CHVote System Specification

Version 2.0

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

September 11, 2018

Bern University of Applied Sciences
CH-2501 Biel, Switzerland
## Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Author(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>14.07.2016</td>
<td>Rolf Haenni</td>
<td>Initial Draft.</td>
</tr>
<tr>
<td>0.2</td>
<td>11.10.2016</td>
<td>Rolf Haenni</td>
<td>Draft to present at meeting.</td>
</tr>
<tr>
<td>0.3</td>
<td>17.10.2016</td>
<td>Rolf Haenni, Reto E. Koenig</td>
<td>Vote casting and confirmation algorithms finished.</td>
</tr>
<tr>
<td>0.4</td>
<td>24.10.2016</td>
<td>Rolf Haenni, Reto E. Koenig</td>
<td>Update of vote casting and confirmation algorithms.</td>
</tr>
<tr>
<td>0.5</td>
<td>18.11.2016</td>
<td>Rolf Haenni, Philipp Locher</td>
<td>Mixing process finished.</td>
</tr>
<tr>
<td>0.6</td>
<td>25.11.2016</td>
<td>Rolf Haenni</td>
<td>String conversion introduced, tallying finished.</td>
</tr>
<tr>
<td>0.7</td>
<td>07.12.2016</td>
<td>Rolf Haenni</td>
<td>Section 5 finished.</td>
</tr>
<tr>
<td>0.8</td>
<td>17.12.2016</td>
<td>Rolf Haenni</td>
<td>Hashing algorithms and cryptographic parameters added.</td>
</tr>
<tr>
<td>0.9</td>
<td>10.01.2017</td>
<td>Rolf Haenni</td>
<td>Section 8 finished.</td>
</tr>
<tr>
<td>0.10</td>
<td>06.02.2017</td>
<td>Rolf Haenni</td>
<td>Security parameters finished.</td>
</tr>
<tr>
<td>0.11</td>
<td>21.02.2017</td>
<td>Rolf Haenni</td>
<td>Section 6 finished, reorganization of Section 7.</td>
</tr>
<tr>
<td>0.11</td>
<td>14.03.2017</td>
<td>Rolf Haenni</td>
<td>Section 7 finished.</td>
</tr>
<tr>
<td>0.12</td>
<td>21.03.2017</td>
<td>Rolf Haenni</td>
<td>Section 8 finished.</td>
</tr>
<tr>
<td>0.13</td>
<td>30.03.2017</td>
<td>Rolf Haenni</td>
<td>Minor corrections, Section 1 finished.</td>
</tr>
<tr>
<td>1.0</td>
<td>12.04.2017</td>
<td>Rolf Haenni</td>
<td>Minor corrections, Section 2 finished.</td>
</tr>
<tr>
<td>1.1</td>
<td>19.04.2017</td>
<td>Rolf Haenni</td>
<td>Conclusion added.</td>
</tr>
<tr>
<td>1.1.1</td>
<td>24.05.2017</td>
<td>Rolf Haenni</td>
<td>Parameter changes.</td>
</tr>
<tr>
<td>1.2</td>
<td>14.07.2017</td>
<td>Rolf Haenni</td>
<td>Major protocol revision, full sender privacy added to oblivious transfer.</td>
</tr>
<tr>
<td>1.3</td>
<td>28.09.2017</td>
<td>Rolf Haenni</td>
<td>Description and algorithms for channel security added.</td>
</tr>
<tr>
<td>1.3.1</td>
<td>29.11.2017</td>
<td>Rolf Haenni</td>
<td>Optimization in Alg. 7.25.</td>
</tr>
<tr>
<td>1.3.2</td>
<td>6.12.2017</td>
<td>Rolf Haenni</td>
<td>Adjusted ballot proof generation and verification.</td>
</tr>
<tr>
<td>1.4</td>
<td>26.3.2018</td>
<td>Rolf Haenni</td>
<td>Proposals for improved usability in Section 9.2.</td>
</tr>
<tr>
<td>1.4.1</td>
<td>12.4.2018</td>
<td>Rolf Haenni</td>
<td>Recapitulation added to 9.2.</td>
</tr>
<tr>
<td>1.4.2</td>
<td>29.6.2018</td>
<td>Rolf Haenni</td>
<td>Two minor errors corrected.</td>
</tr>
<tr>
<td>2.0</td>
<td>31.8.2018</td>
<td>Rolf Haenni</td>
<td>Recommendations by external reviewers implemented.</td>
</tr>
</tbody>
</table>
4.2. Strings ......................................................... 27
4.2.1. Converting Integers to Strings .......................... 27
4.2.2. Converting Strings to Integers .......................... 28
4.2.3. Converting Byte Arrays to Strings ....................... 28
4.3. Hash Algorithms ................................................. 29
4.3.1. Hash Values of Integers and Strings ..................... 29
4.3.2. Hash Values of Multiple Inputs .......................... 29
5. Cryptographic Primitives .......................... 31
5.1. ElGamal Encryption ........................................... 31
5.1.1. Using a Single Key Pair ................................ 31
5.1.2. Using a Shared Key Pair ................................. 32
5.2. Pedersen Commitment ......................................... 32
5.3. Oblivious Transfer .............................................. 33
5.3.1. OT-Scheme by Chu and Tzeng ........................... 34
5.3.2. Full Sender Privacy in the OT-Scheme by Chu and Tzeng 34
5.3.3. Simultaneous Oblivious Transfers ....................... 37
5.3.4. Oblivious Transfer of Long Messages ................... 38
5.4. Non-Interactive Preimage Proofs .......................... 39
5.4.1. Composition of Preimage Proofs ......................... 39
5.4.2. Applications of Preimage Proofs ....................... 40
5.5. Wikström’s Shuffle Proof ..................................... 40
5.5.1. Preparatory Work ......................................... 41
5.5.2. Preimage Proof ............................................. 43
5.6. Schnorr Signatures ............................................ 44
5.7. Hybrid Encryption and Key-Encapsulation .................. 45
III. Protocol Specification .................................. 47
6. Protocol Description ...................................... 48
6.1. Parties and Communication Channels ...................... 48
6.2. Adversary Model and Trust Assumptions ................. 50
6.3. System Parameters .................................................. 51
    6.3.1. Security Parameters ........................................... 51
    6.3.2. Election Parameters .......................................... 56
6.4. Technical Preliminaries ............................................ 58
    6.4.1. Encoding of Votes and Counting Circles .................. 58
    6.4.2. Linking OT Queries to ElGamal Encryptions ............... 58
    6.4.3. Validity of Encrypted Votes ............................... 59
    6.4.4. Voter Identification ........................................ 60
    6.4.5. Election Outcome ............................................ 61
6.5. Protocol Description ............................................... 62
    6.5.1. Pre-Election Phase .......................................... 63
    6.5.2. Election Phase .............................................. 65
    6.5.3. Post-Election Phase ......................................... 70
6.6. Channel Security .................................................. 72

7. Pseudo-Code Algorithms .............................................. 75
    7.1. Conventions and Assumptions ................................ 75
    7.2. General Algorithms ............................................ 76
    7.3. Pre-Election Phase ............................................. 79
    7.4. Election Phase ................................................. 87
    7.5. Post-Election Phase ............................................ 98
    7.6. Channel Security .............................................. 108

IV. System Specification ................................................ 111

8. Security Levels and Parameters ................................... 112
    8.1. Recommended Length Parameters ............................... 112
    8.2. Recommended Group and Field Parameters .................... 113
        8.2.1. Level 0 (Testing Only) ................................. 114
        8.2.2. Level 1 .................................................. 115
        8.2.3. Level 2 .................................................. 116
        8.2.4. Level 3 .................................................. 118
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)
- 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 Hofer (République et Canton de Genève, Switzerland)
- Pascal Junod (Snap Inc., 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)
Part I.

Project Context
1. Introduction

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

While the Geneva Internet voting project continued to expand, concerns about possible vulnerabilities had been raised by security experts and scientists. There were two main points of criticism: the lack of transparency and verifiability and the insecure platform problem [45]. The concept of verifiable elections has been known in the scientific literature for quite some time [11], but the Geneva e-voting system—like most other e-voting systems in the world at that time—remained completely unverifiable. The awareness of the insecure platform problem was given from the beginning of the project [44], but so-called code voting approaches and other possible solutions were rejected due to usability concerns and legal problems [42].

In the cryptographic literature on remote electronic voting, a large amount of solutions have been proposed for both problems. One of the most interesting approaches, which solves the insecure platform problem by adding a verification step to the vote casting procedure, was implemented in the Norwegian Internet voting system and tested in legally binding municipal and county council elections in 2011 and 2013 [8, 25, 26, 48]. The Norwegian project was one of the first in the world that tried to achieve a maximum degree of transparency and verifiability from the very beginning of the project. Despite the fact that the project has been stopped in 2014 (mainly due to the lack of increase in turnout), it still serves as a model for future projects and second-generation systems.

As a response to the third report on Vote électronique by the Swiss Federal Council and the new requirements of the Swiss Federal Chancellery [41, 5], the State of Geneva decided to introduce a radical strategic change towards maximum transparency and full verifiability. For this, they invited leading scientific researchers and security experts to contribute to the development of their second-generation system, in particular by designing a cryptographic voting protocol that satisfies the requirements to the best possible degree. In this context, a collaboration contract between the State of Geneva and the Bern University of Applied
Sciences was signed in 2016. The goal of this collaboration is to lay the foundation for an entirely new system, which will be implemented from scratch.

As a first significant outcome of this collaboration, a scientific publication with a proposal for a cryptographic voting protocol was published in 2016 at the 12th International Joint Conference on Electronic Voting [28]. The proposed approach is the basis for the specification presented in this document. Compared to the protocol as presented in the publication, the level of technical details in this document is considerably higher. By providing more background information and a broader coverage of relevant aspects, this text is also more self-contained and comprehensive than its predecessor.

The core of this document is a set of approximately 60 algorithms in pseudo-code, which are executed by the protocol parties during the election process. The presentation of these algorithms is sufficiently detailed for an experienced software developer to implement the protocol in a modern programming language. Cryptographic libraries are only required for standard primitives such as hash algorithms, pseudo-random generators, and computations with large integers. For one important sub-task of the protocol—the mixing of the encrypted votes—a second scientific publication was published in 2017 at the 21th International Conference on Financial Cryptography [29]. By facilitating the implementation of a complex cryptographic primitive by non-specialists, this paper created a useful link between the theory of cryptographic research and the practice of implementing cryptographic systems. The comprehensive specification of this document, which encompasses all technical details of a fully-featured cryptographic voting protocol, provides a similar, but much broader link between theory and practice.

1.1. Principal Requirements

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

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

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

---

1 See https://github.com/republique-et-canton-de-geneve/chvote-protocol-poc for a complete proof of concept implementation in Java by a developer of the CHVote project.
• **Individual Verifiability:** After submitting an encrypted vote, the voter receives conclusive evidence that the vote has been cast and recorded as intended. This evidence enables the voter to exclude with high probability the possibility that the vote has been manipulated by a compromised voting client. According to [4, Paragraph 4.2.4], this is the proposed countermeasure against the insecure platform problem. The probability of detecting a compromised vote must be 99.9% or higher.

• **Universal Verifiability:** The correctness of the election result can be tested by independent verifiers. The verification includes checks that only votes cast by eligible voters have been tallied, that every eligible voter has voted at most once, and that every vote cast by an eligible voter has been tallied as recorded.

• **Distribution of Trust:** Several independent control components participate in the election process, for example by sharing the private decryption key or by performing individual anonymization steps. While single control components are not fully trusted, it is assumed that they are trustworthy as a group, i.e., that at least one of them will prevent or detect any type of attack or failure. The general goal of distributing trust in this way is to prevent single points of failures.

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

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

1.2. Goal and Content of Document

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

What is currently entirely missing in this document are proper definitions of the security properties and corresponding formal proofs that these properties hold in this protocol. An informal discussion of such properties is included in the predecessor document [28], 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 on June 13, 2018 [12]. 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. A detailed summary of the protocol changes from Version 1.4.2 to Version 2.0 is given in Section A.1

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

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

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

2.1. General Election Procedure

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

**Voting Card Nr. 3587**

<table>
<thead>
<tr>
<th>Question 1: Etiam dictum sem pulvinar elit con vallis vehicula. Duis vitae purus ac tortor volut pat iaculis at sed mauris at tempor quam?</th>
<th>Yes</th>
<th>No</th>
<th>Blank</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A34C</td>
<td>18F5</td>
<td>76BC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question 2: Donec at consectetur ex. Quisque fermentum ipsum sed est pharetra molestie. Sed at nisl malesuada ex mollis consequat?</th>
<th>Yes</th>
<th>No</th>
<th>Blank</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>91F3</td>
<td>71BD</td>
<td>034A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question 3: Mauris rutrum tellus et lorem vehicula, quis ornare tortor vestibulum. In tempor, quam sit amet sodales sagittis, nib quam placerat?</th>
<th>Yes</th>
<th>No</th>
<th>Blank</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>774C</td>
<td>CB4A</td>
<td>76F2</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

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

### 2.2. Election Use Cases

The voting protocol presented in this document is designed to support election events consisting of \(t \geq 1\) simultaneous elections. Every election \(j \in \{1, \ldots, t\}\) is modeled as an independent \(k_j\)-out-of-\(n_j\) election with \(n_j \geq 2\) 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, \(k = (k_1, \ldots, k_t)\) and \(n = (n_1, \ldots, n_t)\) denote corresponding vectors of values. Note that the constraints \(0 < k_j < n_j\) and \(t \geq 1\) 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

---

\(^1\)Here we use voting system as a general term for all server-side parties involved in the election phase of the protocol.
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 $w \geq 1$ 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 $w = (w_1, \ldots, w_{N_E})$ 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 $w \leq 380$. As we will see in Section 9.1.2, we limit the total number of candidates in an election event to $n \leq 1678$, 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 from 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 $t_j \leq t$ elections belong to a municipality, which itself corresponds to a counting circle $j \in \{1, \ldots, w\}$, then citizens not belonging to this municipality are not eligible in those $t_j$ 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 $E = (e_{ij})_{N_E \times t}$, 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 $k' = (k'_1, \ldots, k'_{N_E})$ and their maximum $k' = \max_{i=1}^{N_E} k'_i$ is called maximal ballot size.

To illustrate this general situation, consider the following example with $N_E = 8$ eligible voters, $w = 2$ counting circles, and $t = 9$ different 1-out-of-2 elections (which implies $k = 9$ and $n = 18$). Furthermore, assume that the first five 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, 5, and 6 are restricted to the first and Elections 7, 8, and 9 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 7 has no voting right in Election 8. The situation in this example leads to the following values:

\[
k = \begin{pmatrix} 1 \\ 2 \\ 1 \\ 1 \\ 1 \\ 2 \\ 2 \\ 2 \\ 2 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \quad n = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \end{pmatrix}, \quad w = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \end{pmatrix}, \quad E = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}, \quad k' = \begin{pmatrix} 5 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \\ 6 \end{pmatrix}, \quad k' = 6.
\]

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 \( \sigma = k' / k \) as an indicator of the sparseness of \( E \). 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 \( k_i = 1 \) in all elections, we get \( \sigma = 0.667 \). As a consequence, all affected algorithms will run approximately 30% faster by implementing corresponding optimizations.

In the case of a voter with a restricted eligibility within a counting circle (such as Voters 1, 4 and 7 in the example above), a privacy problem may occur during tallying if the voter’s individual pattern of the eligibility matrix remains visible in the list of decrypted votes (see recommendation in [12, Section 10.6]). To diminish the magnitude of this problem, we propose to extend the ballots of such voters to the size of the ballots of the remaining voters from counting circle \( w_i \) with unrestricted voting rights. For this, we infer from \( w \) and \( E \) the default eligibility matrix \( E^* = (e_{cj}^*)_{w \times t} \). 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 \( j \in \{1, \ldots, t\} \). They can be computed by

\[
e_{cj}^* = 1 - \prod_{i \in I_c} (1 - e_{ij}),
\]

where \( I_c = \{ i \in \{1, \ldots, N_E\} : w_i = c \} \) denotes the set of indices of voters from the same counting circle. Note that this computation is equivalent to applying the logical-or operator to the values \( e_{ij} \) of all voters in \( I_c \).

From \( E^* \) we can infer the default ballot size \( k_c^* = \sum_{j=1}^{t} e_{cj}^* \) of each counting circle \( c \), which defines a vector \( k^* = (k_1^*, \ldots, k_w^*) \) of size \( w \). In the example above with \( w = 2 \) and \( t = 9 \), we get

\[
E^* = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \quad \text{and} \quad k^* = \begin{pmatrix} 6 \\ 5 \end{pmatrix}.
\]

For voter \( i \in \{1, \ldots, N_E\} \) with restricted eligibility, the default ballot size of the voter’s counting circle \( w_i \) can be reached by adding \( k_{w_i}^* - k_i^* \) default votes to the voter’s ballot, one for each supplementary restriction in the eligibility matrix. Therefore, computing the matrix \( E^* \) 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 \( w_i \). When this happens, we get \( e_{ij} = 0 \) and \( e_{w_i,j}^* = 1 \), i.e., such cases can be detected by checking \( e_{ij} < e_{w_i,j}^* \) for all elections \( j \in \{1, \ldots, t\} \). This is the test that we use in Alg. 7.45 for toggling the inclusion of default votes. At tallying, the corresponding total amount of default votes needs to be subtracted from the final result. From a technical point of view, any of the \( n_j \) candidates of election \( j \) could serve as a default candidate in a default vote, but for solving the above-mentioned privacy problem, it is important to ensure that default candidates are unlikely to obtain zero votes from regular voters. We assume that a Boolean vector \( u = (u_1, \ldots, u_n) \), \( u_j \in \mathbb{B} \), specifying suitable default candidates is known to everyone. In Sections 6.3.2 and 6.4.5, we will give more details on how default candidates are defined and used in the protocol.

### 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. At first glance, such \( k \)-out-of-\( n \) elections may seem 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 \( k_{\text{min}} \) and \( k_{\text{max}} \) 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 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 \( c \leq k \) votes to the same candidate. This is called cumulation. In the most flexible case of cumulation, the \( k \) votes can be distributed among the \( n \) candidates in an arbitrary manner. This case can be handled by increasing the size of the candidate list from \( n \) to \( n' = cn \), i.e., each candidate obtains \( c \) distinct entries in the extended candidate list. This leads to an equivalent \( k \)-out-of-\( n' \) election, in which voters may select the same candidate up to \( c \) times by selecting all its entries in the extended list. At the end of the election, an additional accumulation step is necessary to determine the exact number of votes of a given candidate from the final tally. By combining this technique of handling cumulations with the above way of handling blank votes, we obtain \( k' \)-out-of-\( n' \) elections with \( k' = k_{\text{max}} \) 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 maximal \( c = 2 \) votes per candidate. Blank votes are allowed for both the party and the candidate election.

\[ ^2 \text{In this subsection, we ignore the question of counting circles and voters with restricted eligibility.} \]
In a special case of such a party-list elections, a completely blank candidate ballot is prohibited 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.

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

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.

An additional complication arises by allowing a distinction between vote abstentions 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 $k = 1$):

- 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 $n_{\text{max}} = 1678$ (see Section 9.1.2).
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 \( X = \{x_1, \ldots, x_n\} \). For composed sets or subsets of composed sets, we use calligraphic upper-case Latin letters, for example \( \mathcal{X} \subseteq X \times Y \times Z \) for the set or a subset of triples \((x, y, z)\). \(|X|\) denotes the cardinality of a finite set \(X\).

For general tuples, we use lower-case Latin or Greek letters in normal font, for example \(t = (x, y, z)\) for triples from \(X \times Y \times Z\). For sequences (arrays, lists, strings), we use upper-case Latin letters and indices starting from 0, for example \(S = \langle s_0, \ldots, s_{n-1} \rangle \in A^*\) for a string of characters \(s_i \in A\), where \(A\) is a given alphabet. We write \(|S| = n\) for the length of \(S\) and use standard array notation \(S[i] = s_i\) to select the element at index \(i \in \{0, \ldots, n-1\}\). \(S_1 \parallel S_2\) denotes the concatenation of two sequences. For vectors, we use lower-case Latin letters in bold font, for example \(x = (x_1, \ldots, x_n) \in X^n\) for a vector of length \(|x| = n\).

For two-dimensional (or higher-dimensional) matrices, we use upper-case Latin letters in bold font, for example
\[
X = \begin{pmatrix}
x_{1,1} & \cdots & x_{1,n} \\
\vdots & \ddots & \vdots \\
x_{m,1} & \cdots & x_{m,n}
\end{pmatrix} \in X^{m \times n}
\]
for an \(m\)-by-\(n\) matrix of values \(x_{ij} \in X\). We use \(X = (x_{ij})_{m \times n} \in X^{mn}\) as a shortcut notation.

Similarly, \(X = (x_{ijk})_{m \times n \times r} \in X^{m \times n \times r}\) is a shortcut notation for a three-dimensional \(m\)-by-\(n\)-by-\(r\) matrix of values \(x_{ijk} \in X\). For a two-dimensional vector \(X = (x_{ij})_{m \times n}\), we write \((x_{i,1}, \ldots, x_{i,n}) \leftarrow \text{Row}(X, i)\) for selecting the \(i\)-th row vector and \((x_{1,j}, \ldots, x_{m,j}) \leftarrow \text{Col}(X, j)\) for selecting the \(j\)-th column vector of \(X\).

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

For an integer \(x \in \mathbb{Z}\), we write \(\text{abs}(x)\) for the absolute value of \(x\) and \(|x| = \lceil \log_2(\text{abs}(x)) \rceil + 1\) for the bit length of \(x \neq 0\) (let \(|0| = 0\) by definition). The set of all natural numbers of a given bit length \(l \geq 1\) 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.
3.2. Mathematical Groups

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

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

Usually, groups are written either additively as $G = (\{+, -, 0\})$ or multiplicatively as $G = (\{\times, -1\}, 1)$, 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 $k \cdot x = (k \mod q) \cdot x$ and $x^k = x^{k \mod q}$. 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 $G$ and its set of elements $G$.

3.2.1. The Multiplicative Group of Integers Modulo $p$

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

Let $g$ be a generator of $\mathbb{Z}_p^*$ and $x \in \mathbb{Z}_p^*$ an arbitrary group element. The problem of finding a value $k \geq 0$ 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). It is widely believed that DL is hard in $\mathbb{Z}_p^*$. A related problem, called decisional Diffie-Hellman problem (DDH), consists in distinguishing two triples $(g^a, g^b, g^{ab})$
and \((g^a, g^b, g^c)\) for random exponents \(a, b, c\). While DDH is known to be easy in \(\mathbb{Z}_p^\ast\), it is believed that DDH is hard in large subgroups of \(\mathbb{Z}_p^\ast\).

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

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

### 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 \(\mathbb{Z}_p^\ast\). The same holds for groups of prime order \(q\), for example for subgroups \(G_q \subset \mathbb{Z}_p^\ast\).

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 \((\mathbb{Z}_p^\ast, \times, 1)\) 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 \(d \geq 0\) and with coefficients \(a_i \in \mathbb{F}_p\) (degree \(d\) means \(a_d \neq 0\)). Clearly, such polynomials are fully determined by the list \(\mathbf{a} = (a_0, \ldots, a_d)\) 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 \(\mathbf{p} = (p_1, \ldots, p_d), p_i \in (x_i, y_i) \in \mathbb{F}_p^2\), we obtain

\[
a_0 = \sum_{i=0}^{d} y_i \left( \prod_{0 \leq j \leq d, j \neq i} \frac{x_j}{x_j - x_i} \right).
\]

by applying Lagrange’s general method to \(X = 0\).
4. Type Conversion and Hash Algorithms

4.1. Byte Arrays

Let \( B = \langle b_0, \ldots, b_{n-1} \rangle \) denote an array of bytes \( b_i \in B \), where \( B = \mathbb{B}^8 \) 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, \( B = \langle 0A, 23, EF \rangle \) denotes a byte array containing three bytes \( B[0] = 0x0A = 00001010_2 \), \( B[1] = 0x23 = 001000011_2 \), and \( B[2] = 0xEF = 11101111_2 \).

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

\[
\text{Truncate}(B, m) = \langle B[0], \ldots, B[m-1] \rangle, \\
\text{Skip}(B, m) = \langle B[m], \ldots, B[n-1] \rangle,
\]

respectively. Clearly, \( B = \text{Truncate}(B, m) \parallel \text{Skip}(B, m) \) holds for all \( B \in B^* \) and all \( 0 \leq m \leq n \).

Another basic byte array operation is needed for generating unique verification codes on every voting card (see Section 6.3.1 and Algs. 7.16 and 7.34). The goal of this operation is similar to a digital watermark, which we use here for making verification codes unique on each voting card. Below we define an algorithm \textbf{MarkByteArray}(\( B, m, m_{\max} \)), which adds an integer watermark \( m \), \( 0 \leq m \leq m_{\max} \), to the bits of a byte array \( B \).

\begin{algorithm}
\textbf{Algorithm: MarkByteArray}(\( B, m, m_{\max} \))
\begin{itemize}
  \item \textbf{Input:} Byte arrays \( B \in B^* \)
  \item Watermark \( m \), \( 0 \leq m \leq m_{\max} \)
  \item Maximal watermark \( m_{\max} \), \( \|m_{\max}\| \leq 8 \cdot \|B\| \)
  \item \( l \leftarrow \|m_{\max}\| \)
  \item \( s \leftarrow \frac{8 \cdot \|B\|}{l} \)
  \item for \( i = 0, \ldots, l - 1 \) do
    \begin{align*}
      &B \leftarrow \text{SetBit}(B, \lfloor i \cdot s \rfloor, m \mod 2) &// \text{see Alg. 4.2} \\
      &m \leftarrow \lfloor m/2 \rfloor
    \end{align*}
  \item return \( B \) // \( B \in B^* \)
\end{itemize}
\end{algorithm}

Algorithm 4.1: Adds an integer watermark \( m \) to the bits of a given byte array. The bits of the watermark are spread equally across the bits of the byte array.
Algorithm: SetBit($B, i, b$)

**Input:** ByteArray $B \in B^*$
- Index $i$, $0 \leq i < 8 \cdot |B|$
- Bit $b \in \mathbb{B}$

$j \leftarrow \lfloor i/8 \rfloor$

$x \leftarrow 2^j \mod 8$

if $b = 0$ then

$B[j] \leftarrow B[j] \land (255 - x)$  // $\land$ denotes the bitwise AND operator

else

$B[j] \leftarrow B[j] \lor x$  // $\lor$ denotes the bitwise OR operator

return $B$  // $B \in B^*$

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

### 4.1.1. Converting Integers to Byte Arrays

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

$$B = \langle b_0, \ldots, b_{n-1} \rangle,$$

where $b_i = \left\lfloor \frac{x}{256^{n-i-1}} \right\rfloor \mod 256$.

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

<table>
<thead>
<tr>
<th>$x$</th>
<th>$n = 0$</th>
<th>$n = 1$</th>
<th>$n = 2$</th>
<th>$n = 3$</th>
<th>$n = 4$</th>
<th>$n_{\text{min}}$</th>
<th>$\text{ToByteArray}(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\Diamond$</td>
<td>$\langle 00 \rangle$</td>
<td>$\langle 00,00 \rangle$</td>
<td>$\langle 00,00,00 \rangle$</td>
<td>$\langle 00,00,00,00 \rangle$</td>
<td>0</td>
<td>$\Diamond$</td>
</tr>
<tr>
<td>1</td>
<td>$-$</td>
<td>$\langle 01 \rangle$</td>
<td>$\langle 00,01 \rangle$</td>
<td>$\langle 00,00,01 \rangle$</td>
<td>$\langle 00,00,00,01 \rangle$</td>
<td>1</td>
<td>$\langle 01 \rangle$</td>
</tr>
<tr>
<td>255</td>
<td>$-$</td>
<td>$\langle \text{FF} \rangle$</td>
<td>$\langle 00,\text{FF} \rangle$</td>
<td>$\langle 00,00,\text{FF} \rangle$</td>
<td>$\langle 00,00,00,\text{FF} \rangle$</td>
<td>1</td>
<td>$\langle \text{FF} \rangle$</td>
</tr>
<tr>
<td>256</td>
<td>$-$</td>
<td>$-$</td>
<td>$\langle 01,00 \rangle$</td>
<td>$\langle 00,01,00 \rangle$</td>
<td>$\langle 00,00,01,00 \rangle$</td>
<td>2</td>
<td>$\langle 01,00 \rangle$</td>
</tr>
<tr>
<td>65,535</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$\langle \text{FF},\text{FF} \rangle$</td>
<td>$\langle 00,\text{FF},\text{FF} \rangle$</td>
<td>$\langle 00,00,\text{FF},\text{FF} \rangle$</td>
<td>2</td>
</tr>
<tr>
<td>65,536</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$\langle 01,00,00 \rangle$</td>
<td>$\langle 00,01,00,00 \rangle$</td>
<td>3</td>
</tr>
<tr>
<td>16,777,215</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$\langle \text{FF},\text{FF},\text{FF} \rangle$</td>
<td>$\langle 00,\text{FF},\text{FF},\text{FF} \rangle$</td>
</tr>
<tr>
<td>16,777,216</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
<td>$\langle 01,00,00,00 \rangle$</td>
</tr>
</tbody>
</table>

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

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

### 4.1.2. Converting Byte Arrays to Integers

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

Input: Non-negative integer $x \in \mathbb{N}$

$n_{\text{min}} \leftarrow \left\lceil \frac{\|x\|_8}{8} \right\rceil$

$B \leftarrow \text{ToByteArray}(x, n_{\text{min}})$ // see Alg. 4.4

return $B$ // $B \in \mathcal{B}^n$

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: ToByteArray

Input: Non-negative integer $x \in \mathbb{N}$, Length of byte array $n \geq \frac{\|x\|_8}{8}$

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

$\left[ b_{n-i} \leftarrow x \mod 256 \right.$

$x \leftarrow \left\lfloor \frac{x}{256} \right\rfloor$

$B \leftarrow \langle b_0, \ldots, b_{n-1} \rangle$

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 $n \geq \frac{\|x\|_8}{8}$ of the output byte array $B$ implies that the first $n - \left\lceil \frac{\|x\|_8}{8} \right\rceil$ bytes of $B$ are zeros.

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

$$x = \sum_{i=0}^{\frac{n-1}{8}} B[i] \cdot 256^\frac{n-i-1}{8}, \text{ for } n = |B|,$$

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

4.1.3. Converting UCS Strings to Byte Arrays

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

\textbf{Input}: Byte array \( B \in \mathbb{B}^* \)

\begin{algorithm}
\begin{algorithmic}
\State \( x \leftarrow 0 \)
\For {\( i = 0, \ldots, |B| - 1 \)}
\State \( x \leftarrow 256 \cdot x + B[i] \)
\EndFor
\State \return \( x \) \Comment{\( x \in \mathbb{N} \)}
\end{algorithmic}
\end{algorithm}

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

To encode a string \( S \in \mathbb{A}_{\text{ucS}}^* \) as byte array, we use the UTF-8 character encoding as defined in ISO/IEC 10646 (Annex D). Let \( B \leftarrow \text{UTF8}(S) \) denote an algorithm that computes corresponding byte arrays \( B \in \mathbb{B}^* \), in which characters use 1, 2, 3, or 4 bytes of space depending on the type of character. For example, \( \langle 48, 65, 6C, 6C, 6F \rangle \leftarrow \text{UTF8}("Hello") \) is a byte array of length 5, because it only consists of Basic Latin characters, whereas \( \langle 56, 6F, 69, 6C, C3, A0 \rangle \leftarrow \text{UTF8}("Voilà") \) contains 6 bytes due to the Latin-1 Supplement character ‘à’ translating into two bytes.

UTF-8 is the only character encoding used in this document for general UCS strings. In Section 4.3, we use it for computing cryptographic hash values of given input strings, and in Section 7.6, we use it for encrypting and decrypting messages with a symmetric key. The latter application requires applying the UTF-8 encoding in both directions. We write \( S \leftarrow \text{UTF8}^{-1}(B) \) for converting \( B \in \mathbb{B}^* \) back into a UCS string \( S \in \mathbb{A}_{\text{ucS}}^* \). Since implementations of the UTF-8 character encoding are widely available, we do not provide an explicit pseudo-code algorithm.

4.2. Strings

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

4.2.1. Converting Integers to Strings

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

\[ S = \langle s_0, \ldots, s_{k-1} \rangle, \text{ where } s_i = \text{rank}_A^{-1}\left( \frac{x}{N^{k-i-1}} \right) \mod N. \]

We will use this conversion in Alg. 7.16 to print long integers in a more compact form. Note that the following algorithm is almost identical to Alg. 4.4 given in Section 4.1.1 to obtain byte arrays from integers.
Algorithm 4.6: Computes a string representation of length $k$ in big-endian order of a given non-negative integer $x \in \mathbb{N}$ and relative to some alphabet $A$.

### 4.2.2. Converting Strings to Integers

In Algs. 7.24 and 7.36, string representations $S \leftarrow \text{ToString}(x, k, A)$ of length $k$ must be reconverted into their original integers $x \in \mathbb{N}$. In a similar way as in Section 4.1.2, we obtain the inverse of $\text{ToString}(x, k, A)$ by

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

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

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

### 4.2.3. Converting Byte Arrays to Strings

Let $B \in \mathbb{B}^n$ be a byte array of length $n$. The goal is to represent $B$ by a unique string $S \in A^k$ of length $k$, such that $k$ is as small as possible. We will use this conversion in Algs. 7.16, 7.34 and 7.42 to print and display byte arrays in human-readable form. Since there are $|\mathbb{B}^n| = 256^n = 2^{8n}$ byte arrays of length $n$ and $|A^k| = N^k$ strings of length $k$, we derive $k = \left\lceil \frac{8n}{\log_2 N} \right\rceil$ from the inequality $2^{8n} \leq N^k$. To obtain an optimal string representation of $B$, let $x_B \leftarrow \text{ToInteger}(B) < 2^{8n}$ be the representation of $B$ as a non-negative integer. This leads to the following length-optimal mapping from $\mathbb{B}^n$ to $A^k$. 

---

**Algorithm: ToInteger($S$, $A$)**

**Input:** String $S \in A^*$
- Alphabet $A = \{c_1, \ldots, c_N\}$

```plaintext
x ← 0
for $i = 0, \ldots, |S| - 1$ do
  $x \leftarrow N \cdot x + \text{rank}_A(S[i])$
return $x$
```

// $x \in \mathbb{N}$
Algorithm 4.8: Computes the shortest string representation of a given byte array \( B \) relative to some alphabet \( A \).

### 4.3. Hash Algorithms

A cryptographic hash algorithm defines a mapping \( h : \mathbb{B}^* \rightarrow \mathbb{B}^\ell \), which transforms an input bit array \( B \in \mathbb{B}^* \) of arbitrary length into an output bit array \( h(B) \in \mathbb{B}^\ell \) of length \( \ell \), called the hash value of \( B \). In practice, hash algorithms such as SHA-1 or SHA-256 operate on byte arrays rather than bit arrays, which implies that the length of the input and output bit arrays is a multiple of 8. We denote such practical algorithms by \( H \leftarrow \text{Hash}_L(B) \), where \( B \in \mathbb{B}^* \) and \( H \in \mathbb{B}^\ell \) 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 \( \ell \) is used. For this we refer to the technical specification in Chapter 8.

#### 4.3.1. Hash Values of Integers and Strings

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

#### 4.3.2. Hash Values of Multiple Inputs

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

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

We distinguish the special case of \( k = 1 \) to avoid computing \( h(h(B_1)) \) for a single input and to be able to use \( h(B_1, \ldots, B_k) \) as a consistent alternative notation for \( h(b) \).
This definition can be generalized to multiple input values of various types. Let \((v_1, \ldots, v_k)\) be such a tuple of general input values, where \(v_i\) is either a byte array, an integer, a string, or another tuple of general input values. As above, we define the hash value recursively as

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

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

\[
H \leftarrow \text{RecHash}_L(v_1, \ldots, v_k),
\]

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

![Algorithm: RecHash\(_L\)(v\(_1\), ..., v\(_k\))](image)

Algorithm 4.9: Computes the hash value \(h(v_1, \ldots, v_k) \in B^L\) of multiple inputs \(v_1, \ldots, v_k\) in a recursive manner.
5. Cryptographic Primitives

5.1. ElGamal Encryption

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

5.1.1. Using a Single Key Pair

An ElGamal key pair is a tuple \((sk, pk) \leftarrow \text{KeyGen}()\), 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

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

denotes the ElGamal encryption of \(m\) with randomization \(r \in_R \mathbb{Z}_q\). Note that the bit length of an encryption \(e \leftarrow \text{Enc}_{pk}(m, r)\) 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

\[
m \leftarrow \text{Dec}_{sk}(e) = a \cdot b^{-sk}.
\]

For any given key pair \((sk, pk) \leftarrow \text{KeyGen}()\), it is easy to show that \(\text{Dec}_{sk}(\text{Enc}_{pk}(m, r)) = m\) 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:

\[
\text{Enc}_{pk}(m_1, r_1) \cdot \text{Enc}_{pk}(m_2, r_2) = \text{Enc}_{pk}(m_1 m_2, r_1 + r_2).
\]

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

\[
\text{ReEnc}_{pk}(e, r') = e \cdot \text{Enc}_{pk}(1, r') = \text{Enc}_{pk}(m, r + r'),
\]

is clearly an encryption of \(m\) with a fresh randomization \(r + r'\).
5.1.2. Using a Shared Key Pair

If multiple parties generate ElGamal key pairs as described above, let’s say \((sk_j, pk_j) \leftarrow KeyGen()\) 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 \(e = \text{enc}_{pk}(m, r)\) can only be decrypted if all parties collaborate. This idea can be generalized such that only a threshold number \(t \leq s\) of parties is required to decrypt a message, but this property is not needed in this document.

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

\[
\text{Dec}_{sk}(e) = a \cdot b^{-sk} = a \cdot b^{-\sum_{j=1}^{s} sk_j} = a \cdot (\prod_{j=1}^{s} b^{x_j})^{-1} = a \cdot (\prod_{j=1}^{s} b_j)^{-1},
\]

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

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 5.4.2.

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 \(\mathbb{G}_q \subset \mathbb{Z}_p^\times\) of integers modulo \(p = 2q + 1\) as in the ElGamal encryption scheme. Let \(g, h_1, \ldots, h_n \in \mathbb{G}_q\) be independent generators of \(\mathbb{G}_q\), which means that their relative logarithms are provably not known to anyone. For a deterministic algorithm that generates an arbitrary number of independent generators, we refer to the NIST standard FIPS PUB 186-4 [2, Appendix A.2.3]. Note that the deterministic nature of this algorithm enables the verification of the generators by the public.

The Pedersen commitment scheme consists of two deterministic algorithms, one for computing a commitment \(\text{Com}(m, r) = g^r h_1^{m_1} \cdots h_n^{m_n} \in \mathbb{G}_q\) to \(n\) messages \(m = (m_1, \ldots, m_n) \in \mathbb{Z}_q^n\) with randomization \(r \in \mathbb{R}_q\), and one for checking the validity of \(c \leftarrow \text{Com}(m, r)\) when \(m\) and \(r\) are revealed. In the special case of a single message \(m\), we write \(\text{Com}(m, r) = g^r h^m\) 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 $B_\psi = (b_{ij})_{n \times n}$ be the permutation matrix of $\psi$, 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 $B_\psi$ has exactly one 1-bit. If $b_j = (b_{1,j}, \ldots, b_{n,j})$ denotes the $j$-th column of $B_\psi$, then

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

is a commitment to $b_j$ with randomization $r_j$. By computing such commitments to all columns,

$$\text{Com}(\psi, r) = (\text{Com}(b_1, r_1), \ldots, \text{Com}(b_n, r_n)),$$

we obtain a commitment to $\psi$ with randomizations $r = (r_1, \ldots, r_n)$. Note that the size of such a permutation commitment $c \leftarrow \text{Com}(\psi, r)$ 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 $\text{OT}_n^k$, the sender holds a list $m = (M_1, \ldots, M_n)$ of messages $M_i \in \mathbb{B}^\ell$ (bit strings of length $\ell$), of which $k \leq n$ 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 $s = (s_1, \ldots, s_k)$ for the $k$ selections $s_j \in \{1, \ldots, n\}$ of the receiver and $m_s = (M_{s_1}, \ldots, M_{s_k})$ for the $k$ messages to transfer.

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

An $\text{OT}_n^k$-scheme is called secure, if the three algorithms guarantee both receiver privacy and sender privacy. Receiver privacy is defined in terms of indistinguishable selections $s_1$ and $s_2$ 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, $s$ and $m$ are sent to an incorruptible trusted third party, which forwards $m_s$ to the simulator. In the literature, there is a subtle but important distinction between sender privacy and weak sender privacy [38]. 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 $OT_n^2$-scheme, but such general constructions are usually not very efficient. In this document, we use the second $OT_n^k$-scheme presented in [17]. We instantiate the protocol to the same $q$-order subgroup $G_q \subset \mathbb{Z}_p^*$ 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 $g \in G_q \backslash \{1\}$, an encoding $\Gamma : \{1, \ldots, n\} \rightarrow G_q$ of the possible selections into $G_q$, and a collision-resistant hash function $h : \mathbb{B}^* \rightarrow \mathbb{B}^\ell$ with output length $\ell$. In Prot. 5.1, we provide a detailed formal description of the protocol. The query is a vector $a \in G_q^k$ of length $k$ and the response is a tuple $(b, c, d) \in G^k$ consisting of a vector $b \in G^k$ of length $k$, a vector $c \in \mathbb{B}^\ell$ of length $n$, and a single value $d \in G_q$, i.e.,

$$a \leftarrow \text{Query}(s, r),$$

$$(b, c, d) \leftarrow \text{Reply}(a, m, z),$$

$$m_s \leftarrow \text{Open}(b, c, d, s, r),$$

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

Executing $\text{Query}$ and $\text{Open}$ requires $k$ fixed-base exponentiations in $G_q$ each, whereas $\text{Reply}$ requires $n + k + 1$ fixed-exponent exponentiations in $G_q$. 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 [14].

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 $t \leq k$ invalid queries $a_j \notin \{\Gamma(i) \cdot g^r : 1 \leq i \leq n, r \in \mathbb{Z}_q\}$, 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 $M_{a_j} = C_{a_j} \oplus h((b_j \cdot d^{-r_j})^{-z})$, 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.\(^2\)

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

\(^2\)The existence of such attacks against the protocol presented in an earlier version of this document have been discovered by Tomasz Truderung [50, Appendix B].
<table>
<thead>
<tr>
<th>Receiver</th>
<th>Sender</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows $s = (s_1, \ldots, s_k)$</td>
<td>knows $m = (M_1, \ldots, M_n)$</td>
</tr>
<tr>
<td>for $j = 1, \ldots, k$</td>
<td>pick random $z \in_R \mathbb{Z}_q$ for $j = 1, \ldots, k$</td>
</tr>
<tr>
<td>- pick random $r_j \in_R \mathbb{Z}_q$</td>
<td>- compute $b_j = a_j^z$ for $i = 1, \ldots, n$</td>
</tr>
<tr>
<td>- compute $a_j = \Gamma(s_j) \cdot g^{r_j}$</td>
<td>- compute $k_i = \Gamma(i)^z$</td>
</tr>
<tr>
<td>$a = (a_1, \ldots, a_k)$</td>
<td>- compute $C_i = M_i \oplus h(k_i)$</td>
</tr>
<tr>
<td></td>
<td>compute $d = g^z$</td>
</tr>
<tr>
<td>$b = (b_1, \ldots, b_k)$, $c = (C_1, \ldots, C_n), d$</td>
<td>for $j = 1, \ldots, k$</td>
</tr>
<tr>
<td>- compute $k_j = b_j \cdot d^{-r_j}$</td>
<td>- compute $M_{s_j} = C_{s_j} \oplus h(k_j)$</td>
</tr>
<tr>
<td>- compute $M_{s_j} = C_{s_j} \oplus h(k_j)$</td>
<td></td>
</tr>
</tbody>
</table>

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

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 $C \in (\mathbb{B}^\ell)^{nk}$ of size $nk$ instead of a vector $c \in (\mathbb{B}^\ell)^n$ 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:

$$a \leftarrow \text{Query}(s, r),$$

$$(b, C, d) \leftarrow \text{Reply}(a, m, z_1, z_2, \beta_1, \ldots, \beta_k),$$

$$m_s \leftarrow \text{Open}(b, C, d, s, r).$$

Another important difference of the extended scheme is the shape of the queries $a_j = (\Gamma(s_j) \cdot g_1^{r_j}, g_2^{r_j})$, which correspond to ElGamal encryptions for a public key $g_1 = g_2^x$. As a consequence, receiver privacy depends now on the decisional Diffie-Hellman assumption, i.e., it is no longer unconditional. However, the close connection between OT queries and ElGamal encryptions is a key property that we use for submitting ballots (see Section 6.4.2).
The performance of the extended scheme is slightly inferior compared to the basic scheme. On the receiver’s side, executing \textit{Query} requires $2k$ fixed-base exponentiations in $G_q$ (which can all be precomputed), and \textit{Open} requires $k$ fixed-base exponentiations in $G_q$. On the sender’s side, \textit{Reply} requires $n+2k+2$ fixed-exponent exponentiations in $G_q$ (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 the resulting matrix $C$, the overall asymptotic running time for the sender is $O(nk)$.

<table>
<thead>
<tr>
<th>Receiver</th>
<th>Sender</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows $s = (s_1, \ldots, s_k)$</td>
<td>knows $m = (M_1, \ldots, M_n)$</td>
</tr>
</tbody>
</table>

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 $a_j = (a_{j,1}, a_{j,2})$

\[ a = (a_1, \ldots, a_k) \]

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

\[ b = (b_1, \ldots, b_k), \quad C = (C_{ij})_{n \times k}, d \]

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 $\text{OT}^k_n$-scheme with sender privacy receiver, where $g_1, g_2 \in G_q \setminus \{1\}$ are independent generators of $G_q \subset \mathbb{Z}_p^*$, $\Gamma : \{1, \ldots, n\} \rightarrow G_q$ an encoding of the selections into $G_q$, and $h : \mathbb{B}^* \rightarrow \mathbb{B}^\ell$ a collision-resistant hash function with output length $\ell$.  

36
5.3.3. Simultaneous Oblivious Transfers

The OT\textsuperscript{$n$\textsubscript{1}}\textsuperscript{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 \( k_l \leq n_l \) 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 OT\textsuperscript{$n$\textsubscript{1}}\textsuperscript{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 \( \mathbf{s} = (s_1, \ldots, s_k) \).

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 \( 1 \leq l \leq t+1 \). This determines for each \( j \in \{1, \ldots, k\} \) a unique index \( l \in \{1, \ldots, t\} \) satisfying \( k'_l < j \leq k'_{l+1} \), which we can use to define a constraint
\[
n'_l < s_j \leq n'_{l+1}
\]
for every selection \( s_j \) in \( \mathbf{s} \). This guarantees that the first \( k_1 \) messages are selected from \( \mathbf{m}_1 \), the next \( k_2 \) messages from \( \mathbf{m}_2 \), and so on.

Starting from Prot. 5.2, the sender’s algorithm \texttt{Reply} can be generalized in a natural way by introducing an additional outer loop over \( 1 \leq l \leq t \) 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 \texttt{Query} and \texttt{Open} are not affected by this change. It is easy to demonstrate that this generalization of the OT\textsuperscript{$n$\textsubscript{1}}\textsuperscript{k}-scheme of the previous subsection is equivalent to performing \( t \) individual oblivious transfers in parallel. Note that the total number of exponentiations in \( \mathbb{G}_q \) remains the same for all three algorithms.

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

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

In this particular case, the matrix contains \( 2 \cdot 3 + 3 \cdot 4 + 1 \cdot 2 = 20 \) regular entries \( C_{ij} \) and 34 empty entries.
Receiver | Sender
---|---
knows \( s = (s_1, \ldots, s_k) \) | knows \( m = (M_1, \ldots, M_n) \)

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 \( a_j = (a_{j,1}, a_{j,2}) \)

\[ \mathbf{a} = (a_1, \ldots, a_k) \]

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 \( i = n'_1 + 1, \ldots, n'_t + n_l \)
  - compute \( k_i = \Gamma(i)^{z_1} \)
- for \( j = k'_1 + 1, \ldots, k'_t + k_l \)
  - compute \( k_{ij} = k_i \beta_j \)
  - compute \( C_{ij} = M_i \oplus h(k_{ij}) \)

compute \( d = g_1^{z_1} g_2^{z_2} \)

\[ \mathbf{b} = (b_1, \ldots, b_k), \]
\[ \mathbf{C} = (C_{ij})_{n \times k}, d \]

for \( j = 1, \ldots, k \)
- compute \( k_j = b_j \cdot d^{-r_j} \)
- compute \( M_{s_j} = C_{s,j} \oplus h(k_j) \)

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

### 5.3.4. Oblivious Transfer of Long Messages

If the output length \( \ell \) 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 \( \ell \), we can split each message \( M_i \) into \( r \) blocks \( M_{ij} \in \mathbb{B}^\ell \) of length \( \ell \) and process them individually using hash values \( h(k_i, j) \). Here, the index \( j \in \{1, \ldots, k\} \) plays the role of the counter. This is identical to
applying a single concatenated hash value \( h(\ell, 1) \| \cdots \| h(\ell, k) \) of length \( \ell_m \) to \( M_i \). If \( \ell_m \) is not an exact multiple of \( \ell \), we do the same for \( r = [\ell_m/\ell] \) block, but then truncate the first \( \ell_m \) bits from the resulting concatenated hash value \( \ell_m \) to obtain the desired length.

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

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

\[
\text{CheckProof}_\phi(\text{GenProof}_\phi(x, y), y) = 1
\]

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

### 5.4.1. Composition of Preimage Proofs

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

\[
NIZKP[(x_1, x_2) : y_1 = \phi_1(x_1) \land y_2 = \phi_2(x_2)] = NIZKP[(x_1, x_2) : (y_1, y_2) = \phi(x_1, x_2)].
\]

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

\[
NIZKP[(x) : y_1 = \phi_1(x) \land y_2 = \phi_2(x)] = NIZKP[(x) : (y_1, y_2) = \phi(x)],
\]

shows that \( y_1 \) and \( y_2 \) have an equal preimage. In the special case of two exponential functions \( \phi_1(x) = g^x \) and \( \phi_2(x) = h^x \), this demonstrates the equality of discrete logarithms \([16]\).
5.4.2. Applications of Preimage Proofs

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

**Schnorr Identification.** In a Schnorr identification scheme, the holder of a private credential $x \in X$ proves knowledge of $x = \phi^{-1}(y) = \log_g y$, where $g$ is a generator in a suitable group $Y$ in which the DL assumption holds [47]. This leads to one of the simplest and most fundamental instantiation of the above preimage proof,

$$NIZKP[(x) : y = g^x],$$

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

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

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

Generating the proof requires two and verifying the proof four exponentiations in $G_q$.

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

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

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

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

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

5.5. Wikström’s Shuffle Proof

A cryptographic shuffle of a list $e = (e_1, \ldots, e_N)$ of ElGamal encryptions $e_i \leftarrow \text{Enc}_{pk}(m_i, r_i)$ is another list of ElGamal encryptions $\tilde{e} = (\tilde{e}_1, \ldots, \tilde{e}_N)$, which contains the same plaintexts $m_1, \ldots, m_N$ in permuted order. Such a shuffle can be generated by selecting a random permutation $\psi : \{1, \ldots, N\} \rightarrow \{1, \ldots, N\}$ from the set $\Psi_N$ of all such permutations (e.g.,
using Knuth's shuffle algorithm [34]) and by computing re-encryptions \( \tilde{e}_i \leftarrow \text{ReEnc}_{pk}(e_j, \tilde{r}_j) \) for \( j = \psi(i) \). We write

\[
\tilde{e} \leftarrow \text{Shuffle}_{pk}(e, \tilde{r}, \psi)
\]

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

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

\[
\text{NIZKP}[ (\psi, \tilde{r}) : \tilde{e} = \text{Shuffle}_{pk}(e, \tilde{r}, \psi) ].
\]

Unfortunately, since \( \text{Shuffle}_{pk} \) 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 \( c \leftarrow \text{Com}(\psi, r) \) 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 \( \tilde{e} \) from \( e \). 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 \( c \leftarrow \text{Com}(\psi, r) \) 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 \( B_\psi = (b_{ij})_{N \times N} \) is an \( N \)-by-\( N \) matrix over \( \mathbb{Z}_q \) and \( (x_1, \ldots, x_N) \) a vector of \( N \) independent variables, then \( B_\psi \) is a permutation matrix if and only if \( \sum_{j=1}^{N} b_{ij} = 1 \), for all \( i \in \{1, \ldots, N\} \), and \( \prod_{i=1}^{N} \sum_{j=1}^{N} b_{ij} x_i = \prod_{i=1}^{N} x_i \). The first condition means that the elements of each row of \( B_\psi \) must sum up to one, while the second condition requires that \( B_\psi \) has exactly one non-zero element in each row.

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

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

\[
\prod_{j=1}^{N} c_j = \prod_{j=1}^{N} g^{r_j} \prod_{i=1}^{N} h_i^{b_{ij}} = g^{\sum_{j=1}^{N} r_j} \prod_{i=1}^{N} h_i^{\sum_{j=1}^{N} b_{ij}} = g^{\bar{r}} \prod_{i=1}^{N} h_i = \text{Com}(1, \bar{r}). \quad (5.2)
\]
Similarly, for arbitrary values \( \mathbf{u} = (u_1, \ldots, u_N) \in \mathbb{Z}_q^N \), \( \mathbf{\tilde{u}} = (\tilde{u}_1, \ldots, \tilde{u}_N) \in \mathbb{Z}_q^N \), with
\[
\tilde{u}_i = \sum_{j=1}^{N} b_{ij} u_j = u_j \text{ for } j = \psi(i), \text{ and } r = \sum_{j=1}^{N} r_j u_j,\]
the second condition leads to two equalities:
\[
\prod_{i=1}^{N} \tilde{u}_i = \prod_{j=1}^{N} u_j, \tag{5.3}
\]
\[
\prod_{j=1}^{N} c_j^{u_j} = \prod_{j=1}^{N} (g^{\tilde{u}_j})^{u_j} = g^{\sum_{j=1}^{N} r_j u_j} \prod_{i=1}^{N} h_i^{b_{ij} u_j} = g^r \prod_{i=1}^{N} h_i^\tilde{u}_i. \tag{5.4}
\]

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 \( u_i \leftarrow \text{Hash}((\mathbf{e}, \mathbf{e}, \mathbf{c}), i) \).

The second fundamental idea of Wikström’s proof is based on the homomorphic property of the ElGamal encryption scheme and the following observation for values \( \mathbf{u} \) and \( \mathbf{\tilde{u}} \) defined in the same way as above:
\[
\prod_{i=1}^{N} (\tilde{e}_i)^{\tilde{u}_i} = \prod_{j=1}^{N} \text{ReEnc}_{\mathbf{pk}}(e_j, \tilde{r}_j)^{u_j} = \prod_{j=1}^{N} \text{ReEnc}_{\mathbf{pk}}(e_j^{u_j}, \tilde{r}_j u_j) = \text{ReEnc}_{\mathbf{pk}}(\prod_{j=1}^{N} e_j^{u_j}, \sum_{j=1}^{N} \tilde{r}_j u_j) = \text{Enc}_{\mathbf{pk}}(1, \tilde{r}) \cdot \prod_{j=1}^{N} e_j^{u_j}, \tag{5.5}
\]
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:

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

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 list \( \mathbf{c} = \{\tilde{c}_1, \ldots, \tilde{c}_N\} \) of Pedersen commitments with different generators. For \( c_0 = h \) and random values \( \tilde{r} = (\tilde{r}_1, \ldots, \tilde{r}_N) \in \mathbb{Z}_q^N \), we define \( \tilde{c}_i = g^{r_i \tilde{e}_i^{u_i}} \), which leads to \( \tilde{c}_N = \text{Com}(u, \tilde{r}) \) for \( u = \prod_{i=1}^{N} u_i \) and
\[
\hat{r} = \sum_{i=1}^{N} \tilde{r}_i \prod_{j=i+1}^{N} \tilde{u}_j.
\]

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

\[
\begin{align*}
NIZKP &= \left[ (\bar{r}, \bar{r}, r, \bar{r}, \bar{r}, \bar{u}) : \\
\prod_{j=1}^{N} c_j &= \text{Com}(1, \bar{r}) \\
\land \hat{c}_N &= \text{Com}(u, \bar{r}) \land \left[ \prod_{i=1}^{N} (\hat{c}_i = g^{\hat{r}_i}\hat{c}_{i-1}^{\hat{r}}) \right] \\
\land \prod_{j=1}^{N} \hat{c}_j &= \text{Com}(\bar{u}, r) \\
\land \prod_{i=1}^{N} (\hat{c}_i)\hat{u}_i &= \text{Enc}_{pk}(1, \bar{r}) \cdot \prod_{j=1}^{N} e_j^{u_j}
\right].
\end{align*}
\]

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

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

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

### 5.5.2. Preimage Proof

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

\[
\phi(x_1, x_2, x_3, x_4, \tilde{x}, \tilde{x}) = (g^{\tilde{x}_1}, g^{\tilde{x}_2}, \text{Com}(\tilde{x}, x_3), \text{ReEnc}_{pk}(\prod_{i=1}^{N} (\hat{c}_i)^{\tilde{x}_1}, -x_4, (g^{\hat{x}_1}\hat{c}_0^{\tilde{x}_1}, \ldots, g^{\hat{x}_N}\hat{c}_N^{\tilde{x}_N})),
\]

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

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

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

\[(\bar{c}, \hat{c}, \bar{c}, \bar{c}, \hat{c}) = (\prod_{j=1}^{N} c_j / h_j, \hat{c}_N, \prod_{j=1}^{N} c_j u_j, \prod_{j=1}^{N} e_j^{u_j}, (c_1, \ldots, c_N)),\]

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

\[
NIZKP \begin{cases}
\bar{c} = g^\phi \land \hat{e} = \text{ReEnc}_\text{pk}(\prod_{i=1}^{N} (\hat{e}_i)^{\bar{u}_i} \cdot -\hat{r}) \\
\land \hat{e} = \text{ReEnc}_\text{pk}(\prod_{i=1}^{N} (\hat{e}_i)^{\bar{u}_i})
\end{cases}
\]

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

This preimage proof, together with the two lists of commitments \( c \) and \( \bar{e} \), leads to the desired non-interactive shuffle proof \( \text{NIZKP}[(\psi, \bar{r}) : \hat{e} = \text{Shuffle}_\text{pk}(e, \bar{r}, \psi)] \). We denote the generation and verification of such a proof \( \tilde{\pi} = (t, s, c, \bar{e}) \) by

\[
\tilde{\pi} \leftarrow \text{GenProof}_\text{pk}(e, \bar{e}, \bar{r}, \psi) \\
b \leftarrow \text{CheckProof}_\text{pk}(\tilde{\pi}, e, \bar{e}).
\]

respectively. Corresponding algorithms are depicted in Alg. 7.51 and Alg. 7.55. Note that generating the proof requires \( 7N + 4 \) and verifying the proof \( 9N + 11 \) modular exponentiations in \( \mathbb{G}_q \). The proof itself consists of \( 5N + 9 \) elements \( (2N + 4 \) elements from \( \mathbb{Z}_q \) and \( 3N + 5 \) elements from \( \mathbb{G}_q \)).

### 5.6. Schnorr Signatures

The **Schnorr signature scheme** consists of a triple \((\text{KeyGen}, \text{Sign}, \text{Verify})\) of algorithms, which operate on a cyclic group for which the DL problem is believed to be hard [47]. 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 \( k \geq 2 \)) 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 \( g \in \mathbb{G}_q \}, \) and a cryptographic hash function \( h : \mathbb{B}^* \rightarrow \mathbb{B}^\ell \). Note that the output length \( \ell \) of the hash function depends on the scheme’s security parameter.

A key pair in the Schnorr signature scheme is a tuple \((sk, pk) \leftarrow \text{KeyGen}()\), where \( sk \in \mathbb{Z}_q \) 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

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

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

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

44
For any given key pair \((sk, pk) \leftarrow \text{KeyGen}()\), it is easy to show that \(\text{Verify}_{pk}(\text{Sign}_{sk}(m, r), m) = 1\) holds for all \(m \in \mathbb{B}^*\) and \(r \in \mathbb{Z}_q\). Note that a Schnorr signature is very similar to a non-interactive zero-knowledge proof \(\text{NIZKP}[(sk) : pk = g^{sk}]\), in which \(m\) is passed as an additional input to the Fiat-Shamir hash function (a few other subtle differences are due to different traditions of describing Schnorr signatures and non-interactive zero-knowledge proofs in the literature).

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

### 5.7. Hybrid Encryption and Key-Encapsulation

For large messages \(m \in \mathbb{B}^*\), public-key encryption schemes such as ElGamal are often not efficient enough. This is the motivation for constructing hybrid encryption schemes, which combine the advantages of (asymmetric) public-key encryption schemes with the advantages of (symmetric) secret-key encryption schemes. The idea is to use a key-encapsulation mechanism (KEM) to generate and encapsulate an ephemeral secret key \(k \in \mathbb{B}^\ell\), which is used to encrypt \(m\) symmetrically. For a key pair \((sk, pk) \leftarrow \text{KeyGen}\()\), the result of a hybrid encryption is a ciphertext \((c, c') \leftarrow \text{Enc}_{pk}(m)\), which consists of the encapsulated key \(c\) obtained from \((c, k) \leftarrow \text{Encaps}_{pk}()\) and the symmetric ciphertext \(c' \leftarrow \text{Enc}_k'(m)\). The decryption \(m \leftarrow \text{Decaps}_{sk}(c, c')\) works in the opposite manner, i.e., first the symmetric key \(k \leftarrow \text{Decaps}_{sk}(c)\) is reconstructed from \(c\) and then the plaintext message \(m \leftarrow \text{Dec}_{k'}(c')\) is decrypted from \(c'\) using \(k\). Note that a triple of algorithms \((\text{KeyGen, Enc, Dec})\) constructed in this way from a key-encapsulation mechanism \((\text{Encaps, Decaps})\) and a secret-key encryption scheme \((\text{Enc}', \text{Dec}')\) 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 [33].

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

Relative to the above KEM algorithms \(\text{Encaps}\) and \(\text{Decaps}\), a proof for CPA-security can be based either on DDH (standard model) or CDH (random oracle model) [33]. However, by combining this KEM with a practical block cipher such as AES and an appropriate mode
of operation (and possibly a suitable padding algorithm), provable security is replaced by practical security, i.e., it is assumed that the practical block cipher is a good approximation of an ideal block cipher [19]. Nevertheless, given the significant efficiency benefits, instantiations based on current standards such as AES are commonly accepted and widely used in practice.
Part III.

Protocol Specification
6. Protocol Description

The goal of this chapter is to describe the cryptographic voting protocol from various perspectives. We introduce the involved parties, describe their roles, and define the communication channels over which they exchange messages during a protocol execution. The protocol itself has various phases—each with multiple sub-phases—which we describe with sufficient technical details for understanding the general protocol design and the most important computational details. A comprehensive list of security and election parameters is introduced beforehand. We also model the adversary and give a list of underlying trust assumptions. Finally, we discuss the security properties that we obtain from applying the adversary model and trust assumptions to the protocol. For further details in form of low-level pseudo-code algorithms, we refer to Chapter 7. The protocol itself is an extension of the protocol introduced in [28].

6.1. Parties and Communication Channels

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

- The election administrator is responsible for setting up an election event. This includes tasks such as defining the electoral roll, the number of elections, the set of candidates in each election, and the eligibility of each voter in each election (see Section 6.3.2). At the end of the election process, the election administrator determines and publishes the final election result.

- A group of election authorities guarantees the integrity and privacy of the votes submitted during the election period. They are numbered with indices \( j \in \{1, \ldots, s\} \), \( s \geq 1 \). Before every election event, they establish jointly a public ElGamal encryption key \( pk \). They also generate the credentials and codes to be printed on the voting cards. During vote casting, they respond to the submitted ballots and confirmations. At the end of the election period, they perform a cryptographic shuffle of the encrypted votes. Finally, they use their private key shares \( sk_j \) to decrypt the votes in a distributed manner.

- The printing authority is responsible for printing the voting cards and delivering them to the voters. They receive the data necessary for generating the voting cards from the bulletin board and the election authorities.
• The voters are the actual human users of the system. They are numbered with indices \( i \in \{1, \ldots, N_E\} \), \( N_E \geq 0 \). Prior to an election event, they receive the voting card from the printing authority, which they can use to cast and confirm a vote during the election period using their voting client.

• The voting client is a machine used by some voter to conduct the vote casting and confirmation process. Typically, this machine is either a desktop, notebook, or tablet computer with a network connection and enough computational power to perform cryptographic computations. The strict separation between voter and voting client is an important precondition for the protocol’s security concept.

• The bulletin board is the central communication unit of the system. It implements a broadcast channel with memory among the parties involved in the protocol [31]. For this, it keeps track of all the messages received during the protocol execution. The messages from the election administrator and the election authorities are kept in separate dedicated sections, which implies that bulletin board can authenticate them unambiguously. The entire election data stored by the bulletin board defines the input of the verification process.

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

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

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

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

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

For preparing and conducting an election event, as well as for computing the final election result, we assume that at least one honest election authority is following the protocol faithfully. In other words, we take into account that dishonest election authorities may collude with the adversary (willingly or unwillingly), but not all of them in the same election event. Trust assumptions like this are common in cryptographic voting protocols, but they may be difficult to implement in practice. A difficult practical problem is to guarantee that the authorities act independently, which implies, for example, that they use software written by independent developers and run them on hardware from independent manufacturers. This document does not specify conditions for the election authorities to reach a satisfactory degree of independence.
There are two very strong trust assumptions in our protocol. The first one is attributed to the voting client, which is assumed not to be corrupted by an adversary trying to attack vote privacy. Since the voting client learns the plaintext vote from the voter during the vote casting process, it is obvious that vote privacy can not be guaranteed in the presence of a corrupted device, for instance one that is infiltrated with malware. This is one of the most important unsolved problems in any approach, in which voter’s are allowed to prepare and submit their votes on their own (insecure) devices.

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

6.3. System Parameters

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

6.3.1. Security Parameters

The security of the system is determined by four principal security parameters. As the resistance of the system against attackers of all kind depends strongly on the actual choice of these parameters, they need to be selected with much care. Note that they impose strict lower bounds for all other security parameters.
The minimal privacy $\sigma$ 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 $\sigma$ bits. Recommended values today are $\sigma = 112$, $\sigma = 128$, or higher.

The minimal integrity $\tau$ defines the amount of computational work for breaking the integrity of a vote in the same way as $\sigma$ for breaking the privacy of the vote. In other words, the actual choice of $\tau$ determines the risk that an adversary succeeds in manipulating an election. Recommendations for $\tau$ are similar to the above-mentioned values for $\sigma$, 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 $0 < \epsilon \leq 1$ 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 $\epsilon$, 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 $s \geq 1$ determines the amount of trust that needs to be attributed to each of them. This is a consequence of our assumption that at least one election authority is honest, i.e., in the extreme case of $s = 1$, full trust is attributed to a single authority. Generally, increasing the number of authorities means to decrease the chance that they are all malicious. On the other hand, finding a large number of independent and trustworthy authorities is a difficult problem in practice. There is no general rule, but $3 \leq s \leq 5$ 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 $\sigma$, $\tau$, and $\epsilon$. A summary of all parameters and constraints to consider when selecting them will be given in Table 6.1 at the end of this subsection.

a) Hash Algorithm Parameters

At multiple places in our voting protocol, we require a collision-resistant hash functions $h : \mathbb{B}^* \rightarrow \mathbb{B}^\ell$ for various purposes. In principle, we could work with different output lengths $\ell$, 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 $\text{Hash}_L(B)$ throughout the entire document. Its output length $L = 8\ell$ must therefore be adjusted to both $\sigma$ and $\tau$. 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., $\ell \geq 2 \cdot \max(\sigma, \tau)$ bits (resp. $L \geq \frac{\max(\sigma, \tau)}{4}$ bytes) in our particular case.

b) Group and Field Parameters

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

- The encryption group $G_q \subset \mathbb{Z}_p$ is a $q$-order subgroup of the multiplicative group of integers modulo a safe prime $p = 2q + 1 \in \mathbb{S}$. Since $G_q$ 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 $\sigma$ only. The following constraints are consistent with the NIST recommendations:

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

In addition to $p$ and $q$, two independent generators $g, h \in \mathbb{G}_q \setminus \{1\}$ 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 $k \geq 2$ 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 $\tau$ only. The constraints for the bit length of $\hat{p}$ are therefore identical to the constraints for the bit length of $p$,

$$\|\hat{p}\| \geq \begin{cases} 1024, & \text{for } \tau = 80, \\ 2048, & \text{for } \tau = 112, \\ 3072, & \text{for } \tau = 128, \\ 7680, & \text{for } \tau = 192, \\ 15360, & \text{for } \tau = 256. \end{cases} \quad (6.2)$$

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 $\tau$ must be doubled. This implies that $\|\hat{q}\| \geq 2\tau$ is the constraint to consider when choosing $\hat{q}$. Finally, an arbitrary generator $\hat{g} \in \mathbb{G}_{\hat{q}} \setminus \{1\}$ 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 that the constraint for $\mathbb{G}_{\hat{q}}$ also apply to $\mathbb{Z}_{p'}$, i.e., we must consider $\|p'\| \geq 2\tau$ 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 $L_M = 2 \cdot \|\hat{q}\| \text{ bytes.}$
c) Parameters for Voting and Confirmation Codes

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

\[
\ell_X = \left\lceil \frac{2\|\hat{q}_x\|}{\log_2 |A_X|} \right\rceil, \quad \ell_Y = \left\lceil \frac{2\|\hat{q}_y\|}{\log_2 |A_Y|} \right\rceil.
\]

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

d) Parameters for Verification and Finalization Codes

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

\[
8L_R \geq \log \frac{1}{1 - \epsilon}, \quad 8L_F \geq \log \frac{1}{1 - \epsilon}.
\]

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

---

\(^1\)See http://world.std.com/~reinhold/diceware.html.
independent of the election parameters (see next subsection). We can solve this problem by defining $n_{\text{max}}$ to be the maximal number of candidates in every possible election event and to extend the constraint for $L_R$ into

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

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

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

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

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$</td>
<td>Output length of hash function (bytes)</td>
</tr>
<tr>
<td>$p$</td>
<td>Modulo of encryption group $\mathbb{G}_q$</td>
</tr>
<tr>
<td>$g, h$</td>
<td>Independent generators of $\mathbb{G}_q$</td>
</tr>
<tr>
<td>$\hat{p}$</td>
<td>Modulo of identification group $\mathbb{G}_{\hat{q}}$</td>
</tr>
<tr>
<td>$\hat{q}$</td>
<td>Prime order of $\mathbb{G}<em>{\hat{q}}$, modulo of prime field $\mathbb{Z}</em>{\hat{q}}$</td>
</tr>
<tr>
<td>$\hat{g}$</td>
<td>Generator of $\mathbb{G}_{\hat{q}}$</td>
</tr>
<tr>
<td>$L_M$</td>
<td>Length of OT messages (bytes)</td>
</tr>
<tr>
<td>$\hat{q}_x$</td>
<td>Upper bound of private voting credential $x$</td>
</tr>
<tr>
<td>$A_X$</td>
<td>Voting code alphabet</td>
</tr>
<tr>
<td>$\ell_X$</td>
<td>Length of voting codes (characters)</td>
</tr>
<tr>
<td>$\hat{q}_y$</td>
<td>Upper bound of private confirmation credential $y$</td>
</tr>
<tr>
<td>$A_Y$</td>
<td>Confirmation code alphabet</td>
</tr>
<tr>
<td>$\ell_Y$</td>
<td>Length of confirmation codes (characters)</td>
</tr>
<tr>
<td>$n_{\text{max}}$</td>
<td>Maximal number of candidates</td>
</tr>
<tr>
<td>$L_R$</td>
<td>Length of verification codes $R_{ij}$ (bytes)</td>
</tr>
<tr>
<td>$A_R$</td>
<td>Verification code alphabet</td>
</tr>
<tr>
<td>$\ell_R$</td>
<td>Length of verification codes $RC_{ij}$ (characters)</td>
</tr>
<tr>
<td>$L_F$</td>
<td>Length of finalization codes $F_i$ (bytes)</td>
</tr>
<tr>
<td>$A_F$</td>
<td>Finalization code alphabet</td>
</tr>
<tr>
<td>$\ell_F$</td>
<td>Length of finalization codes $FC_i$ (characters)</td>
</tr>
</tbody>
</table>

Table 6.1.: List of security parameters derived from the principal security parameters $\sigma$, $\tau$, and $\epsilon$. We assume that these values are fixed and publicly known to every party participating in the protocol.
6.3.2. Election Parameters

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

In Chapter 2, we already discussed that our definition of an election event, which constitutes of multiple simultaneous $k$-out-of-$n$ elections over multiple counting circles, covers all election use cases in the given context. The most important parameters of an election event are therefore the number $t$ of simultaneous elections and the number $w$ of counting circles. By assuming $t \geq 1$ and $w \geq 1$, 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 $U \in \mathbb{A}_{ucs}^*$. While the protocol is not designed to run multiple election events in parallel, it is important to strictly separate the election data of successive election events. By introducing a unique election event identifier and by adding it to every digital signature issued during the protocol execution (see Section 6.6), the data of a given election event is unanimously tied together. This is the main purpose of the election event identifier. To avoid that the data of multiple elections is inadvertently tied together when the same identifier $U$ is used multiple times, we assume $U$ to contain enough information (e.g., the date of the election day) to allow participating parties to judge whether $U$ is a fresh value or not.

a) Electorate

With $N_E \geq 0$ we denote the number of eligible voters in an election event and use $i \in \{1, \ldots, N_E\}$ as identifier. For each voter $i$, a voter description $V_i \in \mathbb{A}_{ucs}^*$ and a counting circle $w_i \in \{1, \ldots, w\}$ must be specified. 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, voter $i$ is not automatically eligible in every election of an election event. We use single bits $e_{ij} \in \mathbb{B}$ to define whether voter $i$ is eligible in election $j$ or not, and we exclude completely ineligible voters by imposing the constraint $\sum_{j=1}^{t} e_{ij} > 0$. The matrix $E = (e_{ij})_{N_E \times t}$ of all such values is called eligibility matrix.

b) Elections and Candidates

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

---

\[^{2}\text{Related election parameters will be formed during vote casting and confirmation. The number of submitted ballots will be denoted by } N_B \leq N_E, \text{ the number of confirmed ballots by } N_C \leq N_B, \text{ and the number of valid votes by } N \leq N_C.\]
of candidates in an election event. For each such candidate \( i \in \{1, \ldots, n\} \), a candidate description \( C_i \in A_{ucs}^* \) must be provided. Again, we do not further specify the type and format of the information given for each candidate. Among the \( n_j \) candidates of election event \( j \in \{1, \ldots, t\} \), we furthermore assume that at least \( k_j \) of them serve as default candidates (see Section 2.2.2 for further explanations).\(^3\) Default candidates are specified by a Boolean vector \( u = \{u_1, \ldots, u_n\} \in \mathbb{B}^n \), where \( u_i = 1 \) means that candidate \( i \) serves as default candidate. If \( I_j = \{\sum_{i=1}^{j-1} n_i + 1, \ldots, \sum_{i=1}^{j} n_i\} \) denotes the set of indices of all candidates of election \( j \), then \( \sum_{i \in I_j} \geq k_j \) must hold for all \( j \in \{1, \ldots, t\} \).

Other important parameters of an election event are the numbers of candidates \( k_j, 0 < k_j < n_j \), which voters can 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.\(^4\) The total number of selections over all elections, \( k = \sum_{j=1}^{t} k_j \), is limited by a constraint that follows from our particular vote encoding method (see Section 6.4.1). For a given eligibility matrix, we can relax this constraint by replacing it with a constraint for \( k' = \max_{i=1}^{N_E} k'_i \), where \( k'_i = \sum_{j=1}^{t} e_{ij} k_j \) denotes the number of allowed selections of voter \( i \). In the following, \( k' \leq k \) will be called maximal ballot size. Note that \( k' \) can be considerably smaller than \( k \), especially in the case of a sparse eligibility matrix.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>( U )</td>
<td>Unique election event identifier</td>
</tr>
<tr>
<td>( t )</td>
<td>Number of elections</td>
</tr>
<tr>
<td>( N_E )</td>
<td>Number of eligible voters</td>
</tr>
<tr>
<td>( v = (V_1, \ldots, V_{N_E}) )</td>
<td>Voter descriptions</td>
</tr>
<tr>
<td>( w )</td>
<td>Number of counting circles</td>
</tr>
<tr>
<td>( w = (w_1, \ldots, w_{N_E}) )</td>
<td>Assigned counting circles</td>
</tr>
<tr>
<td>( E = (e_{ij})_{N_E \times t} )</td>
<td>Eligibility matrix</td>
</tr>
<tr>
<td>( n )</td>
<td>Total number of candidates</td>
</tr>
<tr>
<td>( n = (n_1, \ldots, n_t) )</td>
<td>Number of candidates</td>
</tr>
<tr>
<td>( c = (C_1, \ldots, C_n) )</td>
<td>Candidate descriptions</td>
</tr>
<tr>
<td>( k )</td>
<td>Total number of selections</td>
</tr>
<tr>
<td>( k = (k_1, \ldots, k_t) )</td>
<td>Number of selections</td>
</tr>
<tr>
<td>( k' = \max_{i=1}^{N_E} \sum_{j=1}^{t} e_{ij} k_j )</td>
<td>Maximum ballot size</td>
</tr>
<tr>
<td>( u = (u_1, \ldots, u_n) )</td>
<td>Default candidates</td>
</tr>
</tbody>
</table>

Table 6.2.: List of election parameters and constraints.

---

\(^3\)The number of default candidates of election \( j \) could also be greater than \( k_j \). In such cases, the extension of ballots from voters with restricted eligibility is not unique. This is not a problem as long as the extension is conducted in a deterministic way. In the extreme case of \( u = (1, \ldots, 1) \), the selection of default candidates is entirely unrestricted and the specification of \( u \) as part of the election parameters could be omitted.

\(^4\)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.
6.4. Technical Preliminaries

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

6.4.1. Encoding of Votes and Counting Circles

In an election that allows votes for multiple candidates, it is usually more efficient to incorporate all votes into a single encryption. In the case of the ElGamal encryption scheme with $\mathbb{G}_q$ as message space, we must define an invertible mapping $\Gamma$ from the set of all possible votes into $\mathbb{G}_q$. A common technique for encoding a selection $s = (s_1, \ldots, s_k)$ of $k$ candidates out of $n$ candidates, $1 \leq s_j \leq n$, 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 $\Gamma(s) = \prod_{j=1}^{k} \Gamma(s_j)$. Inverting $\Gamma(s)$ by factorization is unique as long as $\Gamma(s) < p$ and efficient when $n$ is small [26]. For optimal capacity, we choose the $n$ smallest prime numbers $p_1, \ldots, p_n \in \mathbb{P} \cap \mathbb{G}_q$, $p_i < p_{i+1}$, and define $\Gamma(s_j) = p_{s_j}$ for $j \in \{1, \ldots, k\}$.

Since each encrypted votes is attributed to a counting circle, we extend the above invertible mapping $\Gamma : \{1, \ldots, n\}^k \rightarrow \mathbb{G}_q$ 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 $s$ and a counting circle $w_i \in \{1, \ldots, w\}$ can then be encoded together as $\Gamma'(s, w_i) = p_{n+w_i} \cdot \Gamma(s)$. 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 $w_i$ to an encoded vote can also be conducted under encryption: let $(a, b) = \text{Enc}_{pk}(\Gamma(s), r)$ be an ElGamal encryption of $\Gamma(s)$, then $(p_{n+w_i} \cdot a, b) = \text{Enc}_{pk}(p_{n+w_i}, 0) \cdot \text{Enc}_{pk}(\Gamma(s), r) = \text{Enc}_{pk}(\Gamma'(s, w_i), r)$ is an ElGamal encryption of $\Gamma'(s, w_i)$. We will use this property in the protocol to assign in a verifiable manner the counting circles to the encrypted votes before processing them through the mix-net.

6.4.2. Linking OT Queries to ElGamal Encryptions

If the same encoding $\Gamma : \{1, \ldots, n\} \rightarrow \mathbb{G}_q$ is used for the OT$_k^n$-scheme (see Section 5.3.3) and for encoding plaintext votes, we obtain a natural link between an OT query $a = (a_1, \ldots, a_k)$ and an ElGamal encryption $(a, b) \leftarrow \text{Enc}_{pk}(\Gamma(s), r)$. The link arises by substituting the first generator $g_1$ in the OT-scheme with the public encryption key $pk = g^{sk} \mod p$ 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 $a = \prod_{j=1}^{k} a_j = (\Gamma(s) \cdot pk^r, g^r)$ 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 [28]. It means that submitting $a$ as part of the ballot solves two problems at the same time: sending an OT query and an encrypted vote to the election authorities and guaranteeing that they contain exactly the same selection of candidates.
6.4.3. Validity of Encrypted Votes

The main purpose of the verification codes in our protocol is to provide evidence to the voters that their votes have been cast and recorded as intended. However, our way of constructing the verification codes solves another important problem, namely to guarantee that every submitted encrypted vote satisfies exactly the constraints given by the election parameters \( k, n \), and \( E \), i.e., that every encryption contains a valid vote. Let \( RC_1, \ldots, RC_n \in A^\ell_R \) be the verification codes for the \( n = \sum_{j=1}^n n_j \) candidates of a given voting card. In our scheme, they are constructed as follows [28]:

- For every voter \( i \in \{1, \ldots, N_E\} \) and \( k'_i = \sum_{j=1}^{n_i} e_{ij} k_j \), each authority picks a random polynomial \( A_i(X) \in R \mathbb{Z}_q[X] \) of degree \( k'_i - 1 \). From this polynomial, the authority selects \( n \) random points \( p_{ij} = (x_{ij}, A_i(x_{ij})) \) by picking \( n \) distinct random values \( x_{ij} \in R \mathbb{Z}_q \setminus \{0\} \). The result is a vector of points, \( p_i = (p_{i1}, \ldots, p_{in}) \), of length \( n \). Over all \( N_E \) voting cards, each authority generates a matrix \( (p_{ij})_{N_E \times n} \) of such points. Computing this matrix is part of the election preparation of every election authority. In the remaining of this document, the matrix generated by authority \( j \) will be denoted by \( P_j \).

- During vote casting, every authority transfers exactly \( k'_i \) points from \( P_j \) obliviously to the voting client of voter \( i \), i.e., the voting client receives a matrix \( P_s = (p_{ij})_{s \times k'_i} \) of such points, which depends on the voter’s selection \( s \). The verification code \( RC_{s_j} \) for the selected candidate \( s_j \) is derived from the points \( p_{1j}, \ldots, p_{sj} \) by truncating corresponding hash values \( h(p_{ij}) \) to the desired length \( L_R \), combining them with an exclusive-or into a single value, and finally converting this value into a string \( RC_{s_j} \) of length \( \ell_R \). The same happens simultaneously for all of the voter’s \( k'_i \) selections, which leads to a vector \( R\mathbf{c}_s = (RC_{s_1}, \ldots, RC_{s_{n_s}}) \). During the printing of the voting card, exactly the same calculations are performed for the verification codes of all \( n \) candidates.

- By obtaining \( k'_i \) points from a particular election authority, the voting client can reconstruct the polynomial \( A_i(X) \) of degree \( k'_i - 1 \), if at least \( k'_i \) distinct points from \( A_i(X) \) are available (see Section 3.2.2). If this is the case, the simultaneous \( \text{OT}_{n}^{k'_i} \) query must have been formed properly under the constraints given by \( n, k, \) and \( E \). The voting client can therefore prove the validity of the encrypted vote by proving knowledge of this polynomial. For this, it evaluates the polynomial for \( X = 0 \) to obtain a vote validity credential \( y'_i = A_i(0) \), which can not be guessed efficiently without knowing the polynomial. In this way, the voting client obtains a vote validity credential \( y'_{ij} \) from every authority \( j \). Their integer sum \( y'_{ij} = \sum_{j=1}^s y'_{ij} \) is added to the private confirmation credential \( y_i \) derived from the confirmation code \( Y_i \). Knowing correct values \( y'_{ij} \) is therefore a prerequisite for the voting client to successfully confirm the vote (see following subsection).

The finalization code \( FC \in A^\ell_F \) of a given voting card is also derived from the random points generated by each authority. The procedure is similar to the generation of the verification codes. First, election authority \( j \) computes the hash value of the voter’s \( n \) points in \( P_j \) and truncates it to the desired length \( L_F \). The resulting \( s \) hash values—one from every authority—are combined with an exclusive-or into a single value, which is then converted
6.4.4. Voter Identification

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

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

\[
\hat{x} = \prod_{j=1}^s \hat{x}_j \mod \hat{p} = \prod_{j=1}^s \hat{g}^{x_j} \mod \hat{p} = \hat{g}^{\sum_{j=1}^s x_j} \mod \hat{p} = \hat{g}^x \mod \hat{p}.
\]

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

The generation of the confirmation code \( Y \) is very similar. It is a string representation of the private confirmation credential \( y \in \mathbb{Z}_q \), which is generated by the election authorities in exactly the same way as \( x \). However, for the corresponding public value \( \hat{y} \in \mathbb{G}_q \), the method is slightly different. After picking \( y_j \in R \mathbb{Z}_q \) at random, the authority first computes \( y_j^* = y_j + y'_j \mod q \) by adding \( y'_j = A_j(0) \) from the previous subsection to \( y_j \), and then publishes \( \hat{y}_j = \hat{g}^{y_j^*} \mod \hat{p} \) on the bulletin board. This leads to

\[
\hat{y} = \prod_{j=1}^s \hat{y}_j \mod \hat{p} = \prod_{j=1}^s \hat{g}^{y_j^*} \mod \hat{p} = \hat{g}^{\sum_{j=1}^s y_j^*} \mod \hat{p} = \hat{g}^{y^*} \mod \hat{p},
\]

for

\[
y^* = \sum_{j=1}^s y_j^* \mod q = (\sum_{j=1}^s y_j + \sum_{j=1}^s y'_j) \mod q = y + y' \mod q.
\]

As a consequence, performing a Schnorr identification relative to \( \hat{y} \) requires knowledge of \( y^* = y + y' \mod q \). Note that the corresponding zero-knowledge proof, \( NIZKP[(y^*) : \hat{y} = \hat{g}^{y^*} \mod \hat{p}] \), is a string representation of a secret value \( y^* \).
\( y^* \mod \hat{p} \), offers the same level of security more efficiently than in the case of proving knowledge of \( y \) and \( y^* \) separately. In the remaining of this document, we will call \( y^* \) and \( \hat{y} \) private and public vote approval credentials, respectively.

### 6.4.5. Election Outcome

After successfully mixing and decrypting the votes, the election outcome can be derived from the public data on the bulletin board. We propose to perform this task in two steps. The first step is performed by the election administrator as a very last protocol step. The result of this step, which we call raw election outcome, is written back to the bulletin board for improved convenience (while creating some redundancy). It consists of three matrices \( U = (u_{cj})_{w \times N} \), \( V = (v_{ij})_{N \times n} \), and \( W = (w_{ic})_{N \times w} \), where \( N \) denotes the number of submitted valid votes. For \( i \in \{1, \ldots, N\} \), \( j \in \{1, \ldots, n\} \), and \( c \in \{1, \ldots, w\} \), the meaning of the values contained in these matrices is as follows:

- \( u_{cj} \in \{0, \ldots, N\} \) denotes the number of default votes added for candidate \( j \) in counting circle \( c \);
- \( v_{ij} \in \mathbb{B} \) is set to 1, if plaintext vote \( i \) contains a vote for candidate \( j \), 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 matrices, consider the example from Section 2.2.2 with parameters \( t = 9 \), \( n = 18 \), \( w = 2 \), and \( N_E = 8 \). Among the candidates of all nine 1-out-of-2 elections, let

\[
\mathbf{u} = (1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0)
\]

be the Boolean vector specifying the default candidates (always the first of the two candidates in each election). Furthermore, suppose that \( N = 7 \) voters (all except Voter 5 from the first counting circle) 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. Therefore, we get

\[
\mathbf{U} = \begin{pmatrix}
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]

After mixing the five votes from voters from the first counting circle with the three votes from voters from the second counting circle, the resulting raw election outcome matrices \( V \) and \( W \) could look as follows:
The raw election outcome can be used to sum up the votes for all candidates \( j \in \{1, \ldots, n\} \). If necessary, local results can be computed for each counting circle \( c \in \{1, \ldots, w\} \), and the turnout of each counting circle can be determined:

\[
V_{cj} = \sum_{i=1}^{N} v_{ij}w_{ic} - u_{cj} = \text{total number of votes for candidate } j \text{ in counting circle } c,
\]

\[
V_j = \sum_{c=1}^{w} V_{cj} = \sum_{i=1}^{N} v_{ij} - \sum_{c=1}^{w} u_{cj} = \text{total number of votes for candidate } j,
\]

\[
W_c = \sum_{i=1}^{N} w_{ic} = \text{total number of submitted votes in counting circle } c.
\]

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 \( W_2 = 3 \).

<table>
<thead>
<tr>
<th>Election Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Candidate Index</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Counting Circle 1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Counting Circle 2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

### 6.5. Protocol Description

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

In each of the following subsections, we provide comprehensive illustrations of corresponding protocol sub-phases. The illustrations are numbered from Prot.6.3 to Prot.6.9. Each illustration depicts the involved parties, the necessary information known to each party prior to executing the protocol sub-phase, the computations performed by each party during the protocol sub-phase, and the exchanged messages. Together, these illustration define a precise and complete skeleton of the entire protocol. The details of the algorithms called by the parties when performing their computations are given in Chapter 7. Note that the illustrations do not show the signatures that are generated by the election administrator and the election authorities. These signatures are important to provide authenticity, i.e., they must be generated whenever a message is sent to the bulletin board and verified whenever a message is retrieved from there. As already discussed in Section 6.3.2, a unique election
event identifier $U$ is included in every signature. The distribution of $U$ is included in the protocol illustrations, but other details of the signature generation are discussed in Section 6.6. Corresponding algorithms are given in Section 7.6.

### 6.5.1. Pre-Election Phase

The pre-election phase of the protocol involves all necessary tasks to setup an election event. The main goal is to equip each eligible voter with a personalized voting card, which we identify with an index $i \in \{1, \ldots, N_E\}$. Without loss of generality, we assume that voting card $i$ is sent to voter $i$. We understand a voting card as a string $S_i \in A_{\text{vcs}}^*$, which is printed on paper by the printing authority. This string contains the voter index $i$, the voter description $V_i \in A_{\text{vcs}}^*$, the counting circle $w_i \in \{1, \ldots, w_1\}$, the voting code $X_i \in A_{\ell X}^*$, the confirmation code $Y_i \in A_{\ell Y}^*$, the finalization code $FC_i \in A_{\ell F}^*$, and the candidate descriptions $C_j \in A_{\text{vcs}}^*$ with corresponding verification codes $RC_{ij} \in A_{R}^{\ell R}$ for each candidate $j \in \{1, \ldots, n\}$. The information printed on voting card $i$ is therefore a tuple

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

**a) Election Preparation**

The information for printing the voting cards is generated by the $s$ election authorities in a distributed manner (see Sections 6.4.2 and 6.4.3 for technical background). For this, each election authority $j$ calls an algorithm $\text{GenElectorateData}(n, k, E)$ with the election parameters $n$, $k$, and $E$ as input, which have been published beforehand by the election administrator. The result obtained from calling this algorithm consists of the voting card data $d_j$ to be sent to the printing authority, the shares of the private and public credentials $x_j$, $y_j$, $\hat{x}_j$, and $\hat{y}_j$, and the matrix of random points $P_j$. Further details of the algorithm
are given in Alg. 7.6. For proving knowledge of the shares of the private credentials, a non-
interactive proof $\hat{\pi}_j$ is generated and published on the bulletin board along with $\hat{x}_j$ and $\hat{y}_j$. These first steps are depicted in the upper part of Prot. 6.1.

At the end of the above process, every election authority knows all the shares $\hat{X} = (\hat{x}_1, \ldots, \hat{x}_s)$ and $\hat{Y} = (\hat{y}_1, \ldots, \hat{y}_s)$ of the public credentials of the whole electorate. If the attached cryptographic proofs $\hat{\pi} = (\hat{\pi}_1, \ldots, \hat{\pi}_s)$ are all valid, the authorities call $\text{GetPublicCredentials}(X, Y)$ to obtain the two lists $\hat{x}$ and $\hat{y}$ of aggregated public credentials. They are used to identify the voters during the vote casting and vote confirmation phases (see Section 6.4.4 and Alg. 7.15 for further details).

<table>
<thead>
<tr>
<th>Election Administrator</th>
<th>Bulletin Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows $U, v, w, c, n, k, u, E$</td>
<td></td>
</tr>
<tr>
<td>$U, v, w, c, n, k, u, E$</td>
<td></td>
</tr>
<tr>
<td>$U, n, k, E$</td>
<td></td>
</tr>
<tr>
<td>$(d_j, x_j, y_j, \hat{x}_j, \hat{y}_j, \hat{\pi}_j) \leftarrow \text{GenElectorateData}(n, k, E)$</td>
<td></td>
</tr>
<tr>
<td>$\hat{\pi}_j \leftarrow \text{GenCredentialProof}(x_j, y_j, \hat{x}_j, \hat{y}_j)$</td>
<td></td>
</tr>
<tr>
<td>$\hat{x}_j, \hat{y}_j, \hat{\pi}_j$</td>
<td></td>
</tr>
<tr>
<td>$\hat{X} \leftarrow (\hat{x}_1, \ldots, \hat{x}_s)$</td>
<td></td>
</tr>
<tr>
<td>$\hat{Y} \leftarrow (\hat{y}_1, \ldots, \hat{y}_s)$</td>
<td></td>
</tr>
<tr>
<td>$\hat{\pi} \leftarrow (\hat{\pi}_1, \ldots, \hat{\pi}_s)$</td>
<td></td>
</tr>
<tr>
<td>$X, Y, \hat{\pi}$</td>
<td></td>
</tr>
<tr>
<td>if $\neg \text{CheckCredentialProofs}($X, Y, j$)$ then abort</td>
<td></td>
</tr>
<tr>
<td>$(\hat{x}, \hat{y}) \leftarrow \text{GetPublicCredentials}(X, Y)$</td>
<td></td>
</tr>
</tbody>
</table>


b) Printing of Code Sheets

The voting card data $d_j$ generated by authority $j$ contains for every voting card the authority’s shares of both the private voting and confirmation credentials and the verification and finalization codes. This information is very sensitive and can only be shared with the printing authority. The process of sending $d_j$ to the printing authority is depicted in Prot. 6.2. Recall that this channel is confidential, i.e., it must be secured by cryptographic means. This can be achieved by sending $d_j$ in encrypted form using the key-encapsulation mechanism in combination with a symmetric encryption scheme as described in Section 5.7. Let $c_j \leftarrow \text{GenCiphertext}(pk_{\text{Print}}, U, d_j)$ denote the encryption of $U$ and $d_j$ using the printing authority’s public encryption key $pk_{\text{Print}}$. The printing authority can then decrypt $c_j$ back into $(U', d_j)$ by calling $\text{GetPlaintext}(sk_{\text{Print}}, c_j)$. Including the election event identifier $U$ in this process is a necessary measure to avoid the chosen-ciphertext attack mentioned in
For this reason, the printing process needs to be aborted if the decrypted identifier \( U' \) is different from \( U \). Note that the integrity of the ciphertext \( c_j \) is ensured by other means (see Section 6.6).

The actual voting cards can be generated from the collected voting card data \( \mathbf{D} \leftarrow (d_1, \ldots, d_s) \) and the elections parameters \( \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E} \). The printing authority uses them as inputs for the algorithm \texttt{GetVotingCards}(\( \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{D} \)), which produces corresponding strings \( \mathbf{s} = (S_1, \ldots, S_{N_E}) \) by calling Alg. 7.16. Printouts of these strings are sent to the voters, for example using a trusted postal service.

<table>
<thead>
<tr>
<th>Bulletin Board</th>
<th>Printing Authority</th>
<th>Election Authority ( j \in {1, \ldots, s} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>knows ( U, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E} )</td>
<td>knows ( \mathbf{s}_{\text{Print}} )</td>
<td>knows ( \mathbf{pk}_{\text{Print}}, U, d_j )</td>
</tr>
<tr>
<td>( U, \mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E} )</td>
<td></td>
<td>( c_j \leftarrow \text{GenCiphertext}<em>{\mathbf{pk}}(\mathbf{pk}</em>{\text{Print}}, U, d_j) )</td>
</tr>
<tr>
<td>( (U', d_j) \leftarrow \text{GetPlaintext}<em>{\mathbf{s}</em>{\text{Print}}}(c_j) )</td>
<td>if ( U \neq U' ) abort</td>
<td></td>
</tr>
<tr>
<td>( \mathbf{D} \leftarrow (d_1, \ldots, d_s) )</td>
<td>( s \leftarrow \text{GetVotingCards}(\mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{n}, \mathbf{k}, \mathbf{E}, \mathbf{D}) )</td>
<td></td>
</tr>
<tr>
<td>( \text{Voter } i \in {1, \ldots, N_E} )</td>
<td>( S_i )</td>
<td></td>
</tr>
</tbody>
</table>


c) Key Generation

In the last step of the election preparation, a public ElGamal encryption key \( \mathbf{pk} \in \mathbb{G}_q \) is generated jointly by the election authorities. As shown in Prot.6.3, this is a simple process between the election authorities and the bulletin board. At the end of the protocol, \( \mathbf{pk} \) is known to every authority, and each of them holds a share \( \mathbf{sk}_j \in \mathbb{Z}_q \) of the corresponding private key. It involves calls to two algorithms \texttt{GenKeyPair()} and \texttt{GenPublicKeyProof}(\( \mathbf{sk}_j, \mathbf{pk}_j \)) for generating the key shares and corresponding cryptographic proofs. If the keys and proofs of all other authorities are valid, \( \mathbf{pk} \) results from calling \texttt{GetPublicKey(pk)}. For details of these algorithms, we refer to Section 5.1.2 and Algs. 7.18 to 7.20 and 7.22.

6.5.2. Election Phase

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

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

### a) Candidate Selection

The first step for the voter is the selection of the candidates. In an election event with \( t \) simultaneous elections, voter \( v \) must select exactly \( e_v k_j \) candidates for each election \( j \in \{1, \ldots, t\} \) and \( k_v' = \sum_{j=1}^{t} e_v k_j \) candidates in total. These values can be derived from the election parameters \( k \) and \( E \), which the voting client retrieves from the bulletin board together with the candidate descriptions \( c \) and the number of candidates \( n \). This preparatory step is shown in the upper part of Prot.6.4. By calling \( \text{GetVotingPage}(v, v, w, c, n, k, E) \), the voting client then generates a voting page \( P_v \in A_{\text{res}}^* \), which represents the visual interface displayed to voter \( v \) for selecting the candidates (see Alg.7.23). The voter’s selection \( s = (s_1, \ldots, s_{k_v'}) \) is a vector of values \( s_j \) satisfying the constraint in (5.1) from Section 5.3.3. The voter enters these values together with the voting code \( X_v \) from the voting card.

### b) Vote Casting

Based on the voter’s selection \( s = (s_1, \ldots, s_{k_v'}) \), the voting client generates a ballot \( \alpha = (\hat{x}_v, a, \pi) \) by calling an algorithm \( \text{GenBallot}(X_v, s, pk) \) using the authorities’ common public key \( pk \). The ballot contains an OT query \( a = (a_1, \ldots, a_{k_v'}) \in \mathbb{G}_q^{k_v'} \) for corresponding return codes. By using the public encryption key \( pk \) in the oblivious transfer as a generator of the group \( \mathbb{G}_q \) (see Section 6.4.2), each query \( a_j \) is an ElGamal encryption of the voter’s
Protocol 6.4: Candidate Selection

selection $s_j$. The ballot $\alpha$ also contains the voter’s public credential $\hat{x}_v$, which is derived from the secret voting code $X_v$, and a non-interactive zero-knowledge proof

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

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

The ballot is submitted to the election authorities via the bulletin board. Each authority checks its validity by calling $\text{CheckBallot}(v, \alpha, pk, k, E, \hat{x}, B)$. This algorithm verifies that the size of $a$ is exactly $k'_v = \sum_{j=1}^{k'_v} e_{a_j} k_j$, that the public voting credential $\hat{x}_v$ is included in $\hat{x}$, that the zero-knowledge proof $\pi_\alpha$ is valid (which implies that the voter is in possession of a valid voting code $X_v$), and that the same voter has not submitted a valid ballot before.

To detect multiple ballots from the same voter, each authority keeps track of a list $B_j$ of valid ballots submitted so far. If one of the above checks fails, the ballot is rejected and the process aborts.

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

c) Vote Confirmation

The voting client displays the verification codes $rc_s = (RC_{s_1}, \ldots, RC_{s_{k'_v}})$ for the selected candidates to the voter for comparing them with the codes $rc_v$ printed on voter $v$’s voting
Protocol 6.5: Vote Casting

card. We describe this process by an algorithm call \texttt{CheckReturnCodes}(rc_v, rc_s, s), which is executed by the human voter. In case of a match, the voter enters the confirmation code $Y_v$, from which the voting client computes the confirmation $\gamma = (\hat{y}_v, \pi_\beta)$ consisting of the voter’s public vote approval credential $\hat{y}_v$ and a non-interactive zero-knowledge proof

$$\pi_\beta = \text{NIZKP}[(y_v^s : \hat{y}_v = g^{y_v^s} \mod \hat{p}]$$

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

After submitting $\gamma$ via the bulletin board to every authority, they check the validity of the zero-knowledge proof included. In the success case, they respond with their finalization $\delta_j = (F_{vj}, z_{vj})$. The voting client retrieves the finalization code $FC$ from the values $(F_{v,1}, \ldots, F_{v,s})$ included in $\delta = (\delta_1, \ldots, \delta_s)$ by calling \texttt{GetFinalizationCode}(\delta) and displays it to the voter for comparison. As above, we describe this process by an algorithm call \texttt{CheckFinalizationCode}(FC_v, FC) executed by the human voter. The whole process is depicted in Prot.6.6. Note that the randomizations $(z_{v,1}, \ldots, z_{v,s})$ included in $\delta$ are not needed for computing the finalization code. But their publication enables the verification of the OT responses by external verifiers [28].
Protocol 6.6: Vote Confirmation
6.5.3. Post-Election Phase

In the post-election phase, all \( N \leq N_E \) submitted and confirmed ballots are processed through a mixing and decryption process. The main actors are the election authorities, which perform the mixing in a serial and the decryption in a parallel process. For the decryption, they require their shares \( sk_j \) of the private encryption key, which the have generated during the pre-election phase. Before applying their key shares to the output of the mixing, they verify all previous steps by checking the validity of every ballot collected during the election phase and the correctness of the shuffle proofs. In addition to performing the decryption, they need to demonstrate its correctness with a non-interactive zero-knowledge proof. The very last step of the entire election process is the computation and announcement of the final election result by the election administrator.

a) Mixing

The mixing is a serial process, in which all election authorities are involved. Without loss of generality, we assume that the first mix is performed by the Authority 1, the second by Authority 2, and so on. The process is the same for everyone, except for the first authority, which needs to extract the list of encrypted votes from the submitted ballots and to extend the encryptions of voters with restricted voting rights with default candidates. Recall that during vote casting, each authority keeps track of all submitted ballots and confirmations. In case of Authority 1, corresponding lists are denoted by \( B_1 \) and \( C_1 \), respectively. By calling \( \text{GetEncryptions} \), the first authority retrieves the list \( \tilde{e}_0 \) of encrypted votes, and by calling \( \text{GenShuffle} \), this list is shuffled into \( \tilde{e}_1 \sim \text{Shuffle} (\tilde{e}_0, \tilde{r}_1, \psi_1) \), where \( \tilde{r}_1 \) denotes the re-encryption randomizations and \( \psi_1 \) the random permutation. These values are the secret inputs for a non-interactive proof

\[
\tilde{\pi}_1 = \text{NIZKP} \left[ \left( \psi_1, \tilde{r}_1 \right) : \tilde{e}_1 = \text{Shuffle} \left( \tilde{e}_0, \tilde{r}_1, \psi_1 \right) \right],
\]

which proves the correctness of the shuffle. This proof results from calling the algorithm \( \text{GenShuffleProof} \). The results from conducting the first shuffle—the shuffled list of encryptions \( \tilde{e}_1 \) and the zero-knowledge proof \( \tilde{\pi}_1 \)—are sent to the bulletin board, together with a matrix \( U \) that defines the number of default votes added for each candidate. These initial steps of the mixing phase are depicted in the upper part of Prot. 6.7.

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

b) Decryption

After the mixing, every authority retrieves the complete output of the mix-net—the shuffled lists of encryptions \( \tilde{E} \) and the shuffle proofs \( \tilde{\pi} \)—from the bulletin board. The input \( \tilde{e}_0 \) of the first shuffle is retrieved from the submitted ballots by calling \( \text{GetEncryptions} \). Before starting the decryption, \( \text{CheckShuffleProofs} (\tilde{\pi}, \tilde{e}_0, \tilde{E}, pk, j) \) is called to verify the
Election Authority

\[ j = 1 \]
knows \( U, pk, B_1, C_1, n, k \)

\[ u, w \]

\((\tilde{e}_0, U) \leftarrow \text{GetEncryptions}(B_1, C_1, k, n, u, w)\)
\((e_1, r_1, \psi_1) \leftarrow \text{GenShuffle}(\tilde{e}_0, pk)\)
\(\tilde{\pi}_1 \leftarrow \text{GenShuffleProof}(\tilde{e}_0, e_1, r_1, \psi_1, pk)\)

Election Authority

\[ j \in \{2, \ldots, s\} \]
knows \( U, pk \)

\[ \tilde{e}_{j-1} \]

\((\tilde{e}_j, r_j, \psi_j) \leftarrow \text{GenShuffle}(\tilde{e}_{j-1}, pk)\)
\(\tilde{\pi}_j \leftarrow \text{GenShuffleProof}(\tilde{e}_{j-1}, e_j, r_j, \psi_j, pk)\)

\[ \tilde{e}_j, \tilde{\pi}_j \]

\[ \tilde{e} \leftarrow (\tilde{e}_1, \ldots, \tilde{e}_s) \]
\[ \tilde{\pi} \leftarrow (\tilde{\pi}_1, \ldots, \tilde{\pi}_s) \]

Protocol 6.7: Mixing

correctness of all shuffles. For authority \( j \), this algorithm loops over all shuffle proofs \( \tilde{\pi}_i \), \( i \neq j \), and checks them individually. As shown in Prot. 6.8, the process aborts in case any of these check fails.

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

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

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

c) Tallying

To conclude an election, the election administrator retrieves the partial decryptions of every election authority from the bulletin board. The attached decryption proofs are checked by calling \( \text{CheckDecryptionProofs}(\pi', pk, \tilde{e}_s, B') \). The process aborts if one or more than
Protocol 6.8: Decryption

one check fails. Otherwise, by calling \texttt{GetDecryptions}(\texttt{e}_s, \texttt{B'}) the partial decryptions are assembled and the plaintexts are determined. Recall from Section 6.4.2 that every such plaintext is an encoding $\Gamma'(\texttt{s}, \texttt{w}_i) \in \mathbb{G}_q$ of some voter’s selection of candidates and the voter’s counting circle, and that the individual votes can be retrieved by factorizing this number. By calling \texttt{GetVotes}(\texttt{m}, \texttt{n}, \texttt{w}) this process is performed for all plaintexts. The whole tallying process is depicted in Prot. 6.9.

Protocol 6.9: Tallying

6.6. Channel Security

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

Table 6.4 gives an overview of all signatures generated during the protocol execution. For the reasons discussed earlier in Section 6.3.2, we include the election event identifier \(U\) as a message prefix in every signature. Generally, for generating a signature for multiple messages \(m = (m_1, \ldots, m_r)\), we call \(\text{GenSignature}(sk_X, U, m)\) using the party’s public key \(pk_X\). This algorithm implements Schnorr’s signature scheme as described in Section 5.6 and as implemented in Alg. 7.62. Note that according to Table 6.4, redundant signatures \(\sigma_{\text{param}}^1\), \(\sigma_{\text{param}}^2\), and \(\sigma_{\text{param}}^3\) are generated by the election administrator during the preparation phase. The reason for this redundancy is to provide tailor-made signatures for all involved parties, i.e., depending on the information they retrieve from the bulletin board during the protocol run.

A special case in the list of signatures shown in Table 6.4 is the entry for Prot.6.2, which describes the only signature not submitted to the bulletin board. Recall that the voting card data \(d_j\) 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. Instead of signing \(d_j\), the result \(c_j \leftarrow \text{GenCiphertext}_\varphi(pk_{\text{Print}}, U, d_j)\) of this hybrid encryption is signed and sent to the printing authority. \(pk_{\text{Print}}\) denotes the public encryption key of the printing authority. Again, we assume that a certificate for this key exists and is known to everyone.

<table>
<thead>
<tr>
<th>Issuer</th>
<th>Nr.</th>
<th>Protocol</th>
<th>Parameters</th>
<th>Signatures</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Election administrator</td>
<td>6.1</td>
<td>Election preparation</td>
<td>(v, w, c, n, k, E) (u, w)</td>
<td>(\sigma_{\text{param}}^1), (\sigma_{\text{param}}^2), (\sigma_{\text{param}}^3)</td>
<td>(E) (\sigma_{\text{tally}})</td>
</tr>
<tr>
<td>Election authority</td>
<td>6.9</td>
<td>Tallying</td>
<td>(V, W)</td>
<td>(\sigma_{\text{tally}})</td>
<td>(\sigma_{\text{tally}})</td>
</tr>
<tr>
<td>j (= 1, \ldots, s)</td>
<td>6.1</td>
<td>Election preparation</td>
<td>(\tilde{x}_j, \tilde{y}_j, \tilde{\pi}_j)</td>
<td>(\sigma_{\text{prep}}^j)</td>
<td>(\sigma_{\text{ prep}}^j)</td>
</tr>
<tr>
<td></td>
<td>6.2</td>
<td>Printing</td>
<td>(c_j)</td>
<td>(\sigma_{\text{print}}^j)</td>
<td>(\sigma_{\text{print}}^j)</td>
</tr>
<tr>
<td></td>
<td>6.3</td>
<td>Key generation</td>
<td>(pk_j, \pi_j)</td>
<td>(\sigma_j^{\text{kgen}})</td>
<td>(\sigma_j^{\text{kgen}})</td>
</tr>
<tr>
<td></td>
<td>6.5</td>
<td>Vote casting</td>
<td>(v, \beta_j)</td>
<td>(\sigma_{\text{ cast}}^i) (i \in {1, \ldots, N_B})</td>
<td>(\sigma_{\text{ cast}}^i) (i \in {1, \ldots, N_B})</td>
</tr>
<tr>
<td></td>
<td>6.6</td>
<td>Vote confirmation</td>
<td>(v, \delta_j)</td>
<td>(\sigma_{\text{conf}}^i) (i \in {1, \ldots, N_C})</td>
<td>(\sigma_{\text{conf}}^i) (i \in {1, \ldots, N_C})</td>
</tr>
<tr>
<td></td>
<td>6.7</td>
<td>Mixing</td>
<td>(\tilde{e}_j, \tilde{\pi}_j)</td>
<td>(\sigma_j^{\text{mix}})</td>
<td>(\sigma_j^{\text{mix}})</td>
</tr>
<tr>
<td></td>
<td>6.8</td>
<td>Decryption</td>
<td>(b_j', \pi_j')</td>
<td>(\sigma_j^{\text{dec}})</td>
<td>(\sigma_j^{\text{dec}})</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Verifier</th>
<th>Nr.</th>
<th>Protocol</th>
<th>Parameters</th>
<th>Signatures</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Election administrator</td>
<td>6.9</td>
<td>Tallying</td>
<td>(pk_j, \pi_j) (\tilde{e}_s, \tilde{\pi}_s) (b_j, \pi'_j)</td>
<td>(\sigma^\text{kgen}_j) (\sigma^\text{mix}_j) (\sigma^\text{dec}_j)</td>
<td>(j \in {1, \ldots, s})</td>
</tr>
<tr>
<td>Election authority</td>
<td>6.1</td>
<td>Election preparation</td>
<td>(n, k, E) (\tilde{x}_j, \tilde{y}_j, \tilde{\pi}_j)</td>
<td>(\sigma^\text{param}_2) (\sigma^\text{prep}_j)</td>
<td>(j \in {1, \ldots, s})</td>
</tr>
<tr>
<td>(j \in {1, \ldots, s})</td>
<td>6.3</td>
<td>Key generation</td>
<td>(pk_j, \pi_j) (\tilde{e}_j, \tilde{\pi}_j)</td>
<td>(\sigma^\text{kgen}_j) (\sigma^\text{param}_3) (\sigma^\text{mix}_j)</td>
<td>(j \in {1, \ldots, s})</td>
</tr>
<tr>
<td></td>
<td>6.8</td>
<td>Decryption</td>
<td>(u, w) (\tilde{e}_j, \tilde{\pi}_j)</td>
<td>(\sigma^\text{param}_3) (\sigma^\text{mix}_j)</td>
<td>(j \in {1, \ldots, s})</td>
</tr>
<tr>
<td>Voting client</td>
<td>6.4</td>
<td>Candidate selection</td>
<td>(v, w, c, n, k, E)</td>
<td>(\sigma^\text{param}_1)</td>
<td>(\sigma^\text{kgen}_j) (\sigma^\text{cast}_ij) (\sigma^\text{conf}_ij)</td>
</tr>
<tr>
<td></td>
<td>6.5</td>
<td>Vote casting</td>
<td>(pk_j, \pi_j) (v, \beta_j)</td>
<td>(\sigma^\text{kgen}_j) (\sigma^\text{cast}_ij) (\sigma^\text{conf}_ij)</td>
<td>(\sigma^\text{kgen}_j) (\sigma^\text{cast}_ij) (\sigma^\text{conf}_ij)</td>
</tr>
<tr>
<td></td>
<td>6.6</td>
<td>Vote confirmation</td>
<td>(v, \delta_j)</td>
<td>(\sigma^\text{conf}_ij)</td>
<td>(\sigma^\text{kgen}_j) (\sigma^\text{cast}_ij) (\sigma^\text{conf}_ij)</td>
</tr>
<tr>
<td>Printing authority</td>
<td>6.2</td>
<td>Printing</td>
<td>(v, w, c, n, k, E) (c_j)</td>
<td>(\sigma^\text{param}_1) (\sigma^\text{kgen}_j) (\sigma^\text{cast}_ij) (\sigma^\text{conf}_ij)</td>
<td>(\sigma^\text{kgen}_j) (\sigma^\text{cast}_ij) (\sigma^\text{conf}_ij)</td>
</tr>
</tbody>
</table>

Table 6.5.: Overview of the signatures verified during the election process.
7. Pseudo-Code Algorithms

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

7.1. Conventions and Assumptions

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

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

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

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

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Algorithm</th>
<th>Called by</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>GetPrimes(n)</td>
<td>Algs. 7.25, 7.31 and 7.61</td>
<td>6.5, 6.9</td>
</tr>
<tr>
<td>7.2</td>
<td>( \rightarrow ) IsMember(x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>GetGenerators(n)</td>
<td>Algs. 7.51 and 7.55</td>
<td>6.7, 6.8</td>
</tr>
<tr>
<td>7.4</td>
<td>GetNIZKPChallenge(y, t, ( \kappa ))</td>
<td>Algs. 7.19, 7.21, 7.27, 7.30, 7.38, 7.41, 7.51, 7.55, 7.57 and 7.59</td>
<td>6.3, 6.5, 6.6, 6.7, 6.8, 6.9</td>
</tr>
<tr>
<td>7.5</td>
<td>GetNIZKPChallenges(n, y, ( \kappa ))</td>
<td>Algs. 7.51 and 7.55</td>
<td>6.7, 6.8</td>
</tr>
</tbody>
</table>

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

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

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

\(^1\)See https://gmplib.org
Algorithm: GetPrimes(n)

Input: Number of primes \( n \geq 0 \)

\( x \leftarrow 1 \)

for \( i = 1, \ldots, n \) do

repeat
  if \( x \leq 2 \) then
    \( x \leftarrow x + 1 \)
  else
    \( x \leftarrow x + 2 \)
  if \( x \geq p \) then
    return \( K \) // \( n \) is incompatible with \( p \)
  until IsPrime(x) and IsMember(x) // see Alg. 7.2

\( p_i \leftarrow x \)

\( p \leftarrow (p_1, \ldots, p_n) \)

return \( p \) // \( p \in (\mathbb{G}_q \cap \mathbb{P})^n \)

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

Algorithm: IsMember(x)

Input: Number to test \( x \in \mathbb{N} \)

if \( 1 \leq x < p \) then

\( j \leftarrow \text{JacobiSymbol}(x, p) \) // \( j \in \{-1, 0, 1\} \)

if \( j = 1 \) then

\( \downarrow \text{return true} \)

return false

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

Input: Number of independent generators $n \geq 0$

for $i = 1, \ldots, n$ do
  $x \leftarrow 0$
  repeat
    $x \leftarrow x + 1$
    $h_i \leftarrow \text{ToInteger}(\text{RecHash}_L("chVote", "ggen", i, x)) \mod p$ // see Algs. 4.5 and 4.9
    $h_i \leftarrow h_i^2 \mod p$
  until $h_i \notin \{0, 1\}$ // these cases are very unlikely

$h \leftarrow (h_1, \ldots, h_n)$
return $h$ // $h \in (\mathbb{G}_q \setminus \{1\})^n$

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

Algorithm: GetNIZKPChallenge($y, t, \kappa$)

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

$c \leftarrow \text{ToInteger}(\text{RecHash}_L(y, t)) \mod 2^\kappa$ // see Algs. 4.5 and 4.9
return $c$ // $c \in \mathbb{Z}_{2^\kappa}$

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

Algorithm: GetNIZKPChallenges($n, y, \kappa$)

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

$H \leftarrow \text{RecHash}_L(y)$ // see Alg. 4.9
for $i = 1, \ldots, n$ do
  $I \leftarrow \text{RecHash}_L(i)$ // see Alg. 4.9
  $c_i \leftarrow \text{ToInteger}(\text{Hash}_L(H \parallel I)) \mod 2^\kappa$ // see Alg. 4.5
$c \leftarrow (c_1, \ldots, c_n)$
return $c$ // $c \in \mathbb{Z}_{2^\kappa}$

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

The main actors in the pre-election phase are the election authorities. For the given election definition consisting of values $n$, $k$, and $E$, each election authority generates a share of the electorate data by calling Alg. 7.6. This is the main algorithm of the election preparation, which invokes several sub-algorithms for more specific tasks. Table 7.2 gives an overview of all algorithms of the pre-election phase. The shares of the public credentials of every authority, which are exchanged using the bulletin board, are assembled by every election authorities using Alg. 7.15. 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. 7.16. The corresponding sub-task for creating a single voting card is delegated to Alg. 7.17, but the formatting details are not specified explicitly. Some other algorithms are required for generating shares of the encryption key pair and for assembling the shares of the public key. For a more detailed description of the pre-election phase, we refer to Section 6.5.1.

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Algorithm</th>
<th>Called by</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.6</td>
<td>GenElectorateData($n$, $k$, $E$)</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.7</td>
<td>$\leftarrow$ GenPoints($n$, $k$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.8</td>
<td>$\rightarrow$ GenPolynomial($d$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.9</td>
<td>$\rightarrow$ GetYValue($x$, $a$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.10</td>
<td>$\rightarrow$ GetCodes($p$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.11</td>
<td>$\leftarrow$ GenCredentials($y'$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.12</td>
<td>GenCredentialProof($x_j$, $y_j$, $\hat{x}_j$, $\hat{y}_j$)</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.13</td>
<td>CheckCredentialProofs((\hat{\pi}, \hat{X}, \hat{Y}, i))</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.14</td>
<td>$\leftarrow$ CheckCredentialProof((\hat{\pi}, \hat{x}, \hat{y}))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.15</td>
<td>GetPublicCredentials((\hat{X}, \hat{Y}))</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.16</td>
<td>GetVotingCards($v$, $w$, $c$, $n$, $k$, $E$, $D$)</td>
<td>Printing authority</td>
<td>6.2</td>
</tr>
<tr>
<td>7.17</td>
<td>$\leftarrow$ GetVotingCard($v$, $w$, $c$, $n$, $k$, $X$, $Y$, $FC$, $rc$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.18</td>
<td>GenKeyPair()</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.19</td>
<td>GenPublicKeyProof($sk$, $pk$)</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.20</td>
<td>CheckPublicKeyProofs($\pi$, $pk$)</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.21</td>
<td>$\leftarrow$ CheckPublicKeyProof($\pi$, $pk$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.22</td>
<td>GetPublicKey($pk$)</td>
<td>Election authority</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.2.: Overview of algorithms and sub-algorithms of the pre-election phase.
Algorithm: GenElectorateData(n, k, E)

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

\[
\sum_{j=1}^{t} n_j
\]

for \( i = 1, \ldots, N \) do

\[
k_i' = \sum_{j=1}^{t} e_{ij} k_j
\]

\[
(p_i, y_i') \leftarrow \text{GenPoints}(n, k_i')
\]

\[
(F_i, r_i) \leftarrow \text{GetCodes}(p_i)
\]

\[
(x_i, y_i^*, \tilde{x}_i, \tilde{y}_i) \leftarrow \text{GenCredentials}(y_i')
\]

\[
d_i \leftarrow (x_i, y_i, F_i, r_i)
\]

\[
d \leftarrow (d_1, \ldots, d_{N_E})
\]

\[
x \leftarrow (x_1, \ldots, x_{N_E}) \quad \text{and} \quad y \leftarrow (y_1^*, \ldots, y_{N_E}^*)
\]

\[
\hat{x} \leftarrow (\hat{x}_1, \ldots, \hat{x}_{N_E}) \quad \text{and} \quad \hat{y} \leftarrow (\hat{y}_1, \ldots, \hat{y}_{N_E})
\]

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

return \((d, x, y, \hat{x}, \hat{y}, \mathbf{P})\)

Algorithm 7.6: Generates the authority’s share of the voting card data and credentials for the whole electorate. For this, the algorithm computes for each voter \( i \) the permitted number \( k_i' = \sum_{j=1}^{t} e_{ij} k_j \) of selections of the current election event. Algs. 7.7, 7.10 and 7.11 are called to generate the voting card data and the credentials for each single voter. At the end, the responses of these calls are grouped into corresponding tuples \( d, x, y, \hat{x}, \hat{y}, \mathbf{P} \), and into a matrix \( \mathbf{P} = (p_{ij})_{N_E \times n} \) of random points \( p_{ij} = (x_{ij}, y_{ij}) \), of which \( k_i' \) points will be transferred obliviously to the voters during vote casting (see Prot. 6.5).

Algorithm: GenPoints(n, k)

Input: Number of candidates \( n \geq 2 \)
Number of selections \( 0 < k < n \)

\[
a \leftarrow \text{GenPolynomial}(k - 1)
\]

\[
X \leftarrow \{0\}
\]

for \( i = 1, \ldots, n \) do

\[
x \in R \mathbb{Z}_q \setminus X
\]

\[
x \leftarrow X \cup \{x\}
\]

\[
y \leftarrow \text{GetYValue}(x, a)
\]

\[
p_i \leftarrow (x, y)
\]

\[
y' \leftarrow \text{GetYValue}(0, a)
\]

\[
p \leftarrow (p_1, \ldots, p_n)
\]

return \((p, y')\)

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

Input: Degree $d \geq -1$

if $d = -1$ then
  a ← (0)
else
  for $i = 0, \ldots, d - 1$ do
    $a_i \in \mathbb{Z}_q$
    $a_d \in \mathbb{Z}_q \setminus \{0\}$
  a ← $(a_0, \ldots, a_d)$
return a  \hspace{1em} \text{// } a \in \mathbb{Z}_q^{d+1}$

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

Algorithm: GetYValue($x, a$)

Input: Value $x \in \mathbb{Z}_\hat{q}$

  Coefficients $a = (a_0, \ldots, a_d)$, $a_i \in \mathbb{Z}_\hat{q}$, $d \geq 0$

if $x = 0$ then
  y ← $a_0$
else
  for $i = d, \ldots, 0$ do
    y ← $a_i + x \cdot y \mod \hat{q}$
return y  \hspace{1em} \text{// } y \in \mathbb{Z}_\hat{q}$

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

Algorithm: GetCodes($p$)

Input: Points $p = (p_1, \ldots, p_n)$, $p_i \in \mathbb{Z}_q^2$

$F \leftarrow \text{Truncate(RecHash}_L(p), L_F)$  \hspace{1em} \text{// } \text{see Alg. 4.9}$

for $i = 1, \ldots, n$ do
  $R_i \leftarrow \text{Truncate(RecHash}_L(p_i), L_R)$  \hspace{1em} \text{// } \text{see Alg. 4.9}$
$r \leftarrow (R_1, \ldots, R_n)$
return $(F, r)$  \hspace{1em} \text{// } F \in \mathbb{B}^{L_F}, r \in (\mathbb{B}^{LR})^n$

Algorithm 7.10: Generates an authority’s shares of the secret verification and finalization codes for a single voter.
Algorithm: GenCredentials(y′)

Input: Vote validity credential y′ ∈ ℤ_q

\[ q'_x \leftarrow \frac{q_x}{s}, \quad x \in_R ℤ_{q'_x} \]
\[ q'_y \leftarrow \frac{q_y}{s}, \quad y \in_R ℤ_{q'_y} \]
\[ y^* \leftarrow y + y' \mod q \]
\[ \hat{x} \leftarrow \hat{y}^* \mod \hat{p} \]
\[ \hat{y} \leftarrow \hat{y}^y \mod \hat{p} \]

return \( (x, y, y^*, \hat{x}, \hat{y}) \)  // \( x, y, y^* \in ℤ_q, \hat{x}, \hat{y} \in \mathbb{G}_q \)

Algorithm 7.11: Generates an authority’s shares of the voting, confirmation, and vote approval credentials for a single voter.

Algorithm: GenCredentialProof(x, y, \hat{x}, \hat{y})

Input: Private credentials \( x = (x_1, \ldots, x_{N_E}) \), \( x_i \in ℤ_q \)
Private credentials \( y = (y_1, \ldots, y_{N_E}) \), \( y_i \in ℤ_q \)
Public credentials \( \hat{x} = (\hat{x}_1, \ldots, \hat{x}_{N_E}) \), \( \hat{x}_i \in \mathbb{G}_q \)
Public credentials \( \hat{y} = (\hat{y}_1, \ldots, \hat{y}_{N_E}) \), \( \hat{y}_i \in \mathbb{G}_q \)

for \( i = 1, \ldots, N_E \) do
\[ \omega_i \in_R ℤ_q, \quad v_i \in_R ℤ_{q'_y} \]
\[ t_i \leftarrow (\hat{y}^\omega_i \mod \hat{p}, \hat{y}^v_i \mod \hat{p}) \]
\[ t \leftarrow (t_1, \ldots, t_{N_E}) \]
\[ y \leftarrow (\hat{x}, \hat{y}) \]
\[ c \leftarrow \text{GetNIZKPChallenge}(y, t, \tau) \] // see Alg. 7.4

for \( i = 1, \ldots, N_E \) do
\[ s_i \leftarrow (\omega_i + c \cdot x_i \mod \hat{q}, v_i + c \cdot y_i \mod \hat{q}) \]
\[ s \leftarrow (s_1, \ldots, s_{N_E}) \]
\[ \hat{\pi} \leftarrow (t, s) \]

return \( \hat{\pi} \)  // \( \hat{\pi} \in (\mathbb{G}_q^2)^{N_E} \times (\mathbb{Z}_q^2)^{N_E} \)

Algorithm 7.12: Generates a proof of knowing all private credentials \( x \) and \( y \). For the proof verification, see Alg. 7.14.
Algorithm 7.13: Checks if the credential proofs \( \hat{\pi} \) generated by \( s \) authorities are correct.

**Algorithm: CheckCredentialProofs\( ( \hat{\pi}, \hat{X}, \hat{Y}, i) \)**

**Input:** Credential proofs \( \hat{\pi} = (\hat{\pi}_1, \ldots, \hat{\pi}_s) \), \( \hat{\pi}_j \in (\mathbb{G}_q^{2N_E}) \times (\mathbb{Z}_q^2)^{N_E} \)

- Public credentials \( \hat{X} = (\hat{x}_{ij})_{N_E \times s}, \hat{x}_{ij} \in \mathbb{G}_q \)
- Public credentials \( \hat{Y} = (\hat{y}_{ij})_{N_E \times s}, \hat{y}_{ij} \in \mathbb{G}_q \)
- Authority index \( i \in \{1, \ldots, s\} \)

for \( j = 1, \ldots, s \)
  if \( i \neq j \)
    \[ \hat{x}_j \leftarrow \text{Col}(\hat{X}, j), \hat{y}_j \leftarrow \text{Col}(\hat{Y}, j) \]
  if \( -\text{CheckCredentialProof}(\hat{\pi}_j, \hat{x}_j, \hat{y}_j) \) then
    return false
  return true

Algorithm 7.14: Checks the correctness of a credential proof \( \hat{\pi} \) generated by Alg. 7.12.

**Algorithm: CheckCredentialProof\( ( \hat{\pi}, \hat{x}, \hat{y}) \)**

**Input:** Credential proof \( \hat{\pi} = (t, s), t \in (\mathbb{G}_q^{2N_E}), s = (s_1, \ldots, s_{N_E}), s_i = (s_{i,1}, s_{i,2}) \in \mathbb{Z}_q^2 \)

- Public credentials \( \hat{x} = (\hat{x}_1, \ldots, \hat{x}_{N_E}), \hat{x}_j \in \mathbb{G}_q \)
- Public credentials \( \hat{y} = (\hat{y}_1, \ldots, \hat{y}_{N_E}), \hat{y}_j \in \mathbb{G}_q \)

\( y \leftarrow (\hat{x}, \hat{y}) \)
\( c \leftarrow \text{GetNIZKPChallenge}(y, t, \tau) \)

for \( i = 1, \ldots, N_E \)
  \[ t'_i \leftarrow (\hat{x}_i^{-c} \cdot \hat{y}^{s_{i,1}} \mod \hat{\phi}, \hat{y}^{-c} \cdot \hat{y}^{s_{i,2}} \mod \hat{\phi}) \]

\( t' \leftarrow (t'_1, \ldots, t'_{N_E}) \)

return \( t = t' \)

Algorithm 7.15: Computes the lists \( \hat{x} \) and \( \hat{y} \) of public credentials, which are obtained by multiplying corresponding shares obtained from the election authorities. The public credentials \( \hat{x} \) are used in Prot. 6.5 to verify if a submitted ballot belongs to an eligible voter, whereas the public credentials \( \hat{y} \) are used in Prot. 6.6 to verify that the vote confirmation has been invoked by the same eligible voter.
Algorithm 7.16: Computes the list \( s = (S_1, \ldots, S_{N_E}) \) of voting cards for every voter. A single voting card is represented as a string \( S_i \in A_{ucs}^* \), which is generated by Alg. 7.17.

Algorithm 7.17: Computes a string \( S \in A_{ucs}^* \) which represent a voting card that can be printed on paper and sent to voter \( v \). Specifying the formatting details of presenting the information on the printed voting card is beyond the scope of this document.
Algorithm 7.18: 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. 6.3 by the authorities to generate private shares of a common public encryption key.

\begin{verbatim}
Algorithm: GenKeyPair()

\text{\# $\ell \in \mathbb{Z}_q$} 
\text{$pk \leftarrow g^{\ell} \mod p$}
\text{return $(sk, pk)$ // $(sk, pk) \in \mathbb{Z}_q \times \mathbb{G}_q$}
\end{verbatim}

Algorithm 7.19: Generates a proof of knowledge of a private ElGamal key \(sk\) satisfying \(pk = g^{sk}\). For the proof verification, see Alg. 7.21.

\begin{verbatim}
Algorithm: GenPublicKeyProof\((sk, pk)\)

Input: Private decryption key \(sk \in \mathbb{Z}_q\)
Public encryption key \(pk \in \mathbb{G}_q\)

\omega \in \mathbb{Z}_q
\text{\# $\ell \in \mathbb{Z}_q$} 
\text{$t \leftarrow \omega^{sk} \mod p$}
\text{$c \leftarrow \text{GetNIZKPChallenge}(pk, t, \tau)$} // see Alg. 7.4
\text{$s \leftarrow \omega + c \cdot sk \mod q$}
\text{$\pi \leftarrow (t, s)$}
\text{return $\pi$ // $\pi \in \mathbb{G}_q \times \mathbb{Z}_q$}
\end{verbatim}

Algorithm 7.20: Checks if the public key proofs generated by \(s\) authorities are correct.

\begin{verbatim}
Algorithm: CheckPublicKeyProofs\((\pi, pk)\)

Input: Public key proofs $\pi = (\pi_1, \ldots, \pi_s)$, $\pi_j \in \mathbb{G}_q \times \mathbb{Z}_q$
Encryption key shares $pk = (pk_1, \ldots, pk_s)$, $pk_j \in \mathbb{G}_q$

for $j = 1, \ldots, s$ do
\text{\# $\ell \in \mathbb{Z}_q$} 
\text{if $\neg \text{CheckPublicKeyProof}(\pi_j, pk_j)$ then}
\text{return false // see Alg. 7.21}
\text{return true}
\end{verbatim}

Algorithm 7.21: Checks the correctness of a public key proof $\pi$ generated by Alg. 7.19.
Algorithm: GetPublicKey(pk)

**Input:** Public keys $pk = (pk_1, \ldots, pk_s)$, $pk_j \in \mathbb{G}_q$

$pk \leftarrow \prod_{j=1}^s pk_j \mod p$

**return** $pk$  

$// \; pk \in \mathbb{G}_q$

Algorithm 7.22: Computes a public ElGamal encryption key $pk \in \mathbb{G}_q$ from given shares $pk_j \in \mathbb{G}_q$. 
7.4. Election Phase

The election phase is the most complex part of the cryptographic protocol, in which each of the involved parties (voter, voting client, election authorities) calls several algorithms. An overview of all algorithms is given in Table 7.3. To submit a ballot containing the voter’s selections $s$, the voting client calls Alg. 7.23 to obtain the voting page that is presented to the voter and Alg. 7.22 to obtain the public encryption key. Using the voter’s inputs $X$ and $s$, the ballot is constructed by calling Alg. 7.24, which internally invokes several sub-algorithms. The authorities call Alg. 7.28 to check the validity of the ballot and Alg. 7.31 to generate the response to the OT query included in the ballot. The voting client unpacks the responses by calling Alg. 7.32 and assembles the resulting point matrix into the verification codes of the selected candidates by calling Alg. 7.34. The voter then compares the displayed verification codes with the ones on the voting card and enters the confirmation code $Y$. We describe

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Algorithm</th>
<th>Called by</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.23</td>
<td>GetVotingPage($i, v, w, c, n, k, E$)</td>
<td>Voting client</td>
<td>6.4</td>
</tr>
<tr>
<td>7.22</td>
<td>GetPublicKey($pk$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.24</td>
<td>GenBallot($X, s, pk$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.25</td>
<td>GenSelectedPrimes($s$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.26</td>
<td>GenQuery($q, pk$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.27</td>
<td>GenBallotProof($x, m, r, \hat{x}, a, pk$)</td>
<td>Election authority</td>
<td>6.5</td>
</tr>
<tr>
<td>7.28</td>
<td>CheckBallot($v, \alpha, pk, k, E, \hat{x}, B$)</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.29</td>
<td>HasBallot($v, B$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.30</td>
<td>CheckBallotProof($\pi, \hat{x}, a, pk$)</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.31</td>
<td>GenResponse($v, a, pk, n, k, E, P$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.32</td>
<td>GetPointMatrix($\beta, s, r$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.33</td>
<td>GetPoints($\beta, s, r$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.34</td>
<td>GetReturnCodes($s, P_s$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.35</td>
<td>CheckReturnCodes($rc, rc', s$)</td>
<td>Voter</td>
<td></td>
</tr>
<tr>
<td>7.36</td>
<td>GenConfirmation($Y, P$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.37</td>
<td>GetValue($p$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.38</td>
<td>GenConfirmationProof($y^*, \hat{y}$)</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.39</td>
<td>CheckConfirmation($v, \gamma, \hat{y}, B, C$)</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.40</td>
<td>HasBallot($v, B$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td>7.41</td>
<td>HasConfirmation($i, C$)</td>
<td>Election authority</td>
<td>6.6</td>
</tr>
<tr>
<td>7.42</td>
<td>CheckConfirmationProof($\pi, \hat{y}$)</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.43</td>
<td>GetFinalization($v, P, B$)</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.44</td>
<td>GetFinalizationCode($\delta$)</td>
<td>Voting client</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CheckFinalizationCode($FC, FC'$)</td>
<td>Voter</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3.: Overview of algorithms and sub-algorithms of the election phase.
the (human) execution of this task by a call to Alg. 7.35. The voting client then generates
the confirmation message using Alg. 7.36, which invokes several sub-algorithms. By calling
Algs. 7.39 and 7.42, the authorities check the confirmation and return their shares of the
finalization code. Using 7.43, the voting client assembles the finalization code and displays it
to the voter, which finally executes Alg. 7.44 to compare it with the finalization code printed
on the voting card. Section 6.5.2 describes the election phase in more details.

\textbf{Algorithm: GetVotingPage}(v, v, w, c, n, k, E)

\textbf{Input:}
Voter index \( v \in \{1, \ldots, N_E\} \)
Voter descriptions \( v = (V_1, \ldots, V_{N_E}), V_i \in A_{ucs}^* \)
Counting circles \( w = (w_1, \ldots, w_{N_E}), w_i \in \mathbb{N} \)
Candidate descriptions \( c = (C_1, \ldots, C_n), C_i \in A_{ucs}^* \)
Number of candidates \( n = (n_1, \ldots, n_t), n_j \geq 2, n = \sum_{j=1}^t n_j \)
Number of selections \( k = (k_1, \ldots, k_t), 0 < k_j < n_j \)
Eligibility matrix \( E = (e_{ij})_{N_E \times t}, e_{ij} \in \mathbb{B} \)

\( P \leftarrow \ldots \) // compose string to be displayed to the voter
\( \text{return } P \) // \( P \in A_{ucs}^* \)

Algorithm 7.23: Computes a string \( P \in A_{ucs}^* \), which represents the voting page displayed
to voter \( v \). Specifying the details of presenting the information on the voting page is
beyond the scope of this document.

\textbf{Algorithm: GenBallot}(X, s, pk)

\textbf{Input:}
Voting code \( X \in A_X^* \)
Selection \( s = (s_1, \ldots, s_k), 1 \leq s_1 < \cdots < s_k \)
Encryption key \( pk \in \mathbb{G}_q \)

\( x \leftarrow \text{ToInteger}(X) \) // see Alg. 4.7
\( \hat{x} \leftarrow \hat{g}^x \mod \hat{p} \)
\( q \leftarrow \text{GetSelectedPrimes}(s) \) // \( q = (q_1, \ldots, q_k), \) see Alg. 7.25
\( (a, r) \leftarrow \text{GenQuery}(q, pk) \) // \( a = (a_1, \ldots, a_k), r = (r_1, \ldots, r_k), \) see Alg. 7.26
\( m \leftarrow \prod_{j=1}^k q_j \)
if \( m \geq p \) then
\[ \text{return } \perp \] // \( (k, n) \) is incompatible with \( p \)
\( r \leftarrow \sum_{j=1}^k r_j \mod q \)
\( \pi \leftarrow \text{GenBallotProof}(x, m, r, \hat{x}, a, pk) \) // \( \pi = (t, s), \) see Alg. 7.27
\( \alpha \leftarrow (\hat{x}, a, \pi) \)
\( \text{return } (\alpha, r) \) // \( \alpha \in \mathbb{Z}_q \times (\mathbb{G}_q^2)^k \times ((\mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q) \times (Z_q \times Z_q \times Z_q)), r \in \mathbb{Z}_q^k \)

Algorithm 7.24: Generates a ballot based on the selection \( s \) and the voting code \( X \).
The ballot includes an OT query \( a \) and a NIZKP \( \pi \). The algorithm also returns the
randomizations \( r \) of the OT query, which are required in Alg. 7.33 to derive the transferred
messages from the OT response.
Algorithm: GetSelectedPrimes(s)

Input: Selections s = (s₁, ..., s_k), 1 ≤ s₁ < ... < s_k

\( p \leftarrow \text{GetPrimes}(s_k) \)  // see Alg. 7.1

for \( j = 1, \ldots, k \) do
- \( q_j \leftarrow p_{s_j} \)
- \( q \leftarrow (q_1, \ldots, q_k) \)

return \( q \)  // \( q \in (\mathbb{G}_q \cap \mathbb{F})^k \)

Algorithm 7.25: Selects \( k \) prime numbers from \( \mathbb{G}_q \) corresponding to the given indices \( s = (s_1, \ldots, s_k) \). For example, \( s = (1, 3, 7) \) means selecting the first, the third, and the seventh prime from \( \mathbb{G}_q \).

Algorithm: GenQuery(q, pk)

Input: Selected primes \( q = (q_1, \ldots, q_k) \)
- Encryption key \( pk \in \mathbb{G}_q \)

for \( j = 1, \ldots, k \) do
- \( r_j \leftarrow R \mathbb{Z}_q \)
- \( a_{j,1} \leftarrow q_j \cdot pk^{r_j} \mod p \)
- \( a_{j,2} \leftarrow g^{r_j} \mod p \)
- \( a_j \leftarrow (a_{j,1}, a_{j,2}) \)

\( \mathbf{a} \leftarrow (a_1, \ldots, a_k) \)
\( \mathbf{r} \leftarrow (r_1, \ldots, r_k) \)

return \( (\mathbf{a}, \mathbf{r}) \)  // \( \mathbf{a} \in (\mathbb{G}_q \times \mathbb{G}_q)^k, \mathbf{r} \in \mathbb{Z}_q^k \)

Algorithm 7.26: Generates an OT query \( \mathbf{a} \) from the prime numbers representing the voter’s selections and \( \mathbf{a} \) for a given public encryption key (which serves as a generator of \( \mathbb{Z}_p \)).
Algorithm: GenBallotProof($x, m, r, \hat{x}, a, pk$)

**Input:**
- Voting credentials $(x, \hat{x}) \in \mathbb{Z}_q \times \mathbb{G}_q$
- Product of selected primes $m \in \mathbb{G}_q$
- Randomization $r \in \mathbb{Z}_q$
- OT query $a = (a_1, \ldots, a_k)$, $a_j = (a_{j,1}, a_{j,2}) \in \mathbb{G}_q^2$, $k > 0$
- Encryption key $pk \in \mathbb{G}_q$

Let $\omega_1 \in R\mathbb{Z}_q$, $\omega_2 \in R\mathbb{G}_q$, $\omega_3 \in R\mathbb{Z}_q$

1. $t_i \leftarrow g^{\omega_i} \mod p$
2. $t_2 \leftarrow \omega_2 \cdot pk^{a_i} \mod p$
3. $t_3 \leftarrow g^{a_i} \mod p$

1. $y \leftarrow (\hat{x}, a)$, $t \leftarrow (t_1, t_2, t_3)$
2. $c \leftarrow \text{GetNIZKPChallenge}(y, t, \tau)$ \hspace{1cm} // see Alg. 7.4
3. $s_1 \leftarrow \omega_1 + c \cdot x \mod \hat{q}$, $s_2 \leftarrow \omega_2 \cdot m^c \mod p$, $s_3 \leftarrow \omega_3 + c \cdot r \mod q$
4. $\pi \leftarrow (t, s)$

return $\pi$ \hspace{1cm} // $\pi \in (\mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q) \times (\mathbb{Z}_q \times \mathbb{G}_q \times \mathbb{Z}_q)$

Algorithm 7.27: 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 [47]. For the verification of this proof, see Alg. 7.30.

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

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

$k' \leftarrow \sum_{j=1}^{t} e_{vj} k_j$

if $\neg \text{HasBallot}(v, B)$ and $\hat{x} = \hat{x}_v$ and $k = k'$ then \hspace{1cm} // see Alg. 7.29

- if CheckBallotProof($\pi, \hat{x}, a, pk$) then \hspace{1cm} // see Alg. 7.30
- return true

return false

Algorithm 7.28: Checks if a ballot $\alpha$ obtained from voter $v$ is valid. For this, voter $v$ must not have submitted a valid ballot before, $\hat{x}$ must be the public voting credential of voter $v$, the length $k = |a|$ must be equal to $k' = \sum_{j=1}^{t} k_{v,j}$, and $\pi$ must be valid.
Algorithm: HasBallot\((v, B)\)

Input: Voter index \(v \in \mathbb{N}\)

Ballot list \(B = \{(v_i, \alpha_i, z_i)\}_{i=0}^{N-1}, v_i \in \mathbb{N}\)

\[
\text{foreach} \ (v_i, \alpha_i, z_i) \in B \text{ do} \quad \text{// use binary search or hash table for better performance}
\]
\[
\text{if} \ v = v_i \text{ then} \quad \text{return} \ true
\]

return \ false

Algorithm 7.29: Checks if the ballot list \(B\) contains an entry for voter \(v\).

Algorithm: CheckBallotProof\((\pi, \hat{x}, a, pk)\)

Input: Ballot proof \(\pi = (t, s), t = (t_1, t_2, t_3) \in \mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q, s = (s_1, s_2, s_3) \in \mathbb{G}_q \times \mathbb{G}_q \times \mathbb{G}_q\)

Public voting credential \(\hat{x} \in \mathbb{Z}_q\)

OT query \(a = (a_1, \ldots, a_k), a_j = (a_{j,1}, a_{j,2}) \in \mathbb{G}_q^2, k > 0\)

Encryption key \(pk \in \mathbb{G}_q\)

\(y \leftarrow (\hat{x}, a)\)

\(c \leftarrow \text{GetNIZKPC}hallenge(y, t, \tau)\) \quad \text{// see Alg. 7.4}

\(a_1 \leftarrow \prod_{j=1}^{k} a_{j,1} \mod p, a_2 \leftarrow \prod_{j=1}^{k} a_{j,2} \mod p\)

\(t'_1 \leftarrow \hat{x}^{-c} \cdot \hat{g}^{s_1} \mod \hat{p}\)

\(t'_2 \leftarrow a_{1,1}^{-c} \cdot s_2 \cdot pk^{s_3} \mod p\)

\(t'_3 \leftarrow a_{2,1}^{-c} \cdot g^{s_3} \mod p\)

return \((t_1 = t'_1) \land (t_2 = t'_2) \land (t_3 = t'_3)\)

Algorithm 7.30: Checks the correctness of a NIZKP \(\pi\) generated by Alg. 7.27. The public values of this proof are the public voting credential \(\hat{x}\) and the OT query \(a = (a_1, \ldots, a_k)\).
Algorithm: GenResponse\((v, a, pk, n, k, E, P)\)

**Input:** Voter index \(v \in \{1, \ldots, N_E\}\)
- Queries \(a = (a_1, \ldots, a_k), a_j \in \mathbb{G}_q\)
- Encryption key \(pk \in \mathbb{G}_q\)
- Number of candidates \(n = (n_1, \ldots, n_t), n_j \geq 2, n = \sum_{j=1}^t n_j\)
- Number of selections \(k = (k_1, \ldots, k_t), 0 < k_j < n_j\)
- Eligibility matrix \(E = (e_{ij})_{n \times t}, e_{ij} \in \mathbb{B}\)
- Points \(P = (p_{ij})_{N_E \times n}, p_{ij} = (x_{ij}, y_{ij}), x_{ij} \in \mathbb{Z}_q, y_{ij} \in \mathbb{Z}_q\)

\(z_1, z_2 \in \mathbb{R}_q\)

for \(j = 1, \ldots, k\) do
- \(\beta_j \leftarrow z_1^j \cdot z_2^j \mod p\)
- \(\ell_M \leftarrow \lfloor L_M / L \rfloor\)
- \(p \leftarrow \text{GetPrimes}(n)\) \hspace{1cm} // \(p = (p_1, \ldots, p_n)\), see Alg. 7.1
- \(n' \leftarrow 0, k' \leftarrow 0\)

for \(l = 1, \ldots, t\) do
- if \(e_{vl} \neq 0\) then \hspace{1cm} // optimization by excluding \(e_{vl} = 0\)
  - for \(i = n' + 1, \ldots, n' + n_l\) do
    - \(p'_l \leftarrow p_i^{z_1} \mod p\)
    - \(M_l \leftarrow \text{ByteArray}(x_{vi}, \frac{L_M}{2}) || \text{ByteArray}(y_{vi}, \frac{L_M}{2})\) \hspace{1cm} // see Alg. 4.4
    - for \(j = k' + 1, \ldots, k' + e_{vl} k_l\) do
      - \(k_{ij} \leftarrow p'_l ^{\beta_j} \mod p\)
      - \(K_{ij} \leftarrow \text{Truncate}(\|_{c=1}^{\ell_M} \text{RecHash}_L(k_{ij}, c), L_M)\) \hspace{1cm} // see Alg. 4.9
      - \(C_{ij} \leftarrow M_i \oplus K_{ij}\)
    - \(k' \leftarrow k' + e_{vl} k_l\)
    - \(n' \leftarrow n' + n_l\)

\(b \leftarrow (b_1, \ldots, b_k), C \leftarrow (C_{ij})_{n \times k}, d \leftarrow pk^{z_1} g^{z_2} \mod p\)

\(\beta \leftarrow (b, C, d)\)

\(z = (z_1, z_2)\)

return \((\beta, z)\) \hspace{1cm} // \(\beta \in \mathbb{G}_q^k \times (\mathbb{B}^{L_M})^{nk} \times \mathbb{G}_q, z \in \mathbb{Z}_q^2\)

Algorithm 7.31: Generates the response \(\beta\) for the given OT query \(a\). The messages to transfer are byte array representations of the \(n\) points \((p_{v,1}, \ldots, p_{v,n})\). Along with \(\beta\), the algorithm also returns the randomizations \(z\) used to generate the response.
Algorithm: GetPointMatrix(β, s, r)

Input: OT responses β = (β₁, ..., βₖ), βⱼ ∈ Gₖ × (B<sup>L-M</sup>)<sup>nk</sup> × Gₖ
Selection s = (s₁, ..., sₖ), 1 ≤ s₁ < ... < sₖ ≤ n
Randomizations r = (r₁, ..., rₖ), rⱼ ∈ Zₙ

for i = 1, ..., s do
    pᵢ ← GetPoints(βᵢ, s, r) // pⱼ = (pᵢ₁, ..., pᵢₖ), see Alg. 7.33
P_s ← (pᵢⱼ)ᵢ×k
return P_s // Pₛ ∈ (Z_p<sup>2</sup>)<sup>s×k</sup>

Algorithm 7.32: Computes the s-by-k matrix Pₛ = (pᵢⱼ)ᵢ×k of the points obtained from the s authorities for the selection s. The points are derived from the messages included in the OT responses β = (β₁, ..., βₖ).

Algorithm: GetPoints(β, s, r)

Input: OT response β = (b, C, d), b = (b₁, ..., bₖ), bⱼ ∈ Gₖ, C = (Cᵢⱼ)ₖ×k, Cᵢⱼ ∈ B<sup>L-M</sup>, d ∈ Gₖ
Selection s = (s₁, ..., sₖ), 1 ≤ s₁ < ... < sₖ ≤ n
Randomizations r = (r₁, ..., rₖ), rⱼ ∈ Zₙ

ℓₘ ← ⌊Lₘ/L⌋

for j = 1, ..., k do
    kⱼ ← bⱼ · d⁻¹rⱼ mod p
    Kⱼ ← Truncate(ℓₘ RecHash<sub>L</sub>(kⱼ, c), Lₘ) // see Alg. 4.9
    Mⱼ ← Cₛⱼ,ⱼ ⊕ Kⱼ
    xⱼ ← ToInteger(Truncate(Mⱼ, Lₘ 2<sup>−ℓₘ</sup>)) // see Alg. 4.5
    yⱼ ← ToInteger(Truncate(Skip(Mⱼ, Lₘ 2<sup>−ℓₘ</sup>)) // see Alg. 4.5
    if xⱼ ≥ 2<sup>q</sup> or yⱼ ≥ 2<sup>q</sup> then
        return ⊥ // point not in Z_q²
    pⱼ ← (xⱼ, yⱼ)

p ← (p₁, ..., pₖ)
return p // p ∈ (Z_q²)<sup>k</sup>

Algorithm 7.33: Computes the k transferred points p = (p₁, ..., pₖ) from the OT response β using the random values r from the OT query and the selection s. The algorithm returns ⊥, if some transferred point lies outside Z_q².
Algorithm: GetReturnCodes($s, P_s$)

**Input:** Selection $s = (s_1, \ldots, s_k), 1 \leq s_1 < \cdots < s_k \leq n$
Points $P_s = (p_{ij})_{s \times k}, p_{ij} \in \mathbb{Z}_q$

for $j = 1, \ldots, k$ do
  for $i = 1, \ldots, s$ do
    $R_{ij} \leftarrow \text{Truncate} \left( \text{RecHash}_L(p_{ij}), L_R \right)$  // see Alg. 4.9
    $R_j \leftarrow \text{MarkByteArray}(\oplus_{i=1}^{s} R_{ij}, s_j - 1, n_{\max})$  // see Alg. 4.1
    $RC_{s_j} \leftarrow \text{ToString}(R, A_R)$  // see Alg. 4.8
  $r_{cs} \leftarrow (RC_{s_1}, \ldots, RC_{s_k})$
return $r_{cs}$  // $r_{cs} \in (A_R^{t_R})^k$

Algorithm 7.34: Computes the $k$ verification codes $r_{cs} = (RC_{s_1}, \ldots, RC_{s_k})$ for the selected candidates by combining the hash values of the transferred points $p_{ij} \in P_s$ from different authorities.

Algorithm: CheckReturnCodes($rc, rc', s$)

**Input:** Printed verification codes $rc = (RC_1, \ldots, RC_n), RC_i \in A_R^{t_R}$
Displayed verification codes $rc' = (RC'_1, \ldots, RC'_k), RC'_j \in A_R^{t_R}$
Selections $s = (s_1, \ldots, s_k), 1 \leq s_1 < \cdots < s_k \leq n$
return $\bigwedge_{j=1}^{k}(RC_{s_j} = RC'_j)$

Algorithm 7.35: Checks if every displayed verification code $RC'_j$ 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($Y, P$)

**Input:** Confirmation code $Y \in A_Y^{t_Y}$
Points $P = (p_{ij})_{s \times k}, p_{ij} \in \mathbb{Z}_q$

$y \leftarrow \text{ToInteger}(Y) \mod \hat{q}$
for $i = 1, \ldots, s$ do
  $p_i \leftarrow \text{Row}(P, i)$
  $y'_i \leftarrow \text{GetValue}(p_i)$  // see Alg. 7.37
$y' \leftarrow \sum_{i=1}^{s} y'_i \mod \hat{q}$  // see Alg. 4.7
$y^* \leftarrow y + y' \mod \hat{q}$
$\hat{y} \leftarrow \hat{y}^{y^*} \mod \hat{p}$
$\pi \leftarrow \text{GenConfirmationProof}(y^*, \hat{y})$  // $\pi = (t, s)$, see Alg. 7.38
$\gamma \leftarrow (\hat{y}, \pi)$
return $\gamma$  // $\gamma \in \mathbb{G}_q \times (\mathbb{G}_q \times \mathbb{Z}_q)$

Algorithm 7.36: Generates the confirmation $\gamma$, which consists of the public vote approval credential $\hat{y}$ and a NIZKP of knowledge $\pi$ of the private vote approval credential $y^*$.  

94
Algorithm: GetValue(p)
Input: Points \( p = (p_1, \ldots, p_k) \), \( p_j = (x_j, y_j) \in \mathbb{Z}_q^2 \), \( k \geq 0 \)
\( y \leftarrow 0 \)
for \( i = 1, \ldots, k \) do
  \( n \leftarrow 1, \; d \leftarrow 1 \)
  for \( j = 1, \ldots, k \) do
    if \( i \neq j \) then
      \( n \leftarrow n \cdot x_j \mod q \)
      \( d \leftarrow d \cdot (x_j - x_i) \mod q \)
  \( y \leftarrow y + \frac{n}{d} \mod q \)
return \( y \) \hspace{1cm} // \( y \in \mathbb{Z}_q \)

Algorithm 7.37: Computes a polynomial \( A(X) \) of degree \( k - 1 \) from given points \( p = (p_1, \ldots, p_k) \) using Lagrange’s interpolation method and returns the value \( y = A(0) \).

Algorithm: GenConfirmationProof(\( y^* \), \( \hat{y} \))
Input: Private vote approval credential \( y^* \in \mathbb{Z}_q \)
Public vote approval credential \( \hat{y} \in \mathbb{G}_q \)
\( \omega \in_R \mathbb{Z}_q \)
\( t \leftarrow \hat{y} \omega \mod \hat{p} \)
\( c \leftarrow \text{GetNIZKPChallenge}(\hat{y}, t, \tau) \) \hspace{1cm} // see Alg.7.4
\( s \leftarrow \omega + c \cdot y^* \mod \hat{q} \)
\( \pi \leftarrow (t, s) \)
return \( \pi \) \hspace{1cm} // \( \pi \in \mathbb{G}_q \times \mathbb{Z}_q \)

Algorithm 7.38: Generates a NIZKP of knowledge of the private vote approval credential \( y^* \) that matches with a given public vote approval credential \( \hat{y} \). Note that this proof is equivalent to a Schnorr identification proof [47]. For the verification of \( \pi \), see Alg.7.41.

Algorithm: CheckConfirmation(\( v, \gamma, \hat{y}, B, C \))
Input: Voter index \( v \in \{1, \ldots, N_E\} \)
Confirmation \( \gamma = (\hat{y}, \pi) \), \( \hat{y} \in \mathbb{G}_q \), \( \pi \in \mathbb{G}_q \times \mathbb{Z}_q \)
Public vote approval credentials \( \hat{y} = (\hat{y}_1, \ldots, \hat{y}_{N_E}) \), \( \hat{y}_i \in \mathbb{G}_q \)
Ballot list \( B = \langle (v_i, \alpha_i, z_i) \rangle_{i=0}^{N_B-1}, \; v_i \in \{1, \ldots, N_E\} \)
Confirmation list \( C = \langle (v_i, \gamma_i) \rangle_{i=0}^{N_C-1}, \; v_i \in \{1, \ldots, N_E\} \)
if HasBallot(\( v, B \)) and \( \neg \text{HasConfirmation}(\( v, C \)) \) and \( \hat{y} = \hat{y}_v \) then \hspace{1cm} // see Alg.7.29, 7.40
  if CheckConfirmationProof(\( \pi, \hat{y} \)) then \hspace{1cm} // see Alg.7.41
    return true
else
  return false

Algorithm 7.39: Checks if a confirmation \( \gamma \) obtained from voter \( i \) is valid. For this, voter \( v \) must have submitted a valid ballot before, but not a valid confirmation. The check then succeeds if \( \pi \) is valid and if \( \hat{y} \) is the public vote approval credential of voter \( v \).
Algorithm: HasConfirmation\(v, C\)

Input: Voter index \(v \in \mathbb{N}\)

Confirmation list \(C = \{v_j, \gamma_j\}_{j=0}^{N_v-1}, v_j \in \mathbb{N}\)

\[
\text{foreach} \ (v_j, \gamma_j) \in C \ \\
\quad \text{if} \ v = v_j \ \\
\quad \quad \text{return true}
\]

\[
\text{return false}
\]

Algorithm 7.40: Checks if the confirmation list \(C\) contains an entry for voter \(v\).

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

Input: Confirmation proof \(\pi = (t, s), t \in \mathbb{G}_q, s \in \mathbb{Z}_q\)

Public vote approval credential \(\hat{y} \in \mathbb{G}_q\)

\(c \leftarrow \text{GetNIZKPChallenge}(\hat{y}, t, \tau)\) \quad // see Alg.7.4
\(t' \leftarrow \hat{y}^{-c} \cdot \hat{y}^s \mod p\)
\(\text{return} \ (t = t')\)

Algorithm 7.41: Checks the correctness of a NIZKP \(\pi\) generated by Alg.7.38. The public value of this proof is the public vote approval credential \(\hat{y}\).

Algorithm: GetFinalization\(v, P, B\)

Input: Voter index \(v \in \{1, \ldots, N_E\}\)

Points \(P = (p_{ij})_{E \times n}, p_{ij} \in \mathbb{Z}_q^2\)

Ballot list \(B = \{(v_i, \alpha_i, z_i)\}_{i=0}^{N_E-1}, v_i \in \{1, \ldots, N_E\}\)

\(p \leftarrow \text{Row}(P, v)\)
\(F \leftarrow \text{Truncate(RecHash}_L(p), L_F)\) \quad // see Alg.4.9

\[
\text{foreach} \ (v_i, \alpha_i, z_i) \in B \ \\
\quad \text{if} \ v = v_i \ \\
\quad \quad \delta \leftarrow (F, z_i) \quad // \delta \in B^{L_F} \times \mathbb{Z}_q^2
\]

\[
\text{return} \bot 
\quad // \text{no entry for} \ v \ \text{in} \ B
\]

Algorithm 7.42: Computes the finalization code \(F\) for voter \(v\) from the given points \((p_{v,1}, \ldots, p_{v,n})\) and returns \(F\) together with the randomizations used in the creation of the OT response.

Algorithm: GetFinalizationCode\(\delta\)

Input: Finalizations \(\delta = (\delta_1, \ldots, \delta_n), \delta_j = (F_j, z_j), F_j \in B^{L_F}, z_j \in \mathbb{Z}_q^2\)

\(FC \leftarrow \text{ToString}(\delta^j_{F_j, A_F})\) \quad // see Alg.4.8

\(\text{return} FC\) \quad // \(FC \in A^{L_F}_E\)

Algorithm 7.43: Computes a finalization code \(FC\) by combining the values \(F_j\) received from the authorities.
Algorithm 7.44: CheckFinalizationCode($FC, FC'$)

**Input:** Printed finalization code $FC \in A^F_F$

- Displayed finalization code $FC' \in A^F_F$

**Return** $FC = FC'$

Algorithm 7.44: Checks if the displayed finalization code $FC'$ matches with the finalization code $FC$ from the voting card. Note that this algorithm is executed by humans.
7.5. Post-Election Phase

The main actors in the process at the end of an election are the election authorities. Corresponding algorithms are shown in Table 7.4. To initiate the mixing process, the first election authority calls Alg. 7.45 to cleanse the list of submitted ballots and to extract a sorted list of encrypted votes to shuffle. By calling Algs. 7.48 and 7.51, this list is shuffled according to a random permutation and a NIZKP of shuffle is generated. This step is repeated by every election authority. The final result obtained from the last shuffle is the list of encrypted votes that will be decrypted. Before computing corresponding partial decryptions, each election authority calls Alg. 7.54 to check the correctness of the whole shuffle process. The partial decryptions are then computed using Alg. 7.56 and corresponding decryption proofs are generated using Alg. 7.57. The information exchange during this whole process goes over the bulletin board. After terminating all tasks, the process is handed over from the election authorities to the election administrator, who calls Alg. 7.58 to check all decryption proofs and Alg. 7.61 to obtain the final election result. We refer to Section 6.5.3 for a more detailed description of this process.

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Algorithm</th>
<th>Called by</th>
<th>Protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.45</td>
<td>GetEncryptions((B, C, n, k, u, w, E))</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.46</td>
<td>(\neg) GetDefaultEligibilityMatrix((w, w, E))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.47</td>
<td>(\neg) GetDefaultCandidates((j, n, k, u))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.40</td>
<td>(\neg) HasConfirmation((v, C))</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.48</td>
<td>GenShuffle((e, pk))</td>
<td>Election authority</td>
<td>6.7</td>
</tr>
<tr>
<td>7.49</td>
<td>(\neg) GenPermutation((N))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.50</td>
<td>(\neg) GenReEncryption((e, pk))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.51</td>
<td>GenShuffleProof((e, \bar{e}, \bar{r}, \psi, pk))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.52</td>
<td>(\neg) GenPermutationCommitment((\psi, h))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.53</td>
<td>(\neg) GenCommitmentChain((\bar{u}))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.45</td>
<td>GetEncryptions((B, C, n, k, u, w, E))</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.46</td>
<td>(\neg) GetDefaultEligibilityMatrix((w, w, E))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.40</td>
<td>(\neg) HasConfirmation((v, C))</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.54</td>
<td>CheckShuffleProofs((\bar{\pi}, \bar{e}_0, \bar{E}, pk, j))</td>
<td>Election authority</td>
<td>6.8</td>
</tr>
<tr>
<td>7.55</td>
<td>(\neg) CheckShuffleProof((\bar{\pi}, e, \bar{e}, pk))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.56</td>
<td>GetPartialDecryptions((e, sk))</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.57</td>
<td>GenDecryptionProof((sk, pk, e, b'))</td>
<td>Election authority</td>
<td></td>
</tr>
<tr>
<td>7.58</td>
<td>CheckDecryptionProofs((\pi', pk, e, B'))</td>
<td>Election administrator</td>
<td>6.9</td>
</tr>
<tr>
<td>7.59</td>
<td>(\neg) CheckDecryptionProof((\pi', pk, e, b'))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.60</td>
<td>GetDecryptions((e, B'))</td>
<td>Election administrator</td>
<td></td>
</tr>
<tr>
<td>7.61</td>
<td>GetVotes((m, n, w))</td>
<td>Election administrator</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.4.: Overview of algorithms and sub-algorithms of the post-election phase.
Algorithm: GetEncryptions($B, C, n, k, u, w, E$)

**Input:** Ballot list $B = \langle (v_i, \alpha_i, z_i) \rangle_{i=0}^{N_B-1}$, $v_i \in \{1, \ldots, N_E\}$
Confirmation list $C = \langle (v_i, \gamma_i) \rangle_{i=0}^{N_C-1}$, $v_i \in \{1, \ldots, N_E\}$
Number of candidates $n = (n_1, \ldots, n_t)$, $n_j > 2$, $n = \sum_{j=1}^{t} n_j$
Number of selections $k = (k_1, \ldots, k_t)$, $0 < k_j < n_j$
Default candidates $u = \{u_1, \ldots, u_n\}$, $u_i \in \mathbb{B}$
Counting circles $w = (w_1, \ldots, w_{N_E})$, $w_i \in \mathbb{N}$
Eligibility matrix $E = (e_{ij})_{N_E \times t}$, $e_{ij} \in \mathbb{B}$

$w \leftarrow \max_{i=1}^{N_E} w_i$
p ← GetPrimes($n + w$) \quad // $p = (p_1, \ldots, p_{n+w})$, see Alg. 7.1
$E^* \leftarrow $ GetDefaultEligibilityMatrix($w, w, E$) \quad // $E^* = (e^*_{ij})_{w \times t}$, see Alg. 7.46
for $j = 1, \ldots, t$ do
  \[ I_j \leftarrow $ GetDefaultCandidates($j, n, k, u$) \quad // see Alg. 7.47
for $c = 1, \ldots, w$ do
  \[ e_{cd} \leftarrow 0 \quad // \text{loop over } i = 1, \ldots, N_C
\]
\[ i \leftarrow 1 \quad // \text{loop over } i = 1, \ldots, N_C
\]
\[ \text{foreach } (v, \alpha, z) \in B \text{ do} \quad // \alpha = (\hat{x}, \mathbf{a}, \pi), \mathbf{a} = (a_1, \ldots, a_k), a_j = (a_{j,1}, a_{j,2}) \in \mathbb{G}_q^2
\]
  if HasConfirmation($v, C$) then
    \[ c \leftarrow w_v \]
    \[ a_1 \leftarrow \prod_{j=1}^{k} a_{j,1} \mod p \]
    \[ a_1 \leftarrow a_1 \cdot p_{n+c} \mod p \]
    for $j = 1, \ldots, t$ do
      \[ e_{ij} \leftarrow e^*_{ij} \quad // \text{check for restricted eligibility}
      \]
      \[ \text{foreach } d \in I_j \text{ do} \quad // \text{add default candidate}
      \]
      \[ a_1 \leftarrow a_1 \cdot p_d \mod p \]
      \[ u_{cd} \leftarrow u_{cd} + 1 \]
    \]
    \[ a_2 \leftarrow \prod_{j=1}^{k} a_{j,2} \mod p \]
    \[ c_i \leftarrow (a_1, a_2) \]
    \[ i \leftarrow i + 1 \]
\]
\[ \mathbf{e} \leftarrow \text{Sort}_{\leq} (e_1, \ldots, e_{N_C}) \]
$\mathbf{U} \leftarrow (u_{cd})_{w \times n}$
\[ \text{return } \mathbf{e}, \mathbf{U} \quad // \mathbf{e} \in (\mathbb{G}_q^2)^{N_C}, \mathbf{U} \in \{0, \ldots, N_C\}^{w \times n}
\]

Algorithm 7.45: 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. The added default candidates are counted and stored in the matrix $\mathbf{U}$. Sorting the resulting list $\mathbf{e}$ of encrypted votes is necessary to guarantee a unique order. For this, we define a total order over $\mathbb{G}_q^2$ by $e_i \leq e_j \iff (a_i < a_j) \lor (a_i = a_j \land b_i \leq b_j)$, for $e_i = (a_i, b_i)$ and $e_j = (a_j, b_j)$.  

99
**Algorithm: GetDefaultEligibilityMatrix**\( (w, w, E) \)

**Input:** Number of counting circles \( w \geq 1 \)

- Counting circles \( w = (w_1, \ldots, w_{N_E}), w_i \in \{1, \ldots, w\} \)
- Eligibility matrix \( E = (e_{ij})_{N_E \times t}, e_{ij} \in \mathbb{B} \)

for \( j = 1, \ldots, t \) do
  for \( c = 1, \ldots, w \) do
    \[ e^*_{cj} \leftarrow 0 \]
  for \( i = 1, \ldots, N_E \) do
    \[ c \leftarrow w_i \]
    \[ e^*_{cj} \leftarrow 1 - (1 - e^*_{cj})(1 - e_{ij}) \]  // logical or
  \( E^* \leftarrow (e^*_{cj})_{w \times t} \)
return \( E^* \)  // \( E^* \in \mathbb{B}^{w \times t} \)

Algorithm 7.46: Computes the default eligibility of all counting circles.

**Algorithm: GetDefaultCandidates**\( (j, n, k, u) \)

**Input:**
- Election index \( j \in \{1, \ldots, t\} \)
- Number of candidates \( n = (n_1, \ldots, n_t), n_j \geq 2, n = \sum_{j=1}^t n_j \)
- Number of selections \( k = (k_1, \ldots, k_t), 0 < k_j < n_j \)
- Default candidates \( u = \{u_1, \ldots, u_n\}, u_i \in \mathbb{B} \)

\( k \leftarrow 0, I \leftarrow \emptyset \)
\( i_{\text{min}} \leftarrow \sum_{l=1}^{j-1} n_l + 1, i_{\text{max}} \leftarrow \sum_{l=1}^j n_l \)

for \( i = i_{\text{min}}, \ldots, i_{\text{max}} \) do
  if \( u_i = 1 \) then
    \[ I \leftarrow I \cup \{i\} \]
    \[ k \leftarrow k + 1 \]
  if \( k = k_j \) then
    return \( I \)  // \( I \in \{1, \ldots, n\}^{k_j} \)

return \( \bot \)  // not enough default candidates

Algorithm 7.47: Computes set of indices of the first \( k_j \) default candidates of election \( j \). The algorithm returns an error symbol \( \bot \), if \( u \) contains less than \( k_j \) default candidates for the given election. Therefore, testing corresponding constraints from Table 6.2 can be omitted.

100
Algorithm 7.48: Generates a random permutation $\psi \in \Psi_N$ and uses it to shuffle a given list $e = (e_1, \ldots, e_N)$ of ElGamal encryptions $e_i = (a_i, b_i) \in \mathbb{G}_q^2$. With $\Psi_N = \{ (j_1, \ldots, j_N) : j_i \in \{1, \ldots, N\}, j_1 \neq j_2, \forall i \neq 1 \}$ we denote the set of all $N!$ possible permutations of the indices $\{1, \ldots, N\}$.

Algorithm 7.49: Generates a random permutation $\psi \in \Psi_N$ following Knuth’s shuffle algorithm [34, pp. 139–140].

Algorithm 7.50: Generates a re-encryption $e' = (a \cdot pk^{r'}, b \cdot g^{r'})$ of the given ElGamal encryption $e = (a, b) \in \mathbb{G}_q^2$. The re-encryption $e'$ is returned together with the randomization $r' \in \mathbb{Z}_q$. 

101
Algorithm 7.51: Generates a shuffle proof which is equivalent to proving knowledge of a permutation
verification, see Alg. 7.55. For further background information we refer to Section 5.5.

\[49, 51\], except for the fact that the offline and online phases are merged. For the proof

\[\pi \leftarrow \text{GenShuffleProof}(e, \bar{e}, \bar{r}, \psi, \psi, \psi, \psi, \psi, \psi, \psi, \psi, \psi, \psi, \psi, \psi)\]
Algorithm: GenPermutationCommitment(ψ, h)

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

for \( i = 1, \ldots, N \) do
\[ r_i \in_R \mathbb{Z}_q \]
\[ c_i \leftarrow g^{r_i} \cdot h_i \mod p \]
\[ c \leftarrow (c_1, \ldots, c_N) \]
\[ r \leftarrow (r_1, \ldots, r_N) \]
return \( (c, r) \) // \( c \in \mathbb{G}_q^N, r \in \mathbb{Z}_q^N \)

Algorithm 7.52: Generates a commitment \( c = \text{com}(\psi, r) \) to a permutation \( \psi \) by committing to the columns of the corresponding permutation matrix. This algorithm is used in Alg. 7.51.

Algorithm: GenCommitmentChain(\( \tilde{u} \))

Input: Permuted public challenges \( \tilde{u} = (\tilde{u}_1, \ldots, \tilde{u}_N), \tilde{u}_i \in \mathbb{Z}_q \)
\( \hat{c}_0 \leftarrow h \)

for \( i = 1, \ldots, N \) do
\[ \hat{r}_i \in_R \mathbb{Z}_q \]
\[ \hat{c}_i \leftarrow g^{\hat{r}_i} \cdot \hat{c}_{i-1} \mod p \]
\[ \hat{c} \leftarrow (\hat{c}_1, \ldots, \hat{c}_N) \]
\[ \hat{r} \leftarrow (\hat{r}_1, \ldots, \hat{r}_N) \]
return \( (\hat{c}, \hat{r}) \) // \( \hat{c} \in \mathbb{G}_q^N, \hat{r} \in \mathbb{Z}_q^N \)

Algorithm 7.53: Generates a commitment chain \( \hat{c}_0 \rightarrow \hat{c}_1 \rightarrow \cdots \rightarrow \hat{c}_N \) relative to a list of public challenges \( \tilde{u} \). The chain starts with \( \hat{c}_0 = h \), where \( h \in \mathbb{G}_q \) denotes the second public generator of the group. This algorithm is used in Alg. 7.51.

Algorithm: CheckShuffleProofs(\( \tilde{\pi}, \tilde{e}_0, \tilde{E}, pk, i \))

Input: Shuffle proofs \( \tilde{\pi} = (\tilde{\pi}_1, \ldots, \tilde{\pi}_s) \)
ElGamal encryptions \( \tilde{e}_0 = (\tilde{e}_{1,0}, \ldots, \tilde{e}_{N,0}), \tilde{e}_{i,0} \in \mathbb{G}_q^2 \)
Shuffled ElGamal encryptions \( \tilde{E} = (\tilde{e}_{ij})_{N \times s}, \tilde{e}_{ij} \in \mathbb{G}_q^2 \)
Encryption key \( pk \in \mathbb{G}_q \)
Authority index \( i \in \{1, \ldots, s\} \)

for \( j = 1, \ldots, s \) do
if \( i \neq j \) then
\[ \hat{e}_j \leftarrow \text{Col}(\tilde{E}, j) \]
if \( \neg \text{CheckShuffleProof}(\tilde{\pi}_j, \hat{e}_{j-1}, \hat{e}_j, pk) \) then
\[ \text{return false} \] // see Alg. 7.55

return \( true \)

Algorithm 7.54: Checks if a chain of shuffle proofs generated by \( s \) authorities is correct.
Algorithm: CheckShuffleProof(\(\hat{\pi}, e, \hat{e}, pk\))

**Input:** Shuffle proof \(\hat{\pi} = (t, s, c, \hat{c}), t = (t_1, t_2, t_3, (t_{4,1}, t_{4,2}), (\hat{t}_1, \ldots, \hat{t}_N)),\)
\(s = (s_1, s_2, s_3, s_4, (\hat{s}_1, \ldots, \hat{s}_N), (\hat{\tilde{s}}_1, \ldots, \hat{\tilde{s}}_N)), e = (e_1, \ldots, e_N), \hat{c} = (\hat{e}_1, \ldots, \hat{e}_N)\)

ElGamal encryptions \(e = (e_1, \ldots, e_N), e_i = (a_i, b_i) \in \mathbb{G}_q^2\)
Shuffled ElGamal encryptions \(\hat{e} = (\hat{e}_1, \ldots, \hat{e}_N), \hat{c}_i = (\hat{a}_i, \hat{b}_i) \in \mathbb{G}_q^2\)
Encryption key \(pk \in \mathbb{G}_q\)

\[ h \leftarrow \text{GetGenerators}(N) \quad // \text{see Alg. 7.3} \]
\[ u \leftarrow \text{GetNIZKPChallenges}(N, (e, \hat{e}, c, \hat{c}), \tau) \quad // u = (u_1, \ldots, u_N), \text{see Alg. 7.5} \]
\[ y \leftarrow (e, \hat{e}, c, \hat{c}, pk) \]
\[ \bar{c} \leftarrow \prod_{i=1}^N c_i / \prod_{i=1}^N b_i \mod p \]
\[ u \leftarrow \prod_{i=1}^N u_i \mod q \]
\[ \hat{c} \leftarrow \hat{c}_N / h^\tau \mod p \]
\[ \hat{e} \leftarrow \prod_{i=1}^N e_i^\tau \mod p \]
\[ (a, b) \leftarrow (\prod_{i=1}^N a_i^u \mod p, \prod_{i=1}^N b_i^u \mod p) \]
\[ t'_1 \leftarrow \hat{c} - c \cdot g^{b_1} \mod p \]
\[ t'_2 \leftarrow \hat{c} - c \cdot g^{b_2} \mod p \]
\[ t'_3 \leftarrow \hat{c} - c \cdot g^{b_3} \prod_{i=1}^N h_i^{b_i} \mod p \]
\[ (t'_{4,1}, t'_{4,2}) \leftarrow (a^{-c} \cdot pk^{-s_4} \prod_{i=1}^N a_i^{-s_i} \mod p, b^{-c} \cdot g^{-s_4} \prod_{i=1}^N h_i^{-s_i} \mod p) \]
\[ \hat{c}_0 \leftarrow h \]

for \(i = 1, \ldots, N\) do
\[ t'_i \leftarrow \hat{c}_i - c \cdot g^{b_i} \cdot \hat{c}_{i-1} \mod p \]
\[ \text{return } (t_1 = t'_1) \land (t_2 = t'_2) \land (t_3 = t'_3) \land (t_{4,1} = t'_{4,1}) \land (t_{4,2} = t'_{4,2}) \land \left[ \bigwedge_{i=1}^N (\hat{t}_i = \hat{b}_i) \right] \]

Algorithm 7.55: Checks the correctness of a shuffle proof \(\hat{\pi}\) generated by Alg. 7.51. The public values are the ElGamal encryptions \(e\) and \(\hat{e}\) and the public encryption key \(pk\).

Algorithm: GetPartialDecryptions(e, sk)

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

for \(i = 1, \ldots, N\) do
\[ b'_i \leftarrow b_i^{sk} \mod p \]
\[ b' \leftarrow (b'_1, \ldots, b'_N) \]
\[ \text{return } b' \quad // b' \in \mathbb{G}_q^N \]

Algorithm 7.56: Computes the partial decryptions of a given input list \(e\) of ElGamal encryption using a share \(sk\) of the private decryption key.
Algorithm: GenDecryptionProof\((sk, pk, e, b')\)

**Input:** Decryption key \(sk \in \mathbb{Z}_q\)
- Encryption key \(pk \in \mathbb{G}_q\)
- ElGamal encryptions \(e = (e_1, \ldots, e_N)\), \(e_i = (a_i, b_i)\), \(a_i, b_i \in \mathbb{G}_q\)
- Partial decryptions \(b' = (b'_1, \ldots, b'_N)\), \(b'_i \in \mathbb{G}_q\)

\(\omega \in_R \mathbb{Z}_q\)
\(t_0 \leftarrow g^{\omega} \mod p\)
for \(i = 1, \ldots, N\) do
- \(t_i \leftarrow b'_i \mod p\)
- \(t \leftarrow (t_0, (t_1, \ldots, t_N))\)
- \(b \leftarrow (b_1, \ldots, b_N)\)
- \(y \leftarrow (pk, b, b')\)
- \(c \leftarrow \text{GetNIZKPChallenge}(y, t, \tau)\) \hspace{1cm} // see Alg. 7.4
- \(s \leftarrow \omega + c \cdot sk \mod q\)
- \(\pi \leftarrow (t, s)\)

return \(\pi\) \hspace{1cm} // \(\pi \in (\mathbb{G}_q \times \mathbb{G}_q^N) \times \mathbb{Z}_q\)

Algorithm 7.57: Generates a decryption proof relative to ElGamal encryptions \(e\) and partial decryptions \(b'\). This is essentially a NIZKP of knowledge of the private key \(sk\) satisfying \(b'_i = b_i^e\) for all input encryptions \(e_i = (a_i, b_i)\) and \(pk = g^{sk}\). For the proof verification, see Alg. 7.59.

Algorithm: CheckDecryptionProofs\((\pi', pk, e, B')\)

**Input:** Decryption proofs \(\pi' = (\pi'_1, \ldots, \pi'_s)\), \(\pi'_j \in (\mathbb{G}_q \times \mathbb{G}_q^N) \times \mathbb{Z}_q\)
- Encryption key shares \(pk = (pk_1, \ldots, pk_s)\), \(pk_j \in \mathbb{G}_q\)
- ElGamal encryptions \(e = (e_1, \ldots, e_N)\), \(e_i \in \mathbb{G}_q^2\)
- Partial decryptions \(B' = (b_{ij})_{N \times s}\), \(b'_{ij} \in \mathbb{G}_q\)

for \(j = 1, \ldots, s\) do
- \(b'_j \leftarrow \text{Col}(B', j)\)
- if \(\neg \text{CheckDecryptionProof}(\pi'_j, pk_j, e, b'_j)\) then \hspace{1cm} // see Alg. 7.59
- return false

return true

Algorithm 7.58: Checks if the decryption proofs generated by \(s\) authorities are correct.
**Algorithm:** CheckDecryptionProof($\pi', pk, e, b'$)

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

$b \leftarrow (b_1, \ldots, b_N)$
$y \leftarrow (pk, b, b')$
$c \leftarrow \text{GetNIZKPChallenge}(y, t, \tau) \quad // \text{see Alg. 7.4}$
$t_0' \leftarrow pk^{-c} \cdot g^s \mod p$

\[ \begin{array}{l}
\text{for } i = 1, \ldots, N \text{ do} \\
\quad t_i' \leftarrow (b'_i)^{-c} \cdot b_i^s \mod p
\end{array} \]

\[ (t_0 = t_0') \land \left[ \bigwedge_{i=1}^{N} (t_i = t_i') \right] \]

Algorithm 7.59: Checks the correctness of a decryption proof $\pi$ generated by Alg. 7.57. The public values are the ElGamal encryptions $e$, the partial decryptions $b'$, and the share $pk$ of the public encryption key.

**Algorithm:** GetDecryptions(e, B$'$)

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

\[ \begin{array}{l}
\text{for } i = 1, \ldots, N \text{ do} \\
\quad b'_i \leftarrow \prod_{j=1}^{s} b'_{ij} \mod p \\
\quad m_i \leftarrow \frac{a_i}{b'_i} \mod p \\
\end{array} \]

$m \leftarrow (m_1, \ldots, m_N)$

\[ \text{return } m \quad // \ m \in \mathbb{G}_q^{N} \]

Algorithm 7.60: Computes the list of decrypted plaintexts $m = (m_1, \ldots, m_N)$ by assembling the partial decryptions $b'_{ij}$ obtained from $s$ different authorities.
Algorithm: GetVotes(m, n, w)

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

\[ n \gets \sum_{j=1}^{t} n_j \]
\[ w \gets \max_{i=1}^{NE} w_i \]
\(p \leftarrow \text{GetPrimes}(n + w)\)  // \(p = (p_1, \ldots, p_{n+w})\), see Alg. 7.1

for \(i = 1, \ldots, N\) do
  for \(j = 1, \ldots, n\) do
    if \(m_i \mod p_j = 0\) then
      \(v_{ij} \leftarrow 1\)
    else
      \(v_{ij} \leftarrow 0\)
  for \(c = 1, \ldots, w\) do
    if \(m_i \mod p_{n+c} = 0\) then
      \(w_{ic} \leftarrow 1\)
    else
      \(w_{ic} \leftarrow 0\)

\(V \leftarrow (v_{ij})_{N \times n}\), \(W \leftarrow (w_{ic})_{N \times w}\)

return \((V, W)\)  // \(V \in \mathbb{B}^{N \times n}\), \(W \in \mathbb{B}^{N \times w}\)

Algorithm 7.61: Computes the election result matrix \(V = (v_{ij})_{N \times n}\) and corresponding counting circles \(W = (w_{ic})_{N \times w}\) from the products of encoded selections \(m = (m_1, \ldots, m_N)\) by retrieving the prime factors of each \(m_i\). Each resulting vector \(v_i = (v_{i1}, \ldots, v_{in})\) represents somebody’s vote, and each value \(v_{ij} = 1\) represents somebody’s vote for a specific candidate \(j \in \{1, \ldots, n\}\).
7.6. Channel Security

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

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Algorithm</th>
<th>Called by</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.62</td>
<td>GenSignature((sk, U, m))</td>
<td>Election administrator</td>
<td>6.1, 6.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Election authority</td>
<td>6.1, 6.2, 6.3, 6.5, 6.6, 6.7, 6.8</td>
</tr>
<tr>
<td>7.63</td>
<td>VerifySignature((pk, σ, U, m))</td>
<td>Election administrator</td>
<td>6.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Election authority</td>
<td>6.1, 6.3, 6.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Printing authority</td>
<td>6.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Voting client</td>
<td>6.4, 6.5, 6.6</td>
</tr>
<tr>
<td>7.64</td>
<td>GenCiphertext(_φ(p, U, m))</td>
<td>Election authority</td>
<td>6.2</td>
</tr>
<tr>
<td>7.65</td>
<td>GetPlaintext(_φ(sk, c))</td>
<td>Printing authority</td>
<td>6.2</td>
</tr>
</tbody>
</table>

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

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

In case of the encryption and decryption algorithms, which implement a hybrid encryption scheme based on the key-encapsulation mechanism over \(\mathbb{G}_q\) of Section 5.7, we assume that an invertible function \(\phi : M \rightarrow A_{\text{ics}}^*\) exists for converting messages \(m \in M\) into strings \(\phi(m) \in A_{\text{ics}}^*\) and vice versa. As long as \(\phi^{-1}(\phi(m)) = m\) holds for all \(m \in M\), any mapping that is efficiently computable in both directions is suitable. The actual choice of \(\phi\) is therefore a technical detail of minor importance, which needs not to be specified in this document. In practice, standard string formats such as XML or JSON are often used for this purpose.

Another assumption in the following two algorithms is the availability of an AES-256 block cipher implementation in combination with the CTR mode of operation.\(^2\) For a 256-bit key \(k \in \mathbb{B}^{32}\) (32 bytes), we use \(B' \leftarrow \text{AES-CTR}(k, B)\) to denote the encryption of a byte array \(B \in \mathbb{B}^*\) of length \(L\) into a byte array \(B' \in \mathbb{B}^*\) of the same length \(L\), and \(B \leftarrow \text{AES-CTR}^{-1}(k, B')\) for the corresponding decryption. To solve the problem mentioned in [12, Section 10.9], we use \(k\) also to encrypt the election event identifier \(U\) and include its encryption in the ciphertext.

\(^2\)Using the largest possible AES key length (256 bits instead of 192 or 128 bits) guarantees maximal compatibility with current and future security levels of Chapter 8.
**Algorithm:** \text{GenSignature}(sk, U, m)

**Input:** Signature key \( sk \in \mathbb{Z}_q \)
- Election event identifier \( U \in A_{\text{pcs}}^* \)
- Message \( m \in M, M \) unspecified

\begin{algorithmic}
\Repeat
\State \( r \in_R \mathbb{Z}_q \)
\State \( t \leftarrow \tilde{q}^r \mod \hat{p} \)
\State \( c \leftarrow \text{TolInteger}(\text{RecHash}_L(t, U, m)) \mod \tilde{q} \) \Comment{see Algs. 4.5 and 4.9}
\State \( s \leftarrow r - c \cdot sk \mod \tilde{q} \)
\Until {\( c \neq 0 \) and \( s \neq 0 \)}
\State \( \sigma \leftarrow (c, s) \)
\Return \( \sigma \) \Comment{\( \sigma \in \mathbb{Z}_q^2 \)}
\end{algorithmic}

Algorithm 7.62: Computes a Schnorr signature for a given election event identifier \( U \), message \( m \), and signature key \( sk \). For the verification of this signature, see Alg. 7.63. Using tuples \( m = (m_1, \ldots, m_r) \) as input, the algorithm can be used to sign multiple messages simultaneously.

**Algorithm:** \text{VerifySignature}(pk, \sigma, U, m)

**Input:** Verification key \( pk \in \mathbb{G}_q \)
- Signature \( \sigma = (c, s) \in \mathbb{Z}_q^2 \)
- Election event identifier \( U \in A_{\text{pcs}}^* \)
- Message \( m \in M, M \) unspecified

\begin{algorithmic}
\State \( t' \leftarrow \tilde{g}^* \cdot pk^c \mod \hat{p} \)
\State \( c' \leftarrow \text{TolInteger}(\text{RecHash}_L(t', U, m)) \mod \tilde{q} \) \Comment{see Algs. 4.5 and 4.9}
\Return \( c = c' \)
\end{algorithmic}

Algorithm 7.63: Verifies a Schnorr signature \( \sigma = (c, s) \) generated by Alg. 7.62 using a given public verification key \( pk \).
Algorithm: GenCiphertext_φ(pk, U, m)

Input: Encryption key pk ∈ G_q
   Election event identifier U ∈ A^*_ucs
   Message m ∈ M, M unspecified

1. r ∈ R Z_q
2. k ← RecHash_{32}(pk^r mod ̄p) // see Alg. 4.9
3. c_1 ← ̄y^r mod ̄p
4. c_2 ← AES-CTR(k, UTF8(U))
5. c_3 ← AES-CTR(k, UTF8(φ(m)))
6. c ← (c_1, c_2, c_3)

return c // c ∈ G_q × B^* × B^*

Algorithm 7.64: Computes a hybrid encryption for an election event identifier U, a message m, and a public encryption key pk. With φ : M → A^*_ucs we denote an invertible mapping for converting messages into UCS strings. Alg. 7.65 is the corresponding decryption algorithm.

Algorithm: GetPlaintext_φ(sk, c)

Input: Decryption key sk ∈ Z_q
   Ciphertext c = (c_1, c_2, c_3), c_1 ∈ G_q, c_2 ∈ B^*, c_3 ∈ B^*

1. k ← RecHash_{32}(c_1^k mod ̄p) // see Alg. 4.9
2. U ← UTF8^{-1}(AES-CTR^{-1}(k, c_2))
3. m ← φ^{-1}(UTF8^{-1}(AES-CTR^{-1}(k, c_3)))

return (U, m) // U ∈ A^*_ucs, m ∈ M, M unspecified

Algorithm 7.65: Decrypts a ciphertext c = (c_1, c_2, c_3) for a given private decryption key sk. The algorithms uses the inverse mapping φ^{-1} : A^*_ucs → M from Alg. 7.64.
Part IV.

System Specification
8. Security Levels and Parameters

In this chapter, we introduce three different security levels \( \lambda \in \{1, 2, 3\} \), for which default security parameters are given. An additional security level \( \lambda = 0 \) with very small parameters is introduced for testing purposes. Selecting the “right” security level is a trade-off between security, efficiency, and usability. The proposed parameters are consistent with the general constraints listed in Table 6.1 of Section 6.3.1. In Section 8.1, we define general length parameters for the hash algorithms and the mathematical groups and fields. Complete sets of recommended group and field parameters are listed in Section 8.2. We recommend that exactly these values are used in an actual implementation. In Section 9.1, we specify various alphabets and code lengths for the voting, confirmation, finalization, and verification codes.

8.1. Recommended Length Parameters

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

<table>
<thead>
<tr>
<th>Security Level ( \lambda )</th>
<th>Security Strength ( \sigma, \tau )</th>
<th>Hash Length ( \ell (L) )</th>
<th>( G_q \subset \mathbb{Z}_p^* )</th>
<th>( G_q \subset \mathbb{Z}_p^* )</th>
<th>( L_M )</th>
<th>Crypto-period</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8 (1)</td>
<td>12</td>
<td>12</td>
<td>2</td>
<td>Testing</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
<td>160 (20)</td>
<td>1024</td>
<td>1024</td>
<td>40</td>
<td>Legacy</td>
</tr>
<tr>
<td>2</td>
<td>112</td>
<td>224 (28)</td>
<td>2048</td>
<td>2048</td>
<td>56</td>
<td>( \leq 2030 )</td>
</tr>
<tr>
<td>3</td>
<td>128</td>
<td>256 (32)</td>
<td>3072</td>
<td>3072</td>
<td>64</td>
<td>( &gt; 2030 )</td>
</tr>
</tbody>
</table>

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

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

\[
\text{Hash}_L(B) = \text{Truncate}(	ext{SHA256}(B), L)
\]

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

### 8.2. Recommended Group and Field Parameters

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

<table>
<thead>
<tr>
<th>( e )</th>
<th>Hexadecimal representation of Euler’s number (first 3072 bits of fractional part).</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2.B7E151628ED2A6ABF7158809CF4F3C762E7160F3BB4DA56A784D9045190CFF32E7738926CFBE54FB68D8D8C31D763DA06C80AB1185EB4F7CB757BF5958490CFD47D7C19BB42158D9554F7B46BCED55C4D79FD5F24D6613C31C3839A2DDFB8A9A276BCFBA1C8775C6284DAB79CD4C2B3293D02E9E5EAF02AC60ACC93ED874422A52ECB238FEEE5AB6ADD835FD1A0753D0A8F78537D2B95BB7980DCAEc642C1E9F23B895BC7820BF38737DF8BB300D1334A0DDBD8645C8FA73A6160FFE393C48CBB8CA060FOF8ECC6D31EBE5CCED7F2FOB08001716BC60D45A0EBC1BCD289B06CBBF6A21AD08E1847FF3773856C6D946406EFD0D37BE6700E186D1BF275B9E241DE64749A47DFD6FB69632C3EB061BB6472BF84C26144E49C2D04C324EF10DE513D3F5114BBB5D37493CB8879C7D52FFD72BA0AAE7277DA7BA1BA4AF1488DE836AF14865E6C37AB676FE690B571121382AF341AE94F77BCF06C3B8FF5675F0979074AD9A787BC5B9DB40C5937D3EDE4C3A7939625E</td>
<td></td>
</tr>
</tbody>
</table>

For each security level, we apply the following general rules. We choose the smallest safe prime \( p \in \mathbb{S} \) satisfying \( e_s \leq p < 2^s \), 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 \( e_t \leq \hat{q} < 2^t \) and then the smallest co-factor \( \hat{k} \geq 2 \) satisfying \( \hat{p} = \hat{k}\hat{q} + 1 \in \mathbb{P} \) and \( e_s \leq \hat{p} < 2^s \). 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. 7.3). For the groups \( \mathbb{G}_q \), we use \( \hat{g} = 2^\hat{k} \mod \hat{p} \) as default generator.

---

1. Taken from http://www.numberworld.org/constants.html.
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 $\mathbf{p} = (p_1, \ldots, p_{60})$ of the first 60 primes in $\mathbb{G}_q$, which are required to encode the selected candidates $s$ as a single element $\Gamma(s) \in \mathbb{G}_q$ (see Sections 5.3 and 6.5 for more details).

8.2.1. Level 0 (Testing Only)

$p = 0xB93 = 2963$
$q = 0x5C9 = 1481$
$k = 2$
$g = 4$
$h = 9$

$\hat{p} = 0xEED = 3821$
$\hat{q} = 0xBF = 191$
$\hat{k} = 0x14 = 20$
$\hat{q} = 0x656 = 1622$

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

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

Table 8.4.: The first 60 prime numbers in $\mathbb{G}_q \subset \mathbb{Z}^*_p$ for $p$ and $q$ as defined in Table 8.3.
### 8.2.2. Level 1

| $p$ | 0xB7E151628AED2A6ABF7158809CF4F3C762E7160F3884DA56A784D9045190CFEF324E7738926CFBE5F4BF88D88C31D763DA06C80ABB1185EB4F7C7B5757F5958490CFD47D7C19BB42158D9554F7B46B | 0xB7E151628AED2A6ABF7158809CF4F3C762E7160F3884DA56A784D9045190CFEF324E7738926CFBE5F4BF88D88C31D763DA06C80ABB1185EB4F7C7B5757F5958490CFD47D7C19BB42158D9554F7B46B | CED55C4D79FD5F24D6613C31C3839A2D | DF8A9A276BCFBFA1C877C56284DAB79C | 0xB7E151628AED2A6ABF7158809CF4F3C762E7161D |
| $q$ | 0x5BF0A8B1457695355FB8AC404E7A9E3B1738B079C5A6D2B526C26C828C0B7F79273B9C4936DF25AFC66C618EB | 0xB7E151628AED2A6ABF7158809CF4F3C762E7161D |
| $k$ | 2 | 2 |
| $g$ | 4 | 4 |
| $h$ | 9 | 9 |

Table 8.5.: Groups $\mathbb{G}_q \subset \mathbb{Z}_p^*$ and $\mathbb{G}_q^* \subset \mathbb{Z}_p^*$ for security level $\lambda = 1$ with default generators $g$, $h$, and $\hat{g}$, respectively.
| \( p_1 = 3 \) | \( p_{11} = 59 \) | \( p_{21} = 151 \) | \( p_{31} = 263 \) | \( p_{41} = 353 \) | \( p_{51} = 457 \) |
| \( p_2 = 5 \) | \( p_{12} = 79 \) | \( p_{22} = 157 \) | \( p_{32} = 269 \) | \( p_{42} = 367 \) | \( p_{52} = 463 \) |
| \( p_3 = 7 \) | \( p_{13} = 83 \) | \( p_{23} = 179 \) | \( p_{33} = 271 \) | \( p_{43} = 373 \) | \( p_{53} = 467 \) |
| \( p_4 = 11 \) | \( p_{14} = 89 \) | \( p_{24} = 181 \) | \( p_{34} = 277 \) | \( p_{44} = 379 \) | \( p_{54} = 479 \) |
| \( p_5 = 13 \) | \( p_{15} = 101 \) | \( p_{25} = 199 \) | \( p_{35} = 281 \) | \( p_{45} = 383 \) | \( p_{55} = 509 \) |
| \( p_6 = 23 \) | \( p_{16} = 103 \) | \( p_{26} = 227 \) | \( p_{36} = 283 \) | \( p_{46} = 409 \) | \( p_{56} = 523 \) |
| \( p_7 = 29 \) | \( p_{17} = 109 \) | \( p_{27} = 229 \) | \( p_{37} = 293 \) | \( p_{47} = 419 \) | \( p_{57} = 547 \) |
| \( p_8 = 41 \) | \( p_{18} = 131 \) | \( p_{28} = 239 \) | \( p_{38} = 317 \) | \( p_{48} = 431 \) | \( p_{58} = 557 \) |
| \( p_9 = 43 \) | \( p_{19} = 137 \) | \( p_{29} = 241 \) | \( p_{39} = 337 \) | \( p_{49} = 443 \) | \( p_{59} = 563 \) |
| \( p_{10} = 47 \) | \( p_{20} = 149 \) | \( p_{30} = 251 \) | \( p_{40} = 347 \) | \( p_{50} = 449 \) | \( p_{60} = 569 \) |

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

### 8.2.3. Level 2

\[
p = 0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190CFEF324E7738926CFBE54F80D88C3D763DA06C80ABB1185EB4F7C7B5757F5958490CFD47D7C19BBA2158D9554F7B46BCED55C4D79FD5F24D6613C1C389A22DF8A92765BCF8A1C877C56284DA79CD4C2B329DD0E9E5EAF02AC60ACC93ED87422A52EBC238FEE5A6AD835FD1A0753D0A8F7E537D2B95BB79D8DCAEC642C1E9F23B829B5C2780BF38737DF8BB30D0134A0D0DB645C8F8A73A6160FFE393C48CBB8A060F0FF8EC6D31BE5CCEE7F2F0BB088017163BC60DF45A0EBC1B0CD289B06CBBFEA21AD08E1847F37378D56CED9464D6EF0D3D7BE69D0063\]

\[
q = 0x5BF0A8B1457695355FB8AC0447A79E3B1738B079C5A6D2B53C268228C867F799273B9C49367DF2A5F6C6C618EB1ED0364055D88C2F5A7BE3DABABFACAC24867EA3EBE0CDDA10AC6CAA7BDA35E75E2A626BCFEAF926B309E18E11C16EFC54D135B7EDDF04E3BE2B1426D5BCE6A159949E9074F2F5781563056649F6C3A211257691B7F22DB56EC1AEB8D03A9E8547BC729BE95CADDBCEC6E57632160F4F91DC14DAE13C05F93C9BEF5CD98608995A0658E5C322E5FD39D30B7FF1C9E2465DDE5030787FC76369BDF5A6E776BF97B5D84400BB1DE306FA2D07658DE694D8365DF510D68470C23F9FB9C6AB676CA3206B77869E9BDF34E8031\]

Table 8.7.: Group \( \mathbb{G}_q \subset \mathbb{Z}_p^* \) for security level \( \lambda = 2 \) with default generators \( g \) and \( h \).
Table 8.8.: The first 60 prime numbers in $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for $p$ and $q$ as defined in Table 8.7.

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_{11}$</th>
<th>$p_{21}$</th>
<th>$p_{31}$</th>
<th>$p_{41}$</th>
<th>$p_{51}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>53</td>
<td>137</td>
<td>233</td>
<td>331</td>
<td>433</td>
</tr>
<tr>
<td>7</td>
<td>61</td>
<td>139</td>
<td>257</td>
<td>347</td>
<td>449</td>
</tr>
<tr>
<td>11</td>
<td>71</td>
<td>149</td>
<td>263</td>
<td>349</td>
<td>461</td>
</tr>
<tr>
<td>17</td>
<td>83</td>
<td>157</td>
<td>271</td>
<td>353</td>
<td>479</td>
</tr>
<tr>
<td>19</td>
<td>97</td>
<td>167</td>
<td>281</td>
<td>401</td>
<td>547</td>
</tr>
<tr>
<td>23</td>
<td>101</td>
<td>179</td>
<td>283</td>
<td>419</td>
<td>557</td>
</tr>
<tr>
<td>29</td>
<td>109</td>
<td>193</td>
<td>299</td>
<td>421</td>
<td>559</td>
</tr>
<tr>
<td>37</td>
<td>127</td>
<td>199</td>
<td>317</td>
<td>431</td>
<td>571</td>
</tr>
<tr>
<td>41</td>
<td>131</td>
<td>229</td>
<td>313</td>
<td>442</td>
<td>579</td>
</tr>
<tr>
<td>47</td>
<td>143</td>
<td>233</td>
<td>329</td>
<td>454</td>
<td>599</td>
</tr>
</tbody>
</table>

Table 8.9.: Group $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for security level $\lambda = 2$ with default generator $\hat{g}$.
### 8.2.4. Level 3

\[ p = \text{0xB7E15128ED2A6ABF7158809CF4F3C762E7160F38BDA56A784D905190CFEF324E} \]
\[ 7738926CFBE54F8808D5C31D76DA06C0AB1185EB4F7C7B5757F5958490CF4D7D7C \]
\[ 19B42158D954F7B46BCEC55CD79FDF24D6613C31C3839A2DF8A9A276BCFB1A18 \]
\[ 77C56284DAB7CD42C82932D0E99E5EAF2A0C60ACC93ED87422A526C8238FEE556A6 \]
\[ DD835FD1A0755D0A8F78E537D295BB79D8DCAEC642C1E9F23B8295BC2780BF3873DF \]
\[ 8BB300D1334AD08D645C2BA73A6160FE393C45CBB8CA060F0F8EC6D131EB5CCEE \]
\[ D7F2FB0B08B01716BC6DF450AECB1B2CD289B06CBBFAEA21ADB08E1847F3787D56CDE \]
\[ 946D40EF0D3D37BE67008E186D1BF2759B9241DE64749A47DFDB966323CEB016B4 \]
\[ 72BBF84C26144E49C2D04C3E4E10513DF5114B85D374D3C88797CD72FFD72BA \]
\[ 0AAE77DDA7BA1B4AF1488DE836AF148656C37AB6876EF690B571121382AF34E9 \]
\[ 4F77BC06C83B8F5B750F507974AD97A878BC59BD4B05C9573CDE4ECA79396919C7D \]

\[ q = \text{0x5BF0A8B1457695355FB8AC404E7A9E3B1738B079C56D2B53C26C8228C8676FF79927} \]
\[ 3B9C49367DF2FA5FC6C6C615EBB1ED03640568D2C82FA7BE3D9ABBAFCAC24867EAA3EB \]
\[ 0CDDA10ACE6A97BDA35E76AEAE26BCEFA9F26B309E181E1C1C6D54D3D3B3671D0F4E \]
\[ 3BE2B1426D5BCE6A159949E9074F2F5781563056649663A2152976591C77FD72D5B \]
\[ 6ECA1FE8D03A9E5B7C722BE95CADD8C6E587631260F4F91D1C4ADE13C05F9C93BEF \]
\[ C5D980699A506B5EC322E5F9D39D307FF1C92465DDE503078FC763698DF5A677 \]
\[ 6BF9785DB4400B88B1DE306FA2D07658DE944D8365DF510D68470C23F9FBB6CA6B76 \]
\[ CA3265B778699BDF308407C368DF93A0CD9205E5B23AD236E2F6D8C31961F5830DB2 \]
\[ 395DF26130A2724E168261927786F289E9F8885CAE9A69C543CE3EA97FEB95D \]
\[ 057393BED3D30A578A446C741B578A432F361BDB437F3465BABB9909C1579AD7F4 \]
\[ A7BBDE783641DC7FAB3AF84BC83A56CD3C3E2DCEAE586299E96F261D3C9CB20CE6B \]

\( k = 2 \)
\( g = 4 \)
\( h = 9 \)

Table 8.10.: Group \( G_q \subset \mathbb{Z}_p^* \) for security level \( \lambda = 3 \) with default generators \( g \) and \( h \).

<table>
<thead>
<tr>
<th>( p )</th>
<th>( p_{11} )</th>
<th>( p_{21} )</th>
<th>( p_{31} )</th>
<th>( p_{41} )</th>
<th>( p_{51} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>89</td>
<td>167</td>
<td>313</td>
<td>457</td>
<td>577</td>
</tr>
<tr>
<td>3</td>
<td>101</td>
<td>173</td>
<td>317</td>
<td>461</td>
<td>593</td>
</tr>
<tr>
<td>7</td>
<td>103</td>
<td>181</td>
<td>331</td>
<td>463</td>
<td>599</td>
</tr>
<tr>
<td>11</td>
<td>109</td>
<td>199</td>
<td>367</td>
<td>479</td>
<td>607</td>
</tr>
<tr>
<td>13</td>
<td>113</td>
<td>211</td>
<td>375</td>
<td>491</td>
<td>615</td>
</tr>
<tr>
<td>17</td>
<td>127</td>
<td>229</td>
<td>383</td>
<td>499</td>
<td>643</td>
</tr>
<tr>
<td>23</td>
<td>131</td>
<td>233</td>
<td>397</td>
<td>503</td>
<td>647</td>
</tr>
<tr>
<td>29</td>
<td>139</td>
<td>239</td>
<td>401</td>
<td>507</td>
<td>653</td>
</tr>
<tr>
<td>31</td>
<td>141</td>
<td>251</td>
<td>409</td>
<td>509</td>
<td>657</td>
</tr>
<tr>
<td>47</td>
<td>143</td>
<td>257</td>
<td>409</td>
<td>509</td>
<td>657</td>
</tr>
<tr>
<td>59</td>
<td>151</td>
<td>283</td>
<td>449</td>
<td>563</td>
<td>691</td>
</tr>
<tr>
<td>71</td>
<td>157</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.11.: The first 60 prime numbers in \( G_q \subset \mathbb{Z}_p^* \) for \( p \) and \( q \) as defined in Table 8.10.
$p = \text{0xB7E151628AED2A6ABF7158809CF43C762E7160F37B4DA56A784D9045190CFEF324E}$  
7738926C8E5F4BF8D8CD8317D63A60C8AB1185EB4F7C7B5757F598940CFDF47D7C  
19BB4215D8B74AE8D2D3293D20E9E5EAF02A60ACC93ED874422A52ECEB238FEE65A6B  
DDE35FD1A7075D308F7E5E37D2B95B79808CAEC6421CE9F23B8B95C7280BF38737DF  
8BB3C0D01334A0D0DB6945CBFA73A6160FE939C34CB8B0A60F0FF8ECD631EBC5CEEE  
D7F2F0B8B08810117616CD65F45A0EBC18CDB8906CBF8EA21D80E1487F378D56CED  
94640D6EFD03D37BE6700E186D1BF27B59B241DE6B67494AF47DFF96632C3EB061B64  
72B4F84C2614449C2D04C324FE10D513DF511488B5D37493B8879C752FFD727A  
0AAE7277DA7B4A148F8D8E8B36AF14865E6C7A8676FE96B571121382A3F41AE9  
4F790F02FA1B83E786B40C4AC4B5C3D14E0D8C95557C97648A4038771FC25F84BB

$q = \text{0xB7E151628AED2A6ABF7158809CF43C762E7160F37B4DA56A784D9045190D05D}$

$k = \text{0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF67215E}$  
C1D7BB8A77D7B5C2294EFC4A7B3C6906FC93847CD5FEEF6F101C14003101C2150C4  
450843B67D7B184CA9B7178B67001B502DFAC3E829D3102610EB51B9D470FOE  
FBC232F5025A3D88C658E70D92097C54E4081BBFE2373A9B5076970B38F68B5D302E61  
293DBBC1B158E3FC5412F726264B82A4CFA5EA43F5F6C9D99B88155CEA5EC971  
5322344FF714C8F18D0B19772C421923C7E2CD2A6FE100F0FCB4B88B0CABAAFF47438  
C8C29EE75521F18B03C9816975948F17747F6EBDB816512A0FECEA7D6DEF0A7B6A0  
99167F82337346BDFC1A47586EADFB125A9A71CD960DEDECDE9399A37D740FEFBD940  
3A4ECC70A5841F416E3ED0D70B1A5970EBC446DF220714EB334962754D5C81516  
FCC5ED708EBC21C36C0F3D494E04C15E3C275C18A562BADD0293ADE9075FA254E96  
E73402

$g = \text{0x47AD70733EFE399D1AFF4E387250218B88DF5F4040C31851AE1DF0985D0019950}$  
A958710C6B35B6B3BB45C278318D5C883C3C5B7502D3BC8C77D746E3D1FB287E3  
63C1014417DF83EBADD0F14F6FD986104CDF16C4AEC33BB5906C149C38E65B8837E  
12AB32E7A69C4ABEB0B014FFF1FFBB3173E0B300E1DAE5F52F6260D537879001248  
29751320FE6A1F5BD290F8B46C7EB7815193E5C2460F93A35FD16F7A73D8509CE6B17  
517DF2F88D503E61D60B529A43A00866FA931247C9166C32C99A01823528D3F15686  
0ECC5D9A6D61406F15970A9188EAEFC4A68D8A5DF29105DF3D30049715A9AD  
8E0EB73675D5BFFE7A2F082186FA2F77C472D198844E4DA581BB3D3DE9DF2E61D  
C06FA5249C3200C3DBBF24D5C257879C8323D931ED4AD1F9FA168B38FAAC36DB89A9A  
D89BB6B3F47F81B57856C12A2D2FD708A932DC4C91A48E66237C4076A5D242E844C80  
0EC1E6A31EFC8EB1CA52E57DB760843E83BC225FC62456AB46E39DA3CF45AB11A50

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

9. 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 9.1 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 9.2 various possible workarounds, which each has its own strengths and weaknesses.

9.1. 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 9.1, 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} \approx 5.95 \text{ bits of entropy}. Note that the Base64 alphabet \text{A}_{64} requires two non-alphanumerical characters to reach 6 bits of entropy.

Another special case is the last alphabet in Table 9.1, which contains 6^5 = 7776 different English words from the new \textit{Diceware wordlist} of the Electronic Frontier Foundation.\footnote{See http://world.std.com/~reinhold/diceware.html.} 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.

\footnote{See https://www.eff.org/deeplinks/2016/07/new-wordlists-random-passphrases.}
### Table 9.1: Common alphabets with different sizes and characteristics. Case-insensitivity and fail-safety are desirable properties to facilitate flawless user entries.

<table>
<thead>
<tr>
<th>Name</th>
<th>Alphabet</th>
<th>Case-insensitive</th>
<th>Fail-safe</th>
<th>Bits per character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td>$A_{10} = {0, \ldots, 9}$</td>
<td>•</td>
<td>•</td>
<td>3.32</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>$A_{16} = {0, \ldots, 9, A, \ldots, F}$</td>
<td>•</td>
<td>•</td>
<td>4</td>
</tr>
<tr>
<td>Latin</td>
<td>$A_{26} = {A, \ldots, Z}$</td>
<td>•</td>
<td></td>
<td>4.70</td>
</tr>
<tr>
<td>Alphanumeric</td>
<td>$A_{32} = {0, \ldots, 9, A, \ldots, Z} \setminus {0, 1, I, 0}$</td>
<td>•</td>
<td>•</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>$A_{36} = {0, \ldots, 9, A, \ldots, Z}$</td>
<td>•</td>
<td></td>
<td>5.17</td>
</tr>
<tr>
<td></td>
<td>$A_{57} = {0, \ldots, 9, A, \ldots, Z, a, \ldots, z} \setminus {0, 1, I, 0, 1}$</td>
<td>•</td>
<td></td>
<td>5.83</td>
</tr>
<tr>
<td>Base64</td>
<td>$A_{62} = {0, \ldots, 9, A, \ldots, Z, a, \ldots, z}$</td>
<td>•</td>
<td></td>
<td>5.95</td>
</tr>
<tr>
<td></td>
<td>$A_{64} = {A, \ldots, Z, a, \ldots, z, 0, \ldots, 9, =, /}$</td>
<td>•</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Diceware</td>
<td>$A_{7776} = {&quot;abacus&quot;, \ldots, &quot;zoom&quot;}$</td>
<td>•</td>
<td></td>
<td>12.92</td>
</tr>
</tbody>
</table>

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 $N \geq 2$. The conversion algorithms depend on the assumption that the characters in $A$ are totally ordered and that a ranking function $\text{rank}_A(c_i) = i - 1$ representing this order is available. We propose to derive the ranking function from the characters as listed in Table 9.1. In the case of $A_{16}$, for example, this means that the ranking function looks as follows:

<table>
<thead>
<tr>
<th>$c_i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>rank$<em>{A</em>{16}}(c_i)$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

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).

### 9.1.1. 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 9.1 satisfying fail-safety. For the security levels $\lambda \in \{0, 1, 2, 3\}$ introduced in the beginning of this chapter, Table 9.2 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 $2\tau$-bit integer:

$$\hat{q}_x = \hat{q}_y = \begin{cases} 
2^7, & \text{for } \lambda = 0, \\
2^{159}, & \text{for } \lambda = 1, \\
2^{223}, & \text{for } \lambda = 2, \\
2^{255}, & \text{for } \lambda = 3. 
\end{cases}$$
By looking at the numbers in Table 9.2, 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 9.2.

<table>
<thead>
<tr>
<th>Security Level $\lambda$</th>
<th>Security Strength $\tau$</th>
<th>Required bit length</th>
<th>$\ell_X, \ell_Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>$A_{10}$</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
<td>160</td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td>112</td>
<td>224</td>
<td>68</td>
</tr>
<tr>
<td>3</td>
<td>128</td>
<td>256</td>
<td>78</td>
</tr>
</tbody>
</table>

Table 9.2.: Lengths of voting and confirmation codes for different alphabets and security levels.

9.1.2. 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 $\epsilon$, the maximal number of candidates $n_{\text{max}}$, and the size of the chosen alphabet. For $n_{\text{max}} = 1678$ and security levels $\lambda \in \{0, 1, 2, 3\}$, Table 9.3 shows the resulting code lengths for different alphabets and different deterrence factors $\epsilon = 1 - 10^{-(\lambda+2)}$. This particular choice for $n_{\text{max}}$ 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 $\lambda = 2$.

<table>
<thead>
<tr>
<th>Security Level $\lambda$</th>
<th>Deterrence Factor $\epsilon$</th>
<th>$L_R$</th>
<th>$\ell_R$</th>
<th>$L_F$</th>
<th>$\ell_F$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$A_{10}$</td>
<td>$A_{16}$</td>
<td>$A_{26}$</td>
<td>$A_{36}$</td>
</tr>
<tr>
<td>0</td>
<td>99%</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>99.9%</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>99.99%</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>99.999%</td>
<td>4</td>
<td>10</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 9.3.: Lengths of verification and finalization codes for different alphabets and security levels. For the maximal number of candidates, we use $n_{\text{max}} = 1678$ as default value.

In the light of the results of Table 9.3 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 5-digit numbers for $\lambda \in \{1, 2\}$ or as a 8-digit numbers for $\lambda = 3$.

## 9.2. 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.6.1). 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 optimal solution 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 9.4 gives an update of the values from Table 9.2 for different security levels and alphabets.

In the following two subsections, we describe multiple ways of achieving such a usability improvement. In all proposals, we only discuss the case of the voting credential $x$ and
Table 9.4.: Lengths of voting and confirmation codes for different alphabets and security levels by reducing the required bit length from $2\tau$ to $\tau$ bits.

<table>
<thead>
<tr>
<th>Security Level $\lambda$</th>
<th>Security Strength $\tau$</th>
<th>Required bit length</th>
<th>$\ell_X$, $\ell_Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>$A_{10}$</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
<td>80</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>112</td>
<td>112</td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>128</td>
<td>128</td>
<td>39</td>
</tr>
</tbody>
</table>

Assume that the confirmation credential $y$ is treated equally. The approach presented in the first subsection does not require additional communication during the protocol execution, but it is based on bilinear mappings, which requires rather complex mathematics. Three other approaches are presented in the second subsection, in which an additional channel from the printing authority to the bulletin board is required. We summarize the advantages and disadvantages of all approaches in Section 9.2.3.

9.2.1. Approach 1: Using Bilinear Mappings

This approach is highly compatible with the protocols presented in Chapter 6. It only substitutes the cryptographic methods and underlying mathematics of the authentication mechanism. It is based on a bilinear mapping $\phi : G_1 \times G_2 \rightarrow H$ between groups $(G_1, +, −, 0)$, $(G_2, +, −, 0)$ and $(H, \times, −1, 1)$ satisfying three properties:

- Bilinearity: $\phi(x_1 + x_2, y) = \phi(x_1, y) \times \phi(x_2, y)$ holds for all values $x_1, x_2 \in G_1$ and $y \in G_2$; symmetrically, $\phi(x, y_1 + y_2) = \phi(x, y_1) \times \phi(x, y_2)$ holds for all values $x \in G_1$ and $y_1, y_2 \in G_2$.
- Non-degeneracy: $\phi(x, y) \neq 1$ holds for some values $x \in G_1$ and $y \in G_2$.
- Computability: $\phi(x, y)$ can be computed efficiently for all values $x \in G_1$ and $y \in G_2$.

Among other things, bilinearity implies $\phi(ax, y) = \phi(x, y)^a$, $\phi(x, by) = \phi(x, y)^b$, and therefore $\phi(ax, bz) = \phi(x, y)^{ab}$. In the special case of $G = G_1 = G_2$, called symmetric pairing, this property allows to solve the DDH problem in $G$ (but not in $H$).\(^3\) This seems to be a technical subtlety, but it opens the door for a special cryptographic discipline called pairing-based cryptography, which has numerous applications in different areas. We use it here to define an identification scheme with sufficiently short private credentials.

Pairing-Based Identification Scheme. Let $q = |G|$ be the order of $G = G_1 = G_2$ and $g_1, g_2 \in G$ two independent generators. For generating a private credential of length $\ell \leq \|q\|$, a random value $x \in \{0, \ldots, 2^\ell − 1\}$ is chosen uniformly at random. The corresponding public credential is a pair $\hat{x} = (g_1^{r_1}, g_2^{r_2})$, where $r_1 \in \mathbb{Z}_q$ is picked uniformly at random from the whole range of possible values and $r_2 = r_1 + x \mod q$ is derived from $r_1$ and $x$. For such a

\(^3\)For values $x, ax, bx, cx \in G$, deciding if $ab = c$ is equivalent to checking $\phi(ax, bx) = \phi(x, cx)$.
generating a fresh pair of public credentials

Note that for values which implies

p

sufficient for generating suitable pairs

To avoid that suitable pairs can be found without knowing x, for example by computing \((\hat{x}_1', \hat{x}_2') = (\hat{x}_1, \hat{x}_2) \times (g_1^{r_1'}, g_2^{r_2'})\) for arbitrary values \(r' \in \mathbb{Z}_q\), it is important to demonstrate the freshness of \((\hat{x}_1', \hat{x}_2')\) by proving knowledge of \(r_1'\) and \(r_2'\). Such a proof of knowledge can be constructed as a composition of two Schnorr identification proofs (see Section 5.4). In a non-interactive setting, a proof transcript

\[\pi = \text{NIZKP}(\langle r_1', r_2' \rangle : \hat{x}_1 = g_1^{r_1'} \land \hat{x}_2 = g_2^{r_2'})\]

must therefore be presented along with \(\hat{x}' = (\hat{x}_1', \hat{x}_2')\), where \(\pi = (t_1, t_2, s_1, s_2) \in G^2 \times \mathbb{Z}_q^2\) consists of four values. As a consequence, identifying the holder of the private credential \(x\) according to this scheme requires two steps: checking the above condition relative to \(\hat{x} = (\hat{x}_1, \hat{x}_2)\) and \(\hat{x}' = (\hat{x}_1', \hat{x}_2')\) and verifying the validity of \(\pi\). Note that computing the bilinear mapping \(\phi\) is only necessary in the first verification step, but not for generating \(\hat{x}'\) and \(\pi\).

Consider multiple private credentials \(x_1, \ldots, x_s \in \mathbb{Z}_q\) with corresponding public credentials \(\hat{x}_i = (\hat{x}_{i,1}, \hat{x}_{i,2})\). By computing the sum and the product of these values, we obtain a new valid pair

\[(x, \hat{x}) = (\sum_i x_i \mod q, \prod_i \hat{x}_i) = (\sum_i x_i \mod q, (\prod_i \hat{x}_{i,1}, \prod_i \hat{x}_{i,2}))\]

of private and public credentials. This property is similar to the Schnorr identification scheme (see Section 6.4.4), in which multiple private and public credentials can be aggregated without considerably increasing the length of the private credential (roughly by \(\log s\) bits only).

**Protocol Adjustments.** The above identification scheme could be used to replace the Schnorr identification in the protocol such that the general information flow remains exactly the same. Minor protocol adjustments result from changing the underlying mathematics. First of all, to represent a public credentials \(\hat{x} = (\hat{x}_1, \hat{x}_2)\), we require now two group elements from \(G\) instead of one group element from \(G_\mathbb{F}\). Since all known bilinear mappings operate on elliptic curves, such group elements are points consisting of two coordinates from the underlying finite field. In the case of the Weil pairing

\[\phi : \mathrm{E}(\mathbb{F}_p^r)[q] \times \mathrm{E}(\mathbb{F}_p^r)[q] \to \mathbb{F}_p^r[q],\]
which maps two elements from a $q$-order subgroup of an elliptic curve $E(F_{p^k})$ over an extension field $F_{p^k}$ into an element of a $q$-order subgroup of $F_{p^k}$, where $k > 1$ denotes a small embedding factor (typically $k \in \{6, \ldots, 12\}$), these coordinates are field elements of $F_{p^k}$, which can be represented each by a $k$-tuple of values from $F_p$ [15]. Therefore, we need four such coordinates (i.e., $4k$ values from $F_p$) in total to represent $\hat{x} \in E(F_{p^k})[q] \times E(F_{p^k})[q]$. The same holds for the value $\hat{x}' \in E(F_{p^k})[q] \times E(F_{p^k})[q]$ generated during voter identification. Similarly, for representing the generators $g_1, g_2 \in E(F_{p^k})[q]$ and the commitments included in the proof transcript $\pi$, we need $2k$ values from $F_p$ in each case. Note that $p$ and $q$ as used in this context denote much smaller prime numbers than the ones used elsewhere in this document. To achieve $\tau$ bits of security, $2\tau$ bits are sufficient for both $p$ and $q$, and $\ell = \tau$ bits are sufficient for $x$.

As a consequence, the main protocol change for the voting client is the computation of $\hat{x}'$ and $\pi$ during vote casting, which requires implementing elliptic curve computations on the client side. For the election authorities, generating the shares of $\hat{x}$ is the main change in the pre-election phase. During vote casting, the main change consists in checking the above condition relative to $\hat{x}$ and $\hat{x}'$ (which involves computing the bilinear map twice) and verifying the proof transcript $\pi$. For the printing authorities, computations relative to the private credential $x$ remain the same.

9.2.2. Approach 2: Extending the Printing Authority

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” [4, 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 publish data on the bulletin board 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 $2\tau$ bits to $\tau$ bits. As discussed in Section 6.6 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 9.1 shows the extended communication model.

**Approach 2a:** The first approach is based on the observation that a symmetric encryption key of length $\tau$ is sufficient for achieving a security strength of $\tau$ bits, for example when using AES. Therefore, instead of printing a $2\tau$-bit voting credential $x_i \in \mathbb{Z}_{q^e}$ to the voting card of voter $i$ (see Table 6.1 and Section 9.1.1 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 $\tau$, prints $k_i$ to the voting card, encrypts $x_i$ using $k_i$ into $[x_i] \leftarrow \text{Enc}_{k_i}(x_i)$, and sends the encryption $[x_i]$ to the bulletin board. At the end of the pre-election phase, a list $([x_1], \ldots, [x_{N_E}])$ of such encrypted private voting credentials is available on the bulletin board, 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 bulletin board and decrypts $[x_i]$ into $x_i \leftarrow \text{Dec}_{k_i}([x_i])$. Finally, $x_i$ is used to authenticate voter $i$ as an eligible voter as before. Note that for solving the same usability problem in another system proposed for the Swiss context, exactly this idea has been proposed in [24, Section 6.1].

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 random 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 2b:** 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 via the bulletin board, 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 $2\tau$-bit voting credential contains enough entropy for extracting a $\tau$-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 [36, 35]. Therefore, we let the printing authority compute a secret key $k_i \leftarrow \text{HKDF}_\tau(x_i, s_i, c)$ of length $\tau$ bits, which is used to encrypt $x_i$. The random salt is published on the bulletin board 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 $s_i$ from the bulletin board and decrypting $[x_i]$ using $k_i$, can additionally check the validity of the secret key $k_i = \text{HKDF}_\tau(x_i, s_i, c)$. 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 the voting client. If the check fails, the voting procedure must be aborted.

**Approach 2c:** In this approach, we reverse the role of the key derivation function in the previous approach. Here, the KDF is used to derive a $2\tau$-bit value $x'_i = \text{HKDF}_{2\tau}(x_i, s_i, c)$ from a $\tau$-bit private voting credential $x_i \in \mathbb{Z}_{\hat{p}}$. This means that the constraint $\|\hat{q}_x\| \geq 2\tau$ from Table 6.1 is relaxed into $\|\hat{q}_x\| \geq \tau$. 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 $\hat{x}_i = \hat{g}^{x'_i} \mod \hat{p}$ is now computed by the printing authority based on $x'_i$. The printing authority is also responsible for publishing this value on the bulletin board, 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 bulletin board to compute $x'_i = \text{HKDF}_{2\tau}(x_i, s_i, c)$. 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 three approaches presented above.

### 9.2.3. Comparison of Methods

In the previous subsection, we presented four different methods to mitigate the aforementioned usability problem. In each case, the number of entropy bits for a voter to enter is reduced from $2\tau$ to $\tau$ bits. However, this improvement comes at a price. Since introducing an absolute scale for comparing these prices is rather difficult, we prefer to give an overview of the differences, strengths, and weaknesses of each approach rather than selecting a single winner. 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 9.5, 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 the current protocol, and also the similarity between Approach 2a and Approach 2b. Note that selecting and aggregating the shares $x_{ij}$ of the private voting credential $x_i \in \mathbb{Z}_{q_i}$ looks identical in all four approaches, but the selected values are not equally long. In Approaches 2a and 2b, they consist of at least $2\tau$ bits (the same as in the current protocol), whereas in Approaches 1 and 2c, they consist of $\tau$ bits. This difference results from relaxed restrictions relative to the upper bound $\hat{q}_i$ in two of the four cases.

<table>
<thead>
<tr>
<th>Current Protocol</th>
<th>Approach 1</th>
<th>Approach 2a</th>
<th>Approach 2b</th>
<th>Approach 2c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{ij} \in R \mathbb{Z}_{q_x}$</td>
<td>$x_i \leftarrow \sum_j x_{ij}$</td>
<td>$s_i \in R \mathbb{B}^\tau$</td>
<td>$s_i \in R \mathbb{B}^\tau$</td>
<td>$s_i \in R \mathbb{B}^\tau$</td>
</tr>
<tr>
<td>$k_i \in R \mathbb{B}^\tau$</td>
<td>$k_i \leftarrow \text{HKDF}_\tau(x_i, s_i, c)$</td>
<td>$x'_i \leftarrow \text{HKDF}_2\tau(x_i, s_i, c)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[x_i] \leftarrow \text{Enc}<em>{k_i}(x_i), x_i \leftarrow \text{Dec}</em>{k_i}([x_i])$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\hat{x}<em>{ij} \leftarrow \hat{g}^{x</em>{ij}}$</td>
<td>$\hat{x}<em>{ij} \leftarrow (g_1^{r}, g_2^{r+q</em>{ij}})$</td>
<td>$\hat{x}<em>{ij} \leftarrow \hat{g}^{x</em>{ij}}$</td>
<td>$\hat{x}_i \leftarrow \hat{g}^{x'_i}$</td>
<td></td>
</tr>
<tr>
<td>$\hat{x}<em>i \leftarrow \prod_j \hat{x}</em>{ij}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\hat{x}'_i \leftarrow (g_1^{r'}, g_2^{r'+x_i})$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.5.: Necessary computations in each of the four 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 9.6. It exposes the augmented responsibility assigned to the printing authority in Approaches 2a, 2b, and 2c. In all three cases, this involves generating random values and sending some data to the bulletin board. This is the main disadvantage in comparison to both the current protocol and Approach 1. 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 all other approaches, this is the main disadvantage of Approach 2a, 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 2b and 2c, corresponding tests can be implemented into the voting client or as part of the universal verification process (see explanations given in the previous subsection).

Something that is not visible in Tables 9.5 and 9.6 is the complex mathematics required to implement the bilinear mapping in Approach 1, and also the fact that the proposed identification scheme has not yet been described in a scientific publication. In other words, it is not yet an established cryptographic scheme with formally proven security properties, i.e., further research will be necessary for achieving this. These are the two main weaknesses...
of Approach 1.

Compared to the current protocol, a subtle weakness of all four approaches is the fact that entering the voter index $i$ becomes mandatory. In the current protocol, entering $i$ appears in Prot. 6.4 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 four proposed approaches. In Approach 1, $\hat{x}_i$ is non-deterministic and can therefore not be reconstructed without knowing its randomness. In all other approaches, $\hat{x}_i$ can be reconstructed from $x_i$, but knowing $i$ is necessary for retrieving the right values $[x_i]$ and $s_i$ from the bulletin board, which are needed to derive $x_i$.$^4$

<table>
<thead>
<tr>
<th>Protocol Phase</th>
<th>Party</th>
<th>Task</th>
<th>Current protocol</th>
<th>Approach 1</th>
<th>Approach 2a</th>
<th>Approach 2b</th>
<th>Approach 2c</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>EA$_j$</td>
<td>Select at random, send to PA</td>
<td>$x_{ij}$</td>
<td>$x_{ij}$</td>
<td>$x_{ij}$</td>
<td>$x_{ij}$</td>
<td>$x_{ij}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute and send to BB</td>
<td>$\hat{x}_{ij}$</td>
<td>$\hat{x}_{ij}$</td>
<td>$\hat{x}_{ij}$</td>
<td>$\hat{x}_{ij}$</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Retrieve from BB</td>
<td>$(\hat{x}_{ij})$</td>
<td>$(\hat{x}_{ij})$</td>
<td>$(\hat{x}_{ij})$</td>
<td>$(\hat{x}_{ij})$</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td>$\hat{x}_i$</td>
<td>$\hat{x}_i$</td>
<td>$\hat{x}_i$</td>
<td>$\hat{x}_i$</td>
<td>-</td>
</tr>
<tr>
<td>6.2</td>
<td>PA</td>
<td>Select at random</td>
<td>-</td>
<td>-</td>
<td>$k_i$</td>
<td>$s_i$</td>
<td>$s_i$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td>$x_i$</td>
<td>$x_i$</td>
<td>$x_i, [x_i]$</td>
<td>$x_i, k_i, [x_i]$</td>
<td>$x_i, x'_i, \hat{x}_i$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Send to BB</td>
<td>-</td>
<td>-</td>
<td>$[x_i]$</td>
<td>$s_i, [x_i]$</td>
<td>$s_i, \hat{x}_i$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Send to $V_i$</td>
<td>$i, x_i$</td>
<td>$i, x_i$</td>
<td>$i, k_i$</td>
<td>$i, k_i$</td>
<td>$i, x_i$</td>
</tr>
<tr>
<td>6.4</td>
<td>$V_i$</td>
<td>Enter into $VC_i$</td>
<td>$i, x_i$</td>
<td>$i, x_i$</td>
<td>$i, k_i$</td>
<td>$i, k_i$</td>
<td>$i, x_i$</td>
</tr>
<tr>
<td>6.5</td>
<td>$VC_i$</td>
<td>Retrieve from BB</td>
<td>-</td>
<td>-</td>
<td>$[x_i]$</td>
<td>$s_i, [x_i]$</td>
<td>$s_i$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check integrity of</td>
<td>-</td>
<td>-</td>
<td>$k_i, x_i, s_i$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compute</td>
<td>$\pi$</td>
<td>$\hat{x}'_i, \pi$</td>
<td>$x_i, \pi$</td>
<td>$x_i, \pi$</td>
<td>$x'_i, \pi$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Send to BB</td>
<td>$i, \pi$</td>
<td>$i, \hat{x}'_i, \pi$</td>
<td>$i, \pi$</td>
<td>$i, \pi$</td>
<td>$i, \pi$</td>
</tr>
<tr>
<td>6.6</td>
<td>$EA_j$</td>
<td>Retrieve from BB</td>
<td>$i, \pi$</td>
<td>$i, \hat{x}'_i, \pi$</td>
<td>$i, \pi$</td>
<td>$i, \pi$</td>
<td>$i, \pi$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check integrity of</td>
<td>-</td>
<td>-</td>
<td>$\hat{x}_i, \hat{x}'_i$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Check validity of</td>
<td>$\pi$</td>
<td>$\pi$</td>
<td>$\pi$</td>
<td>$\pi$</td>
<td>$\pi$</td>
</tr>
</tbody>
</table>

Table 9.6.: 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.

$^4$In case of Algorithm 2c, 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. On the other hand, we do not entirely exclude it as an option for achieving an optimal compromise between security and usability.
A recapitulation of the above discussion and comparison is given in Table 9.7. It lists the major strengths and weaknesses for each of the four approaches. As mentioned before, it turns out that no single winner can be selected based on our analysis. However, since Approach 2a seems to be strictly less preferable than Approach 2b or 2c, we recommend excluding it from further consideration. For very different reasons, we also have reservations against Approach 1. The main problem there is the additional complexity for dealing with bilinear maps. Since implementing bilinear maps is known to be very difficult, describing corresponding algorithms in pseudocode be a challenge for this document.

Therefore, we conclude this discussion by recommending either Approach 2b or 2c 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. Nevertheless, we propose to conduct further research relative to Approach 1, which is the only approach that does not augment the printing authority’s responsibility, and to keep it as a possibility for a future protocol update.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
</table>
| 1        | – No new channel from printing authority to bulletin board  
|          | – Information flow identical to current protocol | – Complex mathematics and implementation  
|          | | – Identification scheme not well studied (no publication, no formal security proofs)  
|          | | – Client- and server-side computations more expensive  
|          | | – Additional cryptographic parameters |
| 2a       | – Tasks executed by election authorities remain unchanged | – New channel from printing authority to bulletin board  
|          | | – Random keys generated by printing authority  
|          | | – Validity of secret keys can not be checked |
| 2b       | – Tasks executed by election authorities remain unchanged  
|          | | – Validity of secret key can be checked by voting client | – New channel from printing authority to bulletin board  
|          | | | – Random salt generated by printing authority |
| 2c       | – Tasks executed by election authorities is simplified  
|          | | – Validity of private credentials can be publicly verified | – New channel from printing authority to bulletin board  
|          | | | – Random salt generated by printing authority  
|          | | | – Private credentials revealed after vote casting |

Table 9.7.: Recapitulation of major weaknesses and strength.
Part V.

Conclusion
10. Conclusion

10.1. Recapitulation of Achievements

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

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

- Votes are end-to-end encrypted from the voting client to the final tally. We use a verifiable re-encryption mix-net for breaking up the link between voters and their votes before performing the decryption.
- By comparing some codes, voters can verify that their vote has been recorded as intended. If the verification succeeds, they know with sufficiently high probability that their vote has reached the ballot box without any manipulation by malware or other types of attack. We realize this particular form of individual verifiability with an existing oblivious transfer protocol [28].
- Based on the public election data produced during the protocol execution, the correctness of the final election result can be verified by independent parties. We use digital signatures, commitments, and zero-knowledge proofs to ensure that all involved parties strictly comply with the protocol in every single step. In this way, we achieve a complete universal verification chain from the election setup all the way to the final tally.
- Every critical task of the protocol is performed in a distributed way by multiple election authorities, such that no single party involved in the protocol can manipulate the election result or break vote privacy. This way of distributing the trust involves the code generation during the election preparation, the authentication of the voters, the sharing of the encryption key, the mixing of the encrypted votes, and the final decryption.
By providing these properties, we have addressed all major security requirements of the legal ordinance (see Section 1.1). For adjusting the actual security level to current and future needs, all system parameters are derived from three principal security parameters. This way of parameterizing the protocol offers a great flexibility for trading off the desired level of security against the best possible usability. The strict parametrization is also an important prerequisite for formal security proofs.

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

10.2. 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 Bulletin Board**: Throughout this document, we have assumed the existence of a robust append-only bulletin board, which is available to all protocol participants at all times. However, the implementation of a secure bulletin board is a very difficult problem on its own. The main challenge is to guarantee the consistency of the messages posted to the board without creating a single point of failure. There is a considerable amount of literature on this topic, but so far no consensus about the best approach has been reached [13, 20, 30, 31, 32, 37, 39, 46]. The problem in the context of this document is a little less critical, because copies of all submitted ballots are automatically kept by all election authorities. Lost, manipulated, or added ballots are therefore detected without any additional measures. Nevertheless, the robustness of the board is still critical for the proper functioning of the system.

- **Secure Printing**: The most critical component in our protocol is the printing authority (see Section 6.2). It is the only party that learns enough information to manipulate the election, for example by submitting ballots in the name of real voters. Printing sensitive information securely is known to be a difficult problem. The technical section of the VEleS ordinance accepts a solution based on organizational and procedural measures. Defining them, putting them in place, and supervising them during the printing process is a problem that needs no be addressed separately. This problem gets even more challenging, if one of the proposals of Section 9.2 for improved usability is implemented.

- **Privacy Attack on Voting Device**: The assumption that no adversary will attack the voter’s privacy on the voting device is a very strong one. The problem could be solved
by pure code voting [43], 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.
A. Major Protocol Changes

Two major protocol revisions have been implemented so far. The first revision between Version 1.1.1 and Version 1.2 was the response to a major security problem found by Tomasz Truderung. The necessary changes affected all algorithms related to the oblivious transfer at the heart of the protocol. In Section 5.3.2, we give some details about the flaw of the earlier protocol version, but we do not give an explicit list of all the changes that were necessary to fix it. The second revision from Version 1.4.2 to Version 2.0 is the result of implementing the recommendations obtained from the external reviewers [12]. To establish maximal transparency related to the implementation of these recommendations, we document the details of this revision in Section A.1.

A.1. From Version 1.4.2 to Version 2.0

In Section 10 of their report [12], the external reviewers provide nine explicit recommendations for improving the quality of the protocol [12]. We have taken them into account in Version 2.0 of this document. In the list given below, we describe corresponding modifications made to the protocol and to this document. To implement this revision, it was necessary to introduce some new algorithms and to modify some of the existing algorithms. As a consequence, some of the existing algorithms have been renumbered. Table A.1 gives an overview of the renumbering and the changes applied to the algorithms.

Recommendations 10.1 and 10.2

The election authorities generate two additional NIZKPs during the pre-election phase. Relative to authority $j$, we call them $\pi_j$ for the proof of knowing the share $sk_j$ of the private key and $\hat{\pi}_j$ for the proof of knowing the private shares of the voters’ credentials. To solve notational conflicts with existing proofs, we renamed the shuffle proofs from $\pi_j$ into $\tilde{\pi}_j$, with the tilde symbol indicating that a shuffle has been conducted. For keeping the notation as consistent as possible, this renaming implied a cascade of other (non-critical) renamings, mostly in the context of the shuffling algorithms. For example, shuffling a list of ciphertexts in Section 5.5 is now denoted as $\bar{e} \leftarrow \text{Shuffle}_pk(e, \bar{r}, \psi)$ and the proof generation as $\bar{\pi} \leftarrow \text{GenProof}_{pk}(e, \bar{e}, \bar{r}, \psi)$. Therefore, we added the tilde symbol everywhere in Prots. 6.7 to 6.9 and in Algs. 7.48, 7.51 and 7.53 to 7.55. In some of these algorithms, we also renamed some internal variables to improve the overall notational consistency. For improved simplicity, we removed the initial value of the commitment chain from the parameter list of Alg. 7.53.

For generating the cryptographic proof $\pi_j$, a new algorithm $\text{GenPublicKeyProof}(sk_j, pk_j)$ is called in Prot. 6.3 by each election authority. The proofs are submitted to the bulletin board along with the shares $pk_j$ of the public encryption key. This implies that the signatures $\sigma_j^{\text{gen}}$
<table>
<thead>
<tr>
<th>General Algorithms</th>
<th>Pre-Election Phase</th>
<th>Election Phase</th>
<th>Post-Election Phase</th>
<th>Channel Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 ⇒ 7.1</td>
<td>7.6 ⇒ 7.6</td>
<td>7.17 ⇒ 7.23</td>
<td>7.39 ⇒ 7.45</td>
<td>7.54 ⇒ 7.62</td>
</tr>
<tr>
<td>7.2 ⇒ 7.2</td>
<td>7.7 ⇒ 7.7</td>
<td>7.18 ⇒ 7.24</td>
<td>7.46</td>
<td>7.55 ⇒ 7.63</td>
</tr>
<tr>
<td>7.3 ⇒ 7.3</td>
<td>7.8 ⇒ 7.8</td>
<td>7.19 ⇒ 7.25</td>
<td>7.47</td>
<td>7.56 ⇒ 7.64</td>
</tr>
<tr>
<td>7.4 ⇒ 7.4</td>
<td>7.9 ⇒ 7.9</td>
<td>7.20 ⇒ 7.26</td>
<td>7.48</td>
<td>7.57 ⇒ 7.65</td>
</tr>
<tr>
<td>7.5 ⇒ 7.5</td>
<td>7.10 ⇒ deleted</td>
<td>7.21 ⇒ 7.27</td>
<td>7.49</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.11 ⇒ deleted</td>
<td>7.22 ⇒ 7.28</td>
<td>7.50</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.23 ⇒ 7.29</td>
<td>7.51</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.24 ⇒ 7.30</td>
<td>7.52</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.25 ⇒ 7.31</td>
<td>7.53</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.26 ⇒ 7.32</td>
<td>7.54</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.27 ⇒ 7.33</td>
<td>7.55</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.12 ⇒ 7.15</td>
<td>7.28 ⇒ 7.34</td>
<td>7.56</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.29 ⇒ 7.35</td>
<td>7.57</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.30 ⇒ 7.36</td>
<td>7.58</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.31 ⇒ 7.37</td>
<td>7.59</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.32 ⇒ 7.38</td>
<td>7.60</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.33 ⇒ 7.39</td>
<td>7.61</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.34 ⇒ 7.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.35 ⇒ 7.41</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.36 ⇒ 7.42</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.37 ⇒ 7.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.38 ⇒ 7.44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.1.: Overview of algorithm renumbering from Version 1.4.2 to Version 2.0 of this document. New algorithms are highlighted in green, algorithms with critical changes in red, algorithms with semi-critical changes (e.g., modified ranges of input or output parameters) in blue, and algorithms with uncritical changes (e.g., renamed or new local variables, re-ordered code lines) in gray.

generated by the authorities during key generation now depend on $U$, $pk_j$, and $\pi_j$. Additional verification steps have been added to Prot. 6.3, 6.5 and 6.9 by calling a second new algorithm $\text{CheckPublicKeyProofs}(\pi, pk)$, which internally calls $\text{CheckPublicKeyProof}(\pi_j, pk_j)$ for checking the proofs $\pi_j$ one after another.

Similarly, for generating the proof $\hat{\pi}_j$, a new algorithm $\text{GenCredentialProof}(x_j, y_j, x_j, y_j)$ is called in Prot. 6.1 by each election authority. The proofs are submitted to the bulletin board along with the shares $x_j$ and $y_j$ of the public credentials. Therefore, the signatures $\sigma_{\text{prep}}^j$ generated by the authorities during the election preparation now depend on $U$, $\hat{x}_j$, $\hat{y}_j$, and $\hat{\pi}_j$. Corresponding verification steps have been added to Prot. 6.1 by calling a second new algorithm $\text{CheckCredentialProofs}(\hat{\pi}, \hat{X}, \hat{Y}, i)$, which internally calls $\text{CheckCredentialProof}(\hat{\pi}_j, \hat{x}_j, \hat{y}_j)$ for checking the proofs $\hat{\pi}_j$ one after another.

Introducing the above proof generation and verification algorithms required some changes in the data flow of the election preparation phase. The most obvious change is the extended list of return values in Alg. 7.6. To achieve this change, we had to adjust the internal
structure of Alg. 7.6. The previous calls to Alg. 7.10 (GenSecretVoterData) and Alg. 7.11 (GenPublicVoterData) have been replaced by calls to two new internal algorithms Alg. 7.10 (GetCodes) and Alg. 7.11 (GenCredentials), and the two old internal algorithms have been removed. In our opinion, this change also improved the overall clarity of Alg. 7.6.

Recommendation 10.3

The problem raised here has already been solved in Version 1.3.2 of this document.

Recommendation 10.4

No protocol changes were made to address the problem that voters may not have enough discipline to check all verification codes of multiple blank votes. But we see a possibility of merging these codes into single codes, one for each possible number of blank votes. As this solution can be implemented as a usability measure on top of the voting protocol, i.e., without any changes to the current version, we have not yet made a concrete proposal. But we are planning to do so in the next version of this document.

Similarly, no protocol changes were necessary to address the point that the number of selections must be strictly smaller than the number of candidates. In Section 2.2, this special case has been excluded by definition. We added two footnotes in Sections 6.3.2 and 6.4.3 to increase the reader’s awareness of this important point.

Recommendation 10.5

In Section 6.3.1, we explicitly set $p'$ to $\hat{q}$. In Tables 6.1 and 8.1, we removed $p'$, i.e., $L_M$ now depends on $\hat{q}$. We also removed $p'$ from Table 8.3 and we skipped Tabs. 8.7, 8.11, and 8.15. Replacing $p'$ by $\hat{q}$ also implied corresponding changes in Algs. 7.7 to 7.9, 7.33 and 7.37. Similarly, the range of some input or output parameters changed in Algs. 7.6, 7.31, 7.34, 7.36 and 7.42.

Furthermore, following the explanation given in Recommendation 10.5, we introduced a new variable $y^* = y + y'$ mod $\hat{q}$ for improved clarity in Algs. 7.6, 7.36 and 7.38. In Alg. 7.38, this implied replacing the parameters $y$ and $y'$ by $y^*$. The explanatory text in Section 6.4.4 has been changed accordingly. To distinguish $y^*$ from $y$ (confirmation credential) and $y'$ (vote validity credential), we have introduced the new term called vote approval credential. We use it everywhere in the updated document for both the private value $y^*$ and the public value $\hat{y}$.

Recommendation 10.6

The privacy issue raised in Subsection 10.6 is very delicate, because it may occur even in case of a normal protocol execution. The problem affects mainly voters with restricted eligibility rights within a counting circle. In Version 1.4.2 of the protocol, votes submitted by voters with unrestricted voting rights remain secret within the anonymity set consisting of all members of the counting circle with unrestricted voting rights. If we assume that almost
all voters have unrestricted voting rights within their own counting circle, their vote privacy is protected almost perfectly. On the other hand, as voters with restricted eligibility define their own, possibly much smaller anonymity sets, their vote privacy is strongly restricted (or even entirely broken in extreme cases).

One way of avoiding this problem is to refrain from grouping the encrypted votes into a single vote in Alg. 7.45, i.e., to input the full set of encrypted candidate selections to the mix-net. This solution, which increases the size of the mix-net input from $N$ to $kN$ (assuming that all $N$ submitted ballots contain $k$ encrypted selections), is clearly not very attractive from a performance point of view.

The countermeasure that we implemented in Version 2.0 of the protocol is not perfect, but it will at least increase the size of the anonymity set to a satisfactory degree in most cases. The idea is to augment the number of encrypted selections of voters with restricted eligibility to the size of the ballots of regular voters from their counting circle. For this, we have introduced in Section 2.2.2 the vector $k^* = (k^*_1, \ldots, k^*_w)$, which defines the default ballot size of every counting circle, and the matrix $E^* = (e^*_{ij})_{wxt}$, which defines the default eligibility of the members of a given counting circle. In Alg. 7.45, we augment the $k^*_i$ encrypted selections in a ballot of voter $i$ to $k^*_i + k^*_{ij}$ encrypted selections by attaching $k^*_{ij}$ default candidates, one for every supplementary restriction in the voter’s eligibility matrix. This ensures that all votes from a given counting circle decrypt into the same amount of selections. Clearly, the number of added default candidates must then be subtracted from the final election result to compensate for increasing the size of some ballots.

To implement this idea, a vector $u = \{u_1, \ldots, u_n\}$ of default candidates has been added to the list of input parameters of Alg. 7.45. The election administrator defines $u$ as part of the election parameters. Accordingly, we have added $u$ into the information flow of Protocols 6.1, 6.7 and 6.8. This implied a modification of signature $\sigma_{\text{param}}$ in Section 6.6. We also extended Alg. 7.45 by calling two new internal algorithms GetDefaultEligibilityMatrix($w, w, E$) and GetDefaultCandidates($j, n, k, u$) and by returning a matrix $U$ as an additional output. This implied further minor modifications in Protocols 6.7 and 6.8, in which $U$ is sent to the bulletin board and tested for correctness, respectively. Additional explanations about this process have been added to Sections 2.2.2, 6.3.2 and 6.5.3. A new Section 6.4.5 has been written to better explain the computation of the final election outcome and illustrate it with an example.

Recommendation 10.7

In Alg. 7.7, the initialization of the set of values to exclude has been changed from $X \leftarrow \emptyset$ to $X \leftarrow \{0\}$. This change is necessary to avoid the revealing of the value $y' = A(0)$ in the very unlikely case of picking $x = 0$, even if the probability of this to occur is negligible.

Recommendation 10.8

Testing the consistency of the raw election outcome (which in the updated protocol consists of matrices $U$, $V$, and $W$) is not necessary under the assumption that at least one election authority is honest. Since performing these tests is not very expensive, the authors of [12] argue that there is little reasons to omit them. However, we believe that the responsibility
for performing such tests lies elsewhere. As suggested in [27], checking the consistency of the election data is an important test category of a verification software for offering a complete verification chain. We propose to include such tests there.

**Recommendation 10.9**

The problem described here is a known chosen-ciphertext, which occurs when encryptions and signatures are used in combination. According to [33, Section 12.9], the problem can be solved by including the identifier of the signing party into the ciphertext. Note that in the context of our protocol, the identifiers of all election authorities are assumed to be publicly known for a given election event $U$. Therefore, it is sufficient to include $U$ instead of the authority’s identifier into the ciphertext, i.e., we have added $U$ to the parameter list of Alg. 7.64, which now generates a triple $c = (c_1, c_2, c_3)$ instead of a pair $c = (c_1, c_2)$. Corresponding changes have been made to Alg. 7.65. For improved internal consistency of these algorithms, we have changed the co-domain of the invertible mapping $\phi$ from byte arrays to UCS strings. In the notation of Prot. 6.2, we have replaced $[d_j]$ by $c_j$. Furthermore, decrypting $c_j$ now results in two values $U'$ and $d_j$. An additional check $U = U'$ has been added to Prot. 6.2. If the test fails, the printing authority aborts the process. The inclusion of this test prevents the attack.

Along with the changes made to Algs. 7.64 and 7.65, we also slightly modified the signature generation and verification in Algs. 7.62 and 7.63 by adding $U$ to the parameter list. The only reason for this change is to maintain consistency among all four algorithms used for establishing channel security.

**Other Changes**

- In Section 1.2, we added a link to the sister project, in which formal definitions and security proofs have been worked out.
- In Section 10.2, we removed the remark that formal security proofs are missing and that web browser performance is a remaining open problem (due to recent results from another sister project).
- In Section 2.2.3, we added more detailed information about handling party-list elections with a special rule for invalid votes and about how to distinguish between vote abstentions and empty votes.
- In Section 3.1, we introduced new notations for selecting rows and columns in a matrix. We use them in Algs. 7.13, 7.36, 7.42, 7.54 and 7.58.
Nomenclature

\( \alpha \)  
Ballot

\( a \)  
Left-hand side of encrypted vote

\( A_F \)  
Alphabet for finalization codes

\( A_R \)  
Alphabet for verification codes

\( A_X \)  
Alphabet for voting codes

\( A_Y \)  
Alphabet for confirmation codes

\( \beta_j \)  
Reponse generated by authority \( j \)

\( \beta_i \)  
Reponses for voter \( i \)

\( b \)  
Right-hand side of encrypted vote

\( b_j' \)  
Partial decryptions by authority \( j \)

\( B \)  
Ballot list consisting of tuples \((i, \alpha, w)\) for each valid ballot \((i, \alpha)\)

\( B' \)  
Partial decryptions

\( \mathbb{B} \)  
Boolean set

\( \gamma \)  
Confirmation

\( c \)  
Index over counting circle \( \{1, \ldots, w\} \)

\( C \)  
Vector of candidate descriptions

\( C_i \)  
Candidate description

\( \delta_j \)  
Finalization generated by authority \( j \)

\( \delta_i \)  
Finalizations for voter \( i \)

\( d_{ij} \)  
Voting card data of voter \( i \) generated by authority \( j \)

\( d_j \)  
Voting card data generated by authority \( j \)

\( D \)  
Voting card data

\( \epsilon \)  
Deterrence factor

\( e_{ij} \)  
Eligibility of voter \( i \) in election \( j \)

\( e^*_{c,j} \)  
Default eligibility of members of counting circle \( c \) in election \( j \)

\( \tilde{e}_j \)  
List of shuffled encryptions generated by authority \( j \)

\( E^* \)  
Default eligibility matrix of counting circles

\( E \)  
Eligibility matrix

\( \tilde{E} \)  
Matrix of shuffled encryptions

\( FC_i \)  
Finalization code of voter \( i \)

\( g \)  
Generator of group \( \mathbb{G}_q \)

\( \hat{g} \)  
Generator of group \( \mathbb{G}_{\hat{q}} \)
\( \mathbb{G}_q \) Multiplicative subgroup of integers modulo \( p \) (of order \( q = \frac{p-1}{2} \))

\( \mathbb{G}_{\hat{q}} \) Multiplicative subgroup of integers modulo \( \hat{p} \) (of order \( \hat{q} \))

\( h \) Generator of group \( \mathbb{G}_q \)

\( h_i \) Generator of group \( \mathbb{G}_q \)

\( i \) Index over candidates \( \{1, \ldots, n\} \), index over voters \( \{1, \ldots, N_E\} \), index over submitted ballots \( \{1, \ldots, N_B\} \), index over confirmations \( \{1, \ldots, N_C\} \), index over encrypted votes \( \{1, \ldots, N\} \)

\( j \) Index over authorities \( \{1, \ldots, s\} \), index over selections \( \{1, \ldots, k\} \), index over elections \( \{1, \ldots, t\} \)

\( k_j \) Number of selections in election \( j \)

\( k_{ij} \) Number of selections of voter \( i \) in election \( j \)

\( k_c^* \) Number of selections of members of counting circle \( c \)

\( k_{F} \) String length of finalization codes

\( k_{R} \) String length of verification codes

\( k \) Number of selections in each election

\( k'_{i} \) Number of selections of voter \( i \) in each election

\( k^* \) Number of selections of members of counting circles

\( \lambda \) Security level

\( l \) Auxiliary index in iterations

\( \ell \) Output length of hash function (bits)

\( \ell_{F} \) Length of finalization codes (bits)

\( \ell_{R} \) Length of verification codes (bits)

\( \ell_X \) String length of voting code

\( \ell_Y \) String length of confirmation code

\( L \) Output length of hash function (bytes)

\( L_{F} \) Length of finalization codes (bytes)

\( L_{M} \) Length of OT messages (bytes)

\( L_{R} \) Length of verification codes (bytes)

\( \tau \) Security strength (integrity)

\( m \) Product of selected primes

\( m \) Products of selected primes

\( n \) Number of candidates

\( n \) Number of candidates in each election

\( N \) Number of valid votes

\( N_B \) Size of ballot list \( B \)

\( N_{C} \) Size of confirmation list \( C \)

\( N_E \) Number of eligible voters

\( \mathbb{N} \) Natural numbers

\( \mathbb{N}^+ \) Positive integers

\( \pi \) Ballot or confirmation NIZKP

\( \pi_j \) Public key proof of authority \( j \)
π<sub>j</sub>  Credential proof of authority \( j \)
π<sub>j'</sub>  Decryption proof of authority \( j \)
π<sub>j</sub>  Shuffle proof of authority \( j \)
π  Public key proofs
π  Credential proofs
π<sub>'</sub>  Decryption proofs
π  Shuffle proofs
\( p \)  Prime modulus of group \( G_{\hat{q}} \)
\( \hat{p} \)  Prime modulus of group \( \hat{G}_{\hat{q}} \)
\( p_{ij} \)  Point on polynomials of voter \( i \)
P<sub>i</sub>  Voting page of voter \( i \)
P  Matrix of points
P  Primes of points
pk  Public encryption key
pk<sub>j</sub>  Share of public encryption key
pk<sub>k</sub>  Shares of public encryption key
q  Order of group \( G_{\hat{q}} \)
\( \hat{q} \)  Order of group \( G_{\hat{q}} \), modulus of prime field \( \mathbb{Z}_{\hat{q}} \)
\( \hat{q}_x \)  Upper bound for private voting credentials
\( \hat{q}_y \)  Upper bound for private confirmation credentials
q  Selected primes
rc<sub>i</sub>  Verification codes of voter \( i \)
RC<sub>ij</sub>  Verification code of voter \( i \) for candidate \( j \) (string)
σ  Security strength (privacy)
s  Number of authorities
s<sub>j</sub>  Index of selected candidate
s  Vector of indices of selected candidates
S  Safe primes
S<sub>i</sub>  Voting card of voter \( i \)
sk<sub>j</sub>  Share of private decryption key
t  Number of elections in an election event
u<sub>cj</sub>  Number of default votes added for candidate \( j \) in counting circle \( c \)
u<sub>j</sub>  Default candidate of election \( j \)
u  Vector of default candidates
U  Unique election event identifier
U  Matrix of default votes added
v  Voter index
v<sub>ij</sub>  Single entry of the election result matrix
v  Vector of voter descriptions
V<sub>i</sub>  Voter description (first/last names, address, date of birth, etc.)
V  Election result matrix
\( w \) Number of counting circles
\( w_i \) Counting circle of voter \( i \)
\( w_{ic} \) Single entry of the counting circle matrix
\( \mathbf{w} \) Vector of counting circles assigned to voters
\( \mathbf{W} \) Counting circle matrix of raw election outcome
\( x_i \) Private voting credential of voter \( i \)
\( \hat{x}_i \) Public voting credential of voter \( i \)
\( X_i \) Voting code of voter \( i \)
\( y_i \) Private confirmation credential of voter \( i \)
\( y'_i \) Private vote validity credential of voter \( i \)
\( y^{\pi}_i \) Private vote approval credential of voter \( i \)
\( \hat{y}_i \) Public vote approval credential of voter \( i \)
\( Y_i \) Confirmation code of voter \( i \)
\( z_{ij} \) Randomization used in OT response by authority \( j \) for voter \( i \)
\( \mathbb{Z}_p^* \) Multiplicative group of integers modulo \( p \)
\( \mathbb{Z}_q \) Field of integers modulo \( q \)
\( \mathbb{Z}_{\hat{q}} \) Field of integers modulo \( \hat{q} \)
List of Tables

2.1. Election parameters for common types of elections. ........................................... 19

4.1. Byte array representation for different integers and different output lengths. ........ 25

6.1. List of security parameters derived from the principal security parameters. ....... 55
6.2. List of election parameters and constraints. ..................................................... 57
6.3. Overview of the protocol phases and sub-phases with the involved parties. ........ 63
6.4. Overview of the signatures generated during the protocol execution. ................. 73
6.5. Overview of the signatures verified during the election process. ....................... 74

7.1. Overview of general algorithms for specific tasks. ........................................... 76
7.2. Overview of algorithms and sub-algorithms of the pre-election phase. .............. 79
7.3. Overview of algorithms and sub-algorithms of the election phase. .................... 87
7.4. Overview of algorithms and sub-algorithms of the post-election phase. ............ 98
7.5. Overview of algorithms used to establish channel security. ............................ 108

8.1. Length parameters according to current NIST recommendations. ...................... 112
8.2. Hexadecimal representation of Euler’s number ................................................. 113
8.3. Groups and default generators for security level λ = 0. .................................... 114
8.4. The first 60 prime numbers in $G_q \subset \mathbb{Z}_p^*$ for $p$ and $q$ as defined in Table 8.3. 114
8.5. Groups and default generators for security level λ = 1. .................................... 115
8.6. The first 60 prime numbers in $G_q \subset \mathbb{Z}_p^*$ for $p$ and $q$ as defined in Table 8.5. 116
8.7. Groups and default generators for security level λ = 2. .................................... 116
8.8. The first 60 prime numbers in $G_q \subset \mathbb{Z}_p^*$ for $p$ and $q$ as defined in Table 8.7. 117
8.9. Group $G_q \subset \mathbb{Z}_p^*$ for security level λ = 2 with default generator $\hat{g}$. .......... 117
8.10. Groups and default generators for security level λ = 3. .................................. 118
8.11. The first 60 prime numbers in $G_q \subset \mathbb{Z}_p^*$ for $p$ and $q$ as defined in Table 8.10. 118
8.12. Group $G_q \subset \mathbb{Z}_p^*$ for security level λ = 3 with default generator $\hat{g}$. .......... 119
9.1. Common alphabets with different sizes and characteristics. . . . . . . . . . . . 121
9.2. Lengths of voting and confirmation codes for different alphabets and security
levels. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
9.3. Lengths of verification and finalization codes for different alphabets and se-
curity levels. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
9.4. Lengths of voting and confirmation codes for different alphabets and security
levels by reducing the required bit length from $2\tau$ to $\tau$ bits. . . . . . . . . . 124
9.5. Necessary computations in each of the four proposed methods compared to
the current protocol. Only the case of the voting credentials is shown. Similar
computations are necessary for the confirmation credentials. . . . . . . . . . . 129
9.6. 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. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
9.7. Recapitulation of major weaknesses and strength. . . . . . . . . . . . . . . . . 131

A.1. Overview of algorithm renumbering from Version 1.4.2 to Version 2.0 of this
document. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
List of Protocols

5.1. Two-round $OT^k_n$-scheme by Chu and Tzeng ........................................ 35
5.2. Two-round $OT^k_n$-scheme with sender privacy ........................................ 36
5.3. Two-round $OT^k_n$-scheme with sender privacy ........................................ 38

6.1. Election Preparation. ................................................................. 64
6.2. Printing of Voting Cards. ......................................................... 65
6.3. Key Generation ................................................................. 66
6.4. Candidate Selection ............................................................. 67
6.5. Vote Casting ................................................................. 68
6.6. Vote Confirmation ............................................................. 69
6.7. Mixing ................................................................. 71
6.8. Decryption ................................................................. 72
6.9. Tallying ................................................................. 72
List of Algorithms

4.1. MarkByteArray($B, m, m_{\text{max}}$) .................................................. 24
4.2. SetBit($B, i, b$) ................................................................. 25
4.3. ToByteArray($x$) .............................................................. 26
4.4. ToByteArray($x, n$) .......................................................... 26
4.5. ToInteger($B$) ............................................................... 27
4.6. ToString($x, k, A$) ........................................................... 28
4.7. ToInteger($S, A$) ............................................................ 28
4.8. ToString($B, A$) ............................................................ 29
4.9. RecHash$_L(v_1, \ldots, v_k)$ .................................................. 30

7.1. GetPrimes($n$) ............................................................. 77
7.2. IsMember($x$) .............................................................. 77
7.3. GetGenerators($n$) .......................................................... 78
7.4. GetNIZKPChallenge($y, t, \kappa$) .......................................... 78
7.5. GetNIZKPChallenges($n, y, \kappa$) ........................................ 78
7.6. GenElectorateData($n, k, E$) ............................................. 80
7.7. GenPoints($n, k$) .......................................................... 80
7.8. GenPolynomial($d$) ....................................................... 81
7.9. GetYValue($x, a$) .......................................................... 81
7.10. GetCodes($p$) ............................................................... 81
7.11. GenCredentials($y'$) ..................................................... 82
7.12. GenCredentialProof($x, y, \tilde{x}, \tilde{y}$) ........................... 82
7.13. CheckCredentialProofs($\pi, \tilde{X}, \tilde{Y}, i$) ...................... 83
7.14. CheckCredentialProof($\tilde{\pi}, \tilde{x}, \tilde{y}$) ......................... 83
7.15. GetPublicCredentials($\tilde{X}, \tilde{Y}$) .................................. 83
7.16. GetVotingCards($v, w, c, n, k, E, D$) .............................. 84
7.17. GetVotingCard($v, V, w, c, n, k, X, Y, FC, rc$) ............... 84
7.18. GenKeyPair() ...................................................... 85
7.19. GenPublicKeyProof(sk, pk) .................................. 85
7.20. CheckPublicKeyProofs(π, pk) .................................. 85
7.21. CheckPublicKeyProof(π, pk) .................................. 85
7.22. GetPublicKey(pk) .................................................. 86
7.23. GetVotingPage(v, v, w, c, n, k, E) .......................... 88
7.24. GenBallot(X, s, pk) .............................................. 88
7.25. GetSelectedPrimes(s) ............................................. 89
7.26. GenQuery(q, pk) ................................................... 89
7.27. GenBallotProof(x, m, r, ˚x, a, pk) ......................... 90
7.28. CheckBallot(v, α, pk, k, E, ˚x, B) ......................... 90
7.29. HasBallot(v, B) ........................................................ 91
7.30. CheckBallotProof(π, ˚x, a, pk) ................................. 91
7.31. GenResponse(v, a, pk, n, k, E, P) ......................... 92
7.32. GetPointMatrix(β, s, r) ........................................ 93
7.33. GetPoints(β, s, r) ............................................... 93
7.34. GetReturnCodes(s, P_s) ...................................... 94
7.35. CheckReturnCodes(rc, rc', s) ............................... 94
7.36. GenConfirmation(Y, P) .......................................... 94
7.37. GetValue(p) .......................................................... 95
7.38. GenConfirmationProof(y*, ˚y) ................................. 95
7.39. CheckConfirmation(v, γ, ˚y, B, C) ......................... 95
7.40. HasConfirmation(v, C) .......................................... 96
7.41. CheckConfirmationProof(π, ˚y) ............................... 96
7.42. GetFinalization(v, P, B) ...................................... 96
7.43. GetFinalizationCode(δ) ....................................... 96
7.44. CheckFinalizationCode(FC, FC') .......................... 97
7.45. GetEncryptions(B, C, n, k, u, w, E) .................... 99
7.46. GetDefaultEligibilityMatrix(w, w, E) ................. 100
7.47. GetDefaultCandidates(j, n, k, u) ....................... 100
7.48. GenShuffle(e, pk) ............................................... 101
7.49. GenPermutation(N) ............................................. 101
7.50. GenReEncryption(e, pk) ............................................. 101
7.51. GenShuffleProof(e, e, r, ψ, pk) ................................. 102
7.52. GenPermutationCommitment(ψ, h) ............................ 103
7.53. GenCommitmentChain(ù) ........................................... 103
7.54. CheckShuffleProofs(π, ě0, Ė, pk, i) ........................... 103
7.55. CheckShuffleProof(π, ě, pk) ...................................... 104
7.56. GetPartialDecryptions(e, sk) .................................... 104
7.57. GenDecryptionProof(sk, pk, e, b) .............................. 105
7.58. CheckDecryptionProofs(π', pk, e, B') ......................... 105
7.59. CheckDecryptionProof(π', pk, e, b') ......................... 106
7.60. GetDecryptions(e, B') ............................................. 106
7.61. GetVotes(m, n, w) .................................................. 107
7.62. GenSignature(sk, U, m) ........................................... 109
7.63. VerifySignature(pk, σ, U, m) .................................... 109
7.64. GenCiphertext(pk, U, m) ......................................... 110
7.65. GetPlaintext(sk, e) .................................................. 110


