Berner Fachhochschule (BFH), CH-2501 Biel, Switzerland

Analysis of the Cryptographic Implementation of the Swiss Post Voting Protocol

Philipp Locher, Rolf Haenni, Reto E. Koenig

July 19, 2019

On behalf of the Federal Chancellery

Management Summary

As a starting point of this mandate, we received a large amount of documents describing the cryptographic protocol, the design of the system, technical details of the implementation, and the results received from external reviewers. We also received a large portion of the current system's source code for inspection. The explicit goal of our mandate was to verify as precisely as possible if the cryptographic protocol is implemented adequately in the current system. For this, we accepted the alleged security properties of the protocol as given without looking closely at the formal proofs.

In our source code analysis, particular attention was given to the problems recently found in the context of the source code publication of the extended system. One specific problem allowed an attacker to bypass the individual verification mechanism in such a way that vote manipulations remain undetected. With respect to that specific problem, our analysis of the current system confirms that the problem has been repaired and that corresponding documents have been updated adequately.

In other parts of the system's source code, we encountered four major problematic areas. Provided that corresponding system components are controlled by an attacker, each of them has the potential of weakening the system's security properties.

- The first problem arises from injecting the cryptographic parameters into the system after generating them in an external process. If an attacker manages to control this process or to modify the injected parameters, various security problems may arise. As all system components adopt the injected parameters without performing any consistency checks, these problems are likely to remain undetected.
- The second problem results from the particular implementation of the tables, in which the short choice codes are stored. These tables keep track of the given voting option ordering, which implies that the server responding to a submitted ballot could deduce the selected voting options without decrypting the encrypted votes. By controlling this server, vote secrecy could be violated on a large scale.
- The third problem is a major deviation from the cryptographic protocol in one of the protocol's core algorithms, which is used in the process of casting a vote. This deviation undermines the formal security proofs, which remain conclusive only as long as a one-to-one correspondence exists between implementation and protocol.
- The fourth problem results from a lack of checking the input parameters of some cryptographic algorithms relative to their domain. A best practice in cryptographic protocol design is to perform such checks systematically. This increases the system's robustness against failures and attacks.

In the course of our analysis of both the cryptographic protocol and its implementation, we also encountered a number of minor inconsistencies, discrepancies, and deviations from best practices, which affects the overall robustness of the system.

Revision History

Revision	Date	Description
0.1	May 24, 2019	Document initialization.
0.2	June 3, 2019	Document structure added.
0.3	June 13, 2019	Section 1 finished.
0.4	June 26, 2019	Section 2 finished.
0.5	July 1, 2019	Section 3 finished.
0.6	July 2, 2019	Management summary, final editing.
1.0	July 2, 2019	Document delivered to Federal Chancellery.
1.1	July 19, 2019	Minor corrections.

Contents

1 Introduction

The Swiss Post received in 2017 a certificate for offering their online voting system to 50% of the electorate. This certificate confirms that corresponding requirements of the Ordinance on Electronic Voting (VEleS) are met, in particular those from Art. 4 related to individual verifiability [4]. Since then, the system has been in use in four cantons. In the meantime, the system has continuously been developed with the goal of meeting all VEleS requirements, including those of Art. 5 regarding complete verifiability. To obtain an extended certificate for offering the system up to 100% of the electorate, documentations and source code files have been been made public in February 2019 and a public intrusion test has been conducted. In the course of these actions, external reviewers have reported several flaws and vulnerabilities in the cryptographic design and implementation of the system, for example one that allows an attacker to circumvent the individual verification mechanism [14].

Although these findings have been made by inspecting the extended system, which is not yet used in practice, concerns have been raised that some of these flaws may also be present in the certified system, from which the extended system evolved. As a consequence, the use of the present system has been suspended on March 29, 2019. For reasons of disambiguation, this document only refers to the current system, which we will call *online voting system* (OVS). It is based on an *abstract voting protocol* (AVP), which is a high-level description of the cryptographic computations during the protocol execution and the information flow between the protocol participants. Detailed instructions for implementing the AVP are specified separately in the *voting protocol specification* (VPS), which serves as a bridge between the AVP and the *system's source code* (SSC). The relationship between these three levels of abstraction is depicted in Figure 1.1.

1.1 Goal of Mandate

Proving that a cryptographic protocol satisfies certain properties and implementing the protocol such that these properties remain valid are two totally different tasks. Assuming that the properties of the AVP are sufficiently strong to meet the VEleS requirements relative to individual verifiability, the goal of this mandate was to check if these properties are preserved in the actual implementation. For this, the additional documentation (particularly the VPS) and the SSC written in Java and JavaScript had to be analyzed and compared to the given AVP. Particular attention had to be been given to some implementation pitfalls known from the cryptographic literature or from earlier mistakes made in practical protocol implementations. Assuming that major discrepancies exist between the AVP and the SSC, which may critically weaken some of the protocol's alleged security properties, the purpose of this mandate was to localize them in the course of our analysis. On the other hand, independently of the actual findings, it is impossible to uncover all possible problems. For that, the amount of received documents and source code is too large compared to the relatively short duration of this mandate. Our analysis

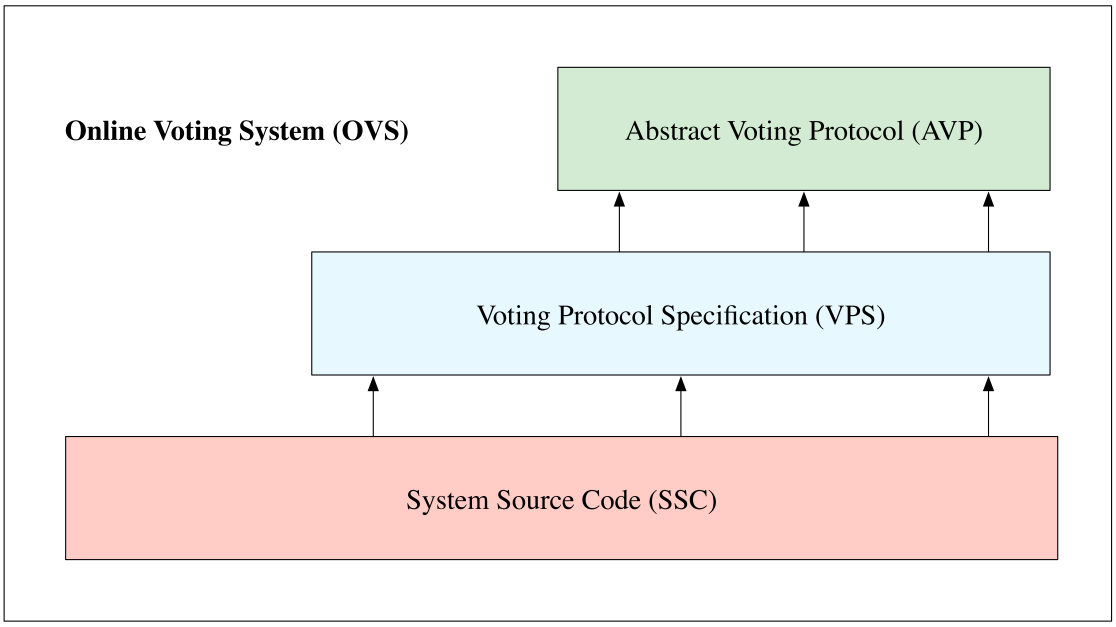

Online Voting System (OVS)
Abstract Voting Protocol (AVP)
Voting Protocol Specification (VPS)
System Source Code (SSC)

Figure 1.1: Overview of the three levels of abstraction of an online voting system. The solid arrows represent direct dependencies.

therefore focused on some of the most critical parts of the cryptographic protocol and its implementation.

1.2 Received Documents

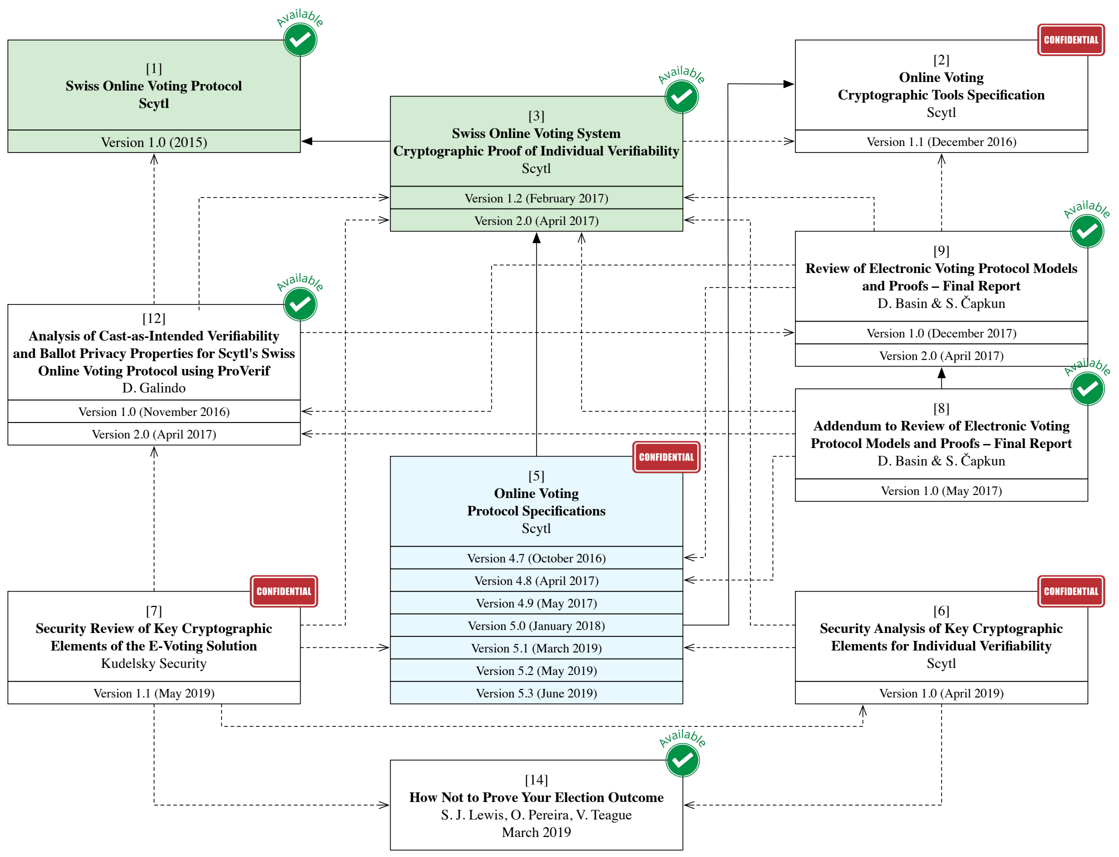
We received a large amount of documents describing the technical details and properties of the OVS, but most of them are not very relevant for building up a cryptographic understanding of the AVP and the VPS. In the limited given time frame, we focused on the documents depicted in Figure 1.2, which turned out to be the most relevant ones for conducting this mandate. Some documents exist in different versions, which we had to consider for obtaining an accurate picture of the protocol and the system as it exists today. While some documents are publicly available on the Swiss Post web site at https://www.post.ch, some others are confidential.

The document graph in Figure 1.2 shows the relationship between the depicted documents, for example direct references are shown as solid arrows. Together with the month/year of publication of corresponding document versions, we get a detailed picture of how these documents evolved over time. A modification of the original AVP description in [1] has been released as a new document with a different title in 2017 [3]. This document has undergone a major revision in April 2017 after feedback from external reviewers had been received [8, 9]. Corresponding modifications have been made at approximately the same time to the VPS document [5]. The resulting Version 1.2 of [3] and

Version 5.0 of [5] are the relevant documents for the certified system, which has been in use from early 2018 to spring 2019. A second round of modification has been triggered in 2019, when Lewis, Pereira, and Teague published a document describing the critical flaw they found in the extended system [14]. The flaw allows a client-side attacker to circumvent the individual verifiability mechanism, i.e., to submit a vote different from the voter's intention. In the aftermath of publishing this paper, an internal document analyzing the reported problem has been created by Scytl [6]. A similar analysis has been conducted by Kudelski Security on behalf of Swiss Post [7]. Both documents come to the conclusion, that the current and the extended OVS are affected in the same way by the encountered problem. The changes made to the latest Versions 5.1, 5.2, and 5.3 of [5] reflect the patches proposed in [6].

Regarding the set of public and non-public documents from Figure 1.1, we made the following observations:

- The description of the AVP in [1] is outdated. Compared to the current protocol description in [3], there are many significant differences, for example the modified set of protocol participants or the name changes of some values in the protocol's information flow. Given that [1] is still available on the Swiss Post web page, and according to its title, we expected it to be the most relevant AVP document. As a consequence, we invested several hours in reading and trying to understand its content. In [3], due to its unambiguous subtitle, we expected to find mainly the protocol's cryptographic proofs, not the latest version of the protocol itself.
- Some of the publicly available documentation has references to confidential documents, for example [2] refers to [3] and [8, 9] both refer to [5]. By holding back important background information, this creates an unfortunate situation to potential readers of public documents from the Swiss Post web site.
- The lack of strict and simple naming conventions (e.g., "Swiss Online Voting System" vs. "Online Voting System" vs. "Online Voting Product" vs. "Online Voting Platform" vs. "Scytl Online Voting" vs. "E-Voting Solution" vs. "EV Solution" vs. "Scytl sVote" vs. "Swiss Post-Scytl Software" vs. "Scytl Standard Software") makes the navigation through the documents unnecessarily complicated, especially for a person with an outsider's view.
- The description of the symbolic proofs in [12] refers to Version 1.2 of [3], not to the current Version 2.0. As mentioned above, some critical changes have been made to the protocol in 2017 as a response to the external review in [8, 9]. Although corresponding publication dates coincide, it remains unclear if these changes have been taken into account in the latest version of [12].
- The role and current state of [2] as a document accompanying the AVP is unclear. It seems that most parts of it have been merged into Version 5.0 of [5] (Appendix 7.2), but it is still referenced prominently by the current Version 1.1 of [3]. While [3] has been updated in 2019 as a response to the findings in [14], [2] has not been modified since 2016.



Available
[1]
Swiss Online Voting Protocol
Scytl
Version 1.0 (2015)
CONFIDENTIAL
[2]
Online Voting
Cryptographic Tools Specification
Scytl
Version 1.1 (December 2016)
Available
[3]
Swiss Online Voting System
Cryptographic Proof of Individual Verifiability
Scytl
Version 1.2 (February 2017)
Version 2.0 (April 2017)
Available
[9]
Review of Electronic Voting Protocol Models and Proofs – Final Report
D. Basin & S. Čapkun
Version 1.0 (December 2017)
Version 2.0 (April 2017)
Available
[12]
Analysis of Cast-as-Intended Verifiability and Ballot Privacy Properties for Scytl's Swiss Online Voting Protocol using ProVerif
D. Galindo
Version 1.0 (November 2016)
Version 2.0 (April 2017)
Available
[8]
Addendum to Review of Electronic Voting Protocol Models and Proofs – Final Report
D. Basin & S. Čapkun
Version 1.0 (May 2017)
CONFIDENTIAL
[5]
Online Voting
Protocol Specifications
Scytl
Version 4.7 (October 2016)
Version 4.8 (April 2017)
Version 4.9 (May 2017)
Version 5.0 (January 2018)
Version 5.1 (March 2019)
Version 5.2 (May 2019)
Version 5.3 (June 2019)
CONFIDENTIAL
[7]
Security Review of Key Cryptographic Elements of the E-Voting Solution
Kudelsky Security
Version 1.1 (May 2019)
CONFIDENTIAL
[6]
Security Analysis of Key Cryptographic Elements for Individual Verifiability
Scytl
Version 1.0 (April 2019)
Available
[14]
How Not to Prove Your Election Outcome
S. J. Lewis, O. Pereira, V. Teague
March 2019

Figure 1.2: Overview of available documents with solid arrows representing direct dependencies and dashed arrows representing references.

We conclude that the available documents contain a large number of unnecessary redundancies and inconsistencies.

1.3 Received Source Code

The initial kickoff meeting at the Swiss Federal Chancellery took place on May 16 and on June 3 we received certain parts of Version 1.4.5 of the SSC. Unfortunately, the missing parts prevented us from compiling and executing the code for testing. Given the complex project structure and the large amount of missing dependencies, navigating efficiently through the code in an IDE was very restricted. In the light of the vast amount of source files (around 40'000 files and folders), we requested a complete project structure including all dependencies, which then can be loaded more easily into a common IDE for testing and inspection. Additionally, we requested that breakpoints can be set in a debugging environment, such that important information about the actual program flow can be deduced from corresponding stack traces. Until the end of our mandate, not all of these requests have been fulfilled to our satisfaction.

Instead, two alternatives to a fully compilable, testable, and debuggable code base have been offered. First, we were given the opportunity to define breakpoints, for which stack traces and memory dumps were generated and delivered to us manually on June 7. Unfortunately, this worked only for three of the twenty breakpoints that we defined. After insisting getting stack traces for all breakpoints, we received on June 14 a laptop with an installed VPN connection to a debuggable system within the infrastructure of the system provider. Unfortunately, it was not possible to debug all parts of the system. This problem was solved on June 20, i.e., only ten days before the end of this mandate. Due to this delay in providing us with the necessary working environment, it was difficult to conduct our mission to the planned extent.

The received source code consists of five components, i.e., cryptolib, ov-channel, ov-client, ov-sdm, and scytl-math. Each of them contains a large number of sub-components, which are organized in a tree structure. Component and sub-components are defined as individual Maven projects (more than 100 in total). In addition to the source code that we received, there are two further components called ov-keystore and ov-logger, which we were told were irrelevant for inspecting the implemented cryptography. Another missing component is the "Admin Portal", which apparently is used to enrich an election definition with cryptographic parameters. We explicitly requested access to its source code on June 18, but our request has not been answered before the end of our mandate.

To the best of our knowledge, the received Version 1.4.5 of the SSC corresponds precisely to the latest Versions 5.1 to 5.3 of the system specification document [5]. From an outsider's perspective, it seems that [5] describes what has been implemented rather than SSC implements what has been specified. As such, we used [5] mainly as an accompanying document for understanding the received source code.

2 Cryptographic Protocol

The main point of orientation for our protocol analysis is the description of the AVP in Section 3 of [3]. Based on the formal security analysis in Section 4 of the same document, the symbolic proofs conducted as described in [12], and the external review of these documents in [8, 9], we assume that the protocol satisfies the required properties relative to individual verifiability and vote secrecy. Therefore, we performed our analysis without giving special attention to any of these accompanying documents, i.e., we considered the soundness of the protocol as given.

Any cryptographic voting protocol decomposes naturally into three distinct phases. The preparation of an election takes place during the *pre-election phase*, the submission of the ballots during the *election phase*, and the derivation of the final election result during the *post-election phase*. The protocol as presented in [3] is also structured in this way, except that the pre-election phase is further split into two sub-phases called *configuration* and *registration*. Besides, the election phase is called *voting phase* and the post-election phase is called *counting phase*. In our analysis of the protocol and its implementation, we follow its natural structure. Corresponding protocol diagrams are depicted in Figures 2.1 to 2.3. They show the involved protocol participants in each phase and the general information flow. For each protocol phase, and for each algorithm in each phase, we propose in this section a checklist of important cryptographic aspects to consider in an implementation of the AVP. In Section 3, these checklists will then be evaluated against the OVS source code that we received. In Section 2.1, we discuss some general protocol aspects, which must be implemented with maximum care.

In our discussion of the protocol, we try to use exactly the same terminology as in [3]. In particular, we consider the following protocol parties and refer to them using respective abbreviations:

- Election Authorities (EA),
- Bulletin Board Manager (BBM),
- Registrars (REG),
- Auditors (AUD),
- Voting Device (VD),
- Voter.

According to the Appendix A.1 of [3], EA and REG are implemented as a single party in the particular setup of the Swiss Post OVS. In general, merging two parties and corresponding trust assumptions may have a great impact on the protocol's security properties. Even if we have no specific objections against this design decision, we consider it a delicate deviation from the protocol (EA and REG are considered as separate parties in the formal and symbolic proofs in [3, 12]). We will not further question this decision, but by making a distinction between EA and REG in our analysis, we follow the description of the protocol.

2.1 General Protocol Aspects

By looking closely at the three phases of the protocol, we encountered some lack of precision at various points. For example, we found it difficult to understand exactly how information is exchanged during protocol runs. Apparently, by receiving messages from various parties and by storing some of the received data on the *bulletin board* BB, the BBM plays a central role in the communication model. But then it remains unclear whether "publishing" refers to sending a message to the BBM, which then forwards the message to the bulletin board, or to sending the message directly to the bulletin board. To the best of our understanding, we consider BBM as a gateway from and to the bulletin board, which itself can be seen as the BBM's local storage (as such it does not appear in our protocol diagrams as a separate protocol participant). Note that by possessing an important secret information, the *codes secret key* $\mathtt{C_{sk}}$, BBM plays another important role in the protocol (a role that formerly was called *return code generator* in [1]), for which it needs to be trusted.

Another general problem that we found is the inconsistency of the parameter lists and return values for some algorithms. For example, the algorithm Register requires the number k of candidates to fulfill its task, but k is not included explicitly in the list of parameters for the algorithm as specified. Similarly, according to our understanding of the protocol, Register must return the *short vote cast code* $\mathtt{sVCC^{id}}$, not the (long) vote cast code $\mathtt{VCC^{id}}$ as specified in the protocol's description. We found numerous such minor issues, which make the protocol's comprehensibility more complex than necessary. In each case, we tried to derive the correct functionality from our general understanding of the protocol or from the additional information presented in [5].

Based on the discussion of the following subsections, we propose applying the checklist of Table 2.1 to the current implementation. It contains various general protocol implementation issues, to which special attention should be drawn during the development of the system. Our analysis of the source code in Section 3.1 will be aligned along this checklist.

2.1.1 Cryptographic Setup

Fundamental from a cryptographic point of view is the generation of the cryptographic setup. Given that ElGamal is used as encryption scheme and that candidates are represented by prime numbers, the cryptographic setup mainly consists of a prime-order subgroup $\mathbb{G}_q \subset \mathbb{Z}_p^*$ for some safe prime $p = 2q + 1$ and an element $g \in \mathbb{G}_q \backslash \{1\}$ generating that subgroup. While knowing $\mathbb{G}_q$ and g is mandatory for almost every algorithm in the protocol, the protocol description says little about how, when, and by whom these parameters are generated. It is also not very clear how these parameters are communicated to the involved parties and how they relate to the security parameter 1^λ passed to the algorithms Setup and Register.

Nr.	**Description of Check**
G1	EA, REG, BBM, VD, and AUD all verify the consistency of the cryptographic parameters $p, q, g, 1^\lambda$ before using them.
G2	Each application of PBKDF2 generates keys K of sufficient length (λ bits or higher) and sufficient security ($2^{\lambda-\phi}$ iterations or higher).
G3	EA, REG, BBM, VD, and AUD perform membership and consistency tests for all algorithm parameters.
G4	Picking elements uniformly at random based on the output of a cryptographically secure PRNG is done properly.
G5	Group operations in $\mathbb{G}_q$, $\mathbb{Z}_q$, and $\mathbb{Z}_p^*$ are computed properly.
G6	Collision-resistance is preserved for hash values of multiple inputs and for input domains different from $\{0,1\}^*$.
G7	Whenever necessary, messages are properly encrypted and/or signed when transmitted from one party to another. Certificates for the involved public keys exist and are valid within the system's PKI.
G8	The messages exchanged during protocol runs contain exactly the specified values. Exchanging additional values is only allowed if they are completely irrelevant for the protocol's cryptographic functionality.

Table 2.1: List of general checks relative to the cryptographic setup, the membership and consistency of algorithm parameters, the generation of randomness, the proper computation of group operations, collision-resistant hashing, and the authenticity of received messages.

In an earlier version of the protocol [1], the task of generating suitable parameters was explicitly assigned to EA (as part of executing Setup). This is no longer the case in the current protocol version. Therefore, we assume that p, q, g, and 1^λ are generated outside of the protocol, i.e., the protocol parties obtain them as global constants, which need not to be exchanged during the protocol execution. However, to ensure that no attack can be conducted based on infiltrating weak parameters, we expect each party to test the consistency of these parameters. Here are the tests that we would expect:

- $p \in \mathbb{P}$ (with probability $1 - \frac{1}{2^\lambda}$ or higher), where $\mathbb{P}$ denotes the set all primes;
- $q \in \mathbb{P}$ (with probability $1 - \frac{1}{2^\lambda}$);
- $p = 2q + 1$;
- $\|p\|$ is consistent with 1^λ (according to current recommendations);
- p is not close to 2^k and 2^{k+1} for $k = \lfloor \log_2 p \rfloor$;
- $g \in \mathbb{G}_q \backslash \{1\}$ (e.g., by computing the Jacobi symbol).

Ideally, p is determined verifiably at random, such that all other parties can convince themselves that no hidden back doors have been installed [11].

Other cryptographic parameters such as the output length of the cryptographic hash function, the key size of the symmetric encryption scheme, the number of iterations of the key derivation function (KDF), and the length of the RSA signature keys should be consistent with 1^λ. If we assume that λ will be 128 or less, the proposed standards SHA256 for hashing, AES-GCM for symmetric encryption, and PBKDF2 for key derivation are legitimate choices.[1] Relative to the KDF, the number of iterations needs to be checked in the current implementation of the protocol. As a general rule of thumb, we assume that c iterations increase the security of the generated key by $\log_2 c$ bits.[2] Therefore, in order to reach λ bits of security, every application of the KDF needs to satisfy $\phi + \log_2 c \geqslant \lambda$ and $\kappa \geqslant \lambda$, where $\phi = \|\mathsf{pwd}\|$ denotes the bit-length of the input password pwd and $\kappa = \|K\|$ the bit-length of the generated key K. In other words, the number of iterations c should be set to $2^{\lambda-\phi}$ or higher.

2.1.2 Membership and Consistency Tests

Every single input parameter of the protocol's algorithms has a well-defined domain. Before calling an algorithm (or initially when running the algorithm), each party should therefore check that all parameters are members of the corresponding domain. For example, the parameter $\mathsf{C_{sk}}$ of the algorithm must be an element of $\mathbb{Z}_q$, where q denotes the order of the given group $\mathbb{G}_q$. In some cases, performing these membership tests may not be relevant for the protocol's security properties, but in some cases they are. Since many attacks based on infiltrating invalid parameters are known in the cryptographic literature, it is a best practice to perform such tests systematically in an implementation of a cryptographic protocol.

To illustrate the importance of such tests, suppose that EA select values $\{v_1, v_2, v_3\}$ in an election with $k = 3$ candidates $i \in \{1, 2, 3\}$, where $v_1, v_2 \in \mathbb{P}$ are prime, but $v_3 = v_1 v_2 \notin \mathbb{P}$ is composite. If none of the other parties checks the consistency of the set $\{v_1, v_2, v_3\}$ during the protocol execution, then by factoring the decrypted ballots in the Tally algorithm, Candidate 3 will not receive a single vote (every vote for Candidate 3 generates one vote for Candidate 1 and one vote for Candidate 2). Clearly, such an attack could be detected by looking at the number of votes contained in each ballot, but the damage could not be reversed entirely in every possible situation. Implementing corresponding tests $v_i \in \mathbb{P}$ (together with testing $v_i \in \mathbb{G}_q$) for every $i \in \{1, \ldots, k\}$ is

[1]While SHA256 and PBKDF2 are still widely used in practice, newer standards are available since many years. The hash algorithm SHA-3 exists since 2012 and the key derivation function Argon2 since 2015. Using the latest international standards is generally recommended in security-critical applications.

[2]In the light of attacks based on special hardware (GPU, FPGA), some may advocate an even more conservative rule of thumb. More recent methods such as Argon2 maximize resistance to such attacks by offering a time-memory trade-off.

therefore of uttermost importance.[3] Another important test guarantees that the product of the t largest values in the set $\{v_1, \dots, v_k\}$ is smaller than p (otherwise computing modulo p in $\mathbb{G}_q$ leads to a completely different factorization).

In the presence of multiple parameters, it may also happen that some parameters are inconsistent to each other. For example, if the $\mathsf{MixTally}(\mathcal{C}, \mathcal{C}', \pi_{\mathtt{mix}})$ algorithm is called with two input lists $\mathcal{C}$ and $\mathcal{C}'$ of different length (or if $\pi_{\mathtt{mix}}$ internally is inconsistent with the length of $\mathcal{C}$ and $\mathcal{C}'$), then obviously something has gone wrong somewhere in the process. To exclude attacks based on infiltrating inconsistent parameters, each party should perform corresponding consistency tests along with the membership tests mentioned above, ideally in a most systematic manner for all sets, lists, and vectors (relative to the number of candidates k, the number of choices t, the number of voters s, or the number of submitted ballots n).

2.1.3 Randomness Generation

Generating a sufficient amount of randomness in reasonable time is a difficult problem in many cryptographic applications. The usual solution consists in deriving pseudo-randomness deterministically from an initial random seed, which is extracted from a real randomness source. Clearly, the quality of the pseudorandomness depends both on the quality of the random seed and on the properties of the pseudorandom generator (PRNG) in use. While there are best practices for both, many things can go wrong in an actual implementation (the literature on attacks of real-world systems based on weak randomness is substantial). It is therefore necessary to implement current standards and best practices with greatest possible care (see for example NIST recommendations SP 800-90A, SP 800-90B, and SP 800-90C).

At various places of the AVP, elements from different sets must be picked uniformly at random. Translating the output of a PRNG (usually a sequence of random bits or bytes) into an algorithm for picking elements uniformly at random from various sets is another potential source for implementation mistakes. In particular cases, even tiny deviations from a uniform distribution may have an impact on the system's security. These algorithms must therefore be designed and implemented appropriately [13, 15].

2.1.4 Group Operations

In the AVP, operations of modular arithmetic appear in various groups. It is important to implement these operation properly, i.e., always using the right modulus. The most important group is the multiplicative subgroup $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of integers modulo p, where q denotes the group's prime order. Multiplications $x \times y$ for elements $x, y \in \mathbb{G}_q$ must therefore be implemented as $xy \bmod p$ and divisions $\frac{x}{y}$ as $xy^{-1} \bmod p$, where $y^{-1} \bmod p$

[3] Alternatively, it would not be difficult to let each party compute these values using a deterministic algorithm.

denotes the multiplicative inverse of y modulo p. Similarly, exponentiations x^z for non-negative exponents $z \in \mathbb{Z}$ must be implemented as $x^z \bmod p$, whereas exponentiations x^{-z} for negative exponents can be implemented either as $(x^{-1})^z \bmod p$ or $(x^z)^{-1} \bmod p$. For improved performance, exponents $z \notin \mathbb{Z}_q$ can be replaced by elements $z \bmod q \in \mathbb{Z}_q$. Generally, computations in the exponent can be implemented as operations in the prime-order field $\mathbb{Z}_q$, i.e., by computing additions, subtractions, multiplications, and divisions modulo q.

Multiplications, divisions, and exponentiations relative to RSA keys $pk = (e, n)$ or $sk = (d, n)$ are operations in the multiplicative group $\mathbb{Z}_n^*$ of integers modulo n. Negative exponents can be treated as explained above. Since the group order $\phi(n) = |\mathbb{Z}_n^*|$ is unknown, computations in the exponent cannot be optimized in the same way as above.

2.1.5 Collision-Resistant Hashing

Collision-resistance is the core property of a cryptographic hash function $H : \{0,1\}^* \rightarrow \{0,1\}^\ell$ with an output length of ℓ bits. It is assumed that current hash functions such as SHA256 or the members of the SHA3 family offer a sufficiently high degree of collision-resistance. While the security of a cryptographic protocol often relies strongly on this property of the underlying hash function, it is easily possible to violate collision-resistance in an actual implementation by not giving enough attention to some subtle details. A common mistake is to hash the concatenation of two inputs to the hash function, i.e., to compute $H(x||y)$ for inputs x and y of arbitrary length. Since many other values x', y' lead exactly to the same concatenation $x'||y' = x||y$, this particular implementation of hashing two values using a collision-resistant hash function H is no longer collision-resistant. Similar problems may arise even for single input values, if a non-injective encoding function is used for encoding them as bit strings.

In a carefully implemented OVS, these problems are avoided in every application of the underlying hash function. In the AVP, there are several applications of H with multiple inputs and input domains different from $\{0,1\}^*$, but neither the protocol description in [3] nor the additional detailed information in [5] specify the actual hashing implementation in sufficient detail (beyond referring to concatenation). Excluding the problems described above is a prerequisite for the correct functioning of the protocol.

2.1.6 Communication and Channel Security

By exchanging messages during the execution of the protocol, it is important for the involved parties to ensure that the transmission of the messages is protected. If insecure channels are used, the necessary security can be achieved using encryption and digital signatures. In both cases, public keys must be exchanged and validated beforehand, usually with the aid of a PKI based on certificates. Sending a message therefore means to encrypt it using the recipient's public key and to sign it using the sender's private key (preferably using the *sign-then-encrypt* method). Conversely, receiving a message means to perform the decryption using the recipient's private key and to validate the signature using the sender's public key. Long messages are usually encrypted in a hybrid manner using a symmetric encryption scheme.

In the given AVP, only messages containing secret values must be encrypted for transmission (all other values are "half-public" within the protocol). Transmitting secret values only appears in two particular cases during the setup procedure, when secret keys are transmitted from EA to REG and BBM (see Section 2.2). Obviously, ensuring the confidentiality of these keys is a prerequisite for enabling the system's security properties. In contrast, ensuring the integrity and authenticity of the exchanged messages is always important in a proper implementation of the system, independently of the actual content of the message.

In addition to guaranteeing the right channel security properties, it is crucial for the proper functioning of the protocol that parties do not exchange messages other than the ones specified in the AVP. For example, sending the signature of a message together with the encrypted message (known as the *encrypt-and-sign* approach) may completely break the message's confidentiality, independently of the properties of the underlying encryption scheme. Other subtle problems may arise whenever additional messages are exchanged. In extreme cases, they may break up the security properties of a carefully designed system.

2.2 Pre-Election Phase

In the pre-election phase as depicted in Figure 2.1, only EA and REG are involved in an active role. By running the Setup algorithm, EA generates two asymmetric key pairs $(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}})$ and $(\mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}})$ and one secret key $\mathtt{C}_{\mathtt{sk}}$. The first private decryption key $\mathtt{EB}_{\mathtt{sk}}$ is kept secret by EA, the second private signature key $\mathtt{VCCs}_{\mathtt{sk}}$ is sent to REG, and the secret key $\mathtt{C}_{\mathtt{sk}}$ is sent to both REG and BBM.[4] By running the Register algorithm, REG generates the data related to each voting card and forwards respective parts of the data to BBM and the voters. Therefore, it is assumed that the number s of generated voting cards corresponds to the total number of eligible voters. Note that the protocol remains unclear about how REG determines s. We therefore simply assume that REG selects s according to the number of entries in the electoral roll, which is available from an offline source.

In our attempt of understanding the pre-election phase in every detail, we encountered some inconsistencies in the formal description. Here is a list of the issues we found and some remarks of how we decided to best deal with them:

[4]We are not aware of any convincing reason for REG and BBM not to generate $\mathtt{VCCs}_{\mathtt{sk}}$ and $\mathtt{C}_{\mathtt{sk}}$ themselves. Letting EA generating these keys may result in unnecessary impersonation attacks. Generally, generating private keying material by other parties is against best practices in cryptographic protocol design.

- Defining the voting options as a set $\{v_1, \ldots, v_k\}$ is insufficient, because then the assignment of the values v_i to the k candidates is ambiguous. We propose to specify these values as a vector $(v_1, \ldots, v_k)$ instead of a set, which defines an explicit order. Option v_i is then assigned to Candidate i for all $1 \leqslant i \leqslant k$.
- The number t of voting options that each voter can select is an important additional election parameter. We assume that it is specified by EA, which then sends it to BBM.
- The *result function* ρ is never used in the protocol. Therefore, it is not necessary for the EA to specify it during the setup phase.
- REG needs to know the number s of voting cards to generate. This value is an important additional election parameter, which we assume is given to REG from an offline source.
- The vector $(v_1, \ldots, v_k)$ is an indispensable additional input parameter of the Register algorithm.
- Obtaining $\mathtt{VCC}^{\mathtt{id}}$ as a return value from calling Register is obviously a mistake in the description of the algorithm. The correct return value is $\mathtt{sVCC}^{\mathtt{id}}$.
- Sending $\{(v_i, \mathtt{sCC}_i^{\mathtt{id}})\}_{i=1}^k$ to the voter makes no sense, because assigning values v_i (prime numbers from $\mathbb{G}_q$) properly to candidates is impossible for humans. Therefore, we propose that Register returns a vector $(\mathtt{sCC}_i^{\mathtt{id}})_{i=1}^k$ instead of a set, which respects the order of the list of official candidates in the same way as their representations $(v_1, \ldots, v_k)$.
- Since REG is responsible for filling up the list ID with the data related to corresponding voting cards and publishing it on the bulletin board, we propose that REG (not EA) is also responsible for initializing this list.

All discrepancies between the description and our understanding of the protocol are highlighted in the protocol diagram of Figure 2.1.

In the following subsections, we will have a closer look at the two pre-election algorithms Setup and Register. The checklist of Table 2.2 is the result of this investigation. It refers to the points that we consider crucial when implementing these algorithms (in addition to the general points listed in Table 2.1).

2.2.1 Algorithm Setup

The private decryption key $\mathtt{EB_{sk}}$ and the codes secret key $\mathtt{C_{sk}}$ are both picked uniformly at random from $\mathbb{Z}_q$, whereas $\mathtt{EB_{pk}} = g^{\mathtt{EB_{sk}}}$ is computed within the encryption group $\mathbb{G}_q$. Both steps fall under the umbrella of the general checks G4 and G5 of Table 2.1, i.e., no additional specific checks must be defined. The same holds for the cryptographic setup of the group $\mathbb{G}_q$ and the generator g.

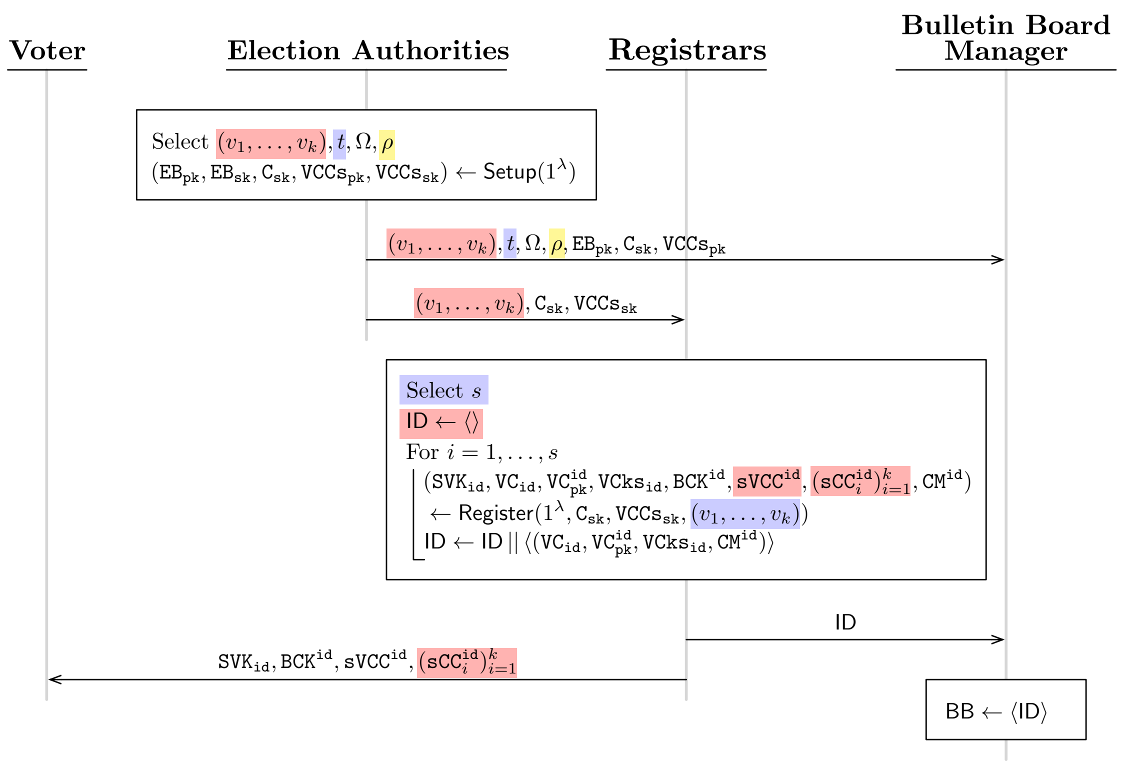

Voter
Election Authorities
Registrars
Bulletin Board Manager
Select $(v_1, \dots, v_k), t, \Omega, \rho$
$(\mathtt{EB}_{\mathtt{pk}}, \mathtt{EB}_{\mathtt{sk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}}) \leftarrow \mathsf{Setup}(1^\lambda)$
$(v_1, \dots, v_k), t, \Omega, \rho, \mathtt{EB}_{\mathtt{pk}}, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{pk}}$
$(v_1, \dots, v_k), \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}$
Select s
$\mathsf{ID} \leftarrow \langle\rangle$
For $i = 1, \dots, s$
$(\mathtt{SVK}_{\mathtt{id}}, \mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, (\mathtt{sCC}^{\mathtt{id}}_i)^k_{i=1}, \mathtt{CM}^{\mathtt{id}})$
$\leftarrow \mathsf{Register}(1^\lambda, \mathtt{C}_{\mathtt{sk}}, \mathtt{VCCs}_{\mathtt{sk}}, (v_1, \dots, v_k))$
$\mathsf{ID} \leftarrow \mathsf{ID} \,||\, \langle(\mathtt{VC}_{\mathtt{id}}, \mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}, \mathtt{VCks}_{\mathtt{id}}, \mathtt{CM}^{\mathtt{id}})\rangle$
ID
$\mathtt{SVK}_{\mathtt{id}}, \mathtt{BCK}^{\mathtt{id}}, \mathtt{sVCC}^{\mathtt{id}}, (\mathtt{sCC}^{\mathtt{id}}_i)^k_{i=1}$
$\mathsf{BB} \leftarrow \langle\mathsf{ID}\rangle$

Figure 2.1: Overview of the pre-election sub-protocol. Deviations from the protocol description in [3] are highlighted in red (modified parameters), blue (additional parameters), and yellow (unused parameters).

Nr.	**Algorithm**	**Description of Check**
S1	Setup	RSA key generation is implemented properly.
S2	Register	The algorithm Register is called exactly s times.
S3	Register	The key derivation function δ is implemented properly. The two salts `IDseed` and `KEYseed` are different.
S4	Register	The symmetric encryption of $\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}$, $\mathtt{sCC}^{\mathtt{id}}_i$, and $(\mathtt{sVCC}^{\mathtt{id}} \| S_{\mathtt{VCC}^{\mathtt{id}}})$ is implemented properly using a current standard such as AES-GCM.
S5	Register	The keyed pseudo-random function f_k is implemented properly, for example as HMAC based on SHA256.
S6	Register	The randomly selected short choice codes $\mathtt{sCC}^{\mathtt{id}}_i$ are checked for uniqueness.
S7	Register	The tables $\mathtt{CM}_{\mathtt{id}}$ are shuffled before sending them as part of ID to BBM. The shuffling algorithm selects the permutation uniformly at random from all $k!$ possible permutations.
S8	Register	The computation of the values $\mathtt{CC}^{\mathtt{id}}_i$ and $\mathtt{VCC}^{\mathtt{id}}$ is implemented according to the protocol.
S9	Register	The creation of the signature $S_{\mathtt{VCC}^{\mathtt{id}}}$ is implemented properly according to the RSA-PSS standard.

Table 2.2: List of checks relative to the pre-election algorithms Setup and Register.

The signature key pair $(\mathtt{VCCs}_{\mathtt{pk}}, \mathtt{VCCs}_{\mathtt{sk}})$ are RSA keys of the form $\mathtt{VCCs}_{\mathtt{pk}} = (e, n)$ and $\mathtt{VCCs}_{\mathtt{sk}} = (d, n)$ for values $n = pq$ and $e, d \in \mathbb{Z}_{\phi(n)}$ satisfying $ed \bmod \phi(n) = 1$. Generating RSA keys is a subtle procedure with several pitfalls. Most importantly, $p, q \in \mathbb{P}$ should primes of equal or almost equal bit length, which are picked uniformly at random, but such that they do not lie too close to each other. If primality of p and q is tested using a probabilistic primality test, then the algorithm's failure probability should be less than $\frac{1}{2^\lambda}$. Furthermore, $\|pq\|$ should depend on the security parameter 1^λ and be consistent with current key length recommendations (for example 2048 bits or more for $\lambda = 112$). For improved efficiency, small (but not too small) values e are acceptable, for example a constant value such as $e = 65537$.

2.2.2 Algorithm Register

This is one of the most critical algorithms in the AVP. Its goal is to prepare the cryptographic data necessary to achieve individual verifiability in the voting process. The algorithm is thus called for all s voters and in each run, the output will always be a different one. Obviously, it is critical to call Register exactly s times, i.e., at least s times to provide a voting card to every voter and at most s times to avoid ballot stuffing.

The first step of $\mathsf{Register}$ is the generation of the random *start voting key* $\mathtt{SVK_{id}}$, which is the value that the voter enters to start the voting process. Two values $\mathtt{VC_{id}}$ and $\mathtt{KSpwd_{id}}$ are derived from $\mathtt{SVK_{id}}$ using the key derivation algorithm δ (see check G2 in Table 2.1) with two different salts called $\mathtt{IDseed}$ and $\mathtt{KEYseed}$, respectively.[5] The value $\mathtt{KSpwd_{id}}$ is then used as a symmetric key to encrypt the private key of an ElGamal key pair $(\mathtt{VC^{id}_{pk}}, \mathtt{VC^{id}_{sk}})$ into a *keystore* $\mathtt{VCks_{id}}$. During vote casting, knowledge of $\mathtt{SVK_{id}}$ is sufficient to derive $\mathtt{VC_{id}}$ and $\mathtt{KSpwd_{id}}$ and therefore to decrypt the keystore into $\mathtt{VC^{id}_{sk}}$. This is the protocol's main voter authentication mechanism.

In the second step of $\mathsf{Register}$, *long choice codes* $\mathtt{CC}_i^{\mathtt{id}}$ and *short choice codes* $\mathtt{sCC}_i^{\mathtt{id}}$ are generated for all voting options $i \in \{1, \ldots, k\}$. The computation of $\mathtt{CC}_i^{\mathtt{id}}$ involves an exponentiation in $\mathbb{G}_q$ and an application of the keyed pseudo-random function f_k (implemented as HMAC based on SHA256). Corresponding short choice codes $\mathtt{sCC}_i^{\mathtt{id}}$ are picked at random and checked for uniqueness. They are encrypted using $\mathtt{CC}_i^{\mathtt{id}}$ as symmetric encryption key and pairs $(H(\mathtt{CC}_i^{\mathtt{id}}), \mathsf{Enc^s}(\mathtt{sCC}_i^{\mathtt{id}}, \mathtt{CC}_i^{\mathtt{id}}))$ are stored in a table $\mathtt{CM_{id}}$. This table will be used during vote casting to map long choice codes $\mathtt{CC}_i^{\mathtt{id}}$ into short codes $\mathtt{CC}_i^{\mathtt{id}}$, which are then presented to the voter. To guarantee vote secrecy in this process, it is crucial to shuffle this table, because otherwise submitted votes can be easily linked to the selected voting options using only the order of the entries in $\mathtt{CM_{id}}$. Surprisingly, this important aspect is not discussed in [3].

The table $\mathtt{CM_{id}}$ contains an additional entry for corresponding *vote cast codes* $\mathtt{VCC^{id}}$ (long) and $\mathtt{sVCC^{id}}$. Similarly to $\mathtt{CC}_i^{\mathtt{id}}$, $\mathtt{VCC^{id}}$ is derived from a randomly picked value $\mathtt{BCK^{id}}$ using the keyed pseudo-random function f_k. This computation involves mapping $\mathtt{BCK^{id}}$ into an element of $\mathbb{G}_q$ (by squaring it modulo p) and raise it to the power of $\mathtt{VC^{id}_{sk}}$. A RSA signature $S_{\mathtt{VCC^{id}}}$ of $\mathtt{sVCC^{id}}$ is created using the private RSA key $\mathtt{VCCs_{sk}}$. The signature is added to the entry for $\mathtt{VCC^{id}}$ in $\mathtt{CM_{id}}$ (using concatenation).[6]

2.3 Election Phase

The election phase can be regarded as the protocol's core. It consists of three communication round trips from the voter (respectively the VD) to the BBM and back. The first round (upper part of Figure 2.2) realizes the voter authentication, which consists in presenting the *start voting key* $\mathtt{SVK_{id}}$ to the BBM. In case this key exists in the list ID, the BBM returns the voter's personal key store $\mathtt{VCks_{id}}$ to the VD, from which the private *vote casting key* $\mathtt{VC^{id}_{sk}}$ is extracted. In the second round (middle part of Figure 2.2), $\mathtt{VC^{id}_{sk}}$ is used to submit the ballot $\mathtt{V}$ containing an encryption of the voter's choices $(j_1, \ldots, j_t)$ to the BBM. Along with this encryption, $\mathtt{V}$ contains so-called *partial choice codes* $\mathtt{pCC}_i^{\mathtt{id}}$,

[5]The salt of key derivation function and the seed of a PRNG should not be mixed up. These are two completely different concepts and serve for different purposes.

[6]The purpose of this signature remains unclear in [3], especially because no signatures are created for the short choice codes $\mathtt{sCC}_i^{\mathtt{id}}$.

from which the BBM can compute the long choice codes $\mathtt{CC}_i^{\mathtt{id}}$ in the same way as the Register algorithm during the pre-election phases. $\mathtt{V}$ also contains three different zero-knowledge proofs $\pi_{\mathtt{sch}}$, $\pi_{\mathtt{exp}}$, and $\pi_{\mathtt{pleq}}$, which tie all submitted values together. If the ballot is well-formed and the proofs are all valid, BBM uses the voter's mapping table $\mathtt{CM}_{\mathtt{id}}$ to derive short choice codes $\mathtt{sCC}_i^{\mathtt{id}}$ from the long ones. They are sent back to VD, which presents them to the voter for comparing them with the codes printed on the voting card. In case of a full match, the voter initiates the third round (lower part of Figure 2.2) by entering the *ballot casting key* $\mathtt{BCK}^{\mathtt{id}}$ to confirm the submitted ballot. Using $\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}$, VD computes the *confirmation message* $\mathtt{CM}^{\mathtt{id}}$, which is forwarded to the BBM. In a similar way as for the choice codes, BBM first computes the *long vote cast code* $\mathtt{VCC}^{\mathtt{id}}$ and then selects the *short vote cast code* $\mathtt{sVCC}^{\mathtt{id}}$ from the mapping table $\mathtt{CM}_{\mathtt{id}}$. Sending $\mathtt{sVCC}^{\mathtt{id}}$ to VD and presenting it to the voter is the last step of this phase.

Similar to the pre-election phase, we also encountered some inconsistencies in the formal description of the election phase. Here is the list of the issues we found and some remarks of how we decided to best deal with them:

- The voting device requires $\mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}$, $\mathtt{EB}_{\mathtt{pk}}$, and $(v_1, \ldots, v_k)$ for calling the CreateVote algorithm.
- The voter's choices are best represented by the set $\{j_1, \ldots, j_t\}$ of the indices of all selected candidates. Assuming from voters to provide corresponding prime numbers $v_{j_i} \in \mathbb{G}_q$—as suggested in the protocol description—is not realistic from a usability point of view. The set $\{j_1, \ldots, j_t\}$ is therefore a more appropriate input to the CreateVote algorithm.
- Algorithm ProcessVote requires $\mathtt{EB}_{\mathtt{pk}}$ as additional parameter.
- Algorithm CreateCC requires t as additional parameter.
- Parameters $\mathtt{VC}_{\mathtt{id}}$ and $\mathtt{V}$ are unused in the algorithm Confirm.
- Updating the BB with entries $(\overline{\mathtt{sVCC}^{\mathtt{id}}}, \overline{S_{\mathtt{VCC}^{\mathtt{id}}}})$ does not link them to corresponding entries $(\mathtt{VC}_{\mathtt{id}}, \mathtt{V})$ containing the votes. We propose to use $\mathtt{VC}_{\mathtt{id}}$ as identifier for every type of entry in BB.
- Each ballot $\mathtt{V}$ created by CreateVote contains three different non-interactive zero-knowledge proofs. We observed that the first proof is redundant in the light of the two other proofs (see Footnote 7).

We also detected a subtle problem with the proposed way of responding to incoming ballot submissions. In the procedure depicted in the middle part of Figure 2.2, suppose that the submitted vote $\mathtt{V}$ passes all tests of ProcessVote, but not all tests of CreateCC (if two referendums are held in parallel, such a ballot can be constructed easily, for example by submitting two votes to the first and zero votes to the second referendum). While such a ballot is clearly invalid, nothing speaks against allowing the voter to submit another ballot, for example from a different voting device. But since the proposed procedure updates BB before calling CreateCC, re-submitting a (valid) ballot would be

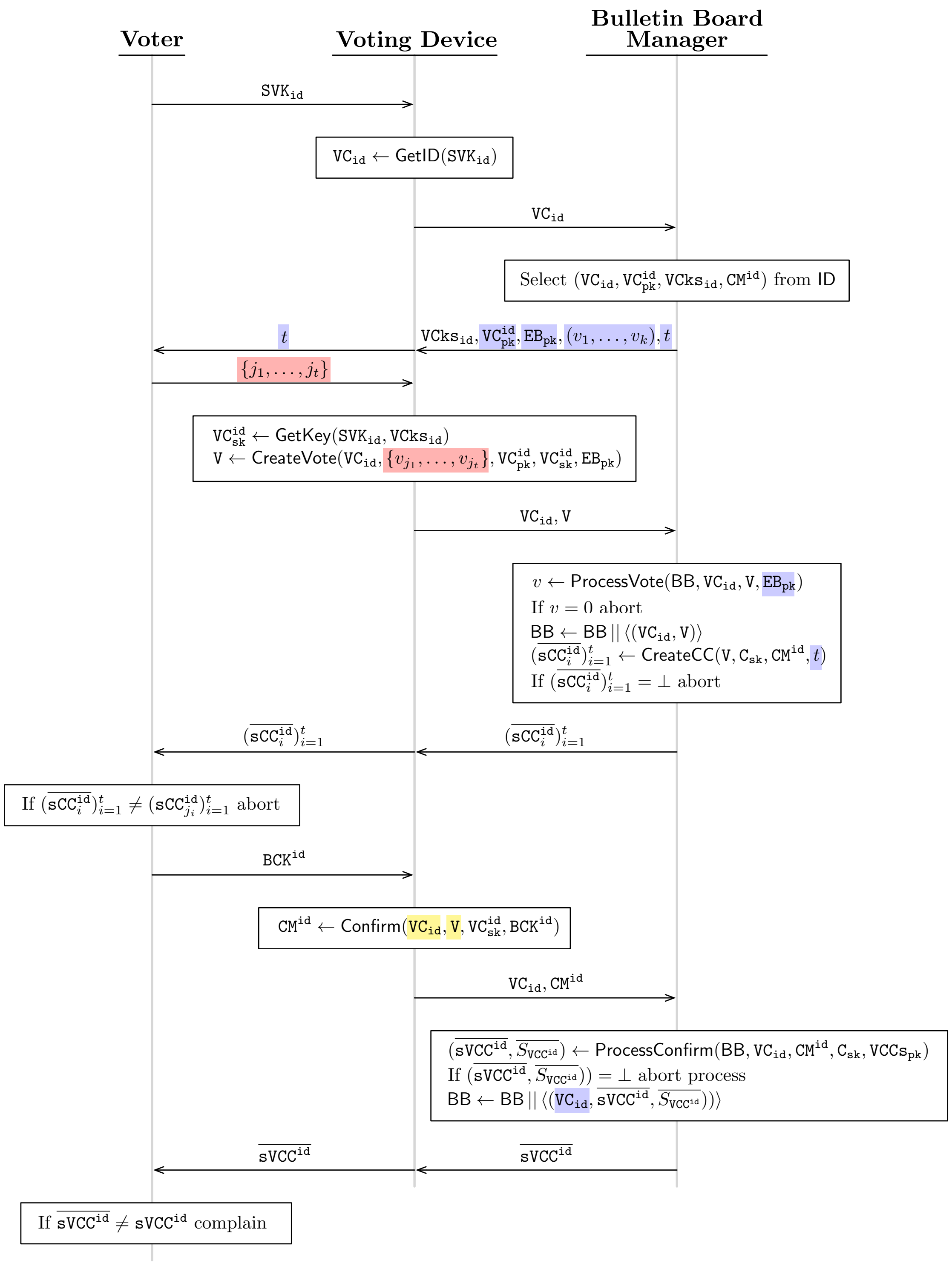

Voter
Voting Device
Bulletin Board Manager
$\mathtt{SVK_{id}}$
$\mathtt{VC_{id}} \leftarrow \mathsf{GetID}(\mathtt{SVK_{id}})$
$\mathtt{VC_{id}}$
Select $(\mathtt{VC_{id}}, \mathtt{VC^{id}_{pk}}, \mathtt{VCks_{id}}, \mathtt{CM^{id}})$ from ID
t
$\mathtt{VCks_{id}}, \mathtt{VC^{id}_{pk}}, \mathtt{EB_{pk}}, (v_1, \ldots, v_k), t$
$\{j_1, \ldots, j_t\}$
$\mathtt{VC^{id}_{sk}} \leftarrow \mathsf{GetKey}(\mathtt{SVK_{id}}, \mathtt{VCks_{id}})$
$\mathtt{V} \leftarrow \mathsf{CreateVote}(\mathtt{VC_{id}}, \{v_{j_1}, \ldots, v_{j_t}\}, \mathtt{VC^{id}_{pk}}, \mathtt{VC^{id}_{sk}}, \mathtt{EB_{pk}})$
$\mathtt{VC_{id}}, \mathtt{V}$
$v \leftarrow \mathsf{ProcessVote}(\mathsf{BB}, \mathtt{VC_{id}}, \mathtt{V}, \mathtt{EB_{pk}})$
If $v = 0$ abort
$\mathsf{BB} \leftarrow \mathsf{BB} \,||\, \langle(\mathtt{VC_{id}}, \mathtt{V})\rangle$
$(\overline{\mathtt{sCC}^{\mathtt{id}}_i})^t_{i=1} \leftarrow \mathsf{CreateCC}(\mathtt{V}, \mathtt{C_{sk}}, \mathtt{CM^{id}}, t)$
If $(\overline{\mathtt{sCC}^{\mathtt{id}}_i})^t_{i=1} = \perp$ abort
$(\overline{\mathtt{sCC}^{\mathtt{id}}_i})^t_{i=1}$
$(\overline{\mathtt{sCC}^{\mathtt{id}}_i})^t_{i=1}$
If $(\overline{\mathtt{sCC}^{\mathtt{id}}_i})^t_{i=1} \neq (\mathtt{sCC}^{\mathtt{id}}_{j_i})^t_{i=1}$ abort
$\mathtt{BCK^{id}}$
$\mathtt{CM^{id}} \leftarrow \mathsf{Confirm}(\mathtt{VC_{id}}, \mathtt{V}, \mathtt{VC^{id}_{sk}}, \mathtt{BCK^{id}})$
$\mathtt{VC_{id}}, \mathtt{CM^{id}}$
$(\overline{\mathtt{sVCC^{id}}}, \overline{S_{\mathtt{VCC^{id}}}}) \leftarrow \mathsf{ProcessConfirm}(\mathsf{BB}, \mathtt{VC_{id}}, \mathtt{CM^{id}}, \mathtt{C_{sk}}, \mathtt{VCCs_{pk}})$
If $(\overline{\mathtt{sVCC^{id}}}, \overline{S_{\mathtt{VCC^{id}}}})) = \perp$ abort process
$\mathsf{BB} \leftarrow \mathsf{BB} \,||\, \langle(\mathtt{VC_{id}}, \overline{\mathtt{sVCC^{id}}}, \overline{S_{\mathtt{VCC^{id}}}}))\rangle$
$\overline{\mathtt{sVCC^{id}}}$
$\overline{\mathtt{sVCC^{id}}}$
If $\overline{\mathtt{sVCC^{id}}} \neq \mathtt{sVCC^{id}}$ complain

Figure 2.2: Overview of the election sub-protocol. Deviations from the protocol description in [3] are highlighted in red (modified parameters), blue (additional parameters), and yellow (unused parameters).

rejected already by ProcessVote. In other words, the voter unnecessarily gets blocked by a previously submitted invalid vote.

In the following subsections, we will have a closer look at the algorithms executed during the election phase. The two checklists of Tables 2.3 and 2.4 are the results of our analysis. The first refers to the algorithms executed by VD and the second to the algorithms executed by BBM. They describe to the points that we consider crucial when implementing these algorithms (in addition to the general points listed in Table 2.1). Note that all VD algorithms are implemented using web-client technologies.

Nr.	**Algorithm**	**Description of Check**
V1	GetID	The key derivation function δ is implemented properly using the same current standard as in Register and the same salt `IDseed`.
V2	GetKey	The key derivation function δ is implemented properly using the same current standard as in Register and the same salt `KEYseed`.
V3	GetKey	The decryption of $\mathtt{VCks}_{\mathtt{id}}$ using the symmetric key $\mathtt{KSpwd}_{\mathtt{id}}$ is implemented properly using the same current standard as in Register.
V4	Confirm	The computation of the confirmation message $\mathtt{CM}^{\mathtt{id}}$ is implemented according to the protocol. It corresponds (partially) to the computation of $\mathtt{VCC}^{\mathtt{id}}$ in Register.
V5	CreateVote	The ElGamal encryption c is implemented properly using a fresh randomization r.
V6	CreateVote	The computation of the partial choice codes $\mathtt{pCC}_i^{\mathtt{id}}$ is implemented according to the protocol.
V7	CreateVote	The modified encrypted vote $\tilde{c}$ is derived from c according to the protocol.
V8	CreateVote	The non-interactive zero-knowledge proofs $\pi_{\mathtt{sch}}$, $\pi_{\mathtt{exp}}$, and $\pi_{\mathtt{pleq}}$ are generated according to the protocol (see Table 2.5).
V9	CreateVote	All non-interactive zero-knowledge proofs are implemented properly. In particular, the Fiat-Shamir heuristic is applied to all public values and commitments. Each proof is generated using a fresh randomization $s \in \mathbb{Z}_q$.

Table 2.3: List of checks relative to the client-side algorithms of the election phase, which are executed by VD.

Nr.	Algorithm	Description of Check
B1	ProcessVote	Checking the existence of a previously submitted ballot with the same $\mathtt{VC_{id}}$ in BB is conducted properly.
B2	ProcessVote	Checking the existence of an entry for $\mathtt{VC_{id}}$ in ID is conducted properly. The public key from that entry must correspond to the submitted value $\mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}$.
B3	ProcessVote	The verification of the non-interactive zero-knowledge proofs $\pi_{\mathtt{sch}}$, $\pi_{\mathtt{exp}}$, and $\pi_{\mathtt{pleq}}$ is implemented properly. In each case, the Fiat-Shamir heuristic is applied to all public values and commitments listed in Table 2.5.
B4	CreateCC	The number of submitted partial choice codes is equal to t.
B5	CreateCC	The long choice codes (or the decrypted short choice codes) are tested for uniqueness.
B6	CreateCC ProcessConfirm	The keyed pseudo-random function f_k is implemented properly, for example as HMAC based on SHA256.
B7	CreateCC ProcessConfirm	The symmetric decryption using the symmetric key $\mathtt{VCC}^{\mathtt{id}}$ is implemented properly using the same current standard as in Register.
B8	ProcessConfirm	The verification of the signature $S_{\mathtt{VCC}^{\mathtt{id}}}$ is implemented properly according to the RSA-PSS standard.

Table 2.4: List of checks relative to the server-side algorithms of the election phase, which are executed by BBM.

2.3.1 Algorithms GetID, GetKey, Confirm, CreateVote

These are the client-side algorithms, which are executed by VD using web-client technologies (JavaScript). GetID, GetKey, and Confirm are relatively simple and their implementation must be in accordance with corresponding computations in the algorithm Register from the pre-election phase. Some of the checks from Table 2.2 must therefore be repeated almost one-to-one for the client-side implementation. Even tiny deviations from the client-side implementation could disrupt the proper functioning of the vote casting process.

The most complex client-side algorithm is CreateVote, which prepares the ballot based on the voter's choices. Its output $\mathtt{V}$ consists of the following elements:

- The ElGamal encryption $c = (c_1, c_2) = \mathsf{Enc}(v, \mathtt{EB_{pk}})$ of the aggregated vote $\prod_{i=1}^{t} v_{j_i}$ using the encryption randomness $r \in \mathbb{Z}_q$;
- The set $\{\mathtt{pCC}_i^{\mathtt{id}}\}_{i=1}^{t}$ of partial choice codes $\mathtt{pCC}_i^{\mathtt{id}} = v_{j_i}^{\mathtt{VC}^{\mathtt{id}}_{\mathtt{sk}}}$;

- The modified encrypted vote $\tilde{c} = (\tilde{c}_1, \tilde{c}_2) = (c_1^{\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}}, c_2^{\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}})$;
- The voter's public key $\mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}$;
- Three zero-knowledge proofs $\pi_{\mathtt{sch}}$, $\pi_{\mathtt{exp}}$, and $\pi_{\mathtt{pleq}}$.[7]

Each of these values must be computed exactly according to the protocol. Most computations are conducted in $\mathbb{G}_q$ (modulo p). In the implementation of the non-interactive zero-knowledge proofs, it is very important to apply the Fiat-Shamir heuristic to all commitments created in the initial step of the proof generation and to the right amount of public inputs. Otherwise, various attacks exist to weaken the security provided by the proofs [10, 14]. Table 2.5 gives an overview of the secret inputs, public inputs, and commitments for each proof included in V. It corresponds exactly to the definition of the proofs in [3].[8] Note that all public inputs and commitments are elements of $\mathbb{G}_q$ and all secret inputs are elements of $\mathbb{Z}_q$, i.e., computations are conducted modulo p respectively modulo q.

Proof	**Type**	**Secret Inputs**	**Public Inputs**	**Commitments**
$\pi_{\mathtt{sch}}$	Exp	r	g, c_1 (and c_2)	g^s, for $s \xleftarrow{\$} \mathbb{Z}_q$
$\pi_{\mathtt{exp}}$	Eq	$\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}$	$g, c_1, c_2, \mathtt{VC}_{\mathtt{pk}}^{\mathtt{id}}, \tilde{c}_1, \tilde{c}_2$	g^s, c_1^s, c_2^s, for $s \xleftarrow{\$} \mathbb{Z}_q$
$\pi_{\mathtt{pleq}}$	Eq	$r \cdot \mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}$	$g, \mathtt{EB}_{\mathtt{pk}}, \tilde{c}_1, \frac{\tilde{c}_2}{\prod_{i=1}^{t} \mathtt{pCC}_i^{\mathtt{id}}}$	$g^s, \mathtt{EB}_{\mathtt{pk}}^s$, for $s \xleftarrow{\$} \mathbb{Z}_q$

Table 2.5: Overview of the non-interactive zero-knowledge proofs contained in each submitted ballot. Note that c_2 is an auxiliary public input of $\pi_{\mathtt{sch}}$, which ties the the right-hand side of the ElGamal encryption to the proof.

2.3.2 **Algorithms** ProcessVote, CreateCC, ProcessConfirm

These are the server-side algorithms of the election phase, which are executed by BBM in response to each submitted ballot and confirmation. All three algorithms include certain validity tests, which may lead to an exception if one of them fails. Exceptions are handled by returning either 0 or the special symbol $\perp$. In each of these cases, the vote casting process is stopped. If ProcessVote returns 0, the submitted ballot V is discarded, which means the voter will be able to repeat the vote casting process from the beginning. If

[7]We observed that $\pi_{\mathtt{sch}}$, which proves knowledge of the encryption randomness $r \in \mathbb{Z}_q$, is redundant in the light of $\pi_{\mathtt{exp}}$ and $\pi_{\mathtt{pleq}}$, because $r = \log_g c_1 = \log_g \tilde{c}_1 / \log_{c_1} \tilde{c}_1$ can be computed easily from $r \cdot \mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}} = \log_g \tilde{c}_1$ and $\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}} = \log_{c_1} \tilde{c}_1$. A proof of knowledge of $r \cdot \mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}$ is included in $\pi_{\mathtt{pleq}}$ and a proof of knowledge of $\mathtt{VC}_{\mathtt{sk}}^{\mathtt{id}}$ is included in $\pi_{\mathtt{exp}}$, i.e., together they imply knowledge of r. Note that the auxiliary public input c_2 of $\pi_{\mathtt{sch}}$ is also included in $\pi_{\mathtt{exp}}$.

[8]From Version 1.0 to Version 5.0 of the specification document [5], the application of the Fiat-Shamir heuristic was not specified in further detail. According to [6], the code implementing the heuristic did not apply the heuristic properly to all public inputs and commitments, but the problem has been solved in a recent patch. Accordingly, the latest versions of [5] contain a more detailed and correct description of the Fiat-Shamir heuristic.

CreateCC returns $\perp$, $\mathtt{V}$ has already been added to BB, which means that the voter will be blocked from submitting another vote (see remark on Page 21). If ProcessConfirm returns $\perp$, the confirmation message $\mathtt{CM}^{\mathtt{id}}$ is discarded, i.e., the voter may attempt to submit another confirmation message that possibly will pass the tests.

The purpose of the algorithm ProcessVote is threefold. It first checks if the public key $\mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}$ included in $\mathtt{V}$ is consistent with the entry for $\mathtt{VC}_{\mathtt{id}}$ in ID.[9] If this is the case, it checks if an entry for $\mathtt{VC}_{\mathtt{id}}$ already exists in BB. If that's not the case, all three non-interactive zero-knowledge proofs are verified. Again, the proper application of the Fiat-Shamir heuristic is a precondition for the correctness of the proof verification.

Algorithm CreateCC first derives the long choice codes $\overline{\mathtt{CC}^{\mathtt{id}}_i}$ by applying the keyed pseudo-random function f_k to the partial choice codes $\mathtt{pCC}^{\mathtt{id}}_i$ included in the ballot (using $\mathtt{C}_{\mathtt{sk}}$ as symmetric key). If an entry exists in the voter's codes mapping table $\mathtt{CM}_{\mathtt{id}}$ for every $H(\overline{\mathtt{CC}^{\mathtt{id}}_i})$, the set of short choice codes $\{\overline{\mathtt{sCC}^{\mathtt{id}}_i}\}_{i=1}^t$ can be retrieved by decrypting the linked ciphertexts. The existence of such entries proves that all selected voting options v_i are valid. In the implementation of this algorithms, the critical points are similar to the ones already mentioned relative to the algorithm Register, which generates the mapping tables. In addition to those points, it is crucial to verify that the long (or the short) choice codes are unique (otherwise the same voting options could be selected multiple times) and that the number of submitted partial choice codes corresponds to t (otherwise ballots containing an amount of votes different from t would be accepted and added to BB). Surprisingly, these important aspects are not discussed in [3].[10]

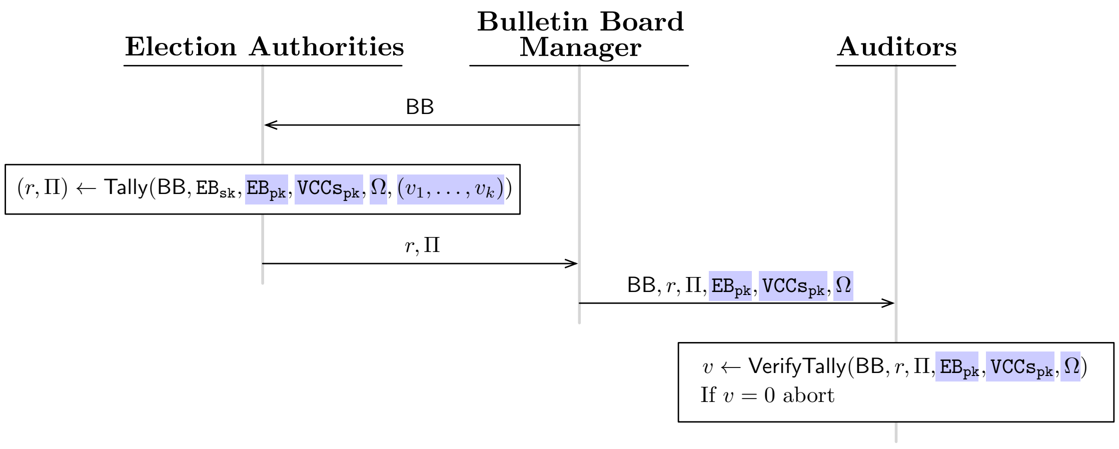
Algorithm ProcessConfirm is very similar to CreateCC, but it only deals with a single value, the confirmation message $\mathtt{CM}^{\mathtt{id}}$. In addition to applying the keyed pseudo-random function f_k to $\mathtt{CM}^{\mathtt{id}}$, selecting the entry for $H(\mathtt{VCC}^{\mathtt{id}})$ from the mapping table $\mathtt{CM}_{\mathtt{id}}$, and decrypting the short vote cast code $\mathtt{sVCC}^{\mathtt{id}}$, the attached RSA signature of $\mathtt{sVCC}^{\mathtt{id}}$ is verified. This is an additional test which could possibly lead to an exception.

2.4 Post-Election Phase

In the post-election phase, the collected election data from BB is given to EA, which is in charge of performing the decryption and the tallying. Their main output is the election result r, which can be seen as the list of plaintext votes from each voter. In addition to r, EA generates some cryptographic evidence Π, which can be used by AUD to verify that the decryption and the tallying has to be conducted properly. As one can see in the protocol diagram of Figure 2.3, there is a single algorithm for each of these tasks.

[9]Instead of returning $\mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}$ to BBM along with the encrypted vote, it would be easier to let the BBM retrieve $\mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}$ from the list ID based on $\mathtt{VC}_{\mathtt{id}}$. Without a consistency test for $\mathtt{VC}^{\mathtt{id}}_{\mathtt{pk}}$, there would be less exceptional cases in the protocol flow.

[10]Invalid votes of that kind would be sorted out at tallying, when the vector of decrypted votes is checked against Ω. However, eliminating them as early as possible in the process is certainly preferable. Then affected voters may even be allowed for re-submitting another (possibly valid) ballot.



Election Authorities
Bulletin Board Manager
Auditors
BB
$(r, \Pi) \leftarrow \mathsf{Tally}(\mathsf{BB}, \mathtt{EB_{sk}}, \mathtt{EB_{pk}}, \mathtt{VCCs_{pk}}, \Omega, (v_1, \ldots, v_k))$
r, Π
$\mathsf{BB}, r, \Pi, \mathtt{EB_{pk}}, \mathtt{VCCs_{pk}}, \Omega$
$v \leftarrow \mathsf{VerifyTally}(\mathsf{BB}, r, \Pi, \mathtt{EB_{pk}}, \mathtt{VCCs_{pk}}, \Omega)$
If $v = 0$ abort

Figure 2.3: Overview of the post-election sub-protocol. Deviations from the protocol description in [3] are highlighted in blue (additional parameters).

Despite the simplicity of this last protocol phase, we still found some inconsistencies in its description in [3]:

- $\mathtt{EB_{pk}}$, $\mathtt{VCCs_{pk}}$, and Ω are additional algorithm parameters of both Tally and VerifyTally.
- $(v_1, \ldots, v_k)$ is an additional algorithm parameter of Tally, otherwise a general factoring algorithm cannot be implemented efficiently.
- While Tally performs consistency checks $\{v_{i,1}, \ldots, v_{i,t}\} \in \Omega$ for each decrypted vote $v_i = \prod_{j=1}^{t} v_{ij}$, VerifyTally does not include such tests. This is an obvious mistake. To perform these test, VerifyTally requires Ω.
- In the course of conducting Tally, EA verifies signatures $S_{\mathtt{VCC^{id}}}$ using their own public key $\mathtt{VCCs_{pk}}$, i.e., they verify their own signatures. By doing so, they get convinced that the decrypted short vote cast codes $\mathtt{sVCC^{id}}$ correspond to the values they generated previously during the pre-election phase. The same could be achieved more easily, for example by keeping a list of all generated short vote cast codes (or a list of corresponding hash values).
- According to [3], Tally calls ProcessVote for every vote in BB. The obvious goal of this task is to repeat the ballot consistency checks. However, because BB has a completely different state when Tally is invoked by EA compared to the state when ProcessVote was invoked during vote casting, this does not work as suggested. We assume therefore that Tally mainly checks that BB contains at most one ballot for each $\mathtt{VC_{id}}$ and that their zero-knowledge proofs are all valid.

While conducting VerifyTally by AUD is part of the AVP as specified in [3], it is—to the best of our knowledge—not part of the implemented and deployed system. We will therefore ignore VerifyTally in our analysis.

The purpose of the Tally algorithm is threefold. First, it performs a cleansing process over all submitted ballots by performing tests similar to those of the $\mathsf{ProcessVote}$ algorithm to each submitted ballot (see remark above). The list $\mathcal{C}$ of encrypted votes is then extracted from the cleansed ballot list. Second, by calling the algorithm $\mathcal{C}' \leftarrow \mathsf{Mix}(\mathcal{C})$, the encrypted votes are re-encrypted and shuffled into a new list $\mathcal{C}'$. The permutation according to which the shuffling is performed is picked uniformly at random from all $n!$ possible permutations (where n denotes the size of $\mathcal{C}$ and $\mathcal{C}'$). Finally, the votes in $\mathcal{C}'$ are decrypted using the private key $\mathtt{EB}_{\mathtt{sk}}$ into values v_i, which are then factorized into sets $V_i = \{v_{i,1}, \ldots, v_{t'}\}$ of prime factors v_{ij}. For each resulting factorized vote, a test $V_i \in \Omega$ is performed to detect invalid combinations of voting options. Independently of how Ω is represented for a given election, this test must consist of at least the following steps:

- $t' = |V_i|$ is equal to t;
- $v_{ij} \in \{v_1, \ldots, v_n\}$ for all $v_{ij} \in V_i$;
- If V_i contains voting options from s different elections (in a combined election event with multiple simultaneous races), then V_i must contain exactly $t_j \geqslant 1$ voting options for every election $j \in \{1, \ldots, s\}$, where $t = \sum_{j=1}^{s} t_i$ is the total number of admissible votes. In such cases, the vector $(t_1, \ldots, t_s)$ and the decomposition of $(v_1, \ldots, v_k)$ into s partitions must be defined as part of the election setup.

Note that it is important to implement factorization efficiently, for example by looping over all given values $(v_1, \ldots, v_k)$ and using them as candidates for possible prime factors. Otherwise, if the search for prime factors is implemented "blindly" (for example using an exhaustive search), then the algorithm could get stuck almost endlessly if a decrypted vote contains very large prime factors.

Nr.	Algorithm	Description of Check
T1	Tally (cleansing)	Testing that BB contains at most one confirmed ballot for each $\mathtt{VC_{id}}$ is properly implemented.
T2	Tally (cleansing)	The verification of the non-interactive zero-knowledge proofs $\pi_{\mathtt{sch}}$, $\pi_{\mathtt{exp}}$, and $\pi_{\mathtt{pleq}}$ is implemented properly (similar to B3 in Table 2.4). The verifications are applied to the proofs in every confirmed ballot from BB.
T3	Tally (cleansing)	The verification of the signatures $S_{\mathtt{VCC^{id}}}$ is implemented properly according to the RSA-PSS standard (see B8 in Table 2.4). The verification is applied to the signature included in every confirmed ballot from BB.
T4	Tally (mixing)	The encrypted votes are properly re-encrypted before (or after) applying the shuffle. A fresh randomization is used for every ciphertext.
T5	Tally (mixing)	The shuffling algorithm selects a permutation uniformly at random and applies it properly to the list of (re-)encrypted votes.
T6	Tally (decryption)	The decryption of the shuffled ElGamal ciphertexts is implemented properly.
T7	Tally (decryption)	The factorizing algorithm is implemented correctly and runs efficiently even for numbers with arbitrarily large factors.
T8	Tally (decryption)	Testing that each factorized vote $\{v_{i,1}, \ldots, v_{i,t}\}$ is an element of the set Ω of admissible vote combinations is implemented properly, such that all types of invalid votes are detected.
T9	Tally	The output list of the cleansing process corresponds to the input list of the mixing process. The output list of the mixing process corresponds to the input list of the decryption process.

Table 2.6: List of checks relative to the post-election algorithm Tally.

3 Source Code Analysis

Due to the circumstances discussed in Section 1.3, we were not able to apply code analysis tools to the source code. Thus we were forced to analyze the code mostly manually, which turned out to be very time-consuming in the light of the vast amount of source files. Our attempts to build up the necessary understanding of the code was further impeded by the *Spring* framework in use. By injecting code at runtime, *Spring* makes analyzing, debugging, and navigating through the code considerably more difficult. While finding code sections implementing a certain task is not difficult as such, figuring out whether the code is really in use turned out to be cumbersome in many cases. Given the very limited time for the analysis of the source code, the following analysis is the result of a best effort approach and cannot be seen as an encompassing and final analysis.

Based on the checklists from Section 2, we analyzed the code by evaluating systematically each single check of each protocol sub-phase. To present the essence of our conclusions in a uniform and simple way, we introduce the color scheme of Table 3.1. Note that only problematic checks (red, orange) will be discussed in detail. Successful checks (green) are listed in the tables, but are not further elaborated.

Eval.	**Description**
🟩	We performed the check and confirm it is properly implemented. Non-security relevant minor deviations are accepted.
🟧 I	While performing the check, we discovered that the implementation of some relevant points is only partially correct.
🟧 II	We were unable to perform the check: relevant source code is missing.
🟧 III	We were unable to perform the check: lack of time.
🟥	While performing the check, we discovered that some of the most relevant points are either incorrectly implemented or completely missing.

Table 3.1: Color scheme used to evaluate the checklists from Section 2.

3.1 General Protocol Aspects

The biggest issue we found in the context of the general protocol aspects affects the cryptographic setup. The generation of the cryptographic parameters p, q, and g and the missing verification do not follow best practices. Other shortcomings were found related to consistency tests of input parameters, the usage of the random source, properly hashing multiple values, the signing of exchanged messages, and messages which contain more information than defined by the protocol. Most of these aspects could not be analyzed completely due to the amount of source code and the lack of time. However, for each of the mentioned aspects there is at least one example, where the system is implemented improperly.

Nr.	Findings	Eval.
G1	The cryptographic parameters p, q, g, 1^λ are not verified by any component or party. Neither a simple consistency check is done nor is the security level verified.	🟥
G2	The keys are properly generated within the security level of 112 bits ($\phi = \lVert \mathsf{pwd} \rVert = 32^{20} = 2^{100}$ and $c = 32000 \approx 2^{15}$ results in $100 + 15 = 115 \geqslant 112$).	🟩
G3	Basic tests are generally done, however fundamental tests like group membership are occasionally missing.	🟧 I/II/III
G4	Picking elements uniformly at random based on the output of a cryptographically secure PRNG is implemented more than once and not always properly.	🟧 I/III
G5	The group operations in $\mathbb{G}_q$, $\mathbb{Z}_q$, and $\mathbb{Z}_p^*$ are computed properly.	🟩
G6	Collision-resistance is not preserved for hash values of multiple inputs.	🟥
G7	Even though there are multiple PKI-trees available and the certificates for the involved public keys exist and are valid, there are security critical documents that are not signed. This is not in accordance to best practice and jeopardizes their authenticity and integrity, as many of these documents are moved between actors and system boundaries.	🟧 I/III
G8	Due to lack of time, it was not possible to do an exhausting verification of all exchanged messages. However, we found a prominent example where the information additionally contained in the message alters the properties of the underlying cryptographic protocol.	🟧 I/III

Table 3.2: Evaluation of the general checks.

G1 The true security level, the system is operating in, cannot be determined. This is due to the fact that multiple security relevant values are not derived from the proposed security level but are hard coded at some point and propagated throughout the code.

Throughout the complete implementation we have not found a single place, where the cryptographic parameters p, q, and g are verified by any consistency checks. It is also not checked whether the bit-lengths of p and q correspond to the security level. Such checks are even missing when the parameters are taken from an external source[11], where authenticity and integrity is missing due to the lack of a signature.

Furthermore, the procedure for the generation of p, q, g, and the list of primes required for vote encoding is implemented in a way, that multiple manually induced steps are

[11]usually by deserializing some values from a file

required. This way, the cryptographic setup is highly error prone. Please note, that even after demanding for it several times we have not received any document describing this manual process in detail.

As there are several findings within this crucial step of the implementation, please find the dedicated section 3.2 for more insights.

G3 Due to the amount of source code and lack of time, we were not able to verify all membership and consistency checks for all algorithm parameters. We were only able to analyze a limited amount of the source code but we can confirm that basic checks are generally done. Parameters are checked for null- and/or empty-value and whether they fit into a certain range. However, fundamental cryptographic checks such as *group membership* are not done systematically and are occasionally missing. For example the encrypted partial choice codes are decrypted without a prior check for group membership. Or in the verification of the plaintext equality proof, the input parameters to the Maurer's unified proof verifier are computed on values, which are not checked for group membership; this questions the soundness of the proofs to guarantee individual verifiability.

The important consistency tests regarding the prime numbers for the voting options (see Section 2.1.2) have not been found and hence are expected to be missing (see also Section 3.2).

G4 There is a complete package (securerandom) within the source code for the server-side, where the delicate matter of choosing entropy pools and acquiring randomness is treated. However, there are locations, where the productive code uses an alternative implementation for acquiring randomness, e.g. BigIntTools within the mixnet package. This way, we could not determine the quality of the randomness used at the server-side of the system.

The implementation of randomness at the client-side is thoroughly implemented. Unfortunately though, the implementation puts usability on top of security. Even if the implementation realizes that there is too little entropy available in order to provide the required security level[12], it will continue the process in order to provide a good user experience. Nevertheless, it must be noted, that such a scenario is somewhat unlikely under normal circumstances.

G6 As far as we were able to analyze the code, there is no mechanism implemented that provides collision resistance when multiple values are hashed. For example, multiple strings are simply concatenated by an empty string resulting in the fact that $H(\text{"}ab\text{"}||\text{"}c\text{"}) = H(\text{"}a\text{"}||\text{"}bc\text{"})$. The same holds for $\mathbb{G}_q$ elements, for example $H(12||3) = H(1||23)$.

G7 Digitally signing and verification of documents is an important security aspect, as many serializations and de-serializations take place during the different voting phases. Thus it is key to work with authentic data. The algorithms used for signing and verification are all defined in the cryptographic policies. However, the usage of signatures and

[12]Even if unlikely, it could be as low as 0.

verification of data transferred between actors within protocol runs is not always given. Paired with the omission of cryptographic checks when deserializing data, neither their origin nor their integrity can be guaranteed. Even if the data is created and serialized within a trusted environment, good practices prohibits trust after data has been serialized. However, some of the data is moved over system boundaries, such as the data serialized to db_dump.json. It contains crucial information about the running system but its integrity and authenticity cannot be verified by any actor. Note, that according to the documentation this file even travels between the air-gaped computers. Another example is the file cryptolibPolicy.properties describing which cryptographic algorithms are in use. Who is the originator of this file, is it still valid, and are all actors working with the same instance? These questions remain open for any internal actor due to the lack of signature.

G8 Unfortunately, we were not able to analyze systematically all exchanged messages due to lack of time. However, we found a prominent example, where the information additionally added to the message alters the properties of the underlying cryptographic protocol. After sending the vote, the voter confirms the properly received choice codes by sending a confirmation message to the BBM. The BBM is then supposed to verify the confirmation message and to send the short vote caste code $\mathtt{sVCC}^{\mathtt{id}}$ together with the signature $S_{\mathsf{VCC}^{\mathtt{id}}}$ back to the voter. But the BBM sends additionally a digitally signed receipt. The receipt allows the voter (or any malicious voting device) to prove to anybody the content of the cast vote without the need to access the bulletin board.

3.2 Cryptographic Setup

For the cryptographic setup, no proper programming code seems available for an automated and verifiable process. Instead, there are manual ceremonies required, that comprise many error prone steps. As the resulting values are required before running the system, it is important to notice, that these ceremonies are run during deploy-time in an unspecified environment. The API for this ceremony is given by a tool called configGenerator.sh with different parameters.

Selection of p, q, and g Right at the start of the ceremony to select p, q, and g, there is a security critical inconsistency present. The API of the tool asks the human user to put in the size of the required prime number p in bytes. Following the call hierarchy in the source code, the entered number is interpreted as the number of bits of p. Hence, if the user follows the API as documented, the setup results in a drop of security by a factor of eight. This lingering between the interpretation of a value as number of bits or number of bytes can be found on multiple occasions within the implementation. The manual process and the misleading API provoke a faulty setup.

Selection of Cryptographic Policies In general, it is important that a cryptographic project always relies on the same cryptographic policies, which comprise the security level and the cryptographic standards in use (for example for signing, encrypting or hashing). Even though, the analyzed system provides such policies by a policy file (cryptolibPolicy.properties), that comprises most of the mentioned entries, this file exists in different variants and versions throughout the system. This adds an unnecessary layer of complexity for analyzing and verification of the systems behavior.

Selection of Security Level There are multiple locations within the system, where security relevant values are used. However, the true security level, the system is operating in, cannot be determined. This is due to the fact that there exists no single point, where security relevant values are derived from the proposed security level. Instead, some of these values are hard coded and propagated throughout the code. To illustrate the situation, two examples (Listing 1 and Listing 2) are provided regarding the certainty level to select probable prime numbers:

```
...
// TODO: [AF] confirm if the certainty level can be set to 90, or if it
    should be made configurable
private final int CERTAINTY_LEVEL = 90;
...
public ElGamalEncryptionParameters
    generateZpSubGroupEncryptionParameters(final int pBits, final int qBits){
    ElGamalEncryptionParamsGenerator elGamalEncryptionParamsGenerator =
    new ElGamalEncryptionParamsGenerator(pBits, qBits, CERTAINTY_LEVEL,
        _cryptoRandomBytes);

    return elGamalEncryptionParamsGenerator.generate();
}
public ElGamalEncryptionParameters
    generateQuadraticResidueEncryptionParameters(final int pBits) {

    QuadResParamsGenerator quadResParamsGenerator =
    new QuadResParamsGenerator(pBits, CERTAINTY_LEVEL, _cryptoRandomInteger);

    return quadResParamsGenerator.generate();
}
...
```

Listing 1: The certainty level used to select probable prime numbers in the class UniversalElGamalEncryptionParamsGenerator (com.scytl.products.ov.encryption.params.generation) is hard coded to 90.

```
public class PrimesUtils {
/**
* A measure of the uncertainty that we are willing to tolerate. It ensures
* that the uncertainty level is below 1 - 1/2<sup> {@code certainty}</sup>
*/
public static final int CERTAINTY = 100;

private PrimesUtils() {
}
...
public static final boolean isProbablePrime(final BigInteger bigInteger) {
return bigInteger.isProbablePrime(CERTAINTY);
}
...
```

Listing 2: The certainty level used to determin whether a number is probable prime in the class PrimesUtils (com.scytl.cryptolib.primitives.primes.utils) is hard coded to 100.

Selection of Prime Numbers for Voting Options An important step of the cryptographic setup of this system is the selection of the prime numbers used as representatives for the voting options. This is part of the manual ceremony, where the user is required to enter the cryptographic parameters (p, q, and g). Then the tool (configGenerator.sh) is explicitly looking for a specific file (primes.txt), where some 10000 numbers are stored. The tool then checks for every number within that file, if the number is a group member of $\mathbb{G}_q$. However, the tool omits the fundamental verification that checks if the treated number truly is a prime number. This is considered a major issue, as any group member will be accepted within this procedure.

As a final extra step, the resulting values have to be updated manually in the system. It is important to notice, that neither the input nor the output of this procedure is signed in any way.

After the prime numbers have been chosen as representatives for the voting options, we could observe, that these prime numbers are not chosen in terms of 'smallest first', but by their alphabetical ordering. This results in a somewhat 'biggest first' strategy, which is both confusing and lowers the maximum amount of options that can be mapped to $\mathbb{G}_q$, especially if the file primes.txt is enlarged once.

3.3 Pre-Election Phase

We found one major issue in the algorithms which are used in the pre-election phase: the choice code mapping tables $\texttt{CM}_{\texttt{id}}$ are not permuted enabling the BBM to break vote secrecy. The check regarding the correct number of generated voting cards could not be conclusively conducted due to lack of time. All other checks of the pre-election phase are implemented properly as far as we can judge. Please note the comment about the cryptographic policies in the Section 3.2 because many algorithms of the pre-election phase are based on standards configured in the policies file.

Nr.	**Findings**	**Eval.**
S1	RSA key generation is implemented properly using openSSL.	🟩
S2	The correct number s of generated voting cards was not conclusively to assess.	III
S3	The key derivation function δ is implemented properly and the two salts `IDseed` and `KEYseed` are different.	🟩
S4	$\texttt{VC}^{\texttt{id}}_{\texttt{sk}}$ is encrypted using a PKCS12 keystore provided by SunJSSE. $\texttt{sCC}^{\texttt{id}}_i$ and $(\texttt{sVCC}^{\texttt{id}} \| S_{\texttt{VCC}^{\texttt{id}}})$ are symmetrically encrypted using the AES-GCM standard.	🟩
S5	The keyed pseudo-random function f_k is implemented properly (HMAC based on SHA256).	🟩
S6	The randomly selected short choice codes $\texttt{sCC}^{\texttt{id}}_i$ are checked for uniqueness.	🟩
S7	The mapping tables $\texttt{CM}_{\texttt{id}}$ are not permuted.	🟥
S8	The values $\texttt{CC}^{\texttt{id}}_i$ and $\texttt{VCC}^{\texttt{id}}$ are computed properly but not following the protocol exactly. Not only the specified values are passed to the function f_k but a number of additional values.	🟩
S9	The signature $S_{\texttt{VCC}^{\texttt{id}}}$ is implemented properly using the RSA-PSS standard. However, not only the $\texttt{sVCC}^{\texttt{id}}$ is signed but also the verification card id.	🟩

Table 3.3: Evaluation of the checks relative to the pre-election algorithms.

S2 The generation of voting cards is performed in parallel processes managed by the *Spring Batch* framework. The concrete number s of voting cards to generate is injected at runtime into the corresponding batch job by the framework. Due to lack of time, it was not possible to conclusively assess whether this number is properly determined (see also T9).

S7 The choice code mapping tables $\texttt{CM}_{\texttt{id}}$ are neither properly shuffled nor is the order of the values indirectly permuted by properly using hash tables. In fact, the order is explicitly preserved by using a data structure called LinkedHashMap. This enables the BBM to break vote secrecy without knowing the private encryption key $\mathtt{EB}_{\mathtt{sk}}$.

3.4 Election Phase

Some checks of the election phase are closely related to the checks of the setup phase and many checks of the client-side and server-side are linked to each other. The protocol would completely collapse if the different components were using different standards for the key derivation function or the asymmetric encryption. Hence, it was expected that most of the checks are implemented properly. We found though one major issue regarding the proofs for the individual verifiability. Due to lack of time, we were not able to verify whether the encrypted votes are properly tested for correctness.

Nr.	Findings	Eval.
V1	The key derivation function δ is implemented properly using the same standard as in Register and the same salt `IDseed`.	🟩
V2	The key derivation function δ is implemented properly using the same standard as in Register and the same salt `KEYseed`.	🟩
V3	The decryption of $\mathtt{VCks}_{\mathtt{id}}$ using the symmetric key $\mathtt{KSpwd}_{\mathtt{id}}$ is implemented properly using the same standard as in Register.	🟩
V4	The computation of the confirmation message $\mathtt{CM}^{\mathtt{id}}$ is implemented according to the protocol.	🟩
V5	The ElGamal encryption c is implemented properly using a fresh randomization r.	🟩
V6	The $\mathtt{pCC}_i^{\mathtt{id}}$ are computed properly. However, they are additionally encrypted which is not according to the protocol.	🟧 I
V7	The modified encrypted vote $\tilde{c}$ is derived from c according to the protocol.	🟩
V8	$\pi_{\mathtt{pleq}}$ is not generated according to the protocol. This is a result of the fact that the partial choice codes are encrypted and hence the equality proof must be performed under encryption. In addition, also not according to the protocol, $\pi_{\mathtt{sch}}$ is linked to the voter and election by applying the Fiat-Shamir heuristic to the voting card id and election id.	🟥
V9	The three non-interactive zero-knowledge proofs are implemented properly. There are minor deviations in the order of the elements by applying the Fiat-Shamir heuristic.	🟩

Table 3.4: Evaluation of the checks relative to the client-side algorithms of the election phase, which are executed by VD.

V6 The partial choice codes $\texttt{pCC}^{\texttt{id}}$ are not transmitted in plain but are additionally encrypted using an ElGamal multi-encryption scheme. In the compliance report [6] it is justified why this is done. However, encrypting the partial choice codes is not specified in the protocol and hence it is also not specified when and by whom the encryption keypair is generated. From our perspective, aspects of channel security and of the cryptographic protocol have been mixed up.

V8 As a consequence of encrypting the partial choice codes, the proof $\pi_{\texttt{pleq}}$ needed to be replaced by another proof. Instead of proving that an encryption holds a certain plaintext, it is now proven that two encryptions hold the same plaintext. As the proofs generated by the voter play a central role for the individual verifiability property of the online voting scheme, changing these proofs undermines the formal security proof for individual verifiability, as it is presented in [3].

Additionally, the proof $\pi_{\texttt{sch}}$ is linked to the voter and election by applying the Fiat-Shamir heuristic to the voting card and election id. This is not an issue in general but affects the statement of the proof and needs to be defined in the protocol.

Nr.	**Findings**	**Eval.**
B1	It is checked properly whether the voter with the provided $\texttt{VC}_{\texttt{id}}$ has already submitted a ballot.	🟩
B2	Checking the existence of an entry for $\texttt{VC}_{\texttt{id}}$ in ID and the corresponding $\texttt{VC}^{\texttt{id}}_{\texttt{pk}}$ is conducted properly.	🟩
B3	The three non-interactive zero-knowledge proofs are verified properly. $\pi_{\texttt{pleq}}$ is not according to the protocol (see V8)	🟩
B4	The correctness rules which are required to verify the vote correctness are defined using a scripting language. Due to lack of time, it was not possible to verify the scripts and its evaluation in-depth.	🟧 III
B5	The short choice codes are properly tested for uniqueness.	🟩
B6	The keyed pseudo-random function f_k is implemented properly.	🟩
B7	The symmetric decryption using the symmetric key $\texttt{VCC}^{\texttt{id}}$ is implemented properly using the same standard as in $\mathsf{Register}$.	🟩
B8	The verification of the signature $S_{\texttt{VCC}^{\texttt{id}}}$ is implemented properly according to the RSA-PSS standard.	🟩

Table 3.5: Evaluation of the checks relative to the server-side algorithms of the election phase, which are executed by BBM.

B4 The correct number of partial choice codes is indirectly verified by checking the correctness of the vote. Vote correctness in implemented by linking each partial choice code to a *correctness id* and applying a certain number of rules (see Appendix A.4 in [3]). The rules are not implemented in Java but in a scripting language (JavaScript) and stored with the election information. Due to lack of time, it was not possible to completely evaluate the ruleset and to verify its proper implementation.

3.5 Post-Election Phase

We did not find any major issue in the post-election phase. We were not able to verify the test for correctness of the decrypted votes due to the same reason as for the encrypted votes in the voting phase. Time was also missing for the non cryptographic but very important last check.

Nr.	Findings	Eval.
T1	It is tested that BB contains at most one confirmed ballot for each $\mathtt{VC_{id}}$.	🟩
T2	The verification of the three non-interactive zero-knowledge proofs is implemented properly. The verification is conducted for every confirmed ballot.	🟩
T3	The verification of the signatures $S_{\mathtt{VCC}^{\mathtt{id}}}$ is implemented properly and conducted for every confirmed ballot.	🟩
T4	The encrypted votes are properly re-encrypted before (or after) applying the shuffle. A fresh randomization is used for every ciphertext.	🟩
T5	The shuffling algorithm selects a permutation uniformly at random and applies it properly to the list of encrypted votes.	🟩
T6	The decryption of the shuffled ElGamal ciphertexts is implemented properly.	🟩
T7	The factorizing algorithm is implemented correctly and runs efficiently. The decrypted vote is checked for only those prime factors which are valid vote options.	🟩
T8	The final test to check whether the decrypted and factorized vote $\{v_{i,1}, \dots, v_{i,t}\}$ is a valid vote is implemented by applying certain rules defined in a scripting language. Due to lack of time, it was not possible to verify the scripts and its evaluation in-depth.	🟧 III
T9	Due to lack of time and the frameworks in use, it was not possible to verify the proper data flow between the different processes.	🟧 III

Table 3.6: Evaluation of the checks relative to the post-election algorithms.

T8 Similarly to the check B4 during the voting phase, we were not able to verify the test for correctness of the decrypted and factorized vote due to lack of time. The applied rules are defined by a script which is evaluated in Java.

T9 Especially in the post-election phase much can go wrong that is not directly related to cryptography. For example, does the list of votes which is passed to the cleansing process contain all cast votes or have some votes (accidentally) been suppressed? Similarly, does the list of votes passed to the mixing process correspond to the proper output of the cleansing process? There is evidence that these critical aspects are properly implemented. None of our test elections during debugging the system had any improper divergence in the final tally. However, we were not able to verify such aspects in the source code. This is due to lack of time and the frameworks in use. *Spring* and *Spring Batch* are extremely powerful but complex frameworks. Batch jobs can be defined on certain resources and the framework takes responsibility to run them in parallel. Hence, no simple loop over the cast votes can be found in the source code. Verifying such aspects is therefore a complex and time consuming task.

References

[1] Swiss online voting protocol. Scytl Secure Electronic Voting, Barcelona, Spain, 2015.

[2] Online voting – cryptographic tools specification. Scytl Secure Electronic Voting, Barcelona, Spain, 2016.

[3] Swiss online voting system – cryptographic proof of individual verifiability. Scytl Secure Electronic Voting, Barcelona, Spain, 2017.

[4] *Verordnung der Bundeskanzlei über die elektronische Stimmabgabe (VEleS) vom 13. Dezember 2013 (Stand vom 1. Juli 2018).* Die Schweizerische Bundeskanzlei (BK), 2018.

[5] Online voting – protocol specifications. Scytl Secure Electronic Voting, Barcelona, Spain, 2019.

[6] Security analysis of key cryptographic elements for individual verifiability. Scytl Secure Electronic Voting, Barcelona, Spain, 2019.

[7] Security review of key cryptographic elements of the e-voting solution. Kudelski Security, 2019.

[8] D. Basin and S. Čapkun. Addendum to review of electronic voting protocol models and proofs. Final report (v1.0), Contego Laboratories, Rüschlikon, Switzerland, 2017.

[9] D. Basin and S. Čapkun. Review of electronic voting protocol models and proofs. Final report (v2.0), Contego Laboratories, Rüschlikon, Switzerland, 2017.

[10] D. Bernhard, O. Pereira, and B. Warinschi. How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In X. Wang and K. Sako, editors, *ASIACRYPT'12, 18th International Conference on the Theory and Application of Cryptology and Information Security*, LNCS 7658, pages 626–643, Beijing, China, 2012.

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

[12] D. Galindo. Analysis of cast-as-intended verifiability and ballot privacy properties for Scytl's swiss online voting protocol using ProVerif. Technical report, University of Birmingham, Birmingham, U.K., 2017.

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

[14] S. J. Lewis, O. Pereira, and V. Teague. How not to prove your election outcome. Technical report, 2019.

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