N \times z}$
for $i = 1, \ldots, N$ do
    $c_i \leftarrow \prod_{j=1}^{s} c_{ij} \bmod p$
    for $j = 1, \ldots, z$ do
        $d_{ij} \leftarrow \prod_{k=1}^{s} d_{ijk} \bmod p$
$\mathbf{c} \leftarrow (c_1, \ldots, c_N)$, $\mathbf{D} \leftarrow (d_{ij})_{N \times z}$
return $(\mathbf{c}, \mathbf{D})$                                  // $\mathbf{c} \in \mathbb{G}_q^N$, $\mathbf{D} \in \mathbb{G}_q^{N \times z}$
```

Algorithm 9.25: Computes the vector $\mathbf{c} = (c_1, \ldots, c_N)$ and the matrix $\mathbf{D} = (d_{ij})_{N \times z}$ of combined partial decryptions.

```
Algorithm: $\mathsf{GetVotes}(\mathbf{e}, \mathbf{c}, \mathbf{c}', \mathbf{D}, \mathbf{D}')$
Input: Encryptions $\mathbf{e} = (e_1, \ldots, e_N)$, $e_i = (a_i, b_i, \mathbf{a}'_i, b'_i) \in \mathbb{E}_z$, $\mathbf{a}'_i = (a'_{i,1}, \ldots, a'_{i,z})$
       First partial decryptions $\mathbf{c} = (c_1, \ldots, c_N) \in \mathbb{G}_q^N$
       Second partial decryptions $\mathbf{c}' = (c'_1, \ldots, c'_N) \in \mathbb{G}_q^N$
       First partial write-in decryptions $\mathbf{D} = (d_{ij}) \in \mathbb{G}_q^{N \times z}$
       Second partial write-in decryptions $\mathbf{D}' = (d'_{ij}) \in \mathbb{G}_q^{N \times z}$
for $i = 1, \ldots, N$ do
    $m_i \leftarrow a_i \cdot (c_i\, c'_i)^{-1} \bmod p$
    for $j = 1, \ldots, z$ do
        $m_{ij} \leftarrow a'_{ij} \cdot (d_{ij}\, d'_{ij})^{-1} \bmod p$
$\mathbf{m} \leftarrow (m_1, \ldots, m_N)$, $\mathbf{M} \leftarrow (m_{ij})_{N \times z}$
return $(\mathbf{m}, \mathbf{M})$                                  // $\mathbf{m} \in \mathbb{G}_q^N$, $\mathbf{M} \in \mathbb{G}_q^{N \times z}$
```

Algorithm 9.26: Computes the vector $\mathbf{m} = (m_1, \ldots, m_N)$ of decrypted plaintext votes and the matrix $\mathbf{M} = (m_{ij})_{N \times z}$ of decrypted write-ins by deducting the partial decryptions c_i, c'_i, d_{ij}, and d'_{ij} from the ElGamal encryptions. This algorithm is an extension of Alg. 8.53 from Section 8.4.

```
Algorithm: $\mathsf{GetElectionResult}(\mathbf{m}, \mathbf{n}, \mathbf{w}, \mathbf{M}, \mathbf{k}, \mathbf{z})$
Input: Encoded selections $\mathbf{m} = (m_1, \ldots, m_N) \in \mathbb{G}_q^N$
       Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$
       Counting circles $\mathbf{w} = (w_1, \ldots, w_{N_E}) \in (\mathbb{N}^+)^{N_E}$
       Encoded write-ins $\mathbf{M} = (m_{ij}) \in \mathbb{G}_q^{N \times z}$
       Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Write-in elections $\mathbf{z} = (z_1, \ldots, z_t) \in \mathbb{B}^t$
$n \leftarrow \sum_{j=1}^{t} n_j$, $w \leftarrow \max_{i=1}^{N_E} w_i$
$\mathbf{p} \leftarrow \mathsf{GetPrimes}(n + w)$                    // $\mathbf{p} = (p_0, \ldots, p_{n+w})$, see Alg. 8.1
for $i = 1, \ldots, N$ do
    for $k = 1, \ldots, n$ do
        if $p_k \mid m_i$ then
            $v_{ik} \leftarrow 1$
        else
            $v_{ik} \leftarrow 0$
    for $j = 1, \ldots, w$ do
        if $p_{n+j} \mid m_i$ then
            $w_{ij} \leftarrow 1$
        else
            $w_{ij} \leftarrow 0$
    if $p_0 \mid m_i$ then
        $u_i \leftarrow -1$
    else
        $u_i \leftarrow 1$
$\mathbf{u} = (u_1, \ldots, u_N)$, $\mathbf{V} \leftarrow (v_{ik})_{N \times n}$, $\mathbf{W} \leftarrow (w_{ij})_{N \times w}$
$(\mathbf{S}, \mathbf{T}) \leftarrow \mathsf{GetWriteIns}(\mathbf{M}, \mathbf{V}, \mathbf{n}, \mathbf{k}, \mathbf{z})$
$ER \leftarrow (\mathbf{u}, \mathbf{V}, \mathbf{W}, \mathbf{S}, \mathbf{T})$
return $ER$    // $ER \in \{-1, 1\}^N \times \mathbb{B}^{N \times n} \in \mathbb{B}^{N \times w} \times (\mathcal{W} \cup \{\varnothing\})^{N \times z} \times (\{\varnothing, 1, \ldots, t\})^{N \times z}$
```

Algorithm 9.27: Computes the election result from the products of encoded selections $\mathbf{m} = (m_1, \ldots, m_N)$ by retrieving the prime factors of each m_i. Each value $v_{ik} = 1$ represents somebody's vote for a specific candidate $k \in \{1, \ldots, n\}$.

```
Algorithm: GetWriteIns($\mathbf{M}, \mathbf{V}, \mathbf{n}, \mathbf{k}, \mathbf{z}$)
Input: Encoded write-ins $\mathbf{M} = (m_{ij}) \in \mathbb{G}_q^{N \times z}$
       Plaintext votes $\mathbf{V} = (v_{ik}) \in \mathbb{B}^{N \times n}$
       Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in (\mathbb{N}^+)^t$, $n = \sum_{j=1}^{t} n_j$
       Number of selections $\mathbf{k} = (k_1, \ldots, k_t) \in (\mathbb{N}^+)^t$, $k_j < n_j$
       Write-in elections $\mathbf{z} = (z_1, \ldots, z_t) \in \mathbb{B}^t$
for $i = 1, \ldots, N$ do
    $\mathbf{v} \leftarrow$ GetRow($\mathbf{V}, i$)
    $\mathbf{e} \leftarrow$ GetEligibility($\mathbf{v}, \mathbf{n}$)                    // $\mathbf{e} = (e_1, \ldots, e_t)$, see Alg. 9.29
    $k \leftarrow 1$                                                 // loop over $k = 1, \ldots, z$
    for $j = 1, \ldots, t$ do
        if $e_j = 1$ and $z_j = 1$ then
            for $l = 1, \ldots, k_j$ do
                if $k > z$ then
                    return $\perp$                                  // $z$ incompatible with $\mathbf{v}$
                $S_{ik} \leftarrow$ GetDecodedStrings($m_{ik}, A_W, \ell_W, c_W$)          // see Alg. 9.2
                $t_{ik} \leftarrow j$
                $k \leftarrow k + 1$
    while $k \leqslant z$ do
        $S_{ik} \leftarrow \varnothing$, $t_{ik} \leftarrow \varnothing$
        $k \leftarrow k + 1$
$\mathbf{S} = (S_{ik})_{N \times z}$, $\mathbf{T} = (t_{ik})_{N \times z}$
return $(\mathbf{S}, \mathbf{T})$                     // $\mathbf{S} \in (\mathcal{W} \cup \{\varnothing\})^{N \times z}$, $\mathbf{T} \in (\{\varnothing, 1, \ldots, t\})^{N \times z}$
```

Algorithm 9.28: Computes the write-in string pairs S_{ik} and assigns them to the corresponding elections $t_{ik} \in \{1, \ldots, t\}$. Some values of the resulting matrices $\mathbf{S}$ and $\mathbf{T}$ are set to $\varnothing$. This is a consequence of extending the write-in encryptions to the maximal size $z_{\max}$ in Alg. 9.17.

```
**Algorithm:** GetEligibility($\mathbf{v}, \mathbf{n}$)
**Input:** Votes $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbb{B}^n$
           Number of candidates $\mathbf{n} = (n_1, \ldots, n_t) \in \mathbb{N}^t$, $n = \sum_{j=1}^{t} n_j$
$n' \leftarrow 0$
for $j = 1, \ldots, t$ do
    $e_j \leftarrow 0$
    for $i = n' + 1, \ldots, n' + n_j$ do
        if $v_i = 1$ then
            $e_j \leftarrow 1$
    $n' \leftarrow n' + n_j$
$\mathbf{e} \leftarrow (e_1, \ldots, e_t)$
return $\mathbf{e}$                                        // $\mathbf{e} \in \mathbb{B}^t$
```

Algorithm 9.29: Derives the eligibility vector $\mathbf{e} = (e_1, \ldots, e_t)$ of an unknown voter from a given plaintext vote $\mathbf{v}$.

Part IV.

System Specification

10. 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 10.1, 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 10.2. We recommend that exactly these values are used in an actual implementation. In Section 11.1, we specify various alphabets and code lengths for the voting, confirmation, verification, and finalization codes.

10.1. Recommended Length Parameters

For each security level, an estimate of the achieved security strengths σ (privacy) and τ (integrity) is shown in Table 10.1. 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 10.1 are in accordance with current NIST recommendations [13, 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 [27]). 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			$\mathbb{G}_q \subset \mathbb{Z}_p^*$		$\mathbb{G}_{\hat{q}} \subset \mathbb{Z}_{\hat{p}}^*$		L_M	Crypto-period
		$\ell_{\min}$	$L_{\min}$	L	$\|p\|$	$\|q\|$	$\|\hat{p}\|$	$\|\hat{q}\|$		
0	16	32	4	32	48	47	48	32	8	Testing
1	80	160	20	32	1024	1023	1024	160	40	Legacy
2	112	224	28	32	2048	2047	2048	224	56	$\leqslant$ 2030
3	128	256	32	32	3072	3071	3072	256	64	> 2030

Table 10.1.: Length parameters according to current NIST recommendations. The length L_M of the OT messages follows deterministically from $\|\hat{q}\|$, see Table 6.1.

Since the minimal hash length that covers all three security levels is 256 bits (32 bytes), we propose to use $L = 32$ as a global parameter and SHA3-256 as general hash algorithm.

Therefore, $H \leftarrow \mathsf{Hash}_L(B)$ means calling the SHA3-256 algorithm with an arbitrarily long input byte array $B \in \mathcal{B}^*$ and assigning its return value to $H \in \mathcal{B}^{32}$. This is our general way of computing hash values for all security levels. It is used in Alg. 4.15 to compute hash values of multiple inputs.

10.2. Recommended Group and Field Parameters

In this section, we specify public parameters for $\mathbb{G}_q \subset \mathbb{Z}_p^*$ and $\mathbb{G}_{\hat{q}} \subset \mathbb{Z}_{\hat{p}}^*$ satisfying the bit lengths of the security levels $\lambda \in \{0, 1, 2, 3\}$ of Table 10.1. 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 [37]. Table 10.2 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 of length s.

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

Table 10.2.: Hexadecimal representation of Euler's number (first 3072 bits).[1]

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 $e_s \leqslant \hat{p} < 2^s$. 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. 8.3). For the groups $\mathbb{G}_{\hat{q}}$, we use $\hat{g} = 2^{\hat{k}} \bmod \hat{p}$ as default generator.

The following four subsections contain tables with values p, q, k, g, h, $\hat{p}$, $\hat{q}$, $\hat{k}$, and $\hat{q}$ for the four security levels. We also give lists $\mathbf{p} = (p_0, \ldots, p_{59})$ of the first 60 primes in $\mathbb{G}_q$, which are required to encode the selected candidates $\mathbf{s}$ as a single group element $\Gamma(\mathbf{s}) \in \mathbb{G}_q$ (see Section 5.3 and chapter 7 for more details). The algorithm given below can be used to

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

generate these parameters for arbitrary bit lengths $s = \|p\| = \|\hat{p}\|$ and $t = \|\hat{q}\|$. Note that for $s \geqslant 4$, the search for a safe prime $p = 2q+1$ can be improved by restricting the primality tests to candidates $p = 12k + 11$ and hence $q = 6k + 5$ for some integer $k \geqslant 0$ [57].

```
Algorithm: GetGroupParameters($s, t, \kappa$)
Input: Bit lenght modulo $s = \|p\| = \|\hat{p}\| \geqslant 4$
       Bit lenght group size $t = \|\hat{q}\|$, $2 \leqslant t < s$
       Failure probability $\leqslant \frac{1}{2^\kappa}$
$E \leftarrow$ "B7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA5..."   // see Table 10.2
$e \leftarrow \lfloor \mathsf{StringToInteger}(\mathsf{Truncate}(E, \lfloor s/4 \rfloor + 1), A_{16})/(2^{4-(s \bmod 4)}) \rfloor$   // see Alg. 4.8
$\hat{e} \leftarrow \lfloor \mathsf{StringToInteger}(\mathsf{Truncate}(E, \lfloor t/4 \rfloor + 1), A_{16})/(2^{4-(t \bmod 4)}) \rfloor$   // see Alg. 4.8
$p \leftarrow e + 11 - (e \bmod 12)$   // make $p - 11$ a multiple of 12
$q \leftarrow \frac{p-1}{2}$   // make $q - 5$ a multiple of 6
while $\neg\mathsf{IsProbablyPrime}(p, \kappa)$ or $\neg\mathsf{IsProbablyPrime}(q, \kappa)$ do
    $p \leftarrow p + 12$
    $q \leftarrow q + 6$
$\hat{q} \leftarrow \hat{e} + 1 - (\hat{e} \bmod 2)$   // make $\hat{q}$ odd
while $\neg\mathsf{IsProbablyPrime}(\hat{q}, \kappa)$ do
    $\hat{q} \leftarrow \hat{q} + 2$
$\hat{p} \leftarrow \lceil \frac{e}{2\hat{q}} \rceil \cdot 2\hat{q} + 1$   // make $\hat{p} - 1$ an even multiple of $\hat{q}$
while $\neg\mathsf{IsProbablyPrime}(\hat{p}, \kappa)$ do
    $\hat{p} \leftarrow \hat{p} + 2\hat{q}$
$\hat{k} \leftarrow \frac{\hat{p}-1}{\hat{q}}$, $\hat{g} \leftarrow 2^{\hat{k}} \bmod \hat{p}$
return $(p, q, \hat{p}, \hat{q}, \hat{k}, \hat{g})$   // $p = 2q+1 \in \mathbb{S}$, $\hat{p} = \hat{k}\hat{q}+1 \in \mathbb{P}$, $\hat{q} \in \mathbb{P}$, $g \in \mathbb{G}_{\hat{q}}$
```

Algorithm 10.1: Derives group parameters for given bit lengths s and t from the binary representation of Euler's number. It assumes the existence of a probabilistic primality test $\mathsf{IsProbablyPrime}(p, \kappa)$, which returns `true` for a given candidate p with a failure probability of $\frac{1}{2^\kappa}$ or less.

10.2.1. Level 0 (Testing Only)

p = 0xB7E151629927 = 202178360940839	$\hat{p}$ = 0xB7FC9CE51713 = 202295591900947
q = 0x5BF0A8B14C93 = 101089180470419	$\hat{q}$ = 0xB7E15173 = 3084996979
$k = 2$	$\hat{k}$ = 0x10026 = 65574
$g = 4$	$\hat{g}$ = 0x8145D710FE7F = 142136960941695
$h = 9$	

Table 10.3.: Groups $\mathbb{G}_q \subset \mathbb{Z}_p^*$ and $\mathbb{G}_{\hat{q}} \subset \mathbb{Z}_{\hat{p}}^*$ for security level $\lambda = 0$ with default generators g, h, and $\hat{g}$, respectively (used for testing only).

$p_0 = 2$	$p_{10} = 67$	$p_{20} = 137$	$p_{30} = 271$	$p_{40} = 367$	$p_{50} = 461$
$p_1 = 3$	$p_{11} = 73$	$p_{21} = 139$	$p_{31} = 281$	$p_{41} = 383$	$p_{51} = 479$
$p_2 = 5$	$p_{12} = 79$	$p_{22} = 149$	$p_{32} = 293$	$p_{42} = 389$	$p_{52} = 487$
$p_3 = 7$	$p_{13} = 83$	$p_{23} = 157$	$p_{33} = 307$	$p_{43} = 397$	$p_{53} = 491$
$p_4 = 11$	$p_{14} = 89$	$p_{24} = 173$	$p_{34} = 313$	$p_{44} = 409$	$p_{54} = 499$
$p_5 = 13$	$p_{15} = 97$	$p_{25} = 181$	$p_{35} = 331$	$p_{45} = 421$	$p_{55} = 503$
$p_6 = 23$	$p_{16} = 101$	$p_{26} = 191$	$p_{36} = 347$	$p_{46} = 431$	$p_{56} = 509$
$p_7 = 37$	$p_{17} = 113$	$p_{27} = 227$	$p_{37} = 349$	$p_{47} = 439$	$p_{57} = 521$
$p_8 = 47$	$p_{18} = 127$	$p_{28} = 251$	$p_{38} = 353$	$p_{48} = 449$	$p_{58} = 523$
$p_9 = 53$	$p_{19} = 131$	$p_{29} = 263$	$p_{39} = 359$	$p_{49} = 457$	$p_{59} = 541$

Table 10.4.: The first 60 prime numbers in $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for p and q as defined in Table 10.3.

10.2.2. Level 1

$p =$ 0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190CFEF324E7738926CFBE5F4BF8D8D8C31D763DA06C80ABB1185EB4F7C7B5757F5958490CFD47D7C19BB42158D9554F7B46BCED55C4D79FD5F24D6613C31C3839A2DDF8A9A276BCFBFA1C877C56284DAB79CD4C2B3293D20E9E5EAF02AC60ACC942593	$\hat{p} =$ 0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190CFEF324E7738926CFBE5F4BF8D8D8C31D763DA06C80ABB1185EB4F7C7B5757F5958490CFD47D7C19BB42158D9554F7B46BCED55C4D79FD5F24D6613C31C3839A2DDF8A9A276BCFBFA1C877C562C77CC8FBA599C5FBDA90A7EC659F50FB5FEA292209
$q =$ 0x5BF0A8B1457695355FB8AC404E7A79E3B1738B079C5A6D2B53C26C8228C867F799273B9C49367DF2FA5FC6C6C618EBB1ED0364055D88C2F5A7BE3DABABFACAC24867EA3EBE0CDDA10AC6CAAA7BDA35E76AAE26BCFEAF926B309E18E1C1CD16EFC54D13B5E7DFD0E43BE2B1426D5BCE6A6159949E9074F2F578156305664A12C9	$\hat{q} =$ 0xB7E151628AED2A6ABF7158809CF4F3C762E7161D
$k = 2$	$\hat{k} =$ 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFECD143303438D0AAD939DEE60194B8DB990AC80D6ACFBA0AA3C285C4ADD467AA7303859CF5F2B38A8C54CC9F95E67E76F5C2313A29D7AC442E7EE08B437562EFC324E7CA505E33CB314E04A54135A4B65F031105BE082EEBA8
$g = 4$	$\hat{g} =$ 0x4ECC560DFEB7F7C6EF0F6B74F3AE8A01DC08FF2A41F1CADB6BFEB2396942EB5E46D5A33EAEFD1AE25AE0C812A82815A04431D991F56FFFD108928AC16DB496AEED72BCCB83A7259A97093FE90991E789F384A478B11FDE984687156832B79C0313BF3660C28043920B0FEBBA1CFC55331F3DA1EFA25A732D0A510CFDA84E00EE
$h = 9$	

Table 10.5.: 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.

$p_0 = 3$	$p_{10} = 59$	$p_{20} = 151$	$p_{30} = 263$	$p_{40} = 353$	$p_{50} = 457$
$p_1 = 5$	$p_{11} = 79$	$p_{21} = 157$	$p_{31} = 269$	$p_{41} = 367$	$p_{51} = 463$
$p_2 = 7$	$p_{12} = 83$	$p_{22} = 179$	$p_{32} = 271$	$p_{42} = 373$	$p_{52} = 467$
$p_3 = 11$	$p_{13} = 89$	$p_{23} = 181$	$p_{33} = 277$	$p_{43} = 379$	$p_{53} = 479$
$p_4 = 13$	$p_{14} = 101$	$p_{24} = 199$	$p_{34} = 281$	$p_{44} = 383$	$p_{54} = 509$
$p_5 = 23$	$p_{15} = 103$	$p_{25} = 227$	$p_{35} = 283$	$p_{45} = 409$	$p_{55} = 523$
$p_6 = 29$	$p_{16} = 109$	$p_{26} = 229$	$p_{36} = 293$	$p_{46} = 419$	$p_{56} = 547$
$p_7 = 41$	$p_{17} = 131$	$p_{27} = 239$	$p_{37} = 317$	$p_{47} = 431$	$p_{57} = 557$
$p_8 = 43$	$p_{18} = 137$	$p_{28} = 241$	$p_{38} = 337$	$p_{48} = 443$	$p_{58} = 563$
$p_9 = 47$	$p_{19} = 149$	$p_{29} = 251$	$p_{39} = 347$	$p_{49} = 449$	$p_{59} = 569$

Table 10.6.: The first 60 prime numbers in $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for p and q as defined in Table 10.5.

10.2.3. 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 10.7.: Group $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for security level $\lambda = 2$ with default generators g and h.

$p_0 = 3$	$p_{10} = 53$	$p_{20} = 137$	$p_{30} = 233$	$p_{40} = 331$	$p_{50} = 433$
$p_1 = 7$	$p_{11} = 61$	$p_{21} = 139$	$p_{31} = 257$	$p_{41} = 347$	$p_{51} = 449$
$p_2 = 11$	$p_{12} = 71$	$p_{22} = 149$	$p_{32} = 263$	$p_{42} = 349$	$p_{52} = 461$
$p_3 = 17$	$p_{13} = 83$	$p_{23} = 157$	$p_{33} = 271$	$p_{43} = 353$	$p_{53} = 479$
$p_4 = 19$	$p_{14} = 97$	$p_{24} = 167$	$p_{34} = 277$	$p_{44} = 373$	$p_{54} = 487$
$p_5 = 23$	$p_{15} = 101$	$p_{25} = 179$	$p_{35} = 281$	$p_{45} = 389$	$p_{55} = 547$
$p_6 = 29$	$p_{16} = 103$	$p_{26} = 181$	$p_{36} = 283$	$p_{46} = 401$	$p_{56} = 557$
$p_7 = 37$	$p_{17} = 109$	$p_{27} = 193$	$p_{37} = 311$	$p_{47} = 419$	$p_{57} = 569$
$p_8 = 41$	$p_{18} = 127$	$p_{28} = 199$	$p_{38} = 313$	$p_{48} = 421$	$p_{58} = 571$
$p_9 = 47$	$p_{19} = 131$	$p_{29} = 229$	$p_{39} = 317$	$p_{49} = 431$	$p_{59} = 599$

Table 10.8.: The first 60 prime numbers in $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for p and q as defined in Table 10.7.

$\hat{p} =$	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 10.9.: Group $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for security level $\lambda = 2$ with default generator $\hat{g}$.

10.2.4. 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 10.10.: Group $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for security level $\lambda = 3$ with default generators g and h.

$p_0 = 2$	$p_{10} = 89$	$p_{20} = 167$	$p_{30} = 313$	$p_{40} = 457$	$p_{50} = 577$
$p_1 = 3$	$p_{11} = 101$	$p_{21} = 173$	$p_{31} = 317$	$p_{41} = 461$	$p_{51} = 593$
$p_2 = 7$	$p_{12} = 103$	$p_{22} = 181$	$p_{32} = 331$	$p_{42} = 467$	$p_{52} = 599$
$p_3 = 11$	$p_{13} = 109$	$p_{23} = 199$	$p_{33} = 367$	$p_{43} = 479$	$p_{53} = 607$
$p_4 = 13$	$p_{14} = 113$	$p_{24} = 211$	$p_{34} = 379$	$p_{44} = 491$	$p_{54} = 619$
$p_5 = 31$	$p_{15} = 127$	$p_{25} = 229$	$p_{35} = 383$	$p_{45} = 499$	$p_{55} = 643$
$p_6 = 61$	$p_{16} = 131$	$p_{26} = 233$	$p_{36} = 397$	$p_{46} = 503$	$p_{56} = 647$
$p_7 = 73$	$p_{17} = 139$	$p_{27} = 239$	$p_{37} = 401$	$p_{47} = 547$	$p_{57} = 659$
$p_8 = 79$	$p_{18} = 151$	$p_{28} = 251$	$p_{38} = 409$	$p_{48} = 557$	$p_{58} = 677$
$p_9 = 83$	$p_{19} = 157$	$p_{29} = 283$	$p_{39} = 449$	$p_{49} = 563$	$p_{59} = 691$

Table 10.11.: The first 60 prime numbers in $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for p and q as defined in Table 10.10.

$\hat{p}$	$=$	0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190CFEF324E
		7738926CFBE5F4BF8D8D8C31D763DA06C80ABB1185EB4F7C7B5757F5958490CFD47D7C
		19BB42158D9554F7B46BCED55C4D79FD5F24D6613C31C3839A2DDF8A9A276BCFBFA1C8
		77C56284DAB79CD4C2B3293D20E9E5EAF02AC60ACC93ED874422A52ECB238FEEE5AB6A
		DD835FD1A0753D0A8F78E537D2B95BB79D8DCAEC642C1E9F23B829B5C2780BF38737DF
		8BB300D01334A0D0BD8645CBFA73A6160FFE393C48CBBBCA060F0FF8EC6D31BEB5CCEE
		D7F2F0BB088017163BC60DF45A0ECB1BCD289B06CBBFEA21AD08E1847F3F7378D56CED
		94640D6EF0D3D37BE67008E186D1BF275B9B241DEB64749A47DFDFB96632C3EB061B64
		72BBF84C26144E49C2D04C324EF10DE513D3F5114B8B5D374D93CB8879C7D52FFD72BA
		0AAE7277DA7BA1B4AF1488D8E836AF14865E6C37AB6876FE690B571121382AF341AFE9
		4F790F02FA1BCE9C73886B4C0ACABDC3DD14E0D8C955577C9764844038771FC25F84BB
$\hat{q}$	$=$	0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190D05D
$\hat{k}$	$=$	0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF67215E
		C15D7BB8A7D7B5CB2294EFCAA4C7B3C6906FC93847CD5FEFF6F1F10C1400310C2150C4
		450843B67D7B0184C0A9B71708B657001B502DFAC3E8E29D3102610EB5B1D9AD470F0E
		FBC232F5025A3D88C58E70D9D2097C5E4E081BBFEE2373A9B5076970B38F6865D03E16
		293DBBBCA1B85E3FC5412F7262643B08A2A4CFA5EA43F5F8C9D9986B88155CEA5EC971
		5322344FF714C84F18D0B19772C421923C7E2CD2A6FE1000FBFCB4BBBBACEBAAF74C38
		CBC29EE75521F18B03C9816975D948F177476F6EBD8816152A0FECEA7DD6EF0AB7B6A0
		99617F82337346BDFC1CA47586EADF125A9DA7C1D960DDECDE399A37D7470FEFBED940
		3A4EC70A5841F41F60E3E0D40D70B1A5970EBEC446DF220714E83349462754D5C81F16
		FCC5ED708EBC21C36C0F3D494E04C15E3C275C18A562BADDA0293ADE9075FAA254E965
		E73402
$\hat{g}$	$=$	0x47DAD70733EFE399D1AFF4FE387250218BB88FD5F4040C31851AE1DF0985D0019950
		A958710C6B935B6B3BB45C278381DC5883CC933C5B7052D3BC8C77D746E3D1FB2B7EF3
		630C1014417D2F83BEAD0E1F4DFD986104CDF16C4AEC33BB5906C8149C83E6C5B8837E
		12AB32E73A69C4ABEB0B014FFF1FBB3173EAD73A1404DAEDF52F62D605D37879001248
		29751320FEDAA1F5B2D90FB846C7EB7815193E5C2460F93A3A5D16FB7A3DBAC9CE31B7
		517D2F88D530E61D06B529A43A0806F6A931247C9166C32CC9BAA019823528D3F156B6
		0ECE5DA9A6D60148661F59670AD98A1B8EAFEBC4A68D8A5D3F29105FD33D994751A9AD
		8E0EB7367D5BFE7A2F082981869FA2F177C472D1988844E4DA58170BB3DDE9DFB2E61D
		C06FA5249C3200CD3BBBF24D5C257879CB23D7931ED4AD1F9FA168B38FAA3C6DB89AA9
		D89BB6DB3F47BF1BE57856C12AD2FD708A932DC4C91A48E662B37C4076A5D2BE54AC80
		0EC1E6A13E1FC8EB61CA52E5D7B7608483E3BC225FBC62456AB46E39DA3CF45AB11A50

Table 10.12.: Group $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for security level $\lambda = 3$ with default generator $\hat{g}$.

11. Usability

For the codes printed on the voting cards and displayed to the voters on their voting device, suitable alphabets need to be fixed. Since the actual choice of an alphabet has a great impact on the system's usability, we will propose and discuss in Section 11.1 several possible alphabets that are commonly used for such purposes. Independently of the chosen alphabets, we will see that for reaching the desired security levels, very long voting and confirmation codes need to be entered by the voters. This creates a usability problem, for which we do not have an optimal solution at hand. Instead, we propose in Section 11.2 various possible workarounds, which each has its own strengths and weaknesses.

11.1. Alphabets and Code Lengths

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 11.1, 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 of entropy. Note that the Base64 alphabet A_{64} requires two non-alphanumerical characters to reach 6 bits of entropy.

Another special case is the last alphabet in Table 11.1, which contains $6^5 = 7776$ different English words from the new *Diceware wordlist* of the Electronic Frontier Foundation.[1,2] 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.

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

[2] See https://www.eff.org/deeplinks/2016/07/new-wordlists-random-passphrases.

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

Table 11.1.: Common alphabets with different sizes and characteristics. Case-insensitivity and fail-safety are desirable properties to facilitate flawless user entries.

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 11.1. 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).

11.1.1. 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 11.1 satisfying fail-safety. For the security levels $\lambda \in \{0, 1, 2, 3\}$ introduced in the beginning of this chapter, Table 11.2 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:

$$\hat{q}_x = \hat{q}_y = \begin{cases} 2^{31} =, & \text{for } \lambda = 0, \\ 2^{159}, & \text{for } \lambda = 1, \\ 2^{223}, & \text{for } \lambda = 2, \\ 2^{255}, & \text{for } \lambda = 3. \end{cases}$$

By looking at the numbers in Table 11.2, 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 seems to be one of the best choices, but it still not very practical. We will continue the discussion of this problem in Section 11.2.

Security Level λ	Security Strength τ	Required bit length	ℓ_X, ℓ_Y					
			A_{10}	A_{16}	A_{26}	A_{32}	A_{57}	A_{7776}
0	16	32	10	8	7	7	6	3
1	80	160	48	40	34	32	28	13
2	112	224	68	56	48	45	39	18
3	128	256	77	64	55	52	44	20

Table 11.2.: Lengths of voting and confirmation codes for different alphabets and security levels.

11.1.2. 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 11.3 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 $\lambda = 2$.

Security Level λ	Deterrence Factor ϵ	L_R	ℓ_R						L_{FA}	ℓ_{FA}					
			A_{10}	A_{16}	A_{26}	A_{36}	A_{62}	A_{64}		A_{10}	A_{16}	A_{26}	A_{36}	A_{62}	A_{64}
0	99%	3	8	6	6	5	5	4	1	3	2	2	2	2	2
1	99.9%	3	8	6	6	5	5	4	2	5	4	4	4	3	3
2	99.99%	3	8	6	6	5	5	4	2	5	4	4	4	3	3
3	99.999%	4	10	8	7	7	6	6	3	8	6	6	5	5	4

Table 11.3.: 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.

In the light of the results of Table 11.3 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 3-digit decimal numbers for $\lambda \in \{0\}$, as 5-digit decimal numbers for $\lambda \in \{1, 2\}$, and as a 8-digit decimal numbers for $\lambda = 3$.

11.2. Proposals for Improved Usability

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 in order to authenticate during voter casting, voters need to enter at least $2\tau = 224$ bits of entropy twice, once for the voting code x and once for the confirmation code y. According to our calculations in the previous section, this corresponds to 39 characters from a 57-character alphabet or equivalently to 18 words from the Diceware word list. Clearly, asking voters to enter such long strings creates a huge usability problem.

Two of the most obvious approaches or improving the usability of the authentication mechanism are the following:

- 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 (see second step in Prot. 7.2). In offline attacks, the workload can be distributed to a large amount of CPUs, which execute the attack in parallel. While breaking the DL problem is still very expensive for 160-bit logarithms (and 1024-bit moduli), especially if multiple discrete logarithms need to be found simultaneously, we do not recommend less than 80 bits security. Note that this number is expected to increase in the future.
- 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 with some special-purpose software installed is required to perform the scanning process. Latest developments in web technologies even allow to the use of built-in cameras directly from the web browser, but this will only work for machines with a built-in camera and an up-to-date web browser installed. We recommend considering this approach as an optional feature, but not yet as a general solution for everyone.

To conclude, the usability of the protocol's authentication mechanism remains a critical open problem. For finding a more suitable solution, we see two general strategies. First, by making offline attacks dependent on values different from the private credentials, and second, by preventing offline attacks targeting directly the underlying DL problem. In both cases, the goal is to make brute-forcing 112-bit credentials the best strategy for an attacker (in security level $\lambda = 2$). The necessary bit lengths of the credentials would then be shortened to one half of the current bit lengths, i.e., 20 characters from a 57-character alphabet or equivalently to 9 words from the Diceware word list. This seems to be within the bounds of what is reasonable for the majority of voters. Table 11.4 gives an update of the values from Table 11.2 for different security levels and alphabets.

Security Level λ	Security Strength τ	Required bit length	ℓ_X, ℓ_Y					
			A_{10}	A_{16}	A_{26}	A_{32}	A_{57}	A_{7776}
0	16	16	5	4	4	4	3	2
1	80	80	25	20	18	16	14	7
2	112	112	34	28	24	23	20	9
3	128	128	39	32	28	26	22	10

Table 11.4.: Lengths of voting and confirmation codes for different alphabets and security levels by reducing the required bit length from 2τ to τ bits.

In the following two subsection, we describe three different ways of achieving such a usability improvement. In all proposals, we only discuss the case of the voting credential x and assume that the confirmation credential y is treated equally. The common disadvantage of all three approaches is the requirement of an additional communication channel from the printing authority to the election authorities. We summarize the advantages and disadvantages of all all three proposals in Section 11.2.2.

11.2.1. Three Proposals for Improved Usability

The appendix of the Federal Chancellery Ordinance on Electronic Voting (VEleS) explicitly allows an additional communication channel from the printing authority back to the "system" [7, Section 4.1]. In the protocol presented in this paper, using this channel has been avoided for multiple reasons, but most importantly for restricting the printing authority's responsibility to their main task of printing the voting cards and sending them to the voters. Using this additional channel means to enlarge the trust assumptions towards the printing authority. Recall that in our adversary model, the printing authority is the only fully trustworthy party, i.e., implementing the printing authority is already a very delicate and difficult problem. Therefore, further increasing the printing authority's responsibility should always be done as moderately as possible. Unfortunately, we have not yet found a single best solution.

Below, we propose three different protocol modifications, which all assume that the printing authority can send data to the election authorities prior to an election. In each of the proposed protocol modifications, we manage to reduce the length of the private voting and confirmation codes from 2τ bits to τ bits. As discussed in Section 7.4 for the general case, we require the data sent over this channel to be digitally signed by the printing authority, and therefore that a certificate for the printing authority's public signature key is available to everyone. Figure 11.1 shows the extended communication model.

Approach 1: The first approach is based on the observation that a symmetric encryption key of length τ is sufficient for achieving a security strength of τ bits, for example when using AES. Therefore, instead of printing a 2τ-bit voting credential $x_i \in \mathbb{Z}_{\hat{q}_x}$ to the voting card of voter i (see Table 6.1 and Section 11.1.1 for more details on system parameters and their bit lengths), the printing authority selects a secret symmetric encryption key $k_i \in \mathbb{B}^\tau$ of length τ, prints k_i to the voting card, encrypts x_i using k_i into $[x_i] \leftarrow \mathsf{Enc}_{k_i}(x_i)$, and

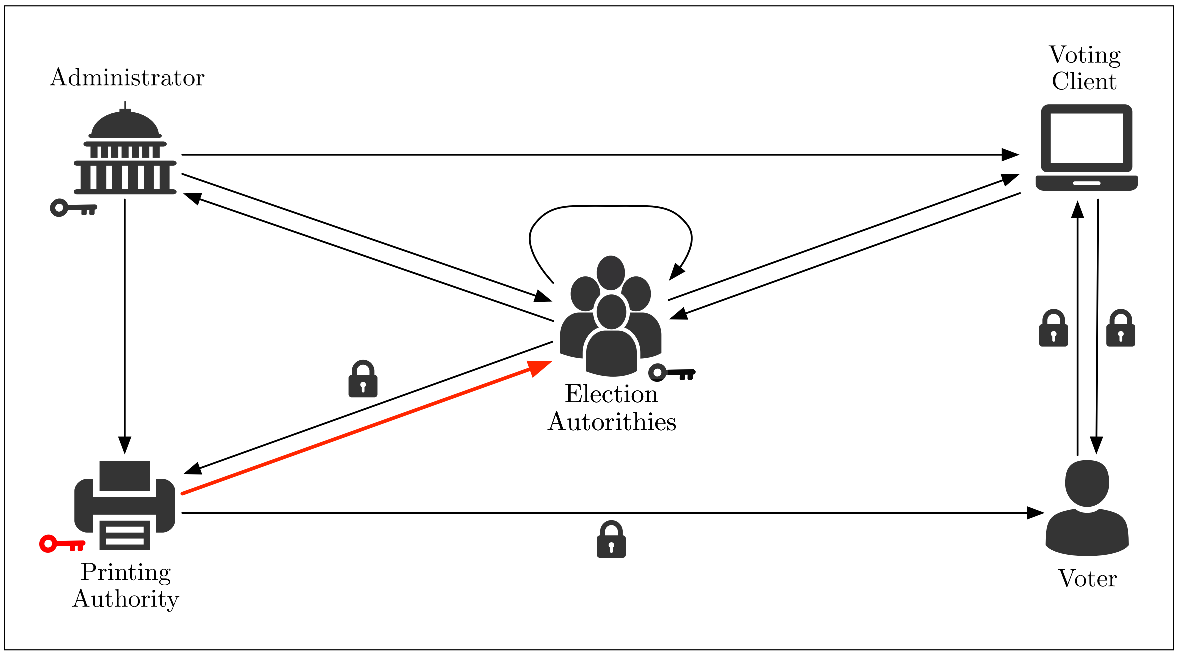


Figure 11.1.: Overview of the parties and channels in the extended communication model. Compared to Figure 6.1, it contains an additional authentic channel from the printing authority to the election authorities. This requires the printing authority to possess a signature key pair.

sends the encryption $[x_i]$ to the election authorities. At the end of the pre-election phase, a list $([x_1], \dots, [x_{N_E}])$ of such encrypted private voting credentials is available to each election authority, one for each eligible voter. During the voting process, voter i enters k_i as printed on the voting card to the voting client, which then retrieves $[x_i]$ from the election authorities and decrypts $[x_i]$ into $x_i \leftarrow \mathsf{Dec}_{k_i}([x_i])$. Finally, x_i is used to authenticate voter i as an eligible voter as before.

The most problematic point in this approach is to let the printing authority generate critical keying material. For this, a reliable randomness source is necessary. Otherwise, an attacker might be capable of reproducing the same keying material and thus fully break the integrity of the system without being noticed. Attributing the randomness generation task to the printing authority is therefore in conflict with the above-mentioned general principle of increasing its responsibility as moderately as possible.

Approach 2: This approach is an adaption of the previous one. The main change to the protocol is the same, i.e., the private credential x_i is transported in encrypted form to the voting client, whereas the secret decryption key is transported to the voter via the voting card. However, instead of letting the printing authority pick the secret encryption key k_i at random, we propose to derive it from the private credential x_i by applying a *key derivation function* (KDF). The idea for this is the observation that a high-entropy 2τ-bit voting credential contains enough entropy for extracting a τ-bit secret encryption key.

We propose to use the HMAC-based standard HKDF, which is designed according to the *extract-then-expand* approach. It offers the option of adding a random salt s_i and some contextual information c to each generated key [40, 41]. Therefore, we let the printing authority compute a secret key $k_i \leftarrow \mathsf{HKDF}_\tau(x_i, s_i, c)$ of length τ bits, which is used to encrypt x_i. The random salt is published along with $[x_i]$, and c is a string which depends on the unique election identifier U. The voting client, upon retrieving $[x_i]$ and s_i and decrypting $[x_i]$ using k_i, can additionally check the validity of the secret key $k_i = \mathsf{HKDF}_\tau(x_i, s_i, c)$. This check is very useful for detecting a cheating printing authority. Note that since this check requires knowledge of k_i, it can only be performed by the voting client. If the check fails, the voting procedure must be aborted.

Approach 3: In this approach, we reverse the role of the key derivation function in the previous approach. Here, the KDF is used to derive a 2τ-bit value $x'_i = \mathsf{HKDF}_{2\tau}(x_i, s_i, c)$ from a τ-bit private voting credential $x_i \in \mathbb{Z}_{\hat{q}_x}$. This means that the constraint $\|\hat{q}_x\| \geqslant 2\tau$ from Table 6.1 is relaxed into $\|\hat{q}_x\| \geqslant \tau$. Like in the general protocol, the private credentials x_i are generated by the election authorities in a distributed manner. Corresponding shares x_{ij} are transmitted to the printing authority, which then applies the KDF to the aggregated value. The main difference here is that the public voting credential $\hat{x}_i = \hat{g}^{x'_i} \bmod \hat{p}$ is now computed by the printing authority based on x'_i. The printing authority is also responsible for sending this value to the election authorities, along with the random salt s_i. As in the general protocol, the voter enters x_i into the voting client, which then retrieves s_i from the election authorities to compute $x'_i = \mathsf{HKDF}_{2\tau}(x_i, s_i, c)$. Finally, the Schnorr identification is performed relative to $\hat{x}_i$ (using x'_i instead of x_i).

The problem with this approach so far is that the printing authority may use voting credentials different from the values x_i obtained from the election authorities. Again, this is not a problem as long as the printing authority is fully trustworthy (which is the case in our adversary model), but the potential of such undetectable protocol deviations assigns unnecessarily large responsibilities to the printing authority. This problem can be mitigated by letting the election authorities publish their shares x_{ij} of the value x_i in response to a successful identification. The correctness of $\hat{x}_i$ and therefore the proper behavior of the printing authority can then be publicly verified for each submitted vote. The effect that x_i does no longer remain secret after submitting a vote is in contrast to the general protocol and the two approaches presented above.

11.2.2. Comparison of Methods

In the previous subsection, we presented three different methods to mitigate the aforementioned usability problem. In each case, the number of entropy bits for a voter to enter is reduced from 2τ to τ bits. We now want to give an overview of the differences, strengths, and weaknesses of each approach. At the end of this section, our analysis allows us to give some general recommendations.

A first overview of the proposed methods is given in Table 11.5, which summarizes the necessary calculations in each approach and compares them to the current protocol. The overview is restricted to calculations relative to the private and public voting credentials, but exactly the same calculations are necessary to deal with corresponding confirmation credentials. The table shows for example the similarity between Approach 1 and Approach 2. Note that selecting and aggregating the shares x_{ij} of the private voting credential $x_i \in \mathbb{Z}_{\hat{q}_x}$ looks identical in all three approaches, but the selected values are not equally long. In Approaches 1 and 2, they consist of at least 2τ bits (the same as in the current protocol), whereas in Approaches 3, they consist of τ bits. This difference results from relaxed restrictions relative to the upper bound $\hat{q}_x$.

Current Protocol	Approach 1	Approach 2	Approach 3
$x_{ij} \in_R \mathbb{Z}_{\hat{q}_x}$			
$x_i \leftarrow \sum_j x_{ij}$			
–	$k_i \in_R \mathbb{B}^\tau$	$s_i \in_R \mathbb{B}^\tau$	
–		$k_i \leftarrow \mathsf{HKDF}_\tau(x_i, s_i, c)$	$x'_i \leftarrow \mathsf{HKDF}_{2\tau}(x_i, s_i, c)$
–	$[x_i] \leftarrow \mathsf{Enc}_{k_i}(x_i)$, $x_i \leftarrow \mathsf{Dec}_{k_i}([x_i])$		–
$\hat{x}_{ij} \leftarrow \hat{g}^{x_{ij}}$			$\hat{x}_i \leftarrow \hat{g}^{x'_i}$
$\hat{x}_i \leftarrow \prod_j \hat{x}_{ij}$			

Table 11.5.: Necessary computations in each of the three proposed methods compared to the current protocol. Only the case of the voting credentials is shown. Similar computations are necessary for the confirmation credentials.

An even more detailed overview of corresponding protocol processes is given in Table 11.6. It exposes the augmented responsibility assigned to the printing authority in all three approaches. This task involves generating random values and sending some data to the election authorities. This is the main disadvantage in comparison to the current protocol. Note that generating a random salt s_i (which is only used for making pre-computations of brute-force attacks more expensive) is much less delicate than generating a random encryption key k_i. Compared to the other approaches, this is the main disadvantage of Approach 1, which does not allow to detect an attack against the random key generation process. Since a printing authority with augmented responsibility is more likely to attract attacks of all kinds, it is important that a corrupt printing authority could at least be detected. In Approaches 2 and 3, corresponding tests can be implemented into the voting client or as part of the universal verification process (see explanations given in the previous subsection).

Compared to the current protocol, a subtle weakness of all three approaches is the fact that entering the voter index i becomes mandatory. In the current protocol, entering i appears in Prot. 7.4 for obtaining the correct voting page, but the role of i as a unique identifier could in principle be taken over by the public voting credential $\hat{x}_i$, which can be derived from x_i alone. Unfortunately, this is no longer the case in any of the three proposals, since knowing i is necessary for retrieving the right values $[x_i]$ and s_i from the election authorities, which are needed to derive $\hat{x}_i$.[3]

[3]In Approach 3, it is possible to derive $\hat{x}_i$ from x_i without knowing i, but only if the random salt of the key derivation function is entirely omitted. As a general rule, we do not recommend using a KDF without a random salt, even if high-entropy input keying material and case-specific contextual information is available.

Protocol Phase	Party	Task	Current protocol	Approach 1	Approach 2	Approach 3
7.2	EA$_j$	Select at random, send to PA	x_{ij}	x_{ij}	x_{ij}	x_{ij}
		Compute and send to BB	$\hat{x}_{ij}$	$\hat{x}_{ij}$	$\hat{x}_{ij}$	–
		Retrieve from BB	$(\hat{x}_{ij})$	$(\hat{x}_{ij})$	$(\hat{x}_{ij})$	–
		Compute	$\hat{x}_i$	$\hat{x}_i$	$\hat{x}_i$	–
7.3	PA	Select at random	–	k_i	s_i	s_i
		Compute	x_i	x_i, $[x_i]$	x_i, k_i, $[x_i]$	x_i, x'_i, $\hat{x}_i$
		Send to BB	–	$[x_i]$	s_i, $[x_i]$	s_i, $\hat{x}_i$
		Send to V$_i$	i, x_i	i, k_i	i, k_i	i, x_i
7.4	V$_i$	Enter into VC$_i$	i, x_i	i, k_i	i, k_i	i, x_i
7.5	VC$_i$	Retrieve from BB	–	$[x_i]$	s_i, $[x_i]$	s_i
		Check integrity of	–	–	k_i, x_i, s_i	–
		Compute	$\hat{x}'_i$, π	x_i, π	x_i, π	x'_i, π
		Send to BB	i, $\hat{x}'_i$, π	i, π	i, π	i, π
	EA$_j$	Retrieve from BB	i, $\hat{x}'_i$, π	i, π	i, π	i, π
		Check integrity of	$\hat{x}_i$, $\hat{x}'_i$	–	–	–
		Check validity of	π	π	π	π
7.6	EA$_j$	Send to BB	–	–	–	x_{ij}

Table 11.6.: Tasks to be executed by the election authorities (EA$_j$), the printing authority (PA), the voters (V$_i$), and the voting clients (VC$_i$) in the different phases of the protocol.

A recapitulation of the above discussion and comparison is given in Table 11.7. It lists the major strengths and weaknesses for each of the discussed approaches. It turns out that no single winner can be selected based on our analysis. However, since Approach 1 seems to be strictly less preferable than Approach 2 or Approach 3, we recommend excluding it from further consideration.

Therefore, we conclude this discussion by recommending either Approach 2 or 3 as a possible compromise solution for solving the usability problem addressed in this section. The augmented responsibility assigned to the printing authority is clearly not very appealing, but also not excluded by the given VEleS regulations.

On the other hand, we do not entirely exclude it as an option for achieving an optimal compromise between security and usability.

Approach	Strengths	Weaknesses
1	– Tasks executed by election authorities remain unchanged	– New channel from printing authority to election authorities – Random keys generated by printing authority – Validity of secret keys can not be checked
2	– Tasks executed by election authorities remain unchanged – Validity of secret key can be checked by voting client	– New channel from printing authority to election authorities – Random salt generated by printing authority
3	– Tasks executed by election authorities are simplified – Validity of private credentials can be publicly verified	– New channel from printing authority to election authorities – Random salt generated by printing authority – Private credentials revealed after vote casting

Table 11.7.: Recapitulation of major weaknesses and strength.

11.3. Reducing the Number of Codes to Verify

In elections with a large number of election options k, checking the correctness of all k verification codes RC'_j according to Alg. 8.31 may result in a cumbersome and time-consuming procedure for human voters. In this section, we propose a method for reducing the number of necessary checks in some specific situations. The general idea is to merge some of the codes without reducing the overall level of individual verifiability. Clearly, merging the codes need to be done twice, when the verification codes are printed and when they are displayed to the voter. The two parties involved are therefore the printing authority and the voting client. For this method to work, they both need to perform the merge operation in exactly the same way. The merging algorithm presented below is therefore the same for both involved parties.

The printed verification codes RC_j and the displayed codes RC'_j are strings of length ℓ_R with characters taken from the alphabet A_R. To define a general string merging procedure, consider a vector $\mathbf{s} = (S_1, \ldots, S_k)$ of strings $S_i = \langle c_{i,0}, \ldots c_{i,n-1} \rangle \in A^n$ of length n from an alphabet A of size $N = |A|$. We propose to merge these strings character-wise, i.e., to compute the j-th character c_j of the merged string $S = \langle c_0, \ldots c_{n-1} \rangle$ from the j-th characters $c_{1,j}, \ldots, c_{n,j}$ of the input strings. Individual characters can be merged by considering the given alphabet as an additive group of order N with the following commutative operation:

$$c_1 \oplus c_2 = rank_A^{-1}(rank_A(c_1) + rank_A(c_2) \bmod N), \text{ for all } c_1, c_2 \in A.$$

The procedure defined in Alg. 11.1 is derived from this simple idea. Note that the algorithm returns the same result for two inputs $\mathbf{s}$ and $\mathbf{s}'$, if they contain the same strings in different order. For an alphabet of size $N = 2$, for example for $A = \{0, 1\}$, the algorithm corresponds to applying the XOR operation bit-wise to the input bit strings.

```
Algorithm: GetMergedString($\mathbf{s}, A$)
Input: Strings $\mathbf{s} = (S_1, \ldots, S_k)$, $S_i = \langle c_{i,0}, \ldots c_{i,n-1} \rangle$, $c_{ij} \in A$
       Alphabet $A$, $N = |A| \geqslant 2$
for $j = 0, \ldots, n-1$ do
    $x \leftarrow \sum_{i=1}^{k} rank_A(c_{ij}) \bmod N$
    $c_j \leftarrow rank_A^{-1}(x)$
$S \leftarrow \langle c_0, \ldots, c_{n-1} \rangle$
return $S$                                                    // $S \in A^n$
```

Algorithm 11.1: Merges k input strings of length n into an output string of length k.

We see at least three ways of using Alg. 11.1 to reduce the number of verification codes to check after submitting a vote. Each of them requires some supplementary information about the candidates. We can either assume that this information can be inferred from the candidate descriptions $\mathbf{c} = (C_1, \ldots, C_n)$, or we require the election administrator to provide this information in form of additional election parameters. In all three cases, some subsets $I \subseteq \{1, \ldots, n\}$ of candidates are involved. If $\mathbf{rc} = (RC_1, \ldots, RC_n)$ are the verification codes of a given voter for all n candidates, then $\mathbf{rc}_I$ denotes the sub-vector of verification codes of the candidates in I and $RC_I \leftarrow \mathsf{MergeStrings}(\mathbf{rc}_I, A_R)$ is the result of merging these codes. Note that relative to RC_I, the order of the codes in $\mathbf{rc}_I$ is not important. In case the voter has selected all $r = |I|$ candidates from I, then the general idea is to replace checking all r codes from $\mathbf{rc}_I$ by checking RC_I only.

The following list gives an overview of the application cases, in which this techniques may help in improving the overall usability of the vote confirmation process. They are derived from the given context of elections in Switzerland. Other application cases may exist in another context.

- In a party-list election for the Swiss National Council, which can be modeled as two independent elections in parallel, one 1-out-of-n_p party election and one cumulative k-out-of-n_c candidate election (see Section 2.2.3), predefined lists of up to k candidates are proposed to the voters by each participating party. These lists belong officially to the election definition. Arbitrary modifications to these list are allowed, but a large number of voters just vote for the proposed candidates of their favorite party, i.e., they adopt the corresponding party list without modifications. Therefore, this is clearly a situation, in which checking a single verification code for the whole predefined list would be much more efficient than checking the $k + 1$ codes of the party itself and of all candidates from the list. Note that such a party list may contain both cumulated or blank candidates (some list are actually smaller than k), but this does not prevent the application of this technique.
- If cumulation with up to c votes for the same candidate is allowed in a k-out-of-n election, we can run it as a non-cumulative k-out-of-n' election with a total of $n' = cn$ candidates (see Section 2.2.3). In this case, each real candidate $j \in \{1, \ldots, n\}$ will be modeled to a set $I_j \subset \{1, \ldots, n'\}$ of virtual candidates of size $|I_j| = c$. To submit $c' \leqslant c$ votes for the same candidate j, the corresponding amount of virtual candidates will then be selected from I_j. For $k = 3$, $n = 5$, and $c = 3$, the $n' = 18$ virtual candidates

can be grouped into sets I_j of size $c = 3$, for example as follows:

$$\{\underbrace{1,2,3}_{I_1}, \underbrace{4,5,6}_{I_2}, \underbrace{7,8,9}_{I_3}, \underbrace{10,11,12}_{I_4}, \underbrace{13,14,15}_{I_5}, \underbrace{16,17,18}_{I_6}\}.$$

First, consider a cumulation of $c' = 3$ votes for the second candidate, which corresponds to selecting $\mathbf{s} = (4, 5, 6)$ or any other permutation of the values in $I_2 = \{4, 5, 6\}$. This case is clearly another good candidate for applying the above technique of merging verification codes. Instead of checking three different verification codes for the selected virtual candidates 4, 5, and 6, it is sufficient to check a single combined verification code $RC_{\{4,5,6\}}$ for the voting option "*Three Votes for Candidate 2*".

To submit a cumulation of $c' < 3$ votes for the second candidate, the situation is a bit more subtle. The problem is that $(4, 5)$, $(4, 6)$, and $(5, 6)$ are all equivalent selections for a cumulation of two votes, and 4, 5, and 6 are all equivalent selections for the limiting case of a single vote. To circumvent this problem, merging the verification codes can be based on some convention. As a general rule, we propose to always select the c' smallest candidate indices from I_j when submitting a cumulation of $1 \leqslant c' \leqslant c$ votes for candidate j. Let $I_j(c') \subseteq I_j$ denote the corresponding set of the c' smallest indices from I_j. If an honest voting client sticks to that rule, matching cumulated verification codes $RC'_{I_j(c')}$ can be generated in a unique way. In the example above, we get the following cumulated verification codes for the second candidate:

$$\begin{aligned} R'_{\{4\}} &= \text{“\textit{One Vote for Candidate 2}”}, \\ R'_{\{4,5\}} &= \text{“\textit{Two Votes for Candidate 2}”}, \\ R'_{\{4,5,6\}} &= \text{“\textit{Three Votes for Candidate 2}”}. \end{aligned}$$

Clearly, a compromised voting client could easily deviate from this rule, for example by submitting $(4, 6)$ instead of $(4, 5)$, but this would be detected by the voter when comparing the resulting mismatched codes.

- Verification codes for multiple blank votes can be handled in a similar way. Recall from Section 2.2.3 that k-out-of-n elections allowing blank votes can be modeled as k-out-of-$(n + b)$ elections, where $b \leqslant k$ denotes the number of allowed blank votes (usually $b = k$), for which special blank candidates are added to the candidate list. Let $I_b \subset \{1, \ldots, n + b\}$ denote the set of their indices and $I_b(b') \subseteq I_b$ the subset of the b' smallest indices. For $I_b = \{n + 1, \ldots, n + b\}$, this convention leads to the following combined verification codes:

$$\begin{aligned} R'_{\{n+1\}} &= \text{“\textit{One Blank Vote}”}, \\ R'_{\{n+1,n+2\}} &= \text{“\textit{Two Blank Votes}”}, \\ &\vdots \\ R'_{\{n+1,\ldots,n+b\}} &= \text{“\textit{b Blank Votes}”}. \end{aligned}$$

If vote abstentions are treated in the same way as blank votes, i.e., by adding multiple abstention candidates to the candidate list, then this technique can be applied in exactly the same way.

11.4. Recovering From Lost Sessions

A usability problem may occur if the vote casting session gets interrupted and the randomizations $\mathbf{r} = (r_1, \ldots, r_k)$ generated by $\mathsf{GenBallot}$ are lost. In that case, the authorities who may have responded properly to the first submitted ballot will not respond to a second ballot submitted in a new vote casting session. The best the election authorities could do in that case is to resend the same response from the first session, but without knowing $\mathbf{r}$, the voting client is unable to proceed with the protocol. This creates a situation where an honest voter gets blocked by the protocol for no compelling reason. In practice, such situations may be caused by a lost network connection, a frozen web browser, or other problems with the machine running the voting client.

As a workaround, we propose sending a symmetric encryption $\mathsf{Enc}'_K(\mathbf{r})$ to a public server immediately after generating $\mathbf{r}$. To recover from a lost session, this encryption can be downloaded and decrypted. Together with the selections $\mathbf{s}$ from the first session (which the voter is supposed to memorize), exactly the same ballot can be resent to the authorities, in which case they accept to resend their first response. The protocol can then be continued in the prescribed way, which ultimately leads to a finalized ballot. For the secret encryption key, we propose to apply a key derivation function (KDF) to the voting code X_v, which contains sufficient entropy. This means that X_v needs to be entered at the beginning of the recovered session to obtain $K = \text{KDF}(X_v)$. From a voter's perspective, entering X_v at the beginning of *every* session is what seems most natural anyway. The usability problem of a lost session is therefore appropriately solved by this approach.

12. Performance Optimizations

The protocol as presented in this document defines a couple of performance bottlenecks, which need to be addressed in a real implementation. Some of these bottlenecks can be eliminated rather easily, while others required more sophisticated optimization techniques. The goal of this chapter is to discuss some of the most effective performance optimizations, which we recommend to consider in an implementation of the protocol. We keep this discussion sufficiently abstract for not restricting it to a particular use case or programming language.

12.1. Efficient Group Membership Tests

To avoid different sorts of attacks, group membership tests are required in many cryptographic primitives. Performing such tests in a systematic way is therefore a best practice in the design and implementation of cryptographic applications. In CHVote, we recommend checking the validity of the input parameters of at least all top-level algorithms (see remarks at the beginning of Chapter 8). For simple groups such as $\mathbb{Z}_q$, group membership tests $x \in \mathbb{Z}_q$ are very efficient by checking $0 \leqslant x < q$, i.e., performing such tests systematically has no noticeable performance impact, even for large inputs. In arbitrary prime-order groups $\mathcal{G}$, group membership $x \in \mathcal{G}$ can be tested by $x^q = 1$ using one exponentiation with an exponent of length $\ell = \|q\|$. For the subgroup $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of integers modulo a safe prime $p = 2q + 1$, which is used in CHVote for the ElGamal encryption scheme, group membership $z_i \in \mathbb{G}_q$ can be tested more efficiently using the Jacobi symbol $(\frac{x}{p}) \in \{-1, 0, 1\}$. For $\ell = 2048$, common $O(\ell^2)$ time algorithms for computing the Jacobi symbol run approximately twenty times faster than modular exponentiation [48]. For very large inputs, even efficient Jacobi symbol algorithms may then become a target for performance optimizations.

Since elements of $\mathbb{G}_q$ are quadratic residues modulo p, we can reduce the cost of the group membership test to a single modular multiplication in $\mathbb{Z}_p^*$. The improvement is based on the observation that every quadratic residue $x \in \mathbb{G}_q$ has two square roots in $\mathbb{Z}_p^*$, whereas quadratic non-residues $x \notin \mathbb{G}_q$ have no square roots in $\mathbb{Z}_p^*$. Group membership $x \in \mathbb{G}_q$ can therefore be demonstrated by presenting one of the two square roots $\sqrt{x} = \pm\, x^{\frac{q+1}{2}} \bmod p$ as a *membership witness* and by checking that $\sqrt{x}^2 \equiv x \pmod{p}$ holds. Thus, provided that such a membership witness is available, group membership can be tested using a single modular multiplication.

To implement this idea in practice, elements $x \in \mathbb{G}_q$ can be represented as pairs $\hat{x} = (\sqrt{x}, x)$, for which group membership can be tested as described above using a single multiplication. To disallow the encoding of an additional bit of information into the square root representation of a quadratic residue, we suggest normalizing the representation by taking always either the smaller or the larger of the two values. For such pairs, multiplication $\hat{z} = \hat{x}\hat{y}$, exponentiation $\hat{z} = \hat{x}^e$, and computing the inverse $\hat{z} = \hat{x}^{-1}$ can be implemented based on corresponding computations on the square roots:

$$\sqrt{xy} \equiv \sqrt{x}\sqrt{y} \pmod{p}, \quad \sqrt{x^e} \equiv \sqrt{x}^e \pmod{p}, \quad \sqrt{x^{-1}} \equiv \sqrt{x}^{-1} \pmod{p}.$$

Thus, only a single additional multiplication $z = \sqrt{z}^2 \bmod p$ is needed in each case to obtain the group element itself. In such an implementation, it is even possible to compute groups elements only when needed, for example before decoding a decrypted ElGamal message. In this way, additional multiplications can be avoided almost entirely during the execution of a cryptographic protocol, during which all computations are conducted on the square roots. Restricting the representation to the square root is also useful for avoiding additional memory and communication costs. In other words, group membership in $\mathbb{G}_q$ can be guaranteed at almost no additional cost. We recommend this optimization for any implementation of the CHVote protocol for speeding up systematic validity checks of all input parameters.

12.2. Product Exponentiation

At several places, the algorithms for generating and verifying a shuffle proof require the computation of *product exponentiations* (also called *simultaneous multi-exponentiations*) of the form

$$z = \prod_{i=1}^{N} b_i^{e_i} \bmod p,$$

where $(b_1, \ldots, b_N) \in \mathbb{G}_q^N$ and $(e_1, \ldots, e_N) \in \mathbb{Z}_q^N$ denote the given vectors of input values (see Algs. 8.45 and 8.48). The input size N is the same in each case. Without optimizations, $3N$ respectively $6N$ modular exponentiations are necessary for computing corresponding expressions.[1] In the summary given in Table 12.1, we see that some of the exponents in Alg. 9.21 are determined by the security strength τ, which is a much smaller value than the bit length $\|q\|$ of the group size q (for example $\tau = 112$ vs. $\ell = 2047$ for $\lambda = 1$, see Table 10.1). This means that the last three product exponentiations in Alg. 9.21 are the dominant ones with respect to the overall performance of the algorithm.

Using Algorithm 2 from [34], product exponentiations can be computed much more efficiently compared to computing the exponentiations individually. In the remainder of this section, we refer to this algorithm as HLG2. It has a single algorithm parameter $m \geqslant 1$, which denotes the size of the sub-tasks into which the problem is decomposed. If $M_m(\ell, N)$ denotes the total number of modular multiplications needed to solve a problem instance of size N and maximal exponent length $\ell = \max_{i=1}^{N} \|e_i\|$, then

$$\widetilde{M}_m(\ell, N) = \frac{M_m(\ell, N)}{N} < \frac{2^m + \ell}{m} + \frac{\ell}{N}$$

denotes the *relative running time* of HLG2. In general, $\widetilde{M}_m(\ell, N)$ depends on both ℓ and N, but the impact of N vanishes for large N. Optimizing m is therefore largely independent of N. All parameters m shown in Table 12.2 are optimal for $N \geqslant 210$ (and nearly optimal

[1]In the extended shuffle proof algorithms of the write-ins protocol from Chapter 9, the number of modular exponentiations involved in product exponentiations is $(z + 4)N$ for Alg. 9.20 and $(z + 7)N$ for Alg. 9.21, respectively.

Algorithm	Computation	e_i	ℓ
8.45 GenShuffleProof	$t_3 \leftarrow g^{\omega_3} \cdot \prod_{i=1}^N h_i^{\tilde{\omega}_i}$ $t_{4,1} \leftarrow pk^{-\omega_4} \cdot \prod_{i=1}^N \tilde{a}_i^{\tilde{\omega}_i}$ $t_{4,2} \leftarrow g^{-\omega_4} \cdot \prod_{i=1}^N \tilde{b}_i^{\tilde{\omega}_i}$	$\tilde{\omega}_i$	$\|q\|$
8.48 CheckShuffleProof	$\tilde{c} \leftarrow \prod_{i=1}^N c_i^{u_i}$ $\tilde{a} \leftarrow \prod_{i=1}^N a_i^{u_i}$ $\tilde{b} \leftarrow \prod_{i=1}^N b_i^{u_i}$	u_i	τ
	$t'_3 \leftarrow \tilde{c}^c \cdot g^{s_3} \cdot \prod_{i=1}^N h_i^{\tilde{s}_i}$ $t'_{4,1} \leftarrow \tilde{a}^c \cdot pk^{-s_4} \cdot \prod_{i=1}^N \tilde{a}_i^{\tilde{s}_i}$ $t'_{4,2} \leftarrow \tilde{b}^c \cdot g^{-s_4} \cdot \prod_{i=1}^N \tilde{b}_i^{\tilde{s}_i}$	$\tilde{s}_i$	$\|q\|$

Table 12.1.: Overview of product exponentiations in the shuffle proof generation and verifications algorithms. With $\ell = \|q\|$ we denote the bit length of group size q, and a security parameter λ.

for smaller values). Applying these optimal parameters to HLG2 considerably decreases the computation time for product computations. Compared to the best general-purpose algorithms such as the *sliding-window method* [48, Alg.14.85], HLG2 performs between 5 times (for $\ell = 112$ and $m = 5$) to 9 times (for $\ell = 3072$ and $m = 9$) better. Therefore, we strongly recommend the application of HLG2 to both shuffle proof algorithms. It will considerably improve the election authority's performance during the post-election phase.

ℓ	80–147	174–349	380–802	845–1839	1896–4148	4231–9284	>9285
m	5	6	7	8	9	10	11

Table 12.2.: Optimal sub-task sizes m in HLG2 for different exponent lengths ℓ.

12.3. Fixed-Based Exponentiation

A second category of modular exponentiations with a large potential for performance improvements are so-called *fixed-based exponentiations* $z_i = b^{e_i}$ for a fixed base $b \in \mathbb{G}_q$ or $b \in \mathbb{G}_{\hat{q}}$. In the CHVote protocol, there are several fixed bases in $\mathbb{G}_q$ (the generators g and h, the generated public key pk, and the prime number representations $p_1, p_2, \ldots, p_n$ of the n candidates) and a single fixed base in $\mathbb{G}_{\hat{q}}$ (the generator $\hat{g}$). Each of them requires precomputing a table of common values, which can be used for computing the actual exponentiations. The optimal size of such a table depends on the number of exponentiations to compute for the particular base. Note that if the same parameters are used over multiple election events, this number can be very large. In such a case, corresponding precomputation tables can be generated by the election authorities during the system setup. It is also possible to include such tables into the source code of certain components or to provide them online by a trusted service.

The best known algorithms for fixed-base exponentiations are the *comb method* by Lim and Lee [44] and Algorithm 3.2 from [34]. They have identical running times in terms of number of modular multiplications. Both of them are parametrized by two values, for example $1 \leqslant k \leqslant \ell$ (window size) and $1 \leqslant m \leqslant \ell/k$ (sub-task size) in the case of the latter, to which we will refer as HLG3.2. If the total number N of exponentiations tends towards infinity, we get optimal parameters $k = 1$ and $m = \ell$, but otherwise the choice of k and m depends on both ℓ and N. For example, $k = 32$ and $m = 12$ are optimal for $\ell = 3071$ and $N = 1000$. In this particular case, the total computation time of HLG3.2 is already more than 11 times better compared to the best general-purpose algorithms, but the advantage improves even further for larger N. We refer to [34] for more detailed information about the algorithm's running time and the selection of optimal parameters.

In Tables 12.3 and 12.4, we summarize the fixed-based exponentiations of a complete protocol run for the voting clients and election authorities (the expected performance benefits is insignificant for all other parties). To simplify the presentation, we only count exponentiations that are repeated at least N_E (number of voters), N (number of submitted ballots), n (number of candidates), or k (number of selections) times. By selecting these parameters for realistic election events, single exponentiations are not significant for the overall performance. Since the number of election authorities will be a small constant in practice, for example $s = 4$, we also ignore some of fixed-based exponentiations that need to be executed s times or less.

Algorithm	Computation	b	ℓ	#modExps
8.23 GenQuery	$a_{j,1} = m_j \cdot pk^{r_j}$	pk	$\|\|q\|\|$	k
	$a_{j,2} = g^{r_j}$	g	$\|\|q\|\|$	k
8.29 GetPoints	$k_j = b_j \cdot d^{-r_j}$	$d^{(1)}, \ldots, d^{(s)}$	$\|\|q\|\|$	k each
8.38 GetAllPoints	$p'_i = p_i^{z_1}$	$p_1, \ldots, p_n$	$\|\|q\|\|$	s each
	$\beta_j = b_j \cdot d^{-r_j} \cdot (p'_{s_j})^{-1}$	$d^{(1)}, \ldots, d^{(s)}$	$\|\|q\|\|$	[from cache]

Table 12.3.: Fixed-base exponentiations of the voting client during a complete protocol run with s authorities, n candidates, and k selections. Note that the second fixed-base exponentiations in Alg. 8.38 are the same as in Alg. 8.29, i.e., they can easily be cached for reuse during the vote casting session.

The picture that we receive from this analysis provides rough estimates of the total number of fixed-based exponentiations during a complete protocol run. Note that the size of the involved exponents is an important distinguishing factor, since exponentiations in $\mathbb{G}_q$ with large exponents of size $\|q\|$ and $\|p\| = \|q\| + 1$ are considerably more expensive than exponentiations in $\mathbb{G}_{\hat{q}}$ with small exponents of size $\|\hat{q}\| = 2\tau$. However, for achieving the best possible performance, we recommend applying a fixed-base exponentiation algorithm such as HLG3.2 to all cases.

By summing up the numbers from Tables 12.3 and 12.4, Table 12.5 provides a guideline for selecting optimal election parameters for a single protocol run. Another approach is to create the precomputation tables during the system setup and to spend as much time as possible on this for speeding up the computations during the execution of the protocol to the maximal possible degree. This creates a trade-off between offline and online costs, which can only be solved optimally by considering all parameters of a given practical use case.

Algorithm	Computation	b	ℓ	#modExps
8.14 GenCredentials	$\hat{x} = \hat{g}^x$	$\hat{g}$	2τ	N_E
	$\hat{y} = \hat{g}^y$	$\hat{g}$	2τ	N_E
	$\hat{z} = \hat{g}^z$	$\hat{g}$	2τ	N_E
8.15 GenCredentialProof	$t_{i,1} = \hat{g}^{\omega_{i,1}}$	$\hat{g}$	2τ	N_E
	$t_{i,2} = \hat{g}^{\omega_{i,2}}$	$\hat{g}$	2τ	N_E
	$t_{i,3} = \hat{g}^{\omega_{i,3}}$	$\hat{g}$	2τ	N_E
8.16 CheckCredentialProof	$t_{i,1} = \hat{x}_i^c \cdot \hat{g}^{s_{i,1}}$	$\hat{g}$	2τ	$(s-1)N_E$
	$t_{i,2} = \hat{y}_i^c \cdot \hat{g}^{s_{i,2}}$	$\hat{g}$	2τ	$(s-1)N_E$
	$t_{i,3} = \hat{z}_i^c \cdot \hat{g}^{s_{i,3}}$	$\hat{g}$	2τ	$(s-1)N_E$
8.26 CheckBallotProof	$t_1 = \hat{x}^c \cdot \hat{g}^{s_1}$	$\hat{g}$	2τ	N
	$t_2 = a_1^c \cdot s_2 \cdot pk^{s_3}$	pk	$\lVert p \rVert$	N
	$t_3 = a_2^c \cdot g^{s_3}$	g	$\lVert q \rVert$	N
8.27 GenResponse	$p'_i = p_i^{z_1}$	$p_1, \ldots, p_n$	$\lVert q \rVert$	N each
	$d = pk^{z_1} g^{z_2}$	pk, g	$\lVert q \rVert$	$2N$
8.36 CheckConfirmationProof	$t_1 = \hat{y}^c \cdot \hat{g}^{s_1}$	$\hat{g}$	2τ	N
	$t_2 = \hat{z}^c \cdot \hat{g}^{s_2}$	$\hat{g}$	2τ	N
8.44 GenReEncryption	$\tilde{a} = a \cdot pk^{\tilde{r}}$	pk	$\lVert q \rVert$	N
	$\tilde{b} = b \cdot g^{\tilde{r}}$	g	$\lVert q \rVert$	N
8.45 GenShuffleProof	$\hat{t}_i = g^{R'_i} h^{U'_i}$	g, h	$\lVert q \rVert$	$2N$
8.46 GenPermutationCommitment	$c_{j_i} = g^{r_{j_i}} h_i$	g	$\lVert q \rVert$	N
8.47 GenCommitmentChain	$\hat{c}_i = g^{R_i} h^{U_i}$	g, h	$\lVert q \rVert$	$2N$
8.48 CheckShuffleProof	$\hat{t}_i = \hat{c}_i^c \cdot g^{\hat{s}_i} \cdot \hat{c}_{i-1}^{\tilde{s}_i}$	g	$\lVert q \rVert$	$(s-1)N$
8.58 GenSignature (3 times)	$pk = \hat{g}^{sk}$	$\hat{g}$	2τ	$3N$
	$t = \hat{g}^{\omega}$	$\hat{g}$	2τ	$3N$

Table 12.4.: Fixed-base exponentiations of the election authorities during a complete protocol run with N_E voters, s authorities, N submitted ballots, and n candidates.

	pk	g	h	$d^{(1)}, \ldots, d^{(s)}$	$p_1, \ldots, p_n$	$\hat{g}$
Voting Client	k	k	–	k each	s each	–
Election Authority	$3N$	$(s+5)N$	$2N$	–	N each	$3(s+1)N_E + 9N$

Table 12.5.: Total number of exponentiations for different fixed bases during a complete protocol run with N_E voters, s authorities, N submitted ballots, n candidates, and k selections.

12.4. Parallelization

With the optimizations proposed in the previous sections, almost all performance bottlenecks can be relaxed considerably. Only a few expensive modular exponentiations remain, for example in Alg. 8.49, where the right-hand sides of the ElGamal encryptions are all different. The performance of these computations can only be improved by executing them in parallel, for example using the different cores of a multi-core CPU. We highly recommend parallelization in such cases, but not only there. Even if fixed-base exponentiation algorithms are remarkably efficient, they are still relatively expensive compared to most other computations.

We therefore recommend to apply parallelization to every loop that contains modular exponentiations, independently of whether the base is fixed or not or the exponents are short or long. All loops over N_E, s, N, n, or k are candidates for parallelization, as long as the involved exponents are either independent or computed beforehand.[2] If such a loop is executed in parallel on a machine with c equally powerful cores, it will terminate approximately c times faster (some communication overhead between the cores is unavoidable).

In the case of product exponentiations, the application of parallelization is less obvious, because an algorithm such as `HLG2` acts as a black box. This means that parallelization must be installed into the algorithm. This is certainly possible, but it is difficult to estimate the expected performance benefit without studying the problem in depth.

12.5. Other Performance Considerations

In the set of election parameters, the highest scalability must be permitted to the number of voters N_E and the number of ballots N (which is usually of the same order of magnitude as N_E). With the previously proposed performance optimizations, all linear algorithms running in $O(N_E)$ or $O(N)$ time should be sufficiently efficient for all considered use cases. However, performance problems may occur in quadratic algorithms running in $O(N_E^2)$ or $O(N^2)$ time. By looking at the given pseudo-code algorithms in Chapter 8, it seems that no such cases exist. However, depending on the running times of the underlying data structures, such cases may occur silently in an actual implementation. We mention this here for making developers aware of this possibility and the resulting performance problems in such cases.

Examples of this type of problem occur in serialization algorithms, which are required for transmitting the messages over a network from one party to another. Serialization means to encode the message into a sequence of primitive values such as bytes or characters. Note that some messages in the CHVote protocol are $O(N_E)$ or $O(N)$ in size. To avoid quadratic running times in such cases, it is important to implement serialization using data structures providing constant-time append or concatenation operations. The `StringBuilder` class in Java is an example of a data structure offering this property (other than the primitive string concatenation operator `+`). By strictly using efficient data structures in serialization algorithms, the creation of unexpected performance problems can be avoided.

[2]This is not the case in the second for-loop of Alg. 8.45, where the exponents R'_i and U'_i are highly dependent. However, by splitting the loop into two loops, one for computing the exponents and one for the modular exponentiations, it is easily possible to establish the precondition for executing the modular exponentiations in parallel. The same remark applies to the for-loop in Alg. 8.47.

Another potential source of performance problems is the RecHash algorithm from Section 4.4.2. While implementations of hash algorithms such as SHA3-256 are remarkably efficient, they are still relatively expensive operations. If recursive hashing from Alg. 4.15 is applied to large structures such as the eligibility matrix $\mathbf{E} \in \mathbb{B}^{N_E \times t}$ of size $N_E \cdot t$, then a performance problem may occur in large elections, especially because the hash value of $\mathbf{E}$ is required multiple times. For keeping the application of the RecHash algorithm efficient in all cases, it is important to cache the hash values of the inputs that occur frequently. Therefore, we recommend implementing Alg. 4.15 always in combination with an appropriate caching mechanism.

Part V.

Conclusion

13. Conclusion

13.1. 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 [8]. 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 [31].
- 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 and performance. The strict parametrization is also an important prerequisite for formal security proofs [17].

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, 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 an implemented system.

13.2. 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.

- *Secure Printing*: According to the adversary model from Section 6.2, the most critical component in our protocol is the printing authority. 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. This problem gets even more challenging, if one of the proposals of Section 11.2 for improved usability is implemented.

- *Privacy Attacks 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 [50], 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 have a considerable impact on the overall costs. At the moment, however, we do not see a better solution.

List of Tables

List of Protocols

List of Algorithms

Bibliography

[1] Advances encyption standard (aes). FIPS PUB 197, National Institute of Standards and Technology (NIST), 2001.

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

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

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

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

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

[7] *Technische und administrative Anforderungen an die elektronischen Stimmabgabe (Version 2.0).* Die Schweizerische Bundeskanzlei (BK), 2018.

[8] *Verordnung der Bundeskanzlei über die elektronische Stimmabgabe (VEleS) vom 13. Dezember 2013 (Stand vom 1. Juli 2018).* Die Schweizerische Bundeskanzlei (BK), 2018.

[9] 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.

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

[11] E. Bach and J. Shallit. *Algorithmic Number Theory, Vol. 1: Efficient Algorithms.* MIT Press, 1996.

[12] E. Bangerter, E. Ghadafi, S. Krenn, A. R. Sadeghi, T. Schneider, N. P. Smart, and B. Warinschi. Initial report on unified theoretical framework of efficient ZK-POK. Initial Report D3.1, CACE, Computer Aided Cryptography Engineering, 2008.

[13] 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.

[14] M. Bellare, A. Boldyreva, K. Kurosawa, and J. Staddon. Multirecipient encryption schemes: How to save on bandwidth and computation without sacrificing security. *IEEE Transactions on Information Theory*, 53(11):3927–3943, 2007.

[15] M. Bellare, A. Boldyreva, and J. Staddon. Randomness re-use in multi-recipient encryption schemes. In Y. Desmedt, editor, *PKC'03, 6th International Workshop on Theory and Practice in Public Key Cryptography*, LNCS 2567, pages 85–99, Miami, USA, 2003.

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

[17] D. Bernhard, V. Cortier, P. Gaudry, M. Turuani, and B. Warinschi. Verifiability analysis of CHVote. *IACR Cryptology ePrint Archive*, 2018/1052, 2018.

[18] D. J. Bernstein. Multi-user Schnorr security, revisited. *IACR Cryptology ePrint Archive*, 2015.

[19] 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.

[20] X. Boyen. A promenade through the new cryptography of bilinear pairings. In *ITW'06, IEEE Information Theory Workshop*, pages 19–23, Punta del Este, Uruguay, 2006.

[21] 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.

[22] 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.

[23] 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.

[24] M. Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. NIST Special Publication 800–38D, NIST, 2007.

[25] 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.

[26] 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.

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

[28] D. Galindo, S. Guasch, and J. Puiggalí. Swiss online voting protocol. Technical report, Scytl Secure Electronic Voting, Barcelona, Spain, 2016.

[29] 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.

[30] 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.

[31] 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.

[32] R. Haenni, R. E. Koenig, P. Locher, and E. Dubuis. CHVote system specification – version 1.0. *IACR Cryptology ePrint Archive*, 2017/325, 2017.

[33] R. Haenni and P. Locher. Performance of shuffling: Taking it to the limits. In M. Bernhard, L. J. Camp A. Bracciali and, S. Matsuo, A. Maurushat, P. B. Rønne, and M. Sala, editors, *Voting'20, FC 2020 International Workshops*, LNCS 12063, pages 369–385, Kota Kinabalu, Malaysia, 2020.

[34] R. Haenni, P. Locher, and N. Gailly. Improving the performance of cryptographic voting protocols. In A. Bracciali, J. Clark, F. Pintore, P. B. Rønne, and M. Sala, editors, *Voting'19, FC 2019 International Workshops*, LNCS 11599, pages 272–288, Frigate Bay, St. Kitts and Nevis, 2019.

[35] R. Haenni, P. Locher, R. E. Koenig, and E. Dubuis. Pseudo-code algorithms for verifiable re-encryption mix-nets. In M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. A. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, editors, *FC'17, 21st International Conference on Financial Cryptography*, LNCS 10323, pages 370–384, Silema, Malta, 2017.

[36] F. Hao. Schnorr non-interactive zero-knowledge proof. RFC 8235, Internet Engineering Task Force (IETF), 2017.

[37] B. R. Johnson and D. J. Leeming. A study of the digits of π, e and certain other irrational numbers. *Sankhyā: The Indian Journal of Statistics*, 52(2):183–189, 1990.

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

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

[40] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In T. Rabin, editor, *CRYPTO'08, 30th Annual International Cryptology Conference*, LNCS 6223, pages 631–648, Santa Barbara, USA, 2010.

[41] H. Krawczyk and P. Eronen. Hmac-based extract-and-expand key derivation function (hkdf). RFC 5869, Internet Engineering Task Force (IETF), 2000.

[42] K. Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In D. Naccache and P. Paillier, editors, *PKC'02, 5th International Workshop on Theory and Practice in Public Key Cryptography*, LNCS 2274, pages 48–63, Paris, France, 2002.

[43] B. Laurie, E. Messeri, and R. Stradling. Certificate Transparency Version 2.0. RFC 9162, Internet Engineering Task Force (IETF), 2021.

[44] C. H. Lim and J. P. Lee. More flexible exponentiation with precomputation. In Y. Desmedt, editor, *CRYPTO'94, 14th Annual International Cryptology Conference on Advances in Cryptology*, LNCS 839, pages 95–107, Santa Barbara, USA, 1994.

[45] 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.

[46] 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.

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

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

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

[50] 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.

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

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

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

[54] 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.

[55] 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.

[56] 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.

[57] M. J. Wiener. Safe prime generation with a combined sieve. *IACR Cryptology ePrint Archive*, 2003/186, 2003.

[58] 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.