University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a Verifiable Remote Postal Voting System

Niels Kübler
Winterthur, Switzerland
Student ID: 10-712-123

Supervisor: Christian Killer, Jan von der Assen
Date of Submission: October 17, 2022

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Switzerland allows its citizens to participate in elections and votes by casting ballots via postal mail. This practice, referred to as "Remote Postal Voting", is a convenient way for voters to cast their ballots without visiting a polling station. However, the application of Remote Postal Voting has raised security concerns regarding vote- and election manipulations by voting officials or third parties. These concerns have been confirmed in practice, where malicious activities have been discovered on multiple occasions. This thesis aims at providing verifiability for Remote Postal Voting procedures by presenting the design and implementation of a verifiable Remote Postal Voting system. The design integrates well into the existing Swiss Remote Postal Voting procedures. It leverages blockchain technology, Homomorphic Encryption, Non-Interactive Zero Knowledge Proofs, and Threshold Cryptography to meet privacy and verifiability requirements. The evaluation shows that the proposed system scales well in the Swiss Remote Postal Voting scenario, whereas further optimization would be necessary for larger countries.

Kurzfassung

Die Schweiz erlaubt ihren Staatbürgerinnen auf dem brieflichen Weg an Abstimmungen und Wahlen teilzunehmen. Dieses Vorgehen, oft als “Remote Postal Voting” bezeichnet, stellt einen einfachen und bequemen Weg der Stimmabgabe dar, ohne dass Wählerinnen zu diesem Zweck ein Stimmlokal aufsuchen müssen. Allerdings gibt es bezüglich des Vorgehens der brieflichen Stimmabgabe Sicherheitsbedenken, insbesondere in Bezug auf Wahl- und Abstimmungsmanipulation, sei es durch Wahlhelfer oder Drittparteien. Diese Bedenken wurden in der Praxis mehrfach durch einschlägige Vorfälle bestätigt. Das Ziel dieser Arbeit ist es, die Vorgänge der brieflichen Stimmabgabe verifizierbar zu machen, indem sowohl das Design als auch die Implementation eines verifizierbaren brieflichen Abstimmungssystems präsentiert wird. Dabei lässt sich das vorgestellte Design nahtlos in die aktuellen Vorgänge der brieflichen Stimmabgabe in der Schweiz integrieren, indem eine Kombination von Blockchain Technologie, Kryptographie, wie beispielsweise homomorphe Verschlüsselung, Non-Interactive Zero Knowledge Proofs und Threshold Kryptographie angewandt wird. Dadurch lassen sich die Anforderungen bezüglich des Datenschutzes und der Verifizierbarkeit erfüllen. Die Evaluation zeigt, dass das vorgeschlagene System im Rahmen der Schweizerischen brieflichen Stimmabgabe skaliert, während für den Einsatz in grösseren Ländern noch Optimierungen erforderlich wären.

Acknowledgments

I want to express my gratitude to my supervisor Christian Killer for his support and priceless feedback during all phases of writing this thesis. Furthermore, I want to thank Prof. Dr. Burkhard Stiller for allowing me to work on this thesis and for all the other projects I could work on at the CSG during my studies.

Last but not least, I want to thank my family, especially my fiancée Carolin for her unrelenting support and patience during my studies.

Contents

Abstract **i**

Kurzfassung **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Thesis Outline 2

2 Background **3**

2.1 Remote Postal Voting in Switzerland 3

2.1.1 Voting Artifacts 3

2.1.2 Voting Procedures 4

2.1.3 Security Threats and Manipulations 4

2.1.4 Discussion 5

2.2 Properties of Voting Protocols 6

2.2.1 Privacy Notions 6

2.2.2 Verifiability Notions 7

3 Cryptography **9**

3.1 Homomorphic Encryption 9

3.1.1 ElGamal Cryptosystem 9

3.1.2 Exponential ElGamal 10

3.2 Threshold Cryptography 11

3.3 Non-Interactive Zero-Knowledge Proofs 11
3.3.1 Chaum-Pedersen Proof 12
3.3.2 Disjunctive Chaum-Pedersen Proof 13
3.3.3 Schnorr Proof 15

4 Related Work **17**
4.1 Verifiable Postal Voting 17
4.2 Towards Verifiable Remote Postal Voting with Paper Assurance 18
4.3 STROBE-Voting 19
4.4 RemoteVote 20
4.5 SAFE Vote 21
4.6 Comparison and Discussion 22
4.6.1 Trust Assumptions 22
4.6.2 Swiss RPV Criteria 22
4.6.3 Protocol Suitability 23

5 Design **25**
5.1 Requirements 25
5.1.1 Functional Requirements 26
5.1.2 Non-functional Requirements 26
5.1.3 Constraints 26
5.2 Helverify Protocol 27
5.2.1 Setup 27
5.2.2 Ballot Creation 28
5.2.3 Delivery 29
5.2.4 Casting 29
5.2.5 Storage 30
5.2.6 Tallying 30
5.2.7 Verification 31

5.2.8 Destruction 31

5.3 Architecture 31

5.3.1 Voting Authority 31

5.3.2 Consensus Node 31

5.3.3 Interplanetary File System 32

5.3.4 Voter 32

5.3.5 Proof-of-Authority Blockchain 32

6 Implementation **33**

6.1 Cryptography Library 33

6.1.1 ElGamal Cryptosystem 34

6.1.2 Zero-Knowledge Proofs 34

6.2 Consensus Nodes 35

6.2.1 Blockchain Node 35

6.2.2 Backend 36

6.3 Voting Authority 37

6.3.1 Backend 38

6.3.2 Frontend 45

6.4 Voter Frontend 49

6.4.1 Ballot Authenticity 49

6.4.2 Ballot Choice and Spoilt Ballot 51

6.4.3 Results and Summary 52

6.5 Proof-of-Authority Blockchain 53

6.5.1 Configuration 53

6.5.2 Election Smart Contract 53

6.6 Interplanetary File System 55

7 Evaluation **57**

7.1 Performance and Scalability 57

7.1.1 Ballot Creation 58

7.1.2 Result Calculation 60

7.2 Security Level 61

7.2.1 Trust Boundaries 61

7.2.2 Implications 62

7.3 RPV Properties 63

7.3.1 Privacy 63

7.3.2 Verifiability 63

7.4 Requirements 64

7.5 Projection to Real-World Votes 65

7.6 Discussion 66

8 Summary, Conclusion, and Future Work **69**

8.1 Summary 69

8.2 Conclusion 69

8.3 Future Work 70

Abbreviations **79**

List of Figures **80**

List of Tables **82**

A Installation Instructions **85**

Chapter 1

Introduction

Nowadays, electronic voting systems are in use all over the world to run elections and votes. The broad term of remote voting systems can be divided into Remote Electronic Voting (REV) systems and Remote Postal Voting (RPV) systems. The former describes systems that allow voters to cast their ballots from any location using an electronic device (*e.g.,* a PC or tablet computer). The latter defines systems using postal mail to cast the voters' ballots. The use of RPV depends on the legal frameworks of the respective country. Thus, RPV has been in place for many years in some countries, sometimes with only minimal degrees of digitization. Compared to modern REV systems (*e.g.,* [26, 63]), RPV systems often lack adequate means of verification. Consequently, several approaches have been proposed for providing verifiability in an RPV setting [5, 6, 20, 51].

REV and RPV systems are subject to the trade-off between privacy and verifiability. In other words, the difficulty lies in providing as much verifiability as possible without sacrificing the voter's privacy. For this purpose, REV and RPV systems rely heavily on the use of cryptography, such as Homomorphic Encryption (HE), Mix Networks (MN), Commitment Schemes (CS), and Zero-Knowledge Proofs (ZKP). Consequently, such systems are usually more complex than their analog paper-based predecessors.

This work focuses on Swiss RPV procedures. Switzerland is a direct democracy, allowing Swiss citizens of age 18 and above to vote at municipal, cantonal, and federal levels [15, 22]. Thus, compared to other countries, Swiss citizens have far-reaching voting rights. Consequently, votes and elections occur frequently, which has led to multiple accepted ways of casting a ballot. Apart from casting a ballot at the municipalities' polling stations, Switzerland also allows RPV since 1994 [55], meaning eligible voters can cast their ballot by postal mail or by posting the voting envelope into the municipalities' mail box [42]. Note that the RPV procedures are decentralized as they are implemented by cantons and municipalities, mirroring the federal structure of Switzerland [42].

While RPV is convenient for voters, this approach raises concerns in terms of verifiability: In Switzerland, there have been multiple incidents involving fraudulent activities during elections or votes (*e.g.,* [2, 39, 40, 46, 47]) to manipulate the outcome of the election or vote. Thus, this raises the following question: How can a voter be sure that her ballot was cast, recorded, and counted as she intended? With the current procedures, voters can

check their ballot before putting it into the voting envelope to ensure it was cast correctly. Alternatively, voters could cast their ballot at the polling station, ensuring that the ballot was cast and recorded correctly. Thus, there is currently no way of verifying that a vote or election has been performed correctly, as current procedures rely on statistical analyses of the results to discover anomalies (*e.g.,* [47]). The lack of verifiability is particularly serious if voters cast their ballots by postal mail.

This work presents *Helverify*: the design and implementation of a verifiable RPV system for use in Switzerland, the first of its kind to the author's knowledge. Consequently, the current Swiss RPV procedures primarily influence the requirements for such a system (*e.g.,* regarding scalability, performance, and usability). The system designed and implemented during this work aims at augmenting the current Swiss RPV procedures instead of replacing them, allowing to fall back on said procedures in case something goes wrong. *Helverify* leverages the combination of a Proof-of-Authority (PoA) Blockchain (BC) and a distributed file system (*i.e.,* IPFS) to provide a Public Bulletin Board (PBB) for storing the election evidence. The combination of these two technologies allows to leverage immutability, public accessibility, and scalability, which are crucial properties of a PBB. Moreover, the decentralized PBB design permits seamless integration into the current Swiss RPV procedures, which are decentralized by design [42]. Furthermore, HE, threshold cryptography, and NIZKPs have been applied to achieve End-To-End Verifiability (E2E-V) while preserving Ballot Privacy (BP) at the same time.

1.1 Thesis Outline

The remainder of this thesis is structured as follows: Chapter 2 delves into the fundamentals of the Swiss RPV procedures, with serves as a basis for the protocol design. Next, Chapter 3 presents the mathematical foundations of the cryptographic primitives used in the *Helverify* prototype, namely HE, Threshold Cryptography, and Non-Interactive ZKPs (NIZKP). Chapter 4 compares relevant related work in the domain of RPV. Chapter 5 shows the *Helverify* protocol design and the requirements considered for its creation. Chapter 6 documents the technical and architectural aspects of the *Helverify* prototype's implementation. Chapter 7 delves into various characteristics of the *Helverify* prototype, such as performance, scalability, security, and more. Finally, Chapter 8 concludes this thesis, providing an overview over the work done, the derived conclusion, and potential areas of future work.

Chapter 2

Background

This chapter features an overview of the context, basic principles, and notions used throughout this work. Section 2.1 provides an introduction to the voting procedures of Switzerland, with a focus on RPV. Furthermore, this chapter delves into the basics of voting protocol properties, consisting of privacy and verifiability notions.

2.1 Remote Postal Voting in Switzerland

In Switzerland, Swiss citizens aged 18 and above can vote on the federal [22], cantonal and municipal levels [15]. Municipalities are responsible for maintaining an Electoral Register (ER) [42], which has the purpose of keeping a record of all the eligible voters living within the municipality [23]. The voter has three possibilities for casting her ballot: The first option is the traditional ballot casting at the urn [42]. As an alternative, voters have the right to cast their ballot through postal voting, sending the voting envelope to the municipality using the Swiss Post mail service [42]. The third option involves submitting the voting envelope to the mailbox of the local municipality [42]. In the context of this work, the focus lies on the postal voting approach.

2.1.1 Voting Artifacts

Offering RPV to cast votes requires producing and distributing specific voting artifacts. As the names of these artifacts are in German, the following English names and definitions provided by [42] are used throughout this work:

The Paper Ballot Envelope (PBE) ("Stimmkuvert" [14]) serves as a container for the filled Paper Ballots (PB) ("Stimmzettel" [14]). The Voting Signature Card (VSC) ("Stimmrechtsausweis" [14]) is a pre-printed personalized document that the voter needs to sign to render her vote valid. Finally, the Two-Way Voting Envelope (VE) ("Kuvert zur brieflichen Abstimmung" [14]) is the envelope in which the voter receives all artifacts (PBs, PBE, VSC) needed for RPV. Additionally, the VE is also used as an envelope for sending the completed voting artifacts back to the municipality [42].

2.1.2 Voting Procedures

According to [42], the process of RPV in Switzerland (*cf.* Figure 2.1) can be divided into six subsequent phases [42]:

1. **Setup**: This phase consists of the production and the subsequent assembly of the VEs. Each VE consists of a VSC, a BPE, and one or more BPs. Municipalities usually outsource this process to external suppliers.
2. **Delivery**: The Swiss Post delivers the VEs to eligible voters.
3. **Casting**: After filling in the PBs and concealing them inside the PBE, the voter puts the signed VSC and the PBE into the VE. The voter now faces the three options mentioned above for casting the VE. For the following steps, however, we assume that the voter submits the VE via RPV.
4. **Storage**: When the VEs arrive at the municipalities' premises, they are stored safely until the tallying phase starts.
5. **Tallying**: First, election officials validate the signature on the VSC, as an invalid (or missing) signature renders the ballot invalid. The PBs are then tallied. Some municipalities manually count the PBs, while others use counting machines or scales to weigh the PBs. After tallying, the municipalities report the results to the cantons, which forward the results to the federal level. PBs and VSCs are stored until the results are final and validated, as these two artifacts are indispensable for recounting.
6. **Destruction**: The last phase consists of physically destroying all PBs and VSCs.

2.1.3 Security Threats and Manipulations

[42] provides an extensive list of threats regarding the current postal voting procedures in Switzerland, identifying potential threats in all but the last phase. Amongst others, threats involve tampering with the ER during the setup phase, interfering with the physical delivery of the VEs, manipulating stored VEs, and manipulations during the tallying phase, to name a few examples [42].

Real-world examples show that these threats are not solely theoretical: During the elections for the cantonal parliament of the canton of Thurgau in 2020, a statistical plausibility check revealed irregularities in the ratio of unaltered and altered ballot lists in the city of Frauenfeld [47]. Investigations revealed that a person involved in the tallying process replaced several PBs containing votes for a particular party with ballots in favor of another party [47]. In 2021, the city of Moutier in the canton of Berne voted on whether it should join the canton of Jura [2]. In this case, a controversy emerged because many people registered as city residents before the vote [2]. The cantonal authorities suspected that some of these new residents only registered to be eligible to vote while not intending to live in Moutier [2]. Another manipulation incident occurred in 2012 during the elections for the city mayor in Porrentruy in the canton of Jura, where someone was caught stealing

VEs to cast manipulated ballots [39, 46]. Similarly, during the cantonal elections in 2012, a person stole VEs from people's mailboxes in the canton of Valais to cast votes instead of them [40, 46].

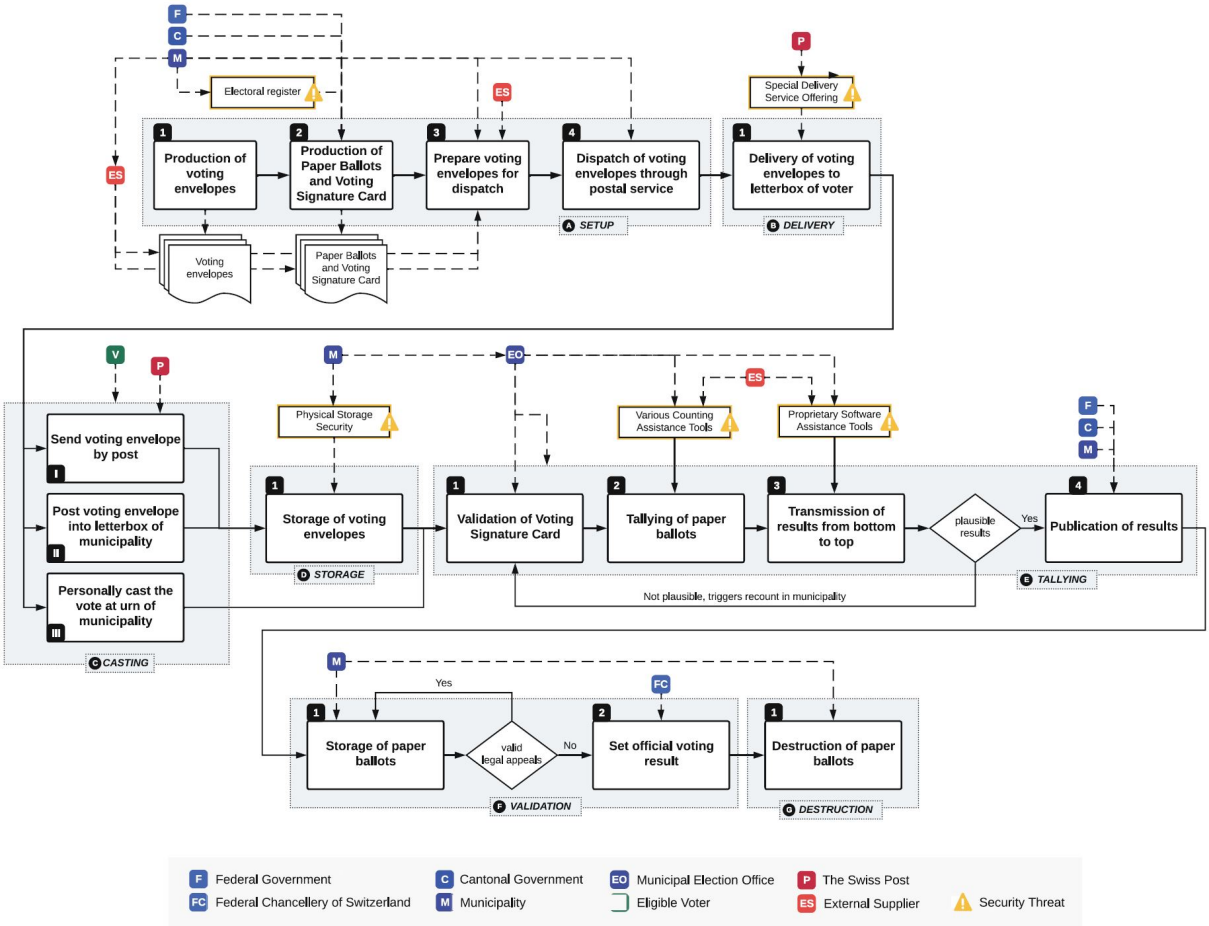

Figure 2.1: Process flow of Remote Postal Voting in Switzerland [42]

2.1.4 Discussion

The Swiss RPV system heavily relies on trusted relationships between the involved actors (*i.e.,* authorities, external suppliers, the Swiss Post, and election helpers) [42]. Despite the decentralization of the system [42], together with the distribution of the trust, which is an advantage of the Swiss RPV [42], manipulation is still possible (*e.g.,* [2, 39, 40, 46, 47]). These incidents raise a question: How can the voters and the election authorities discover manipulation (attempts)? In its current state, the Swiss RPV provides very little verifiability. Thus, the solution is integrating additional verifiability mechanisms into the existing procedures. Chapter 4 presents possible solutions to this problem, leading to a design proposal for the Swiss context in Chapter 5.

2.2 Properties of Voting Protocols

This section introduces the required definitions and terminology for analyzing and comparing voting protocols in Chapter 4. The definitions of the following privacy and verifiability notions are essential for discussing the suitability of a protocol toward the Swiss RPV use case.

Note that the author of this work has already discussed the privacy and verifiability notions in the context of REV in [45]. However, as these properties are also essential to classify RPV protocols, the relevant properties and notions for this work are shortly explained for completeness (*cf.* Sections 2.2.1 and 2.2.2).

2.2.1 Privacy Notions

The privacy properties discussed in this work represent a revised version of the definitions in [45], namely for the following terms: Ballot Privacy, Receipt-Freeness, Coercion Resistance, and Everlasting Privacy.

Privacy is a fundamental challenge in voting systems. In 1981, Chaum [18] delineated a concept for holding privacy-preserving elections, thereby sparking research on privacy in voting systems [37]. This work builds on the following privacy definition:

Definition 1 Ballot Privacy (BP)**:**
BP implies that "no outside observer can determine for whom a voter voted" [37].

Benaloh and Tuinstra [7] noticed that in previous protocols, a voter could keep a receipt of her vote, leading to potential vote selling and coercion. To prevent this issue, they [7] suggested that such systems must be receipt-free:

Definition 2 Receipt-Freeness (RF)**:**
RF denotes that "a voter cannot prove after the election how she voted" [37].

In 2005, Juels et al. [38] presented an even stricter privacy notion, which protects the voter from blackmail [37]:

Definition 3 Coercion Resistance (CR)**:**
CR means that "a voter cannot interact with a coercer *during the election* to prove how she is voting" [37].

The final privacy notion considers the means of achieving privacy in a voting system:

Definition 4 Everlasting Privacy (EP)**:**
EP is achieved if "even a computationally unbounded party does not gain any information about individual votes" [54].

2.2.2 Verifiability Notions

The verifiability notions also represent a revised notion of the definitions discussed in [45], specifically for the terms Individual- and Universal Verifiability, while leaving the definitions of End-To-End-Verifiability, Cast-as-Intended, Recorded-as-Cast, and Tallied-as-Recorded unaltered.

Individual and Universal Verifiability are early verifiability notions (*cf.* [37]). For the context of this thesis, the following definitions apply:

Definition 5 Individual Verifiability (IV)**:**
IV means that "a voter can verify that the ballot containing her vote is in the published set of 'all' [...] votes" [37].

Definition 6 Universal Verifiability (UV)**:**
UV implies that "anyone can verify that the result corresponds with the published set of 'all' votes" [37].

End-to-End Verifiability is a more recent verifiability notion, consisting of the following three notions [37]:

Definition 7 Cast-as-Intended (CaI)**:**
CaI verifiability is provided if "a voter can verify that [...] her choice was correctly denoted on the ballot by the system" [37].

Definition 8 Recorded-as-Cast (RaC)**:**
RaC verifiability is achieved if "a voter can verify that [...] her ballot was received the way she cast it" [37].

Definition 9 Tallied-as-Recorded (TaR)**:**
TaR verifiability means that "a voter can verify that [...] her ballot counts as received" [37].

Finally, this leads to the notion of:

Definition 10 End-To-End Verifiability (E2E-V)**:**
E2E-V is achieved if a system provides CaI, RaC, and TaR verifiability at once [37].

When analyzing a voting protocol, either IV and UV or E2E-V is used to describe verifiability [43]. In the context of this work, the comparison in Chapter 4 (*cf.* Table 4.1) lists IV/UV and E2E-V (*i.e.,* as CaI, RaC, and TaR) together to improve comparability.

Finally, the eligibility of voters to participate in an election or vote can be verified:

Definition 11 Eligibility Verifiability (EV)**:**
EV means that "anyone can check that each vote in the election outcome was cast by a registered voter and there is at most one vote per voter" [44].

Chapter 3

Cryptography

This chapter elaborates on the cryptographic mechanisms used in this work. It starts with introducing two cryptographic primitives, namely HE and Threshold Cryptography, followed by NIZKPs.

Note that the author of this work has discussed the basic definitions of the terms *HE* and *Zero-Knowledge Proofs* in [45]. However, this work discusses these two terms in more detail, focusing on the mathematical foundations of specific schemes. They are crucial for designing and implementing an RPV system.

3.1 Homomorphic Encryption

HE algorithms allow performing operations on ciphertext without prior decryption [58], a valuable property for electronic voting. More formally (*cf.* Equation 3.1):

Definition 12 Homomorphic Encryption (HE)**:**
An encryption scheme is "*homomorphic over an operation '$\star$' if it supports the following equation*:

$$E(m_1) \star E(m_2) = E(m_1 \star m_2), \forall m_1, m_2 \in M \tag{3.1}$$

where E is the encryption algorithm and M is the set of all possible messages" [1].

3.1.1 ElGamal Cryptosystem

A popular HE scheme for electronic voting systems is the ElGamal [27] cryptosystem, which is based "*on the difficulty of computing discrete logarithms over finite fields*" [27].

The ElGamal cryptosystem consists of four algorithms [8]. The `KeyGen` algorithm relies on the parameters p and q (both large primes) [8] and a generator g [1]. `KeyGen` (*cf.* Equation 3.2) generates a secret key sk by selecting a random number from $\mathbb{Z}_q$, which is used to derive a public key pk [8]:

Definition 13 ElGamal `KeyGen()` algorithm [8]**:**

$$(pk, sk) = (g^{sk} \pmod p, sk) \tag{3.2}$$

The `Encrypt` (*cf.* Equation 3.3) algorithm creates a ciphertext representation of a message by first choosing a random number r from $\mathbb{Z}_q$ and then performing the following operation to obtain the ciphertexts c and d [8]:

Definition 14 ElGamal `Encrypt(`pk`,`m`)` algorithm [8]**:**

$$(c, d) = (g^r \pmod p, m \cdot pk^r \pmod p) \tag{3.3}$$

To decrypt the ciphertexts c and d, the receiver applies the `Decrypt` (*cf.* Equation 3.4) algorithm [8], which reveals the original message:

Definition 15 ElGamal `Decrypt(`sk`,(`c`,`d`))` algorithm [8]**:**

$$m = \frac{d}{c^{sk}} \pmod p \tag{3.4}$$

Alternatively, the ElGamal cryptosystem is also compatible with an elliptic curve group instead of a cyclic group [8]. Moreover, this version of the ElGamal cryptosystem is homomorphic for multiplications [1, 8].

To combine two ciphertexts, we can use the `Add` (*cf.* Equation 3.5) algorithm, which performs a homomorphic multiplication [8]:

Definition 16 ElGamal `Add(`(c,d)`,`(c',d')`)` algorithm [8]**:**

$$(c'', d'') = (c \cdot c' \pmod p, d \cdot d' \pmod p) \tag{3.5}$$

3.1.2 Exponential ElGamal

The Exponential ElGamal version is homomorphic for additions [8], making it suitable for homomorphic tallying. For this purpose, the `Encrypt` algorithm (*cf.* Equation 3.6) needs a slight change [8]:

$$d = g^m \cdot pk^r \pmod p \tag{3.6}$$

The `Add` algorithm now results in addition [8]. The downside of this approach is the `Decrypt` algorithm: It requires solving the discrete logarithm problem to restore m, which for small numbers can be circumvented by brute-forcing the result [8].

3.2 Threshold Cryptography

Threshold cryptography prevents single entities from decrypting messages independently [8, 41], which is especially useful for preserving privacy in voting systems.

This work relies on a threshold version of the ElGamal cryptosystem, where all participants must cooperate [8] to decrypt messages. Provotum 2.0 [41] applies the same principle for ballot encryption.

In the beginning, all participants generate standard ElGamal key pairs (pk_i, sk_i), which are then combined into a single public key pk (*cf.* Equation 3.7) [8, 41]:

Definition 17 ElGamal Distributed Key Generation [8, 41]**:**

$$pk = \prod_{i=1}^{n} pk_i \pmod{p} \tag{3.7}$$

Then, encryption is performed with the ElGamal encryption algorithm (*cf.* Equation 3.3) using the public key pk [8, 41]. To restore the original message, the participants must first decrypt (c, d), which yields their shares s_i (*cf.* Equation 3.8), as these shares (n denotes the number of shares) are needed for restoring the original message (*cf.* Equation 3.9) [8, 41]:

Definition 18 ElGamal Cooperative Decryption [8, 41]**:**

$$s_i = c^{sk_i} \pmod{p} \tag{3.8}$$

$$m = \frac{d}{\prod_{i=1}^{n} s_i} \pmod{p} \tag{3.9}$$

3.3 Non-Interactive Zero-Knowledge Proofs

RPV and REV systems usually require one entity to be able to prove a fact to another entity, while keeping any additional details secret. These systems often use ZKPs for this purpose. As previously discussed in [45], the term ZKP is defined as follows:

Definition 19 Zero-Knowledge Proof (ZKP)**:**
"Zero-knowledge proofs are defined as those proofs that convey no additional knowledge other than the correctness of the proposition in question." [31]

ZKPs are interactive per se, which means that the prover and the verifier must be able to send messages to each other [31]. This interactivity is usually not desirable for voting systems, as the interaction between a voter and the Voting Authority is not recurring. This interaction includes the verifier sending a challenge to the prover, which must then respond accordingly [8]. To adapt ZKPs toward the requirements of voting systems, they can be made non-interactive. For this purpose, Fiat and Shamir [29] defined a

transformation allowing to convert interactive ZKPs to NIZKPs: Instead of the challenge-response mechanism, the prover creates a challenge by hashing a commitment [8]. The following proofs all use this technique to make them non-interactive.

3.3.1 Chaum-Pedersen Proof

The Chaum-Pedersen Proof [17] was initially designed as a signature scheme [17]. *Helverify* uses the Chaum-Pedersen proof in two variations. The first version proves the correct decryption of a ciphertext share when cooperatively decrypting a ciphertext (*cf.* Provotum [41]). The other version of the Chaum-Pedersen Proof proves that a ciphertext contains a specific value (*i.e.,* in this case, the value 1). While, at first, this might not seem of use, it is beneficial for proving that the homomorphic addition of multiple ciphertexts is equal to 1. *Helverify* uses this approach to ensure the correct construction of ballots.

Proof of Correct Decryption

Definition 20 Proof of Correct Decryption [8, 41]**:**
To create the proof, the prover must provide the ciphertext (a, b), the public key share pk_i, the private key share sk_i, and the public parameters p, q, and g as inputs. The prover selects a random number r from $\mathbb{Z}_q$. Then, she calculates the two values u and v:

$$(u, v) = (a^r \pmod p, g^r \pmod p)$$

To make the proof non-interactive, the challenge c is obtained by applying a hash function H on the public key share pk_i, the ciphertext (a, b), and the values (u, v):

$$c = H(pk_i, a, b, u, v) \pmod q$$

Next, the prover calculates the values s and d, where sk_i is the private key share of the prover:

$$s = r + c \cdot sk_i \pmod q$$
$$d = a^{sk_i} \pmod p$$

Finally, the proof π is constructed as follows:

$$\pi = (d, u, v, s)$$

Definition 21 Verification of Correct Decryption [8, 41]**:**
The inputs for the verification consist of the ciphertext (a, b), the public key share pk_i, and the public parameters p, q, and g. To verify the proof, the verifier must first recreate the challenge hash c:

$$c = H(pk_i, a, b, u, v)$$

Afterward, the verifier checks if the following equation holds:

$$a^s = u \cdot d^c \pmod p \wedge g^s = v \cdot (pk_i)^c \pmod p$$

A successful check verifies that the decryption has been performed correctly.

Proof of Ciphertext Value

Definition 22 Proof that a ciphertext contains the value 1 (modified version of Disjunctive Chaum-Pedersen Proof (DCP) [8, 41])**:**
The input for creating this proof consists of the ciphertext (a, b), the public key h, the private key x, and the public parameters p, q, and g. As in the proof of correct decryption, the prover starts by selecting a random number r from $\mathbb{Z}_q$. Afterward, she calculates the values (u, v):

$$(u, v) = (g^r \pmod p, h^r \pmod p)$$

Again, the challenge is the result of applying a hash function H to the ciphertext (a, b) and the parameters (u, v):

$$c = H(h, a, b, u, v) \pmod q$$

Finally, the prover calculates s as

$$s = r + c \cdot x \pmod q$$

which is the final parameter needed to construct the proof π:

$$\pi = (u, v, s)$$

Definition 23 Verification that a ciphertext contains the value 1 (modified version of DCP [8, 41])**:**
For the verification, the verifier must provide the ciphertext (a, b), the public key h, and the public parameters p, q, and g as inputs. Again, the first step of the verification is to recreate the challenge hash c:

$$c = H(h, a, b, u, v) \pmod q$$

The actual verification consists of checking the following equation:

$$g^s \pmod p = u \cdot a^c \pmod p \wedge h^s \pmod p = v \cdot (\frac{b}{g})^c \pmod p$$

If this equation evaluates to *true*, it means that the given ciphertext (a, b) contains the value 1.

3.3.2 Disjunctive Chaum-Pedersen Proof

Cramer et al. [19] provide a mechanism applicable to the Chaum-Pedersen proof, which allows a verifier to verify that the prover knows one out of two secrets [8]. Like Provotum [41], *Helverify* uses the DCP to prove that a ciphertext represents either 0 or 1, which helps ensure the correct ballot creation.

Definition 24 Proof if the message is 0 [8, 41]**:**
As inputs, the prover provides the ciphertext (a, b) together with the public key h and the randomness x, which have been used to encrypt the message. Also, the prover provides the public parameters p, q, and g. First, the prover selects three random values r'_0, r_1, and c_1 from $\mathbb{Z}_q$. Then, the prover simulates a proof for the scenario where the message is equal to 1 (as this is not the case) by calculating the values u_1 and v_1:

$$u_1 = \frac{g^{r_1}}{a^{c_1}} \pmod p$$

$$v_1 = \frac{h^{r_1}}{(\frac{b}{g})^{c_1}} \pmod p$$

Afterward, the prover calculates the prove parameters u_0 and v_0 for the actual message (*i.e.,* message is equal to 0):

$$u_0 = g^{r'_0} \pmod p$$

$$v_0 = h^{r'_0} \pmod p$$

As in all the other proofs discussed in this chapter, the construction of the challenge involves applying a hash function H as follows:

$$c = H(h, a, b, u_0, v_0, u_1, v_1) \pmod q$$

The final two values needed for the proof are c_0 and r_0:

$$c_0 = c - c_1 \pmod q$$

$$r_0 = r'_0 + c_0 \cdot x \pmod q$$

The resulting proof π looks as follows:

$$\pi = (u_0, u_1, v_0, v_1, c_0, c_1, r_0, r_1)$$

Definition 25 Proof if the message is 1 [8, 41]**:**
The input parameters are the same as in the first scenario. As before, the prover chooses three random values r'_1, r_0, and c_0 from $\mathbb{Z}_q$. In this scenario, the actual message contains the value 1. Thus, the verifier simulates the proof for the case where the message is 0 by calculating the values u_0 and v_0:

$$u_0 = \frac{g^{r_0}}{a^{c_0}} \pmod p$$

$$v_0 = \frac{g^{r_1}}{b^{c_1}} \pmod p$$

The proof for the actual message is produced in analogy to the first scenario, this time yielding the variables u_1 and v_1:

$$u_1 = g^{r'_1} \pmod p$$

$$v_1 = h^{r'_1} \pmod p$$

Again, the challenge is the result of applying a hash function H to the calculated public proof elements:

$$c = H(h, a, b, u_0, v_0, u_1, v_1) \pmod q$$

The prover calculates the values c_1 and r_1:

$$c_1 = c - c_0 \pmod q$$

$$r_1 = r_1' + c_1 \cdot x \pmod q$$

The resulting proof π looks exactly like the proof in the other scenario:

$$\pi = (u_0, u_1, v_0, v_1, c_0, c_1, r_0, r_1)$$

Definition 26 Verification of message being either 0 or 1 [8, 41]**:**
For the verification, the verifier provides the ciphertext (a, b) together with the public key h and the public parameters p, q, and g. The verifier then checks the following five equations to find out if the proof is valid:

$$g^{r_0} \pmod p = u_0 \cdot a^{c_0} \pmod p$$

$$g^{r_1} \pmod p = u_1 \cdot a^{c_1} \pmod p$$

$$h^{r_0} \pmod p = v_0 \cdot b^{c_0} \pmod p$$

$$h^{r_1} \pmod p = v_1 \cdot (\frac{b}{g})^{c_1} \pmod p$$

$$c_0 + c_1 \pmod q = H(h, a, b, u_0, v_0, u_1, v_1) \pmod q$$

If these equations all yield *true*, the verifier knows that the corresponding ciphertext contains either the message 0 or 1.

3.3.3 Schnorr Proof

The Schnorr Proof [60] is often used to prove that someone owns the private key to a particular public key (*e.g.,* in [41]). *Helverify* uses the Schnorr Proof [60] for the same purpose.

Definition 27 Proof of Private Key Ownership [60, 41]**:**
The prover must provide the public key h and the corresponding private key x, together with the public parameters p, q, and g, to create the proof. Then, the prover picks a random number a from $\mathbb{Z}_q$ and calculates the value b:

$$b = g^a \pmod p$$

The prover applies the hash function H to the public key h and b to create the challenge c, making the proof non-interactive:

$$c = H(h, b) \pmod q$$

The prover calculates the parameter d as follows:

$$d = a + c \cdot x \pmod q$$

The proof π consists of the values c and d:

$$\pi = (c, d)$$

Definition 28 Verification of Private Key Ownership [60, 41]**:**
The verifier provides the public key h to be verified as an input, together with the public parameters p, q, and g. As a first step, the verifier must recalculate the value b:

$$b = \frac{g^d}{h^c} \pmod{p}$$

Then, the verifier reconstructs the challenge c'

$$c' = H(h, b) \pmod{q}$$

which is then compared against the challenge c according to the following equation:

$$c' = c \pmod{q} \wedge g^d \pmod{p} = b \cdot h^c \pmod{p}$$

If this equation holds, then the verifier can be sure that the prover owns the private key to the corresponding public key h.

Chapter 4

Related Work

This chapter elaborates on related work proposing protocols to enable verifiability in RPV systems. Furthermore, these works are compared and analyzed concerning the feasibility of application in Swiss RPV processes.

4.1 Verifiable Postal Voting

Benaloh et al. [6] propose "Verifiable Postal Voting", a protocol to enhance existing RPV procedures by adding cryptographic information to the PBs. The protocol is based on three pieces of paper that the voter prints at home: The first paper (`VR`) contains the voters' choice in plaintext. The second paper (`VI`) contains an encrypted version of the content of `VR`, together with the random values `RI` and `RE` used to encrypt both `VI` and `VE` respectively. Finally, the third piece of paper (`VE`) contains a re-encrypted version of `VI` alongside a digital signature of the voter. The voter puts all these three papers into the envelope and sends it to the municipality. Note that the voter keeps a copy of `VE`, which is later used for verification [6].

When arriving at the tallying facility, officials open the envelopes and verify the signatures on `VE` [6]. Afterward, `VI` is physically stapled to either `VR` or `VE`. Next, the papers are sorted according to their contents, resulting in 4 shuffled piles (`VR`, `VI` + `VE`, `VR` + `VI`, `VE`). Finally, from each of these piles, specific information is published to the PBB (*cf.* Figure 4.1). Thus, the PBB contains all three papers of each voter but only one of two links (`RE` or `RI`) to preserve privacy [6].

This protocol achieves BP under the assumption that the tallying personnel complies with the process rules defined in the protocol [6]. However, the system does not achieve RF, as the voter can prove how she voted if she keeps `VI` as a receipt. IV and CaI verifiability are achieved by letting the voter check the plaintext vote `VR` before sealing the envelope. As all plaintext votes (`VR`) are published to the PBB, the protocol fulfills UV. The protocol also achieves Counted-as-Cast (CaC) verifiability by letting the tallying observers decide for each envelope whether to check for the equality of `VE` and `VI` or `VR` and `VI`, respectively [6].

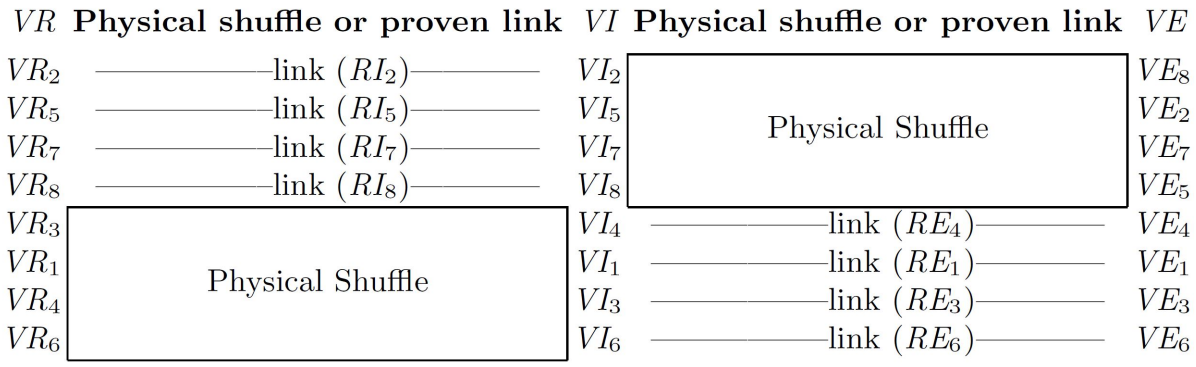

Figure 4.1: Contents of the Public Bulletin Board [6]

4.2 Towards Verifiable Remote Postal Voting with Paper Assurance

McMurtry et al. [51] provide a similar approach for a verifiable RPV system. This approach also leverages cryptography to enhance the traditional approach to conducting RPV, *e.g.,* using CS in combination with other cryptographic primitives.

1. **Setup**: To set up the election, the Voting Authority (VA) stores a double envelope with the encrypted (inner) and the plaintext (outer) Voter IDs onto the PBB [51].

2. **Cast**: The voter then generates two random secrets for producing two commitments. These commitments are then posted to the PBB. The voter now chooses her vote. Afterward, a MAC is generated and published on the PBB. The voter can now print two pieces of paper: Paper 1 contains the vote in plaintext, the inputs for the two commitments, and two ZKPs. Paper 2 consists of the Voter ID. The voter now places Paper 1 in a PBE, puts it into the VE together with Paper 2, and sends it to the tallying location via postal service [51].

3. **Processing**: After the VE has arrived at the tallying location, the VA opens the VE and removes Paper 2, which allows them to validate the voter's identity. Then, the VA gets the encrypted Voter ID from the inner envelope on the PBB. The encrypted Voter ID is then stapled to Paper 1 blindly. Afterward, the VA destroys Paper 2. Before Paper 1 and the encrypted Voter ID are stored on the PBB, both ZKPs are validated [51].

4. **Tallying**: The VA trustees cooperatively decrypt the voters' secrets to start the tallying process. They then validate whether the commitments are valid. Finally, the system generates a new MAC from the decrypted parameters and compares it to the original MAC as posted by the voter [51].

The suggested protocol provides BP assuming that an attacker does not collude with the VA, the postal service, or the voter's device [51]. Furthermore, RF is achieved, assuming that all actors strictly follow the protocol. Moreover, the protocol fulfills CaI verifiability

by letting the voter verify the plaintext on Paper 1 before casting. Recorded-as-Intended (RaI) verifiability is achieved by first performing the CaI check, then validating that the Voter ID on Paper 2 is correct, followed by checking the inclusion of the Voter ID in the PBB after the voting phase. The protocol achieves UV by verifying the proofs generated during the tallying phase and verifying the uniqueness of Voter IDs on the PBB. The presence of an EV mechanism provided by the VA is assumed [51].

4.3 STROBE-Voting

With STROBE-Voting [5], Benaloh presents a protocol for achieving E2E-V in both postal and in-person voting processes.

1. **Setup**: For each eligible voter, the VA generates two ballots containing the same options. Both ballots are encrypted using the ElGamal [27] cryptosystem on a per-row basis (*cf.* Figure 4.2), which means that each candidate option is represented as an encrypted vector of zeros and a one [5]. Subsequently, each encrypted vector is hashed and truncated to generate shortcodes, which serve as a reference for the voter to ensure verifiability. Next, all encrypted options on the ballot are input into a hash function, producing a ballot hash, a hash string that ensures that all encrypted options remain unaltered. Finally, the printed ballot (*cf.* Figure 4.3) contains the ballot hash and the shortcodes for each option [5].

2. **Voting**: When receiving the PBs, the voter fills out one of the PBs and keeps the other for verification. The voter memorizes the selected options' shortcodes and the ballot hash of the selected PB and then casts the PB [5].

3. **Receival**: When the VEs arrive at the VA's premises, the VA publishes encryptions, shortcodes, and the ballot hashes of the cast and the retained PB. For the PB cast, the selected shortcodes (together with the ballot hash) are also published. The retained PB is decrypted and published, together with the randomness factors used to encrypt it in the first place. Finally, the VA publishes multiple NIZKPs to ensure the correctness of the PBs [5].

4. **Tallying**: The process involves homomorphically adding the encrypted votes and decrypting the tally results [5].

Candidate voted for	Vote (vector of encryptions)	Sample selection code
Alice	$\langle \mathcal{E}(1), \mathcal{E}(0), \mathcal{E}(0), \mathcal{E}(0) \rangle$	Q4
Bob	$\langle \mathcal{E}(0), \mathcal{E}(1), \mathcal{E}(0), \mathcal{E}(0) \rangle$	D6
Carol	$\langle \mathcal{E}(0), \mathcal{E}(0), \mathcal{E}(1), \mathcal{E}(0) \rangle$	L7
did-not-vote	$\langle \mathcal{E}(0), \mathcal{E}(0), \mathcal{E}(0), \mathcal{E}(1) \rangle$	R9

Figure 4.2: Encryption of ballot options in STROBE-Voting [5]

Alice	⬭	Q4
Bob	⬭	D6
Carol	⬭	L7
none		R9

Ballot code: `XC3K0-A21BM-8WP8Q-MWQ6E-UYW9Y-ZPBL5-93LRE-M3J62-MJ1W7-87DYF`

Figure 4.3: Example of PB in STROBE-Voting [5]

STROBE-Voting [5] provides E2E-V by applying the following mechanisms: The voter memorizes shortcodes selected and validates against the published shortcodes for the particular ballot hash. Additionally, the correctness of the cast PB is checked by verifying the NIZKPs and the PB's twin. Furthermore, it achieves BP by applying encryption to the PBs [5].

4.4 RemoteVote

RemoteVote [20] has been inspired by STROBE-Voting [5]. The PB (*cf.* Figure 4.4) consists of the options, two encrypted ballots, each defining a column of shortcodes for all options, and a ballot ID that combines the identifiers of each encrypted ballot [20].

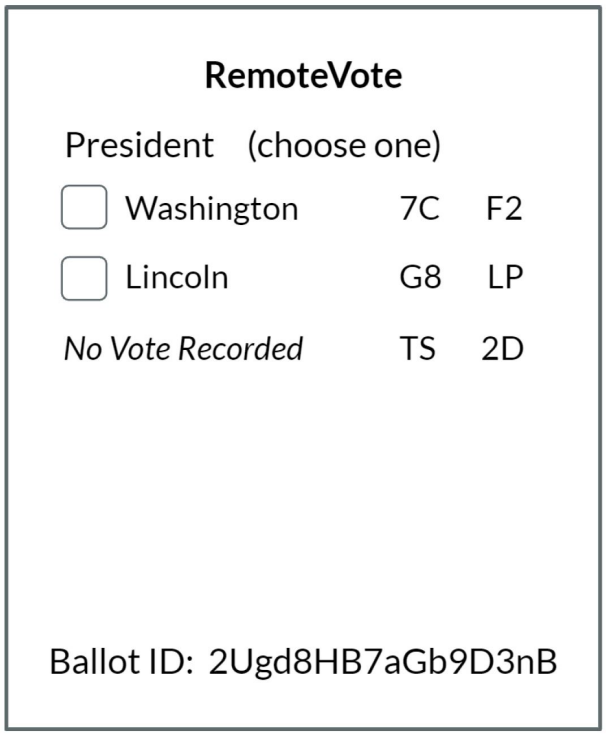

Figure 4.4: Example of PB in RemoteVote [20]

1. **Election Setup**: First, the VA constructs the ballots according to the requirements of the specific election, which involves encrypting the options together with the corresponding commitments and the generation of shortcodes and ballot codes [20].

2. **Paper Ballot Creation**: For each voter, the VA combines two encrypted ballots into one physical PB, which follows the structure described above (and in Figure 4.4). Note that the ballot ID, the longcodes of the encrypted ballots, and the shortcodes, together with the generated commitments, are published on the PBB, allowing for later verification. The PBs are then sent to the voters by postal mail [20].

3. **Vote Casting**: The voter selects the candidates on the PB by filling in the ballot by hand while memorizing selected candidates' shortcodes, together with the ballot ID, as these attributes allow the voter to perform verification later [20].

4. **Receival**: When the VA receives the PBs, it posts the PBs' evidence (ballot ID and selected shortcodes) to the PBB. From this point, voters can verify if their vote has been recorded correctly [20].

5. **Tallying**: An MN anonymizes the commitments before calculating the final tally by revealing the committed values [20].

Like STROBE-Voting [5], RemoteVote [20] achieves E2E-V by allowing the voter to verify that their shortcodes have been processed correctly. From a privacy standpoint, RemoteVote achieves BP and RF. Additionally, it fulfills EP, as it uses commitments for the tallying process instead of HE [20].

4.5 SAFE Vote

SAFE Vote [20] can be regarded as a sibling protocol to RemoteVote [20], which manifests in identical steps 1, 4, and 5. However, the PB (*cf.* Figure 4.5) is slightly different, as it contains one row of shortcodes, a scratch-off label hiding a random value, the ballot ID, and a QR-Code [20].

1. **Election Setup**: Identical to RemoteVote [20].

2. **Paper Ballot Creation**: The VA produces the PB by printing a QR-Code containing the ballot commitment onto the PB and the random value used to produce the commitment as a scratch-off label [20].

3. **Vote Casting**: Before the voter selects her candidates, she can verify that the shortcodes printed on the ballot are valid by removing the scratch-off label and scanning the QR-Code. With these two items, the voter can reconstruct the ballot and check whether the reconstructed ballot matches the actual PB [20].

4. **Receival**: Identical to RemoteVote [20].

5. **Tallying**: Identical to RemoteVote [20].

SAFE Vote fulfills the same properties as RemoteVote while providing individual CaI verification compared to the collective CaI verification in RemoteVote [20].

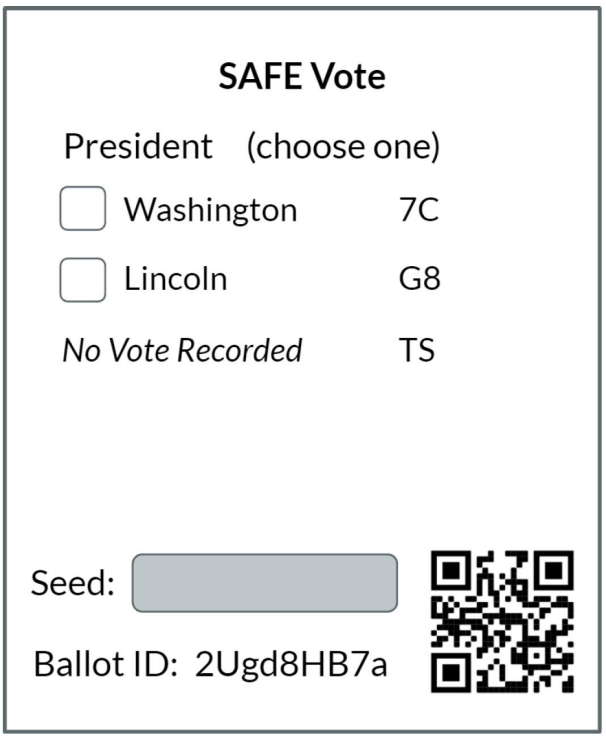

Figure 4.5: Example of PB in SAFE Vote [20]

4.6 Comparison and Discussion

This section discusses the suitability of the presented protocols regarding their use in the context of Swiss RPV. Table 4.1 compares the protocols by highlighting their properties and cryptographic primitives.

4.6.1 Trust Assumptions

The presented protocols use different trust assumptions to achieve the listed properties. Verifiable Postal Voting [6] assumes that the election procedures are public and are asserted by observers, with at least one honest observer [6]. McMurtry et al. [51] base the privacy of their system on the assumption that, amongst others, the election authority, the postal services, and the voter's device are honest [51]. Further, STROBE-Voting [5] presumes that the ballot printers do not store secret information about the ballots they print [5]. RemoteVote and SAFE Vote do not list any explicit assumptions [20].

4.6.2 Swiss RPV Criteria

A desirable property of an RPV protocol for use in Switzerland is the compatibility with existing voting procedures. Thus, a protocol should be compatible with ballot delivery and casting by postal mail. Likewise, such a protocol should not interfere with other voting channels, such as in-person voting. Furthermore, the goal is to achieve as much

verifiability (*e.g.,* E2E-V) while keeping the protocol complexity as low as possible, as introducing complexity into a system can lead to security issues [50], which is undesirable in a voting system. Moreover, a general requirement is the ease of use for voters and VA alike when running the protocol. A system with poor usability could negatively impact voter turnout, which is unwanted.

Last but not least, an RPV system should remain operable despite the failure of computer systems. This resilience could be achieved by basing the system on the existing RPV procedures, which can serve as a fallback. After all, this work does not aim to build an REV system but to augment the existing postal voting procedures toward verifiable RPV.

Table 4.1: Comparison of RPV protocols

Protocol	BP	RF	EP	IV	UV	CaI	RaC	TaR	EV	Cryptographic Primitives
Verifiable Postal Voting [6]	✓	✗	✗	✓	✓	✓	✗	✗	✓	HE/MN, DS, NIZKP, (CS)
Towards Verifiable Remote Postal Voting with Paper Assurance [51]	✓	(✓)	✗	[✓]	✓	✓	✓	✗	✓	HE, MN, NIZKP, MAC, CS
STROBE-Voting [5]	✓	✓ [20]	✗	[✓]	[✓]	[✓]	[✓]	[✓]	✓*	HE, NIZKP
RemoteVote [20]	✓	✓	✓	[✓]	[✓]	✓	✓	✓	✓	HE, CS, MN
SAFE Vote [20]	✓	✓	✓	[✓]	[✓]	✓	✓	✓	✓	HE, CS, MN

[]=deduced, ()=partially, *=compatible with existing mechanisms

4.6.3 Protocol Suitability

Determining a suitable protocol as a basis for the design of this work requires comparing the highlighted protocols to the criteria explained above. [6] and [51] do not provide E2E-V and are not designed for postal ballot delivery. The three remaining protocols [5, 20] are compatible with postal ballot delivery and ballot casting while providing E2E-V. Furthermore, they do not impede other voting channels. When comparing the cryptographic primitives used, STROBE-Voting [5] uses HE and NIZKPs resulting in relatively low complexity. RemoteVote [20] and SAFE Vote [20] use HE, CS, and an MN. From a privacy standpoint, STROBE-Voting [5] trades EP in favor of a more straightforward design while still achieving a similar degree of verifiability compared to RemoteVote and SAFE Vote.

The design specified in Chapter 5 will be based on STROBE-Voting [5], as it provides a suitable combination of privacy and verifiability despite its simple design. It also offers a single-ballot variant, which fits nicely into the context of Swiss RPV, requiring only minor additions to existing voting procedures. Moreover, the design of STROBE-Voting [5] is easily extendable with ideas from RemoteVote [20] or SAFE Vote [20], as both are based on STROBE-Voting [5].

Chapter 5

Design

This chapter delves into the requirements of *Helverify* for its use as a verifiable Swiss RPV protocol. The protocol itself is based on the mechanisms of STROBE-Voting [5], with adaptions toward the Swiss RPV context. Furthermore, this chapter discusses the system architecture, which is the basis for the implementation (*cf.* Chapter 6).

5.1 Requirements

Before delving into concrete use cases, it is essential to elicit and classify the requirements of an RPV system. The requirements for this work are mainly influenced by the current Swiss RPV procedures [42] and the current Swiss legislation [22, 23]. The following sections discuss the functional and non-functional requirements and constraints to be considered in the context of Swiss RPV.

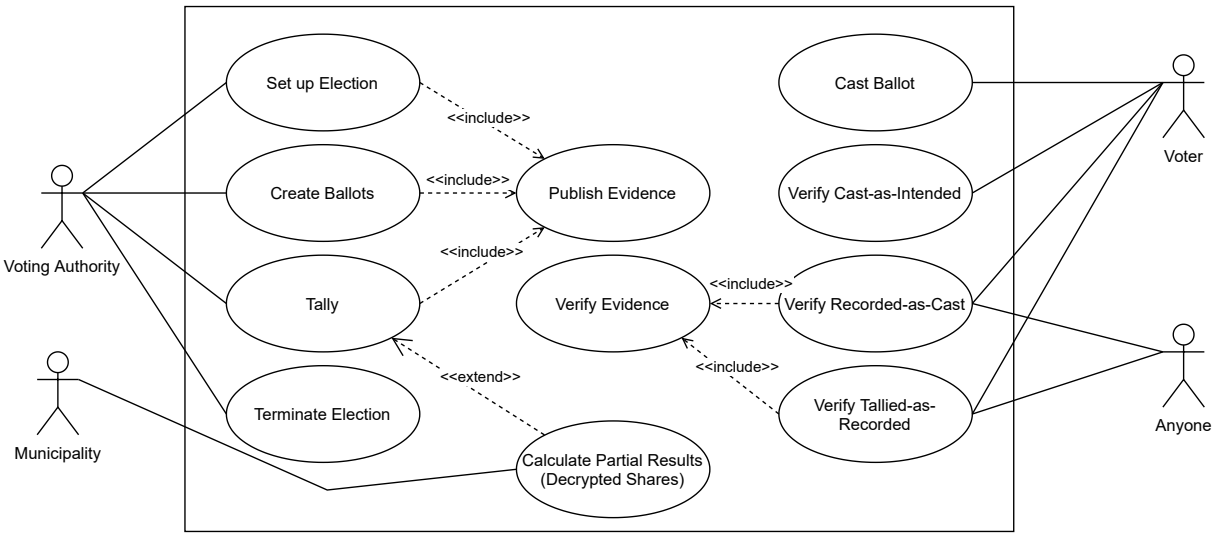

Figure 5.1: Use Cases of *Helverify*

5.1.1 Functional Requirements

The functional requirements (A) for a verifiable Swiss RPV system are relatively straightforward, as they mainly focus on providing verifiability:

- **A1.** The voting process must provide E2E-V for the voter.
- **A2.** Verification procedures must be optional for voters.
- **A3.** The VA must be able to create the PBs and publish evidence of correct ballot creation.
- **A4.** The VA runs the tallying process and publishes evidence of correct tallying.
- **A5.** The voter must be able to cast a ballot.

5.1.2 Non-functional Requirements

Quality aspects are fundamental in RPV systems, as the user base of such a system is potentially very large. Thus, the non-functional requirements (B) primarily revolve around scalability and usability:

- **B1.** The system must scale for national elections and votes.
- **B2.** Verifiability mechanisms must be easy to use for voters and observers.
- **B3.** The voter's privacy must be protected (BP).

5.1.3 Constraints

RPV systems must comply with many legal and procedural guidelines, referred to as constraints in the context of requirements engineering. Thus, to design a usable RPV system in the context of Swiss RPV, it is crucial to account for the following constraints:

- **C1.** The voting protocol must be compatible with existing Swiss RPV procedures.
- **C2.** The system must allow EV procedures as in the Swiss RPV system.
- **C3.** RPV procedures must be resilient to computer infrastructure outages.
- **C4.** The voting processes must be compatible with on-site ballot casting.
- **C5.** The system should use a decentralized PBB to store evidence.

Fulfilling the listed requirements and their corresponding use cases (*cf.* Figure 5.1) requires the definition of a protocol, which is presented in the next section.

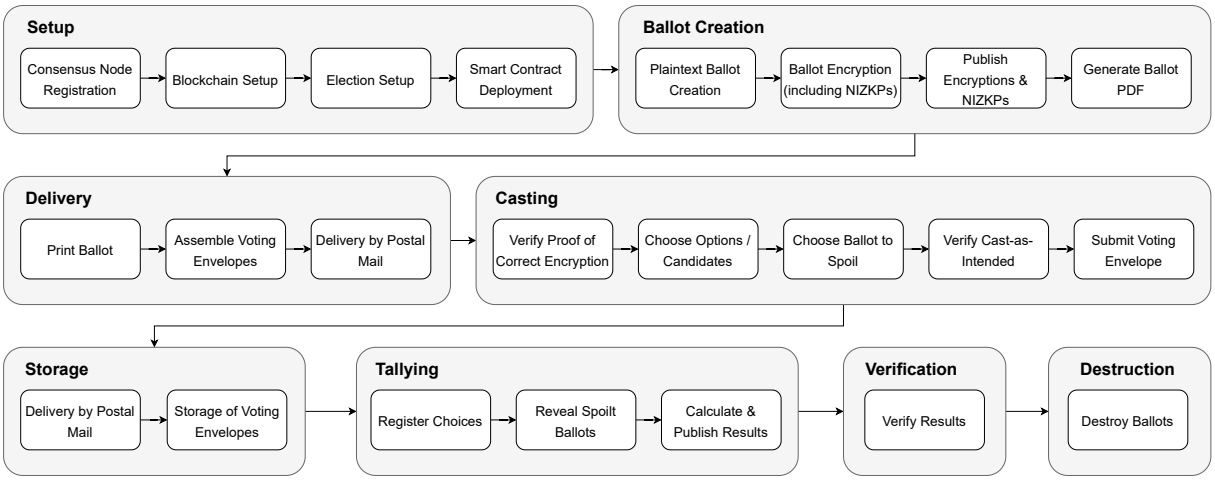

Figure 5.2: *Helverify* Protocol Phases

5.2 Helverify Protocol

The *Helverify* protocol follows the phases of [42], as these phases represent the current procedures of Swiss RPV. Furthermore, the mechanisms applied in these phases follow the suggestions of STROBE-Voting [5]. Figure 5.2 depicts the *Helverify* protocol phases.

Figure 5.3 shows a high-level overview of the process of running an election with *Helverify*. In particular, it shows the required steps from the VA's but also the voter's perspective, including their interactions. The following sections delve into the steps of this process in more detail.

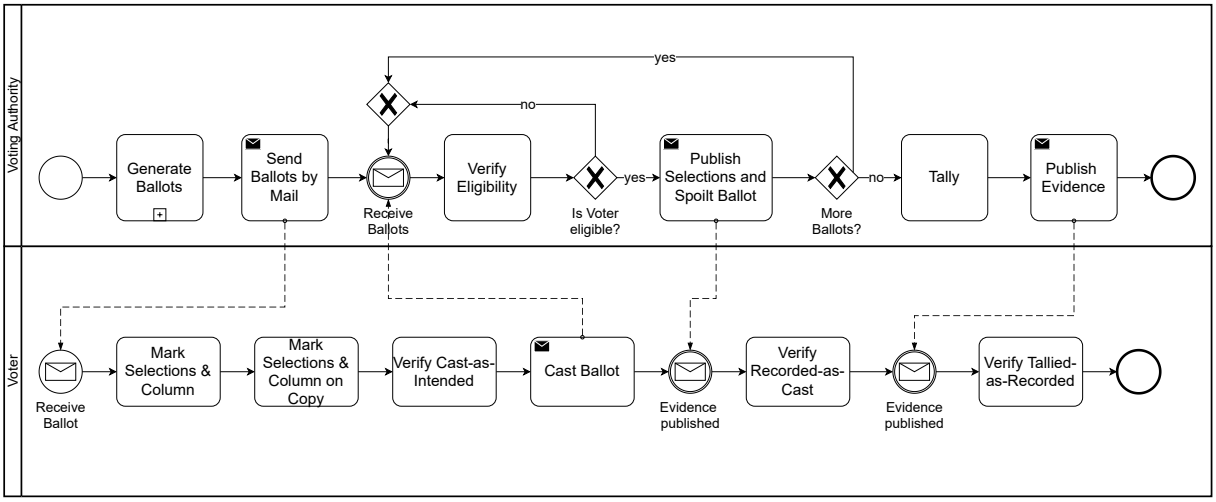

Figure 5.3: Process of running an election in *Helverify*

5.2.1 Setup

At the beginning of the setup phase (*cf.* Figure 5.4), the VA sets up a Proof-of-Authority (PoA) BC in cooperation with multiple municipalities. This mechanism is inspired by

Provotum [41], which follows a similar approach. The municipalities are responsible for running Consensus Nodes forming the PoA BC. Once the municipalities have registered their Consensus Nodes, the VA generates a new BC configuration, which it deploys onto the Consensus Nodes. After successfully deploying the BC, the VA creates a new election by specifying the parameters, such as candidates/options, title, and encryption. Furthermore, the VA requests each Consensus Node to create a public-private key pair and then collects the public keys of all Consensus Nodes. Next, the VA uses these public keys to create a composite (election) public key. For this Distributed Key Generation (DKG) process, each Consensus Node has to provide proof that it owns the private key to the submitted public key, which is done by generating a Schnorr Proof [60]. Finally, the Smart Contract (SC) for the election is deployed on the BC. At this point, the BC is ready to store evidence.

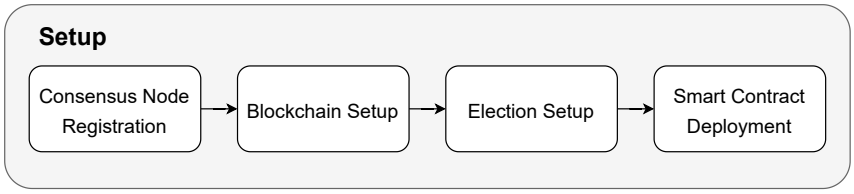

Figure 5.4: *Helverify* Protocol Setup Phase

5.2.2 Ballot Creation

Next, the VA (or a delegated third party) creates the ballots (*cf.* Figure 5.5). The ballots correspond to the STROBE-Voting scheme [5]: Each paper ballot consists of a pair of 'virtual' ballots. A virtual ballot contains a vector of zeros and ones for each candidate or option, where the position inside the vector designates an option/candidate. If the value of a vector element is equal to one, it means that this vector represents the candidate at the element's position. A valid ballot must only contain one occurrence of one per row and per column, respectively. Each value inside this vector is encrypted with the election public key using the exponential ElGamal cryptosystem. To guarantee the correctness of the created ballot, the VA must generate the following proofs for each ballot and publish them to the PBB [5]:

- Proof that the sum of the encrypted values in each row equals one (Chaum Pedersen Proof [17]).
- Proof that the sum of the encrypted values in each column equals one (Chaum Pedersen Proof [17]).
- Proof that each encrypted value is either an encryption of zero or one (Disjunctive Chaum-Pedersen Proof [19]).

The virtual ballots also contain shortcodes and a ballot code, which are created as follows [5]: The VA produces a hash value of each vector of encryptions and takes the first two characters of the resulting hexadecimal representation. These characters serve as

shortcodes, which are printed next to the corresponding candidate/option on the ballot. The ballot code is the hashed value of all encrypted vectors of both virtual ballots, and it is also printed onto the ballot. Finally, the VA publishes the encrypted ballots and the corresponding evidence on the PBB [5]. The final step of the setup phase consists of creating PDF files of the generated ballots for printing.

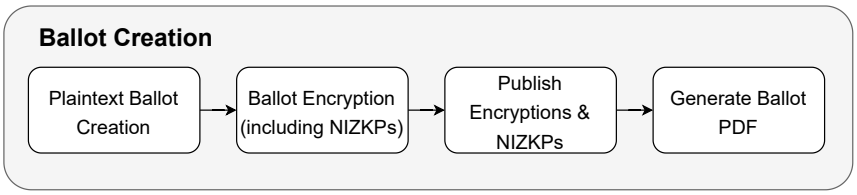

Figure 5.5: *Helverify* Protocol Ballot Creation Phase

5.2.3 Delivery

The first step in the delivery phase (*cf.* Figure 5.6) is to print the ballot PDFs on paper. Following the Swiss RPV process [42], the resulting PBs, PBE, and VSC are placed inside the VE. The Swiss Post then delivers the assembled ballots to the voters. Thus, the delivery phase is the same as in [42].

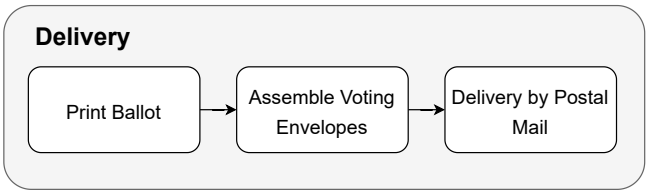

Figure 5.6: *Helverify* Protocol Delivery Phase

5.2.4 Casting

This phase encompasses the ballot casting process (*cf.* Figure 5.7). When the voter receives the VE, she can verify that the ballots have been correctly encrypted by verifying the proofs from the PBB [5]. If the encryptions are correct, the voter chooses a candidate/option by marking the respective checkbox on the PB. The remainder of this phase follows the current Swiss RPV procedures [42], with minor additions from [5] for verifiability: If the voter wants to perform RaC and TaR verification at the end of the election, she crosses the checkbox of one of the two short code columns, indicating that her selection from this set of shortcodes must be published to the PBB (*cf.* single ballot variant in [5]). The voter memorizes the shortcode(s) of the selected option(s) and the ballot code for later verification. Furthermore, the voter notes the shortcodes of the second (unselected) column and their corresponding options/candidates [5]. At this point, the voter verifies that she has selected the intended option (CaI). Next, the voter signs the VSC and puts the PB into the PBE and seals it. Finally, the voter puts the signed VSC and the sealed PBE into the VE, seals it, and sends the VE to the municipality by postal mail [42].

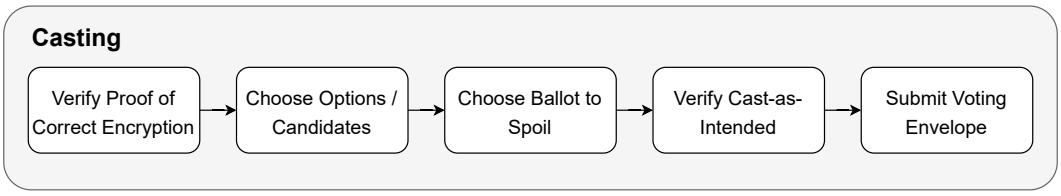

Figure 5.7: *Helverify* Protocol Casting Phase

5.2.5 Storage

The storage phase (*cf.* Figure 5.8) remains unaltered from the original process described in [42]. In other words, the municipalities store the VEs sent by the voters until the tallying process begins.

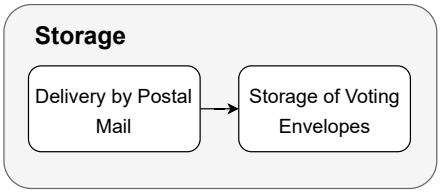

Figure 5.8: *Helverify* Protocol Storage Phase

5.2.6 Tallying

The first step of the tallying phase (*cf.* Figure 5.9) is to verify the voter's eligibility by checking the signature on the VSC [42]. Next, the VA publishes two pieces of evidence on the PBB [5]:

- The shortcodes from the selected short code column representing the voter's choice.
- For the other set of shortcodes (*i.e.,* which the voter did not select), the corresponding decrypted virtual ballot containing the options in plain text and the shortcodes.

The published shortcode(s) enable the voter to verify that her vote has been RaC and the decrypted virtual ballot allows verifying that the shortcodes correspond to the correct options [5]. Note that a malicious VA is caught producing incorrect shortcodes with the probability of $\frac{1}{2}$ for each ballot, as it does not know in advance which virtual ballot the voter chooses [5]. Thus, for n validating voters, the probability of detecting a cheating VA is $\frac{1}{2^n}$ [5]. To determine the final results of the election, the VA performs a homomorphic addition of all selected options, which yields a ciphertext containing the tallies for each candidate/option. To retrieve the result in plaintext, *Helverify* follows the approach of Provotum [41] in applying Cooperative Decryption (CD): The VA sends the ciphertext to the Consensus Nodes, which produce a decrypted share by decrypting the ciphertext with their private keys. The Consensus Nodes also generate a proof of correct decryption (Chaum-Pedersen Proof [17]). The VA verifies the proofs and then combines the shares, which yields the final result in plaintext. Finally, the VA publishes the results on the PBB [5].

5.2.7 Verification

In the verification phase (*cf.* Figure 5.9), the published results and the corresponding evidence [5] allows voters and observers to perform RaC and TaR verification.

5.2.8 Destruction

The destruction phase (*cf.* Figure 5.9) involves the physical destruction of the PBs and VSCs [42] and the tear-down of the PoA BC.

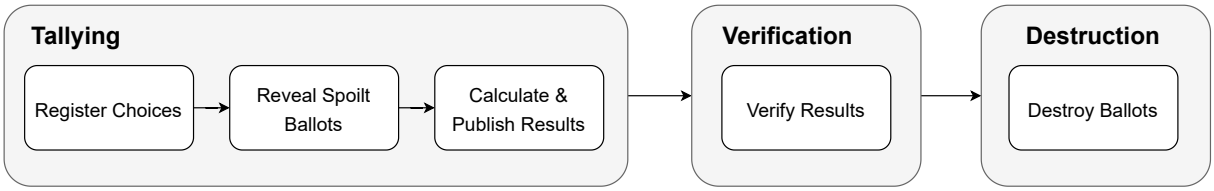

Figure 5.9: *Helverify* Protocol Tallying, Verification, and Destruction Phases

5.3 Architecture

This section provides an overview of *Helverify*'s architecture consisting of the components depicted in Figure 5.10. The following paragraphs explain the capabilities and purposes of these components.

5.3.1 Voting Authority

The VA component consists of a frontend with a corresponding backend. It serves the purpose of allowing election officials to set up the blockchain infrastructure and elections. Furthermore, officials can generate ballots and export them as PDF files. In the tallying phase, the VA component allows registering ballot selections while publishing evidence of the process. Finally, election officials can initiate the tallying process. For this purpose, the ballot selections are homomorphically added and then cooperatively decrypted by the Consensus Nodes. Lastly, the results are persisted on to the election SC.

5.3.2 Consensus Node

The Consensus Nodes serve two purposes: Operating the PoA BC, which hosts the election SC, and participating in the DKG and CD processes for ballot encryption and decryption. Each Consensus Node contains a 'Blockchain Node'. The Blockchain Nodes of all Consensus Nodes are responsible for operating the BC network. The Consensus Node Backend orchestrates both the Blockchain Node and the cryptographic operations by providing an API, which the VA Backend consumes.

5.3.3 Interplanetary File System

Depending on the key length of the ElGamal cryptosystem chosen for an election, ciphertext representations of a ballot can exceed the capabilities of BC-based SCs. Thus, *Helverify* uses a private Interplanetary File System (IPFS) [35] swarm to store encryptions and proofs for the ballots, improving the scalability of the solution. The VA Backend uses the IPFS swarm for storing and retrieving encryptions and evidence (*i.e.,* proofs). Finally, the Voter Frontend uses the IPFS swarm to retrieve evidence for verification.

5.3.4 Voter

The Voter Frontend allows the voter to perform multiple forms of verification. Firstly, the voter can verify that the ballot she received has properly been encrypted by checking the proof of correct encryption for each of the encrypted values in the ballot. After the VA has registered the selected choices from the ballot, the voter can verify that the VA has registered the the ballot choices correctly. In addition, the spoilt ballot verification allows the voter to be assured that the ballot encryption has been performed correctly. Thus, the Voter Frontend allows the voter to perform the verification steps described in [5].

5.3.5 Proof-of-Authority Blockchain

Helverify uses an Ethereum BC for hosting its PBB. Due to the scalability and privacy requirements, *Helverify* uses a BC with the PoA consensus algorithm. As mentioned earlier, the Consensus Nodes operate the BC. *Helverify*'s BC is a platform for hosting the election SC, which provides the PBB functionality for a particular election. The election SC stores references to the storage locations of ballots on IPFS, the selections of a particular ballot, and the results of an election.

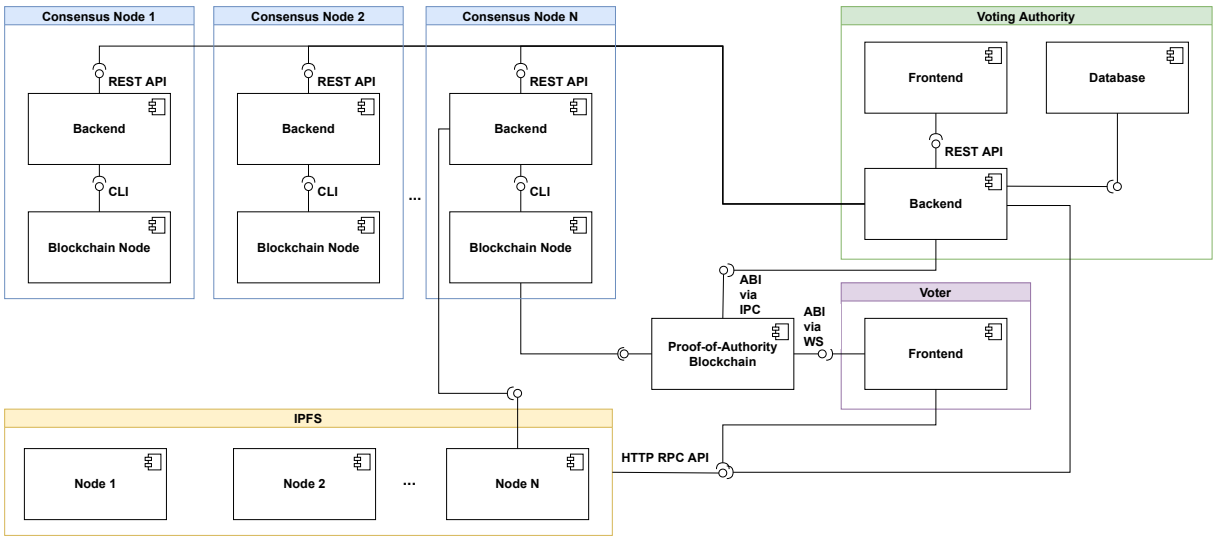

Figure 5.10: Architecture of *Helverify*

Chapter 6

Implementation

This chapter highlights the fundamental aspects of the *Helverify* implementation. From a technological standpoint, several options were available initially, ranging from NodeJS over Rust to C#. The selection of an appropriate technology stack for this implementation was based on the following criteria:

1. Quality of cryptographic libraries
2. Availability of documentation
3. Support for safe cryptographic key lengths (*i.e.,* minimum 2048 bit according to NIST [4])
4. Ease of use

After evaluating the available options, the choice of technology for this implementation is the following:

- C# with ASP.NET 6 [59] for the backend applications providing REST APIs (including cryptographic primitives)
- TypeScript [53] with React [52] for the frontend applications
- Solidity [62] for the election SC
- Docker [24] for containerizing the individual components

6.1 Cryptography Library

The cryptography library is a crucial component of *Helverify*, providing implementations of the ElGamal cryptosystem and various ZKPs. The implementation follows the definitions provided in Chapter 3. This library is implemented in C#, using the Bouncy Castle [48] cryptographic library. The following sections highlight the cryptographic primitives provided in detail.

6.1.1 ElGamal Cryptosystem

The library includes both the standard and exponential versions of the ElGamal cryptosystem. In particular, this comprises key generation, encryption, and decryption algorithms. Furthermore, the library also provides implementations for DKG and CD, which allow the creation of composite public keys and the decryption and combination of decrypted shares to retrieve the plain text cooperatively. Another important feature is the homomorphic addition of ciphertexts, which is essential for the tallying process in the election.

6.1.2 Zero-Knowledge Proofs

The *Helverify* cryptography library also contains the implementations of four types of NIZKPs: Namely, proof of private key ownership, proof of correct decryption, proof of containing zero or one, and proof of containing one. Each proof consists of a proof generation (*e.g.,* Listing 1) and a verification method (*e.g.,* Listing 2).

```
1   /// <summary>
2   /// Creates a proof that the specified ciphertext has been decrypted correctly.
3   /// </summary>
4   /// <param name="a">First component of an ElGamal ciphertext</param>
5   /// <param name="b">Second component of an ElGamal ciphertext</param>
6   /// <param name="publicKeyShare">Public key share</param>
7   /// <param name="privateKeyShare">Private key share</param>
8   /// <returns>Proof of correct decryption</returns>
9   public static ProofOfDecryption Create(BigInteger a, BigInteger b,
10      DHPublicKeyParameters publicKeyShare, DHPrivateKeyParameters privateKeyShare)
11  {
12      BigInteger p = publicKeyShare.Parameters.P;
13      BigInteger q = p.Subtract(BigInteger.One)
14          .Multiply(BigInteger.Two.ModInverse(p))
15          .Mod(p);
16      BigInteger g = publicKeyShare.Parameters.G;
17      BigInteger pk = publicKeyShare.Y;
18      BigInteger sk = privateKeyShare.X;
19
20      BigInteger r = new BigInteger(privateKeyShare.X.BitLength, SecureRandom)
21          .Mod(q);
22
23      BigInteger u = a.ModPow(r, p).Mod(p);
24      BigInteger v = g.ModPow(r, p).Mod(p);
25
26      BigInteger c = HashHelper.GetHash(q, pk, a, b, u, v).Mod(q);
27
28      BigInteger s = r.Add(c.Multiply(sk).Mod(q)).Mod(q);
29
30      BigInteger d = a.ModPow(sk, p).Mod(p);
31
32      return new ProofOfDecryption(d, u, v, s);
33  }
```

Listing 1: Exemplary proof generation (proof of correct decryption)

```
1         /// <summary>
2         /// Allows to verify that the specified ciphertext has been
3         /// decrypted correctly.
4         /// </summary>
5         /// <param name="a">First component of an ElGamal ciphertext</param>
6         /// <param name="b">Second component of an ElGamal ciphertext</param>
7         /// <param name="publicKeyShare">Public key share</param>
8         /// <returns>True if decryption has been performed correctly,
9         /// false otherwise.</returns>
10        public bool Verify(BigInteger a, BigInteger b,
11            DHPublicKeyParameters publicKeyShare)
12        {
13            BigInteger p = publicKeyShare.Parameters.P;
14            BigInteger q = p.Subtract(BigInteger.One)
15                .Multiply(BigInteger.Two.ModInverse(p)).Mod(p);
16            BigInteger g = publicKeyShare.Parameters.G;
17            BigInteger pk = publicKeyShare.Y.Mod(p);
18
19            BigInteger c = HashHelper.GetHash(q, pk, a, b, U, V).Mod(q);
20
21            bool firstCondition = a.Mod(p).ModPow(S, p).Mod(p)
22                .Equals(U.Multiply(D.ModPow(c, p).Mod(p)).Mod(p));
23
24            bool secondCondition = g.Mod(p).ModPow(S, p).Mod(p)
25                .Equals(V.Multiply(pk.ModPow(c, p).Mod(p)).Mod(p));
26
27            return firstCondition && secondCondition;
28        }
```

Listing 2: Exemplary proof verification (proof of correct decryption)

6.2 Consensus Nodes

As the name suggests, the Consensus Nodes are responsible for running *Helverify*'s Ethereum PoA private BC. For this purpose, each Consensus Node exposes a REST API. Thus, the Consensus Node backend allows configuring the BC and provides endpoints for key pair management and CD, as the Consensus Node serves as a trusted entity for DKG and CD in an election.

6.2.1 Blockchain Node

Helverify uses Go Ethereum (GETH) [64] to run private Ethereum nodes, forming the BC for deploying the election SCs. The Blockchain Node consists of a series of scripts that encapsulate GETH interactions:

- `init.sh`: Sets up a clean GETH environment and creates a new Ethereum account for the node.
- `init-genesis.sh`: Initializes the genesis block for the PoA BC.

- `enode.sh`: Extracts the node's identifier, which is needed for discovering other peers.
- `start-consensusnode.sh`: Starts the GETH node, which exposes it to the network.
- `start-mining.sh`: Starts the block sealing process, rendering the node ready for executing BC transactions.

These scripts allow the backend to control the node's state and configuration conveniently.

6.2.2 Backend

The Consensus Node backend is implemented in C# using ASP.NET 6 [59]. It provides a selection of REST API endpoints (*cf.* Figure 6.1), which allow configuring the BC, generating key pairs, and performing CD of ballots. The following sections discuss these three aspects in more detail.

Blockchain Configuration

As mentioned in Section 6.2.1, the backend interacts with the BC node through a series of scripts. The scripts are specifically tailored towards their usability in the API methods. Thus, the structure of the API largely follows the structure of the scripts:

- `/api/blockchain/account`: Uses the `init.sh` script to create a new account and returns the account's address.
- `/api/blockchain/genesis`: Takes a genesis block and initializes the node using `init-genesis.sh`.
- `/api/blockchain/peer`: Starts the node using `start-consensusnode.sh` and then announces the other nodes' addresses by calling `enode.sh`.
- `/api/blockchain/nodes`: Takes the node identifiers of the other BC nodes in the network and announces them to this node.
- `/api/blockchain/sealing`: Calls the `start-mining.sh` script to start the sealing process on the node, after which the Consensus Node is operable.

Key Pair Management

The Consensus Node participates in an election's encryption and decryption. Thus, each Consensus Node holds an ElGamal key pair per election. An election's public key consists of the combination of all the Consensus Nodes' public keys. Thus, it generates and stores these key pairs on the file system by providing the following endpoints:

- `/api/key-pair`: Generates a new ElGamal key pair for the specified election using the ElGamal public parameters p and g. The generated key pair is then stored on the file system and the public key is returned.
- `/api/key-pair/public-key`: Returns the Consensus Node's ElGamal public key for the specified election.

Cooperative Decryption

Due to the encryption of ballots and results with the election's public key, the decryption of the respective ciphertexts involves partial decryptions by each Consensus Node. For this purpose, Consensus Nodes expose the following endpoints to decrypt shares of ballots and results:

- `/api/decryption`: Calculates and returns the decrypted share of a single ElGamal ciphertext together with a proof of correct decryption.
- `/api/decryption/ballot`: Retrieves an encrypted ballot from IPFS to calculate the decrypted share of each ciphertext in the ballot. It returns the decrypted shares together with the respective proof of correct decryption.

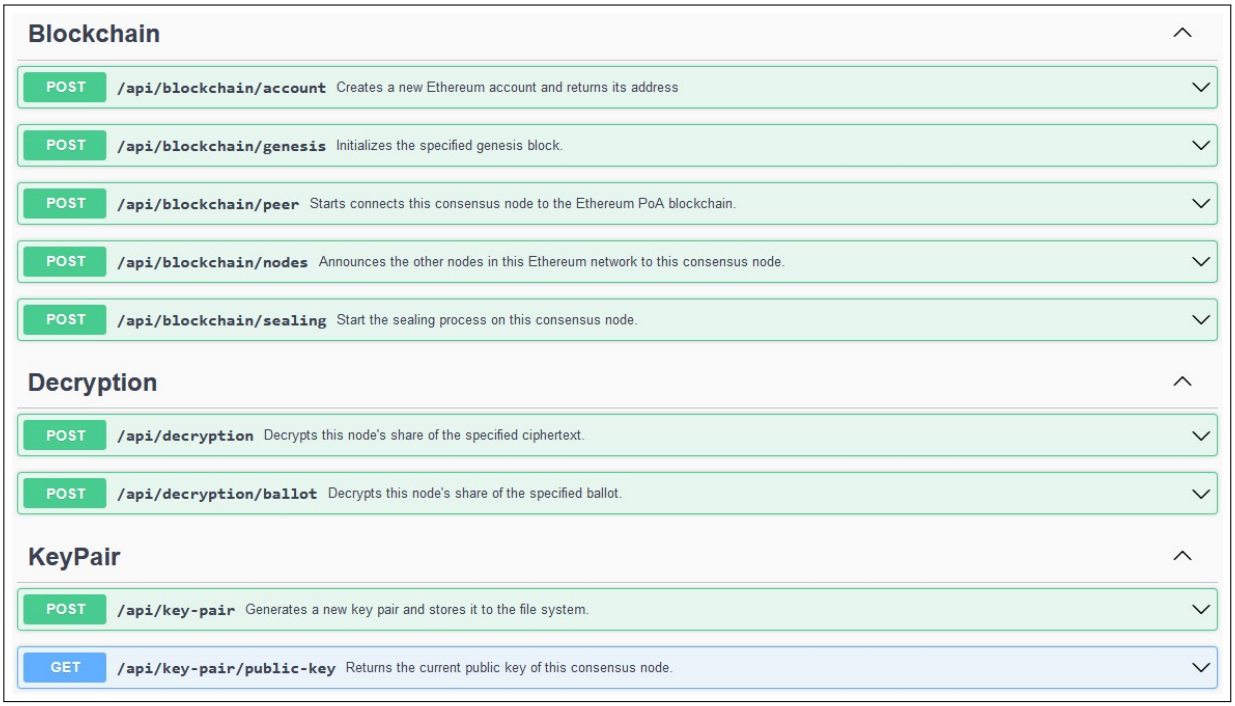

Figure 6.1: Consensus Node Backend REST API

6.3 Voting Authority

The VA application consists of two main components: the backend and the frontend. The following sections delve into the implementation specifics of these components.

6.3.1 Backend

The VA backend is implemented in C# using ASP.NET 6 [59]. It exposes a REST API to the VA frontend, which encapsulates the backend's functionality. In terms of functionality, the VA backend implements all the VA's use cases, *i.e.,* from BC setup and election management to ballot recording and evidence publishing. While the essential endpoints are explained in the following sections, a complete overview of the VA backend REST API is depicted in Figure 6.8.

Blockchain Setup

The BC must be set up once and is usable for multiple elections. The VA backend provides a REST API endpoint (*i.e.,* `/api/blockchain`) which allows the creation of a new BC network on the specified Consensus Nodes. The setup process (*cf.* Figure 6.2) consists of the following steps initiated by the Consensus Node backend:

1. The VA backend orders each Consensus Node to create a new BC account. The Consensus Node creates the account and returns the account address to the VA backend. The VA backend also creates an account, which it later uses to interact with the BC.

2. With all the account addresses collected, the VA backend generates the genesis block configuration, which specifies the sealers and the pre-funded accounts. All Consensus Nodes are sealers for the PoA BC. In addition, the sealers' accounts and the VA backend's account are pre-funded, which enables them to issue contract state-changing calls later.

3. The VA backend then propagates the genesis block configuration to all Consensus Nodes, which use this configuration to initialize the genesis block on their side.

4. Afterward, the VA backend tells the Consensus Nodes to start their BC nodes. Each node then returns its enode ID to the VA backend.

5. With the enode IDs collected, the VA backend creates a list of nodes participating in the network and distributes this list to all Consensus Nodes, which store this information on the file system.

6. Next, the VA backend orders the Consensus Nodes to start the sealing process. During this step, the Consensus Nodes learn about the other peers in the network by loading the enode IDs from the file system.

7. Finally, the VA backend initializes the genesis block and starts a BC node, which serves as an RPC endpoint (*i.e.,* HTTP and WebSocket) for later BC interactions.

After these steps, the BC is ready for the deployment of SCs. Thus, the next step is to set up an election.

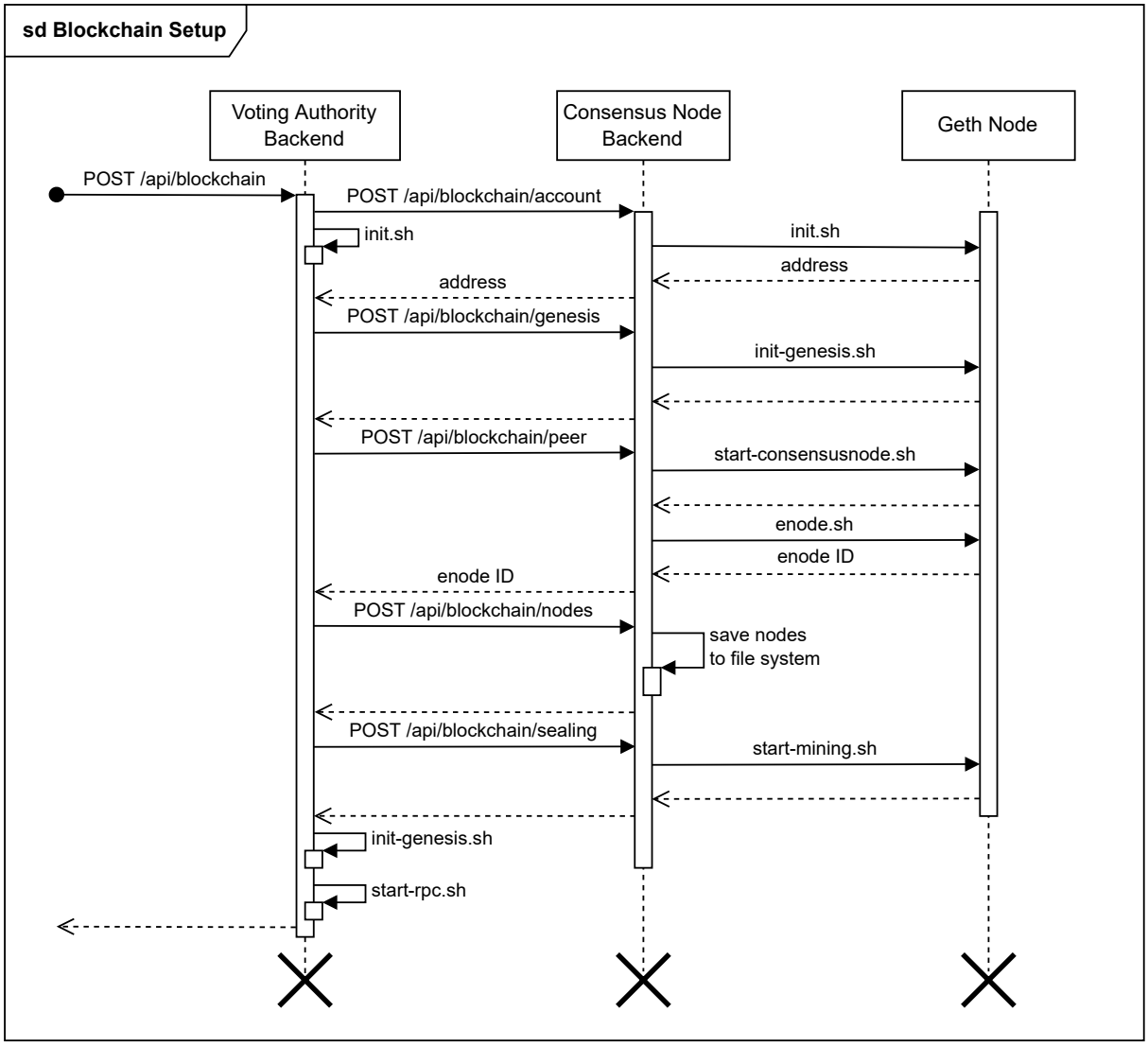

Figure 6.2: Sequence diagram showing the BC setup orchestrated by the VA backend.

Election Setup

Creating a new election is a three-step process (*cf.* Figure 6.3):

1. The first step involves persisting the election data (*i.e.,* question, candidates/options, cryptographic parameters) to the VA backend via a REST API call (`/api/elections`). This data is then stored in a MongoDB database.

2. The second step involves calling the `/api/elections/{id}/public-key` endpoint, which leads the VA backend to call every registered Consensus Node to generate an ElGamal key pair for this election. After receiving the public keys of all Consensus Nodes, the VA backend combines them into one composite election public key (*i.e.,* DKG), which it stores on the MongoDB database.

3. Via the `/api/elections/{id}/contract` endpoint, the VA backend deploys the election SC onto the PoA BC. This step also includes initializing the SC by register-

ing the election information, such as candidates/options, ElGamal parameters, and all public keys (of the election and all Consensus Nodes).

The election infrastructure and data are set up at this point, which means the election is ready for ballot generation.

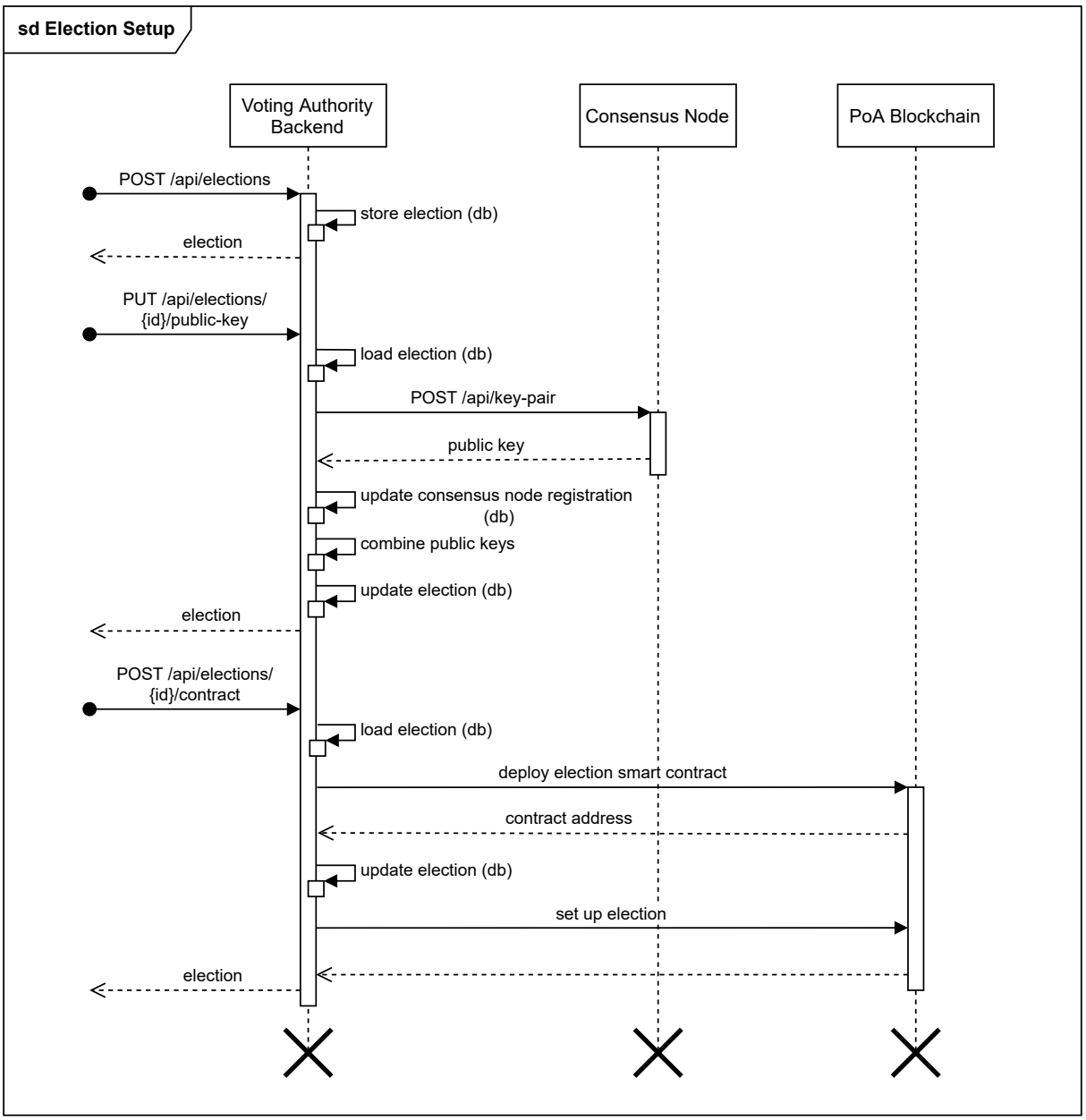

Figure 6.3: Sequence diagram showing the setup process of an election.

Ballot Generation

The VA backend provides an endpoint (`/api/election/{id}/ballots`) that allows generating new ballots for an election. The process of generating a ballot (*cf.* Figure 6.5) works as follows:

1. The first step is to generate a ballot template from the election parameters, yielding an object that contains all the election options/candidates in plaintext.

2. The ballot template allows the generation of encrypted ballots, also referred to as virtual ballots.

3. The VA backend stores the generated (encrypted) virtual ballots on IPFS. Note that the encryptions are sorted according to their shortcodes because otherwise, the position of an encryption would reveal which candidates/options they represent [5]. Furthermore, the MongoDB database stores the plain text paper ballots, where they are available for ballot printing.

4. Due to size limitations on the BC, the VA backend only stores the ballot ID, the ballot codes of both virtual ballots, and the IPFS references, also called Content Identifiers (CID) [33], of the virtual ballots on the SC.

In the process of generating ballots, there are multiple different ballot representations. Figure 6.4 depicts the relationships between these representations. Each election has its `BallotTemplate`. The `BallotTemplate` contains a reference to the `Election` to which it belongs. Furthermore, it contains a list of plaintext options representing the options/candidates for this particular election (*i.e.,* `PlainTextOption`). A `PlainTextOption` represents one candidate/option, consisting of the candidate's name, the shortcode (after encryption), and a vector of zeros and a one, where the one represents the current candidate. The `BallotTemplate` can encrypt the plaintext options to yield a `VirtualBallot`. The `VirtualBallot` consists of plain- and ciphertext representations of all candidates/options. The `EncryptedOption`s consist of the shortcode, the hash of the encryptions, and the encryptions the number vector (`EncryptedOptionValue`, consisting of the ElGamal ciphertext and a proof that the encryption contains either the value zero or one). The `VirtualBallot` also contains the proofs for guaranteeing that each candidate is only represented once and that each vector of numbers only represents one candidate. Furthermore, the `VirtualBallot` is only usable if all generated shortcodes are unique [5]. When storing a virtual ballot to IPFS, the plaintext values are not included, as these values would compromise ballot privacy. The construction of a `PaperBallot` takes two `VirtualBallot`s. Note that the `PaperBallot` contains all options in plaintext, along with the shortcodes, ballot codes, and ballot ID, as this information is needed for printing ballots later.

Each option/candidate has an accompanying shortcode. A shortcode is derived as follows (*cf.* Listing 3, lines 19-23): First, the SHA-256 hash of all encrypted values of the option is calculated and then reduced to the first two characters. In case of duplicate shortcodes in a `VirtualBallot`, the ballot is discarded and a new `VirtualBallot` is generated.

Ballot Printing

The `/api/elections/{electionId}/ballots/pdf` endpoint allows to print ballots to PDF. Printing the ballots to PDF is a fairly straightforward process:

```
        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="publicKey">Public key of the election</param>
        /// <param name="plainTextOption">Option in plaintext, list of zeros and a one
        /// (position of one = selected candidate / option)</param>
        public EncryptedOption(DHPublicKeyParameters publicKey, IList<int> plainTextOption)
        {
            Values = new List<EncryptedOptionValue>();

            foreach (int option in plainTextOption)
            {
                EncryptedOptionValue encryptedOptionValue
                    = new EncryptedOptionValue(option, publicKey);

                Values.Add(encryptedOptionValue);
            }

            HashHelper hashHelper = new HashHelper();

            Hash = hashHelper.Hash(Values.Select(v => v.Cipher).ToArray());

            ShortCode = Hash.Substring(0, 2);
        }
```

Listing 3: Short Code Calculation

1. The specified number of paper ballots are loaded from the MongoDB database.
2. A PDF is produced according to a pre-defined template for each ballot. Afterward, the paper ballot is updated to mark it as printed.
3. The VA backend produces a Zip archive of all generated PDFs and returns the archive for download.

Ballot Recording

The `/api/elections/{electionId}/ballots/{ballotId}/evidence` endpoint serves the purpose of registering the voter's choice as marked on the printed paper ballot. Additionally, the VA backend publishes evidence in the shape of the spoilt ballot marked by the voter. The process of recording ballot choices (*cf.* Figure 6.6) works as follows:

1. The first step involves loading the respective paper ballot from the database.
2. The VA backend retrieves the IPFS references of the ballot from the contract.
3. Next, the voter's selections (*i.e.,* the selected shortcodes of the chosen code column) are persisted on the election SC.
4. The other column (*i.e.,* the column that has not been selected) indicates the virtual ballot to be spoilt. Spoiling a ballot involves decrypting the values before storing the ballot as plaintext on the PBB.

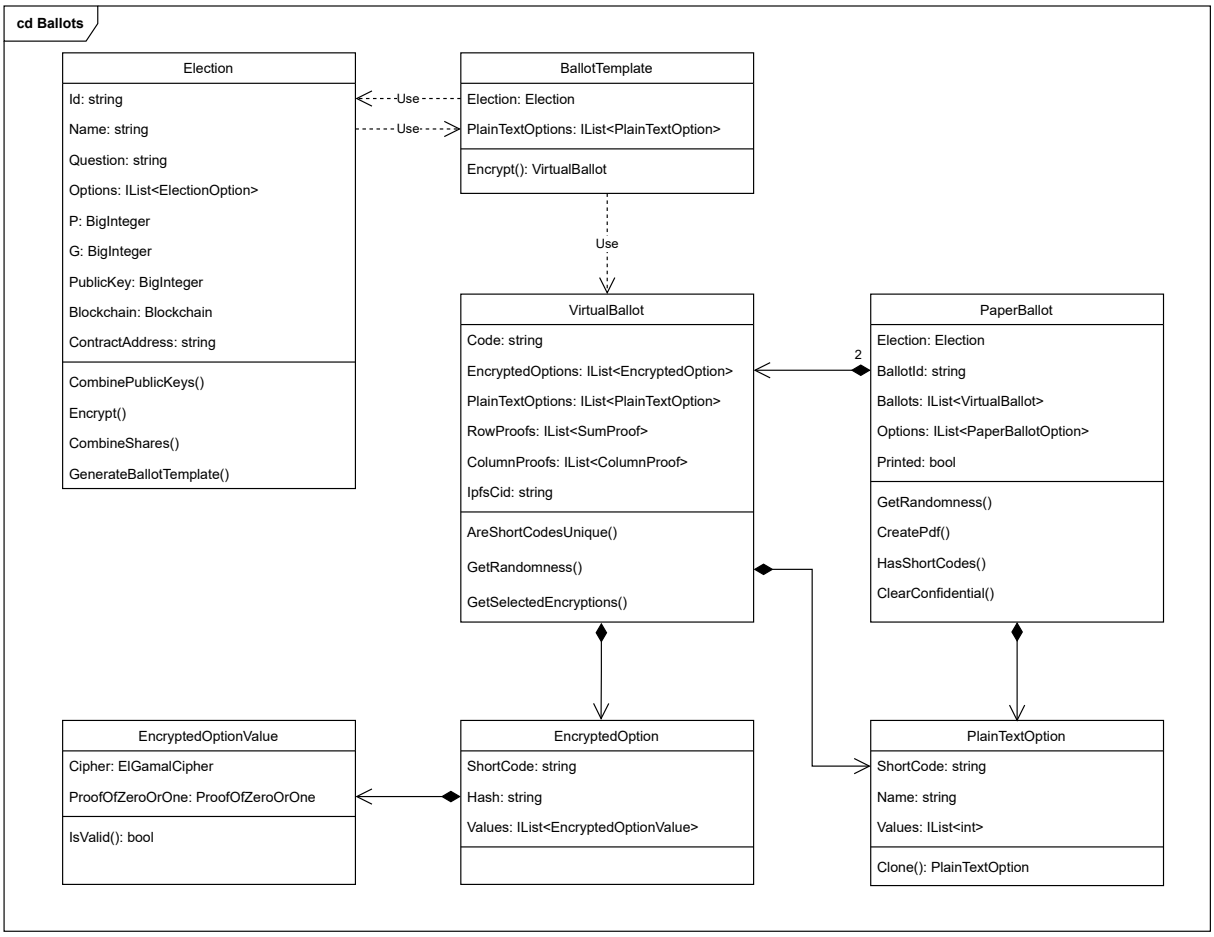

Figure 6.4: Class diagram showing the structure and composition of ballots in plaintext and encrypted states.

5. The Consensus Nodes cooperatively decrypt the virtual ballot to be spoilt. For this purpose, each Consensus Node retrieves the encryptions from IPFS by using the IPFS CID of the ballot to be spoilt and then produces a decrypted share, which the VA backend collects.

6. The VA backend combines the decrypted shares to restore the original plaintext of the virtual ballot.

7. The restored virtual ballot is published on IPFS.

8. The IPFS CID of the spoilt ballot is published on the election SC.

Tallying and Publishing Evidence

The last feature of the VA backend is tallying and publishing the results of an election (*cf.* Figure 6.7), including evidence that the results are correct. A call to the `/api/elections/{id}/tally` endpoint starts the following process:

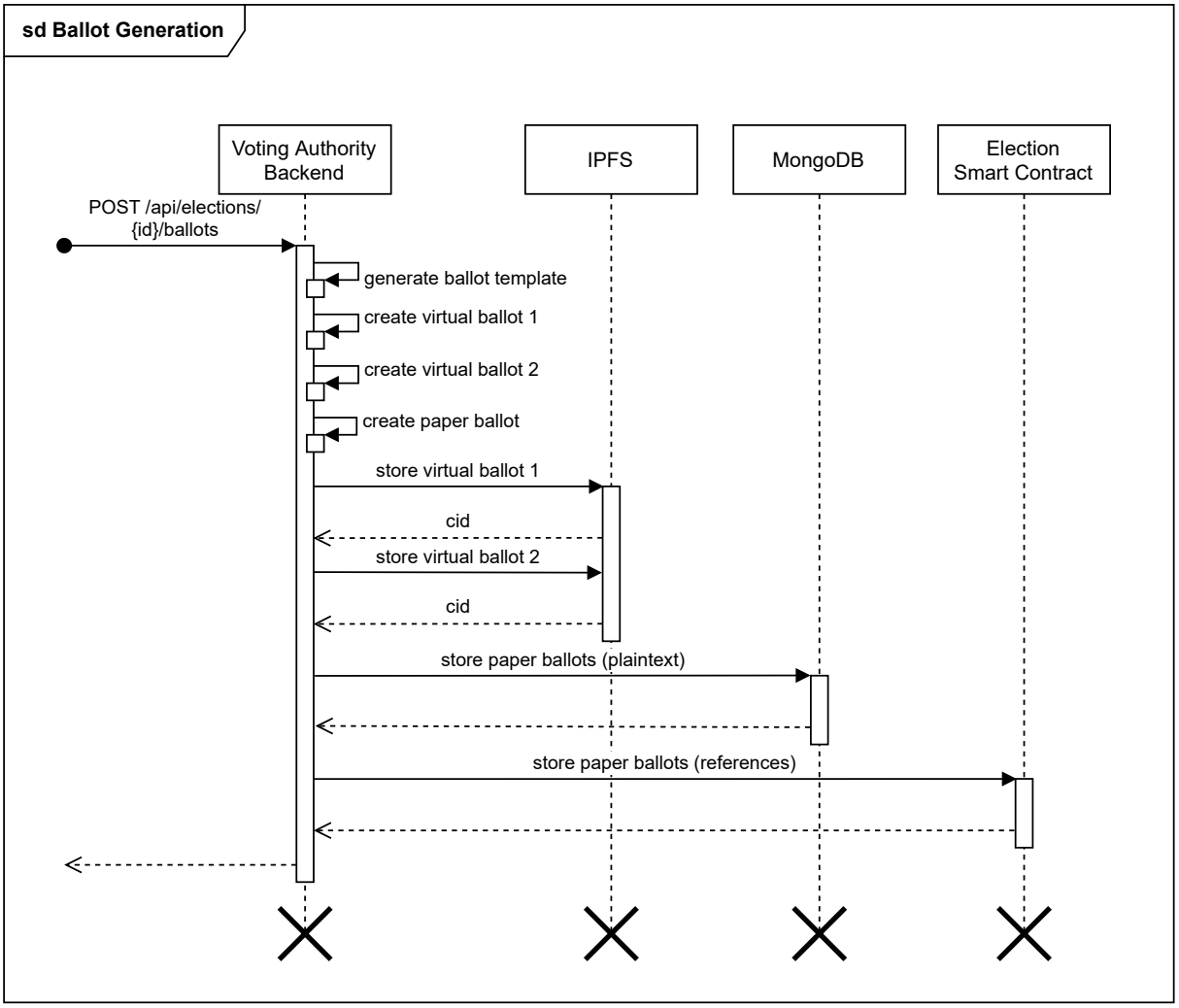

Figure 6.5: Sequence diagram showing the ballot generation process of an election.

1. The election data is loaded from the database.
2. The VA backend retrieves the number of cast ballots from the election SC.
3. The VA backend loads each ballot ID from the election SC in this step. With the ballot ID, the VA backend can retrieve the ballot references from the election SC, including the selected options/candidates of the ballot.
4. Using the ballot references, the VA backend loads the selected options of each ballot from IPFS in encrypted form.
5. The VA backend homomorphically adds all encrypted selections, yielding one ciphertext per candidate containing the tally of said candidate.
6. Each candidate's ciphertext is cooperatively decrypted by all Consensus Nodes, yielding the tally for each candidate after the VA backend has combined the decrypted shares. The decrypted shares also include the proofs of correct decryption.
7. The VA backend publishes the results and the corresponding proofs on IPFS.

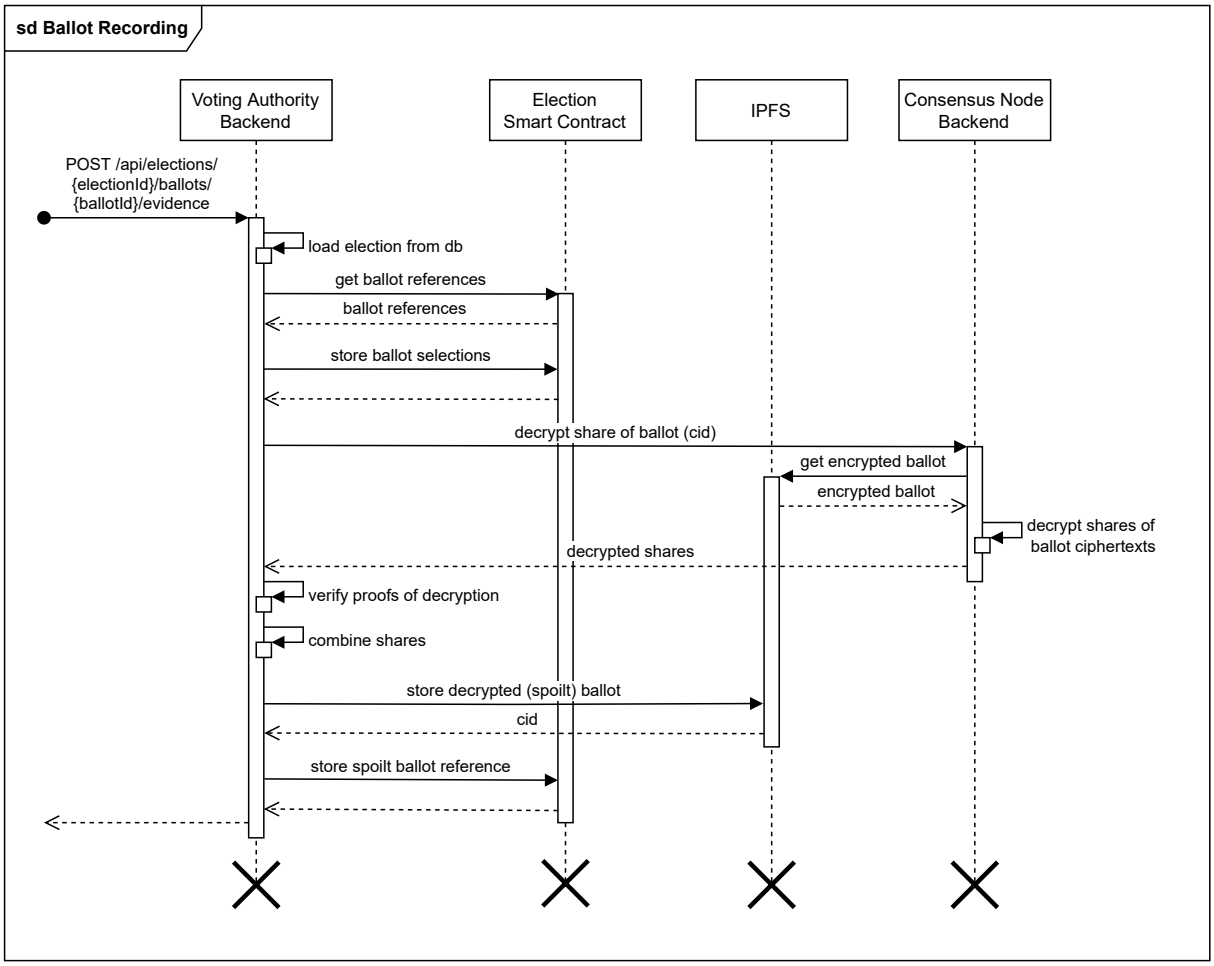

Figure 6.6: Sequence diagram showing the ballot recording process of an election.

8. The last step consists of the VA backend storing the IPFS reference (*i.e.,* CID) and the final results on the election SC. Thus, the results are available for verification from now.

6.3.2 Frontend

The VA frontend allows the election organizers to set up elections and handle all life-cycle events of an election. From a technology standpoint, the VA frontend is implemented in React [52] using TypeScript [53]. Figure 6.9 depicts the VA frontend, particularly the details view of a specific election. The following sections elaborate on the most significant use cases implemented in the frontend.

Blockchain Setup

Before creating elections, the first step is to set up a new BC consisting of multiple Consensus Nodes. For the registration, the user must specify the Consensus Nodes' backend

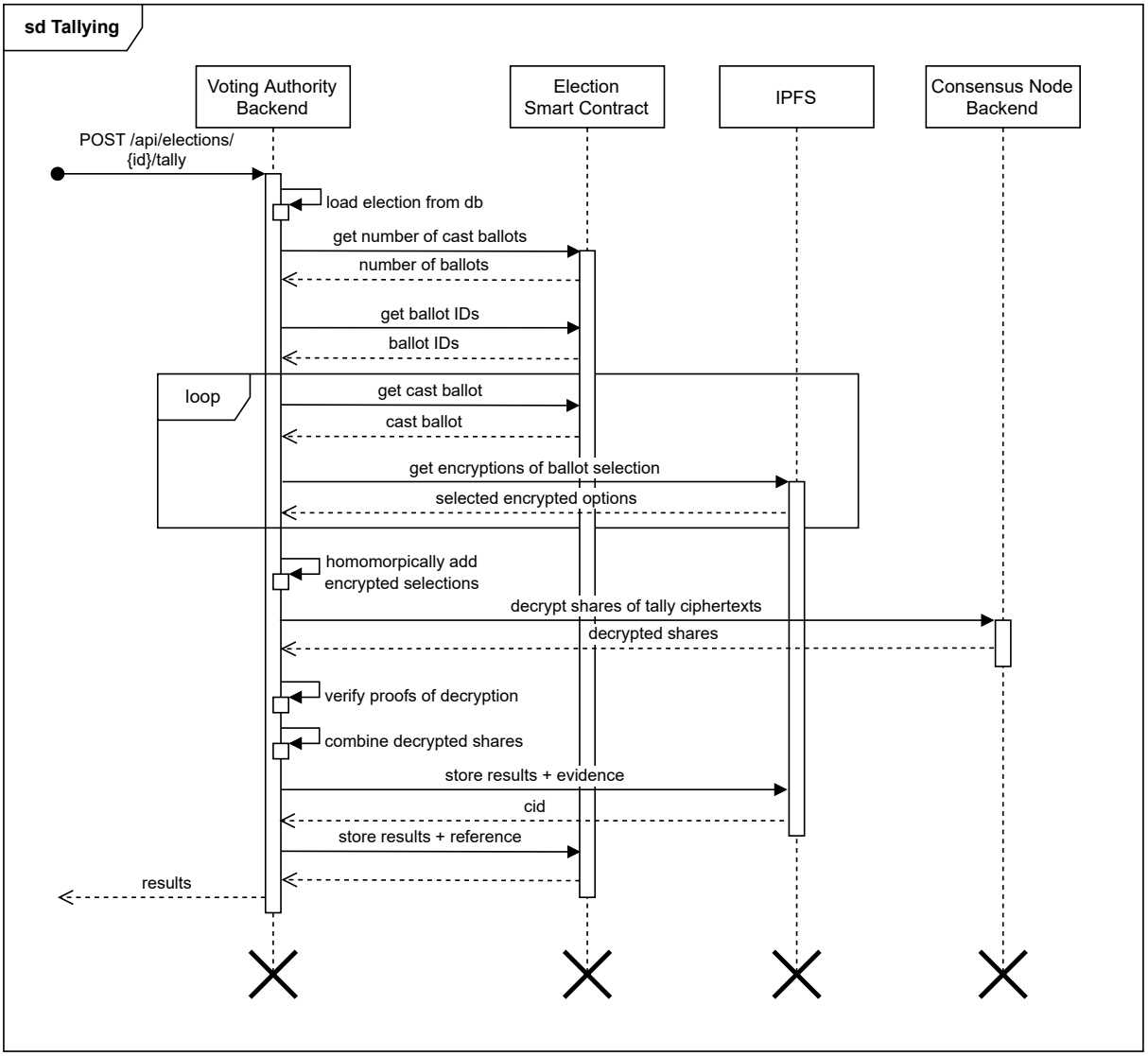

Figure 6.7: Sequence diagram showing the tallying and evidence publishing process of an election.

addresses (*i.e.,* URL to the REST API). The Consensus Nodes are supposed to be operated by municipalities or cantons, thereby distributing the trust to multiple trusted entities. Note that the setup procedure only has to be done once. Afterward, multiple elections can run on the same BC.

Election Setup

Election officials can set up new elections using the VA frontend. For this purpose, the VA frontend offers a step-by-step process guiding the user through the different aspects of creating an election. Initially, the user enters the election data, such as name, question, candidates/options, and cryptographic parameters. In the next step, the system generates an election public key, followed by the deployment of the election SC in the last step. At this point, the election is ready for ballot generation.

BallotPdf

GET	/api/elections/{electionId}/ballots/pdf	Generates PDF ballots packed into a ZIP file.

Ballots

GET	/api/elections/{electionId}/ballots	Shows the ballot data needed for printing a paper ballot.
POST	/api/elections/{electionId}/ballots	Generates new ballots, stores the encryptions on IPFS, publishes the evidence and the IPFS CIDs on the smart contract, and persists the plaintext print ballots onto the database.
POST	/api/elections/{electionId}/ballots/{ballotId}/evidence	Publishes the evidence of a casted ballot, consisting of the selected short codes and the ballot to be spoiled.

Blockchain

POST	/api/blockchain	Initializes the Proof-of-Authority blockchain using the consensus nodes registered.
GET	/api/blockchain	Returns the blockchain instance. Exists only once.

Elections

POST	/api/elections	Create a new election.
GET	/api/elections	Provides a list of all elections.
GET	/api/elections/{id}	Provides the election with the specified id.
PUT	/api/elections/{id}	Updates a specific election.
DELETE	/api/elections/{id}	Removes an election.
PUT	/api/elections/{id}/public-key	Combines and stores the public keys of all registered consensus nodes of the specified election.
POST	/api/elections/{id}/contract	Deploys a smart contract for the specified election.
POST	/api/elections/{id}/tally	Calculates the final tally and publishes the results with evidence.
GET	/api/elections/{id}/results	Returns the final tally.
GET	/api/elections/{id}/statistics	Returns the current election statistics, such as ballot counts.

Figure 6.8: Voting Authority Backend REST API

Ballot Generation and Printing

The VA frontend offers modal dialogues for generating and printing ballots. First, the user specifies how many ballots she wants to create. Note that more ballots can be generated later, if necessary. After generating the ballots, the next step consists of printing the ballots. Figure 6.10 shows an example of a paper ballot. A paper ballot comprises the election name, the questions, and the options/candidates. For later ballot recording and verification, the ballot also contains a QR code in the top right corner, which allows the effortless loading of the the ballot- and election parameters. The ballot ID is printed below the options. It helps to distinguish two ballots from each other. From a verification standpoint, the ballot contains two columns of shortcodes: The voter selects one of the two columns. For the voter's choices, the shortcodes of the selected column will be published after recording the ballot [5]. The other column is spoilt, meaning that the virtual ballot

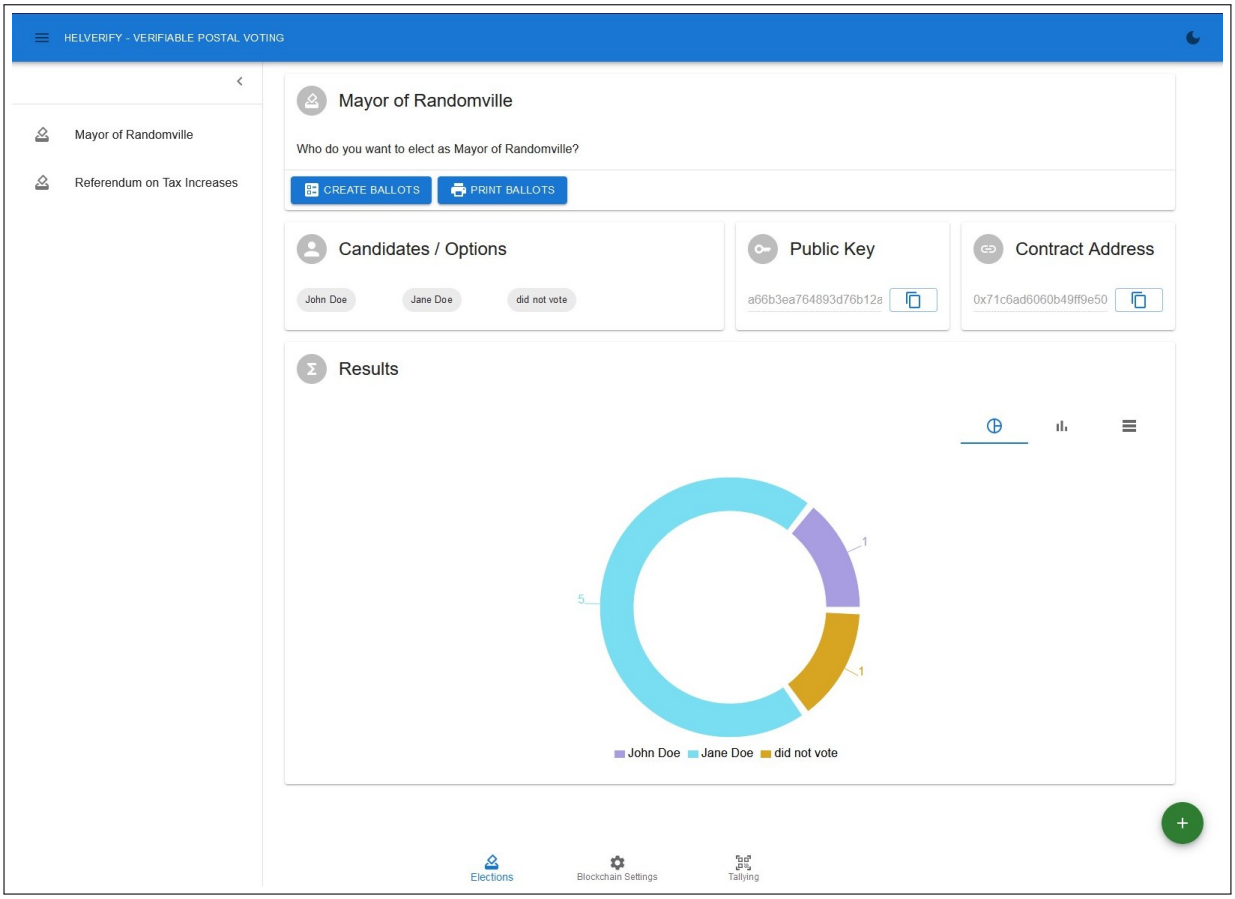

Figure 6.9: Voting Authority UI

is decrypted and shown together with the respective shortcodes, allowing to verify the correct encryption of the ballot [5]. The printed ballots are in PDF format, consisting of two pages: The first page contains the actual ballot. The second page, however, contains a watermarked copy, which the voter can use to memorize her choices for later verification. The system provides all ballots generated in a Zip file for download. The VA can hand this Zip file to a service partner for bulk printing.

Ballot Recording

Next, the voters fill in their choices and send the ballots back to the VA. Then, the election officials register the choices marked on the ballot into *Helverify*. For this purpose, an official scans the QR code of the ballot and fills in the user's selection into the ballot recording form. This form is a digital version of the paper ballot allowing the officials to enter the voter's selections and the selected column. The author knows that this could be considered inefficient and error-prone in real-world scenarios, as it relies on manually entering the voter's choices. However, an OCR approach could be implemented in such scenarios for more efficient ballot recording.

Mayor of Randomville

Who do you want to elect as Mayor of Randomville?

		☐	☐
John Doe	☐	c5	8c
Jane Doe	☐	11	8b
did not vote	☐	60	27

304128db2af41797563fd8387f7b80ac3b75767554c9bc2cb20b8174894baaf6

Figure 6.10: Printed Ballot

Results

When the VA has registered all ballot selections, the officials can use the VA frontend to calculate the final results and publish them with the corresponding evidence. After publishing the results, the VA and voter frontends visualize the election outcomes by providing multiple visualizations (*i.e.,* pie chart, bar charts, and a table, *cf.* Figure 6.9).

6.4 Voter Frontend

The voter frontend allows the voter to verify the election proceedings, with different means of verification available in each phase of an election. Furthermore, the voter frontend directly interacts with the election SC and the IPFS nodes to access evidence and ballots.

6.4.1 Ballot Authenticity

When receiving the paper ballot by postal mail, the voter can verify that the ballot is well-formed and properly encrypted (as in [5]). First, the voter scans the QR code on the ballot (or the ballot copy), which loads the required data from the election SC (*i.e.,* the ballot references) and the IPFS network (*i.e.,* the ballot data). The voter frontend offers the dialogue depicted in Figure 6.11 to perform this verification. While the figure shows all verification results in detail, the details are hidden by default not to confuse voters. Instead, the frontend only shows a single verification status. The verification includes the following checks (according to the mechanisms defined in [5]):

1. The ballot ID is correct, meaning it results from hashing the ballot codes of both virtual ballots.

```
/**
 * Verifies that the provided ciphertext indeed contains the value zero or one.
 * @param a C component of the ElGamal ciphertext
 * @param b D component of the ElGamal ciphertext
 * @param h Election public key
 * @param p ElGamal parameter p
 * @param g ElGamal parameter g
 */
verify(a: bigInt.BigInteger, b: bigInt.BigInteger, h: bigInt.BigInteger,
    p: bigInt.BigInteger, g: bigInt.BigInteger) {
    const q: bigInt.BigInteger = p.subtract(bigInt.one)
        .multiply(bigInt(2).modInv(p)).mod(p);

    const c = HashHelper.getHash(q,
        [h, a, b, this.u0, this.v0, this.u1, this.v1]);

    const gInverse: bigInt.BigInteger = g.modInv(p).mod(p);

    const condition1: boolean = g.modPow(this.r0, p)
        .equals(this.u0.multiply(a.modPow(this.c0, p).mod(p)).mod(p));
    const condition2: boolean = g.modPow(this.r1, p)
        .equals(this.u1.multiply(a.modPow(this.c1, p).mod(p)).mod(p));
    const condition3: boolean = h.modPow(this.r0, p)
        .equals(this.v0.multiply(b.modPow(this.c0, p).mod(p)).mod(p));
    const condition4: boolean = h.modPow(this.r1, p)
        .equals(this.v1.multiply(b.multiply(gInverse)
        .mod(p).modPow(this.c1, p)).mod(p));
    const condition5: boolean = this.c0.add(this.c1).mod(q).equals(c);

    return condition1 && condition2 && condition3 && condition4 && condition5;
}
```

Listing 4: Verification of proof that a ciphertext either contains the value 0 or 1.

2. For each virtual ballot:

   (a) Each set of encryptions (*i.e.,* vector of a candidate/option) represents precisely one candidate/option.

   (b) Each option/candidate is only represented once.

   (c) Each ciphertext contains either the value 0 or 1.

   (d) The shortcodes are correct (*i.e.,* the two first characters of the SHA-256 hash of the corresponding ciphertexts).

   (e) The ballot code is correct (*i.e.,* it is the SHA-256 hash of all encryptions).

The verification algorithms of the proofs mentioned above have been implemented in TypeScript. Listing 4 shows the TypeScript implementation of the proof that a ciphertext either contains the value 0 or 1. The implementations of the other proof verification algorithms follow the same principle.

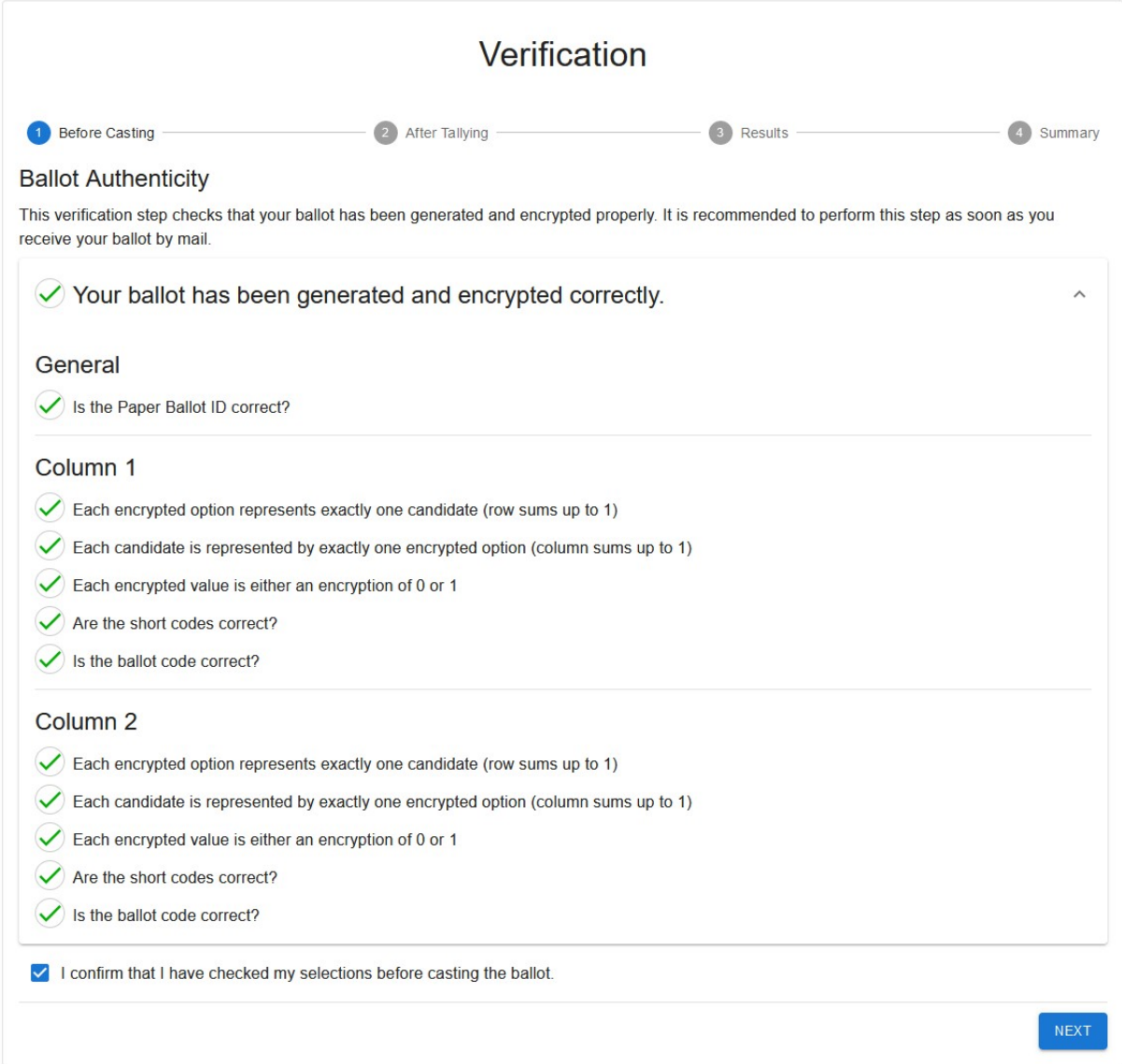

Figure 6.11: Ballot Authenticity Verification

6.4.2 Ballot Choice and Spoilt Ballot

The next step in the verification procedures consists of verifying that the VA has registered the voter's choices correctly and that the spoilt ballot's shortcodes match the shortcodes on the ballot copy (as in [5]). Figure 6.12 shows this verification step on the voter frontend. First, the voter checks that the shortcodes displayed in the voter frontend match the selected options on the selected column of her ballot. If they match, the voter can be sure that her choices are registered correctly. As suggested by STROBE-Voting [5], the spoilt ballot allows the voter to expose a cheating VA with a probability of $\frac{1}{2}$. Additionally, the voter frontend re-encrypts the spoilt ballot to check whether the resulting ciphertext matches the original ciphertext stored on IPFS.

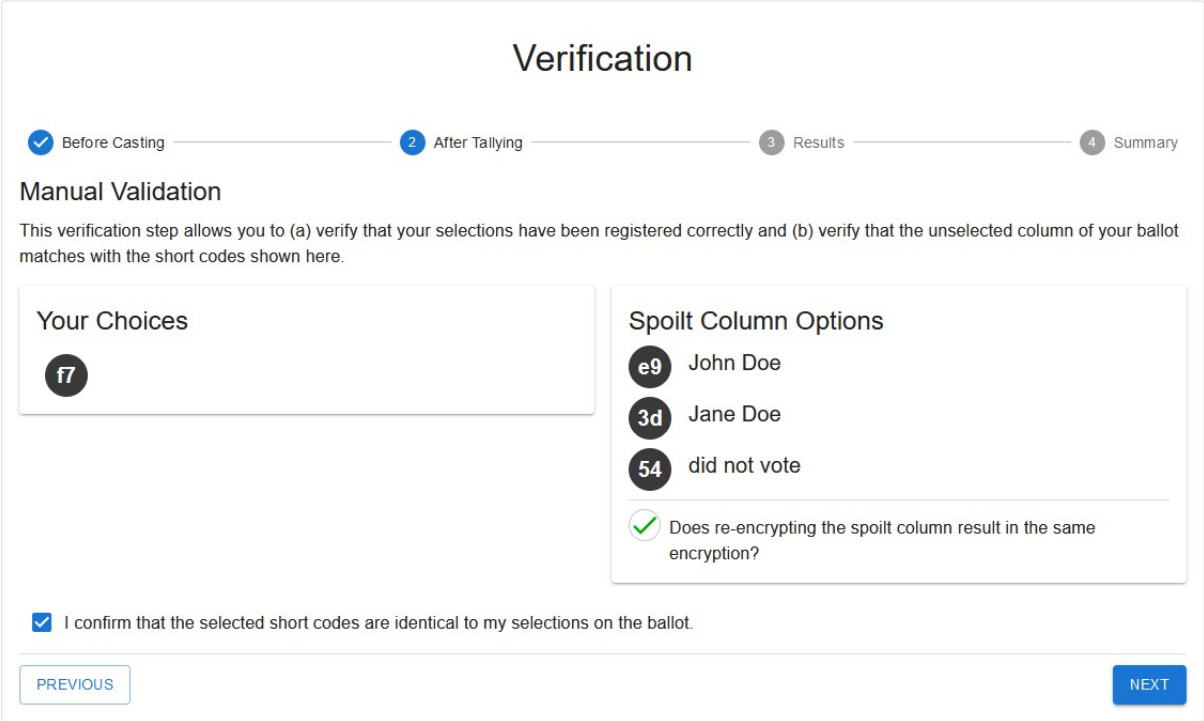

Figure 6.12: Choice and Spoilt Ballot Verification

6.4.3 Results and Summary

After verifying the ballot choices, the voter can inspect the election results. This step includes the verification of the proofs that the results have been decrypted correctly. Finally, a summary is presented to the voter (*cf.* Figure 6.13), which shows the properties achieved during the verification. The summary marks the end of the verification process.

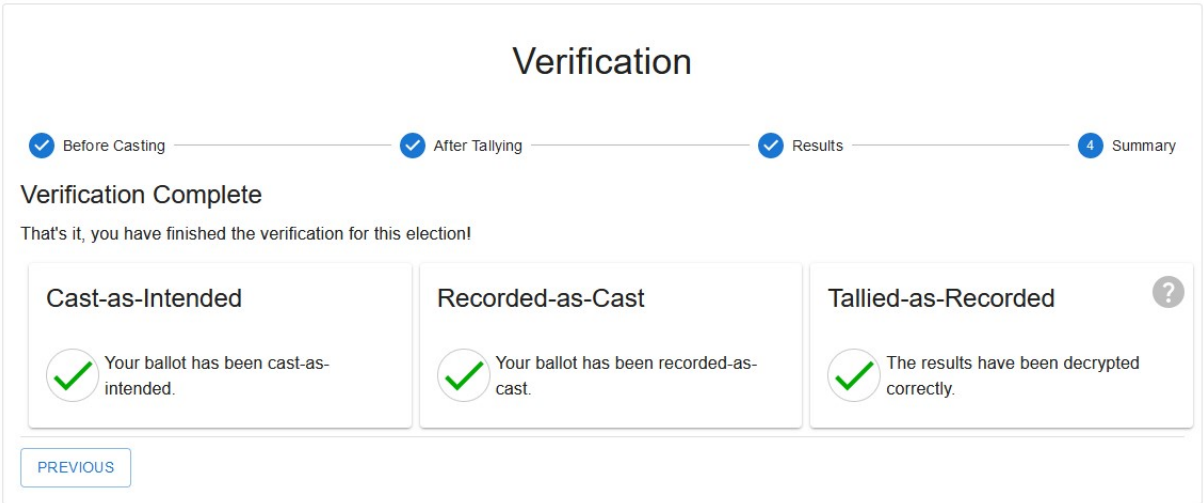

Figure 6.13: Verification Summary

```
{
  "config": {
    "chainId": 13337,
    "homesteadBlock": 0,
    "eip150Block": 0,
    "eip155Block": 0,
    "eip158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0,
    "petersburgBlock": 0,
    "clique": {
      "period": 2,
      "epoch": 30000
    }
  },
  "difficulty": "1",
  "gasLimit": "100000000"
}
```

Listing 5: Excerpt of a genesis block used by *Helverify*

6.5 Proof-of-Authority Blockchain

To run the election SC, *Helverify* operates a private Ethereum [28] PoA BC, consisting of the Consensus Nodes mentioned in 6.2. The following sections feature the details of the BC configuration and delve into the implementation of the election SC.

6.5.1 Configuration

The idea of using Consensus Nodes in *Helverify* is to have municipalities and cantons run these nodes, thereby forming the Ethereum PoA private network. Hence, *Helverify* uses Ethereum's Clique [56] consensus algorithm. Thus, each Consensus Node is a signer in the Ethereum network, allowing it to mine blocks [56].

In terms of configuration, *Helverify* uses a custom chain ID (*i.e.,* 13337) to not collide with any official Ethereum network. Furthermore, the gas limit is set to 180'000'000 to allow large transactions, thereby improving transaction scalability. For the same reason, *Helverify* uses a block period of 8 seconds, as this proved to be the sweet spot between performance and stability. Listing 5 shows parts of the genesis block generated for each election. Note that the `extradata` (*i.e.,* the definition of signers) and `alloc` (*i.e.,* pre-funded accounts) sections have been omitted for better readability.

6.5.2 Election Smart Contract

The election SC serves as a PBB for elections in *Helverify*. Thus, the SC is responsible for storing election parameters, ballot references, selections, evidence, and results in an

immutable manner. From a technological perspective, the election SC is implemented in Solidity [62].

The following functions allow the VA to alter the state of the contract:

- `setUp`: Allows the VA to initialize the SC with the election parameters. These parameters include candidates, cryptographic parameters, and public keys (of election and Consensus Nodes).
- `storeBallot`: After the VA has generated a ballot, it stores the ballot on IPFS and persists the references on the SC (*cf.* Listing 6). Note that this function allows storing multiple ballots at once, increasing the throughput on the BC (*i.e.,* ballots per transaction).
- `publishBallotSelection`: Stores the shortcodes of a virtual ballot's selections.
- `spoilBallot`: Stores the IPFS reference to a spoilt (*i.e.,* decrypted) ballot.
- `publishResult`: Publishes the final tally for each candidate, together with the evidence accompanying the results on IPFS.

```
string[] public ballotIds;

struct PaperBallot{
    string ballotId;
    string ballot1Code;
    string ballot1Ipfs;
    string ballot2Code;
    string ballot2Ipfs;
}

function storeBallot(PaperBallot[] memory ballots) public {
    if(msg.sender != votingAuthority){
        revert("Only voting authority is allowed to store ballots.");
    }

    for(uint i = 0; i < ballots.length; i++){
        string memory ballotId = ballots[i].ballotId;

        paperBallots[ballotId] = ballots[i];

        ballotIds.push(ballotId);
    }
}
```

Listing 6: Excerpt of Election Smart Contract that allows to store ballots

Apart from these state-altering functions, the SC also offers accessors (*i.e.,* `view` functions) that conveniently allow observers and verifiers to access the data stored on the SC. Furthermore, these `view` functions are free to call, meaning anyone can call them to retrieve the verification data. This division between free functions and functions with cost was deliberately chosen, as this allows only pre-funded accounts (*i.e.,* the VA) to carry out state-changing operations on the SC while still allowing verification for anyone.

6.6 Interplanetary File System

Helverify uses IPFS [35] as decentralized storage for storing ballots in plain and encrypted form. The decision to use IPFS for storage instead of storing the ballot data on the election SC is due to the poor suitability of BCs for storing medium to large-size data. Thus, the election SC only stores references (*i.e.,* CID) to the IPFS network. Moreover, due to the delicate nature of *Helverify* as a voting system, data must be stored in an immutable manner. Both the BC and IPFS achieve this property. In the case of IPFS, content is addressed by hashing the content itself, where the hash value is the CID of the content [33]. Thus, attempts to alter the content eventuate in a new CID [34], so if the CID is immutably stored on the BC, immutability is preserved.

The *Helverify* prototype uses a private IPFS swarm [36], meaning that a set of trusted IPFS nodes form the network. The swarm runs on instances based on the go-ipfs Docker image [25]. Listing 7 shows the configuration of a private swarm IPFS node.

```
1         ipfs-node1:
2         image: ipfs/go-ipfs
3         volumes:
4             - ./ipfs1/export:/export
5             - ./ipfs1/data:/data
6             - ./ipfs1/home:/home/ipfs
7         environment:
8             - ipfs_staging=/export
9             - ipfs_data=/data
10            - IPFS_SWARM_KEY_FILE=/home/ipfs/swarm.key
11        ports:
12            - 4001:4001
13            - 5001:5001
14            - 8080:8080
```

Listing 7: Docker Compose configuration of a private IPFS node

Chapter 7

Evaluation

Helverify is the prototype of a verifiable RPV system for use in Switzerland. It aims at empowering the voter to detect malicious activity during election procedures. For this purpose, *Helverify* adapts the principles of STROBE-Voting [5], which, combined with the procedures of Swiss RPV [42], allow providing E2E-V for voting procedures while preserving BP at the same time. Consequently, *Helverify* combines HE, threshold cryptography, and NIZKPs to achieve these properties. Furthermore, it uses a combination of an Ethereum PoA BC and a private IPFS swarm to host an immutable (and auditable) PBB. The protocol steps are visualized in Figures 5.2 and 5.3.

This chapter focuses on evaluating various aspects of *Helverify*: First, the performance and scalability are evaluated concerning ballot creation and tallying, which heavily depend on cryptographic operations. Afterward, the section on *Helverify*'s security highlights key lengths, trust assumptions, and their implications on the overall system security level. Moreover, this chapter elaborates on the RPV properties and the requirements (as specified in 5.1) *Helverify* achieves. Finally, the evaluation concludes with a discussion on the benefits and limitations of the *Helverify* prototype.

7.1 Performance and Scalability

This section discusses the influence of particular variables (*i.e.,* number of options, key length) on the duration of executing specific use cases for varying numbers of ballots (*i.e.,* 1'000, 10'000, 100'000, and 1'000'000). In particular, the scenarios consist of creating ballots and calculating the final results. All of the measurements are based on *Helverify* running with a BC period of 8 seconds and a gas limit of 180'000'000, which proved stable while offering decent performance. The performance figures in this section have been measured on an AMD Ryzen 5950X (16C/32T) with 64 GB of system memory available, running Ubuntu 22.04 LTS [16] with Docker [24]. Note that, for this evaluation, a private IPFS swarm was used for all measurements, with the data being stored at a single IPFS node. The reason for this setup lies in the reproducibility of the results, as it still allows spotting differences in storage durations caused by the variables in the measurements of this chapter.

7.1.1 Ballot Creation

The two main influencing factors when creating ballots are the cryptographic key length and the number of options/candidates available for an election. Figure 7.1 shows the ballot creation duration in relation to the number of ballots created with 2048-bit and 4096-bit keys, respectively. Within each series of measurements (*cf.* Table 7.1) of the same key length, the relation between the number of ballots and duration is roughly linear; a slight performance decrease is noticeable, though: For 1'000 ballots at 2048-bit key length, each ballot is created in 0.0172 seconds, while at 1'000'000 ballots, the time increases to 0.02868 seconds per ballot. The same pattern can be observed for 4096-bit key length (*i.e.,* 0.045 seconds per ballot at 1'000 ballots, 0.05682 seconds per ballot at 1'000'000 ballots). This effect could partially be due to performance advantages for short workloads, as the CPU can clock at higher frequencies during short workloads. When comparing ballot creation duration with 2048-bit and 4096-bit key lengths, it can be observed that the ratio between these durations decreases with an increasing number of ballots (*i.e.,* 2.61 at 1'000, 1.98 at 1'000'000). This behavior could be due to the influence of the BC period, which introduces variance in the results, especially for low numbers of ballots. Namely, the timing of when a block is mined has more impact on short timespans, which is the case for 1'000 and 10'000 ballots. Thus, increasing the number of ballots created reduces this effect.

Table 7.1: Duration measurements in minutes for creating ballots

Scenario	**1'000**	**10'000**	**100'000**	**1'000'000**
2048-bit 3 options	0.29	3.27	40.50	478.00
4096-bit 3 options	0.75	7.85	93.10	947.00
2048-bit 6 options	0.85	8.92	106.58	1'117.00

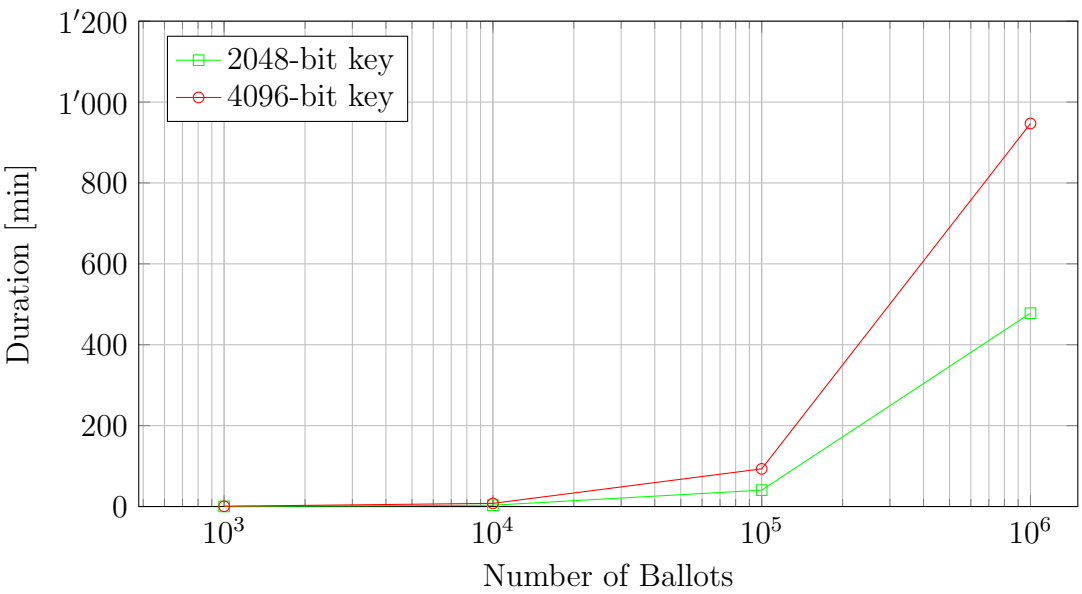

Figure 7.1: Ballot creation duration for different key lengths (3 options)

Figure 7.3 depicts the durations of creating ballots depending on the number of options/candidates specified for the election. Again, the durations for creating 1'000, 10'000,

100'000, and 1'000'000 ballots were measured (*cf.* Table 7.1). When changing the number of options from three to six, the time for creating 1'000'000 ballots increases by 133.68%. As a major part of ballot creation consists of generating NIZKPs, the number of proofs to be created significantly impacts the creation duration: In *Helverify*, the number of proofs required for n options can be described by the polynomial $n^2 + 2n$ (*cf.* Figure 7.2). Consequently, for three options ($n = 3$), one ballot requires 15 proofs, while for six options ($n = 6$), each ballot requires 48 proofs. Other factors, however, such as BC transaction processing, stay the same, as the key length did not change in this case.

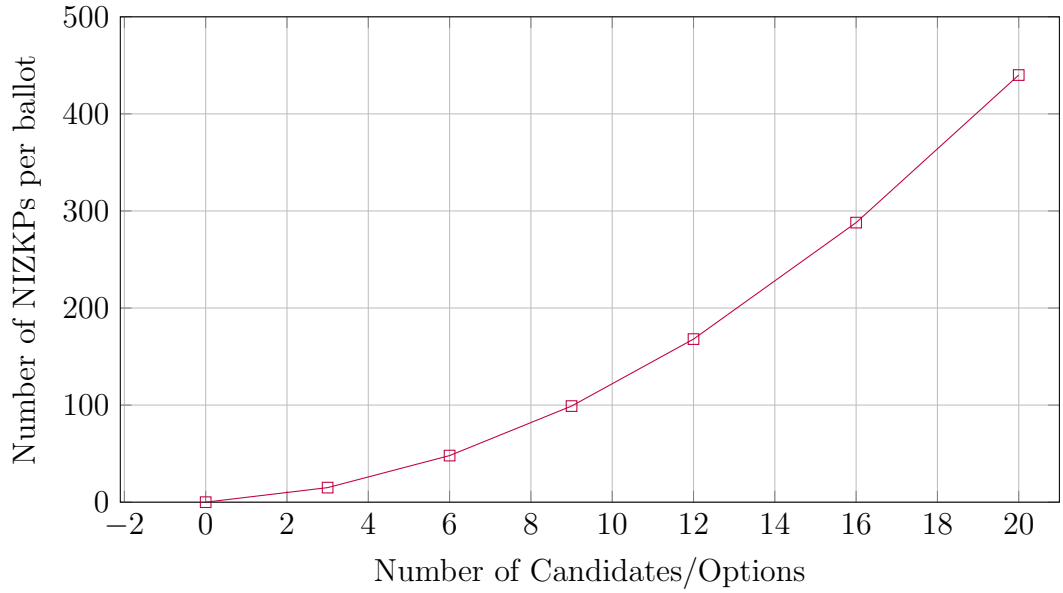

Figure 7.2: Number of NIZKPs required for number of options/candidates

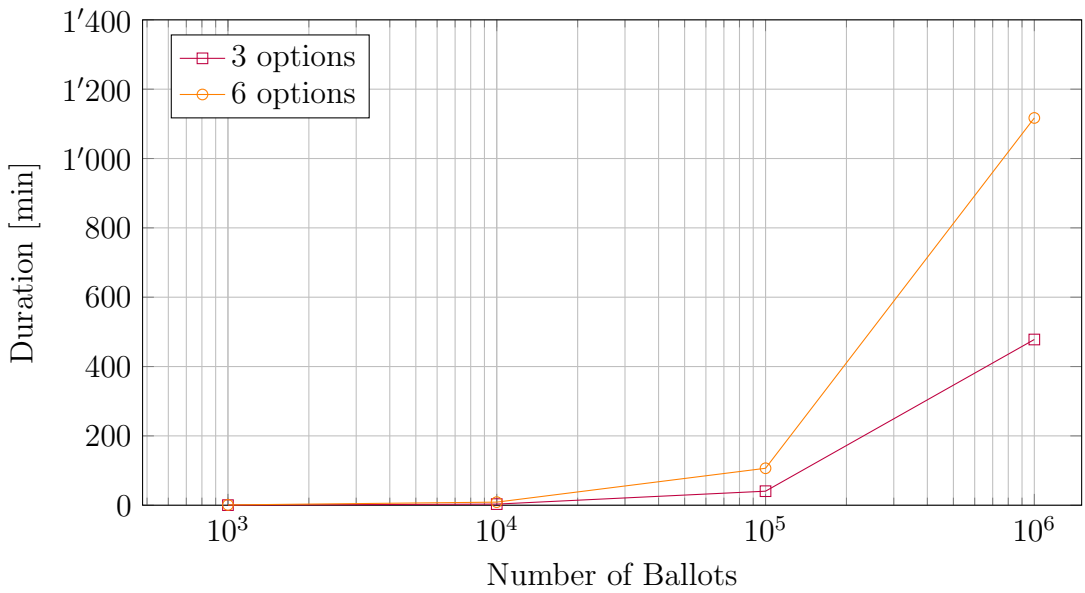

Figure 7.3: Ballot creation duration for different number of options (2048-bit key length)

7.1.2 Result Calculation

The performance and scalability analysis for the result calculation requires registered votes in *Helverify*. A dedicated evaluation endpoint on the VA backend allows selecting random choices for each ballot. As in the previous analysis, the measurements consist of runs with 1'000, 10'000, 100'000, and 1'000'000 ballots, respectively (*cf.* Table 7.2). Furthermore, the variables remain the same (*i.e.,* key length and the number of options/candidates).

Table 7.2: Duration measurements in minutes for tallying ballots

Scenario	**1'000**	**10'000**	**100'000**	**1'000'000**
2048-bit 3 options	0.08	0.59	6.05	61.30
4096-bit 3 options	0.34	2.05	20.60	215.00
2048-bit 6 options	0.15	1.10	11.28	113.07

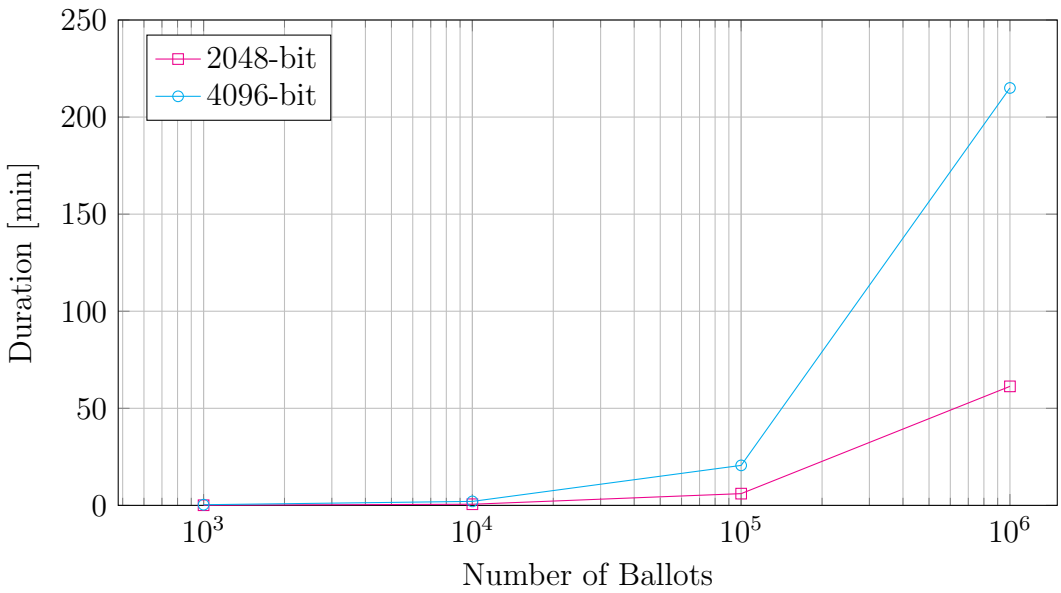

Figure 7.4: Tallying duration for different key lengths (3 options)

Figure 7.4 shows the results of tallying specific numbers of ballots with different key lengths while constantly using three options/candidates. The results show that using longer keys comes at a performance penalty: This is mainly due to the computational effort of performing homomorphic addition, which takes between 78.81% (2048-bit key, 10'000 ballots) and 81.31% (4096-bit key, 10'000 ballots) of the total result calculation duration. In this case, doubling the key length (2048-bit to 4096-bit) increases the duration from 61.3 to 215 minutes (+250.7%) for tallying 1'000'000 ballots. Furthermore, the data size of a ballot increases with growing key lengths, which contributes to the longer ballot creation durations. On the other hand, other interactions, such as SC and MongoDB calls, remain the same for varying key lengths and are, thus, negligible. Lastly, the time for cooperative decryption can vary, as the decryption is parallelized in a divide-and-conquer manner. This mechanism was implemented to optimize decryption, leveraging the capabilities of multi-core CPUs. Consequently, for some tallies, the decryption can be faster than others, mainly influenced by the partition borders for brute-forcing the exponential ElGamal ciphertext.

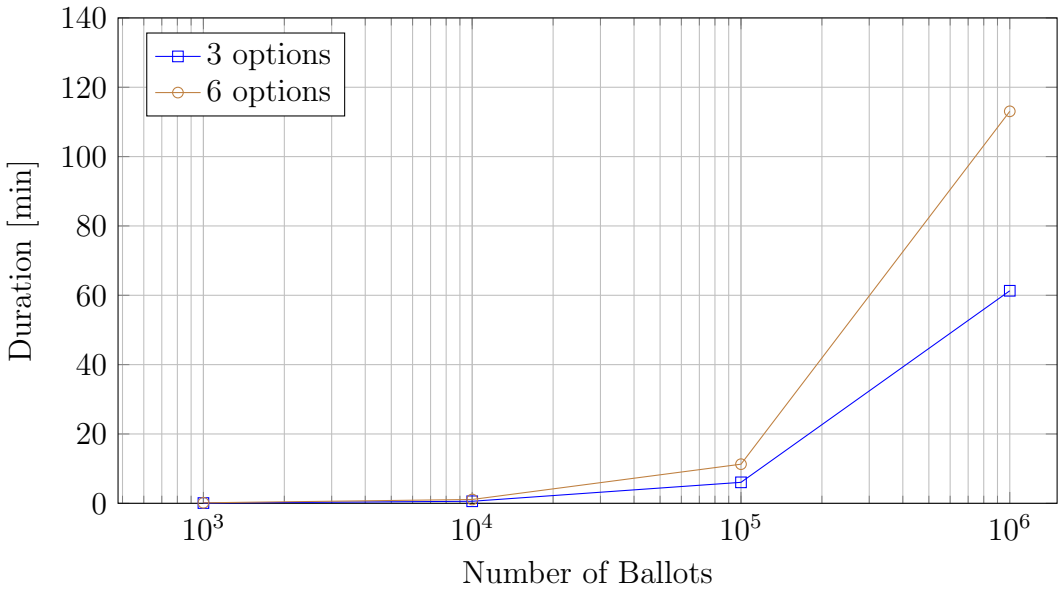

Figure 7.5: Tallying duration for different number of options (2048-bit key length)

Increasing the number of options/candidates (*cf.* Figure 7.5 and Table 7.2) does not come with the same performance penalty as ballot creation. At 1'000'000 ballots, the time to calculate the final results increases from 61.3 (3 options, 2048-bit key) to 113 minutes (6 options, 2048-bit key), which is less than double the time (+84%). Again, this could be due to the influence of the NIZKP creation during ballot generation, which is absent during the result calculation.

7.2 Security Level

From a security standpoint, key lengths and trust assumptions are critical subjects, as they influence the overall security level of *Helverify*. This section delves into *Helverify*'s key length, trust boundaries, and the implications.

NIST recommends using key lengths of at least 2048-bit for Diffie-Hellman keys [4]. In this respect, *Helverify* fulfills the requirements: During the evaluation, key lengths from 2048-bit keys up to 8192-bit have been tested. While performance at 2048-bit is decent, there are decreases in performance at 4096-bit and significant impacts when using 8192-bit key length. This effect can be reduced by upgrading to server-grade hardware, especially regarding CPUs.

7.2.1 Trust Boundaries

Regarding trust assumptions, *Helverify* distributes the trust between multiple entities (*cf.* trust boundaries in Figure 7.6): The PBB contains all the ballot encryptions, selections, spoilt ballots, results, and election evidence. These artifacts are publicly available; thus,

the PBB forms a trust zone. The VA's components are located inside a confidential trust zone, as it contains the PBs in plaintext. Thus, the VA backend is the only component allowed to write into the PBB. Other components, be it in this or other zones, are only allowed to read from the PBB. The Consensus Node zone is the second confidential zone of *Helverify*. It is classified as confidential because each Consensus Node stores multiple key pairs: The backends each store a key pair per election and a single key pair for the BC account. Moreover, the verification zone is another publicly accessible part of *Helverify*. It contains the Voter Frontend, the verification application for the voters. Finally, the voter zone classifies as confidential, as it involves the voter storing the ballot copy, which she needs for verifying the election procedures via the Voter Frontend. The ballot copy is considered sensitive data, as it contains the voter's selections and the spoilt column choice, mirroring the original cast ballot.

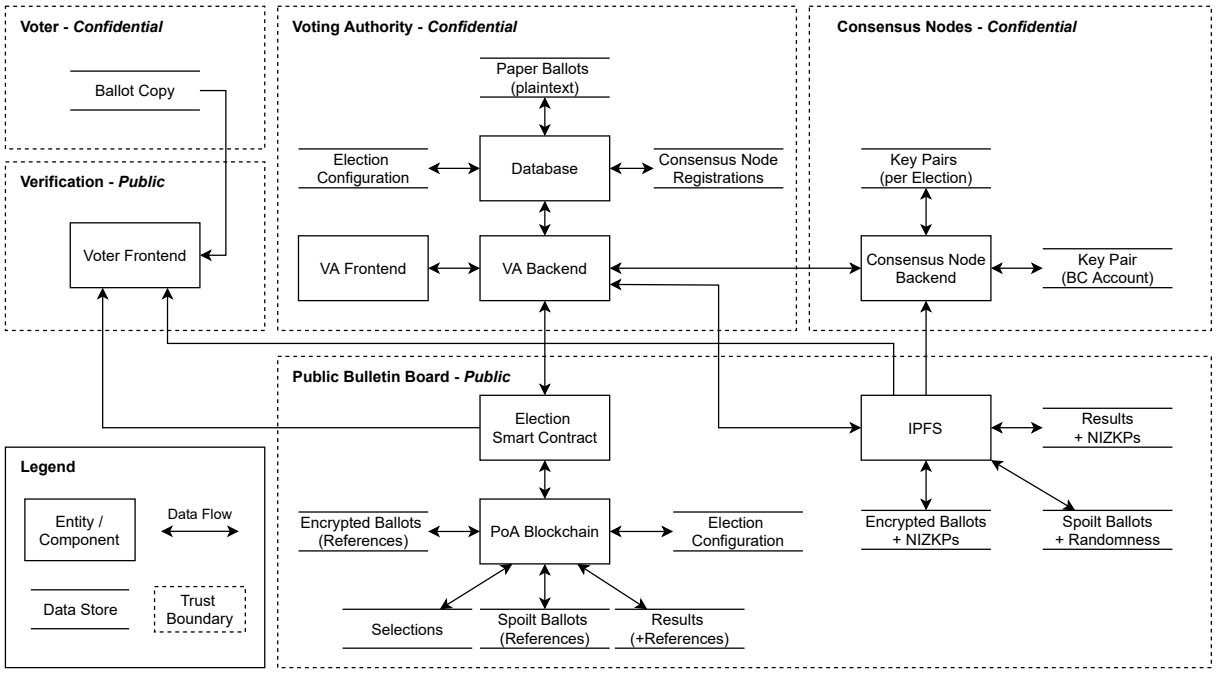

Figure 7.6: *Helverify*'s trust boundaries (notation inspired by [61])

7.2.2 Implications

The trust boundaries, as shown in Figure 7.6, form the following trust assumptions, which must hold for the system to be secure:

- **Voter**: The voter must keep the ballot (copy) secret. Otherwise, third parties could learn about the voter's selection. However, the ballot copy does not necessarily prove how the voter voted, as she could deceive third parties by selecting different choices on the copy than in the original ballot.
- **Voting Authority**: The PBs must be kept secret. Revealing the PBs would mean revealing the mapping between ballot codes and their plaintext options, indirectly exposing voters' selections. Furthermore, the VA must behave honestly.

- **Consensus Nodes**: The private keys of all key pairs (*i.e.,* election and BC) must remain secret. Due to applying threshold cryptography, an attacker could decrypt ballots and results only if she learns about all Consensus Nodes' private keys of an election. If the BC account private key of a Consensus Node is exposed, an attacker could sign blocks in the name of the Consensus Node. As with the VA, the Consensus Nodes must behave honestly.
- **Public Bulletin Board and Verification**: These zones are public; thus, the information is freely available. However, writing to the PBB must be restricted (*i.e.,* only by VA). Additionally, data integrity is guaranteed by the immutability property of the BC and IPFS.

Thus, if these assumptions hold, *Helverify* can be considered secure. However, this raises the need for mechanisms to detect if these trust assumptions are violated. Hence, the following mechanisms are available for detecting malicious activities and trust violations: The voter can use the evidence on the PBB to verify the VA's compliance with election procedures using the Voter Frontend. Thus, the voter can detect a VA behaving maliciously. Moreover, the voter and VA rely on the Consensus Nodes to act honestly. Hence, using multiple Consensus Nodes applying threshold cryptography (*i.e.,* DKG and CD) prevents a single entity from decrypting ballots, which lowers the trust assumption on each Consensus Node and the VA.

7.3 RPV Properties

As *Helverify* is based on STROBE-Voting [5] and the Swiss RPV procedures [42], it achieves the same properties. Said properties are discussed in this section, including the aspects of privacy and verifiability, which are crucial to any voting system.

7.3.1 Privacy

When it comes to privacy, *Helverify* achieves BP by applying HE. Applying HE allows the VA and the Consensus Nodes to calculate the final results without decrypting individual ballots, except for the spoilt ballots, which are not associated with selected options/candidates. Furthermore, *Helverify* fulfills RF, assuming that the voter does not deliberately reveal her choices. Even the voter's ballot copy does not necessarily prove to a coercer that the voter voted in a specific way, as the voter could also create another copy to deceive a coercer. According to [20], STROBE-Voting [5] fulfills RF, which is also the case for *Helverify*.

7.3.2 Verifiability

Regarding verifiability, *Helverify* achieves full E2E-V by applying the following mechanisms: The voter plays a crucial role in the verification process. First, the voter must

ensure that the choices marked on the ballot are correct from her perspective before inserting the PB in the PBE. This verification guarantees CaI fulfillment, as in the current Swiss RPV procedures. As the evidence of the cast ballots is publicly accessible, the voter can verify that the published selection shortcodes match the selections marked on her ballot copy. Furthermore, she can also verify that the spoilt ballot shortcodes match the spoilt column shortcodes. By performing these two checks [5], the voter ensures that RaC is achieved. Finally, verifying TaR is more involved: The voter can verify that the final results have been decrypted correctly. However, this is where the automatic verification possibilities reach the limit. If the voter (or any observer) wants to ensure that all votes are represented correctly in the final results, they must inspect the BC themselves. Furthermore, they need to re-run the homomorphic addition of the selected ciphertexts to retrieve the ciphertext representation of the results. These ciphertexts serve as inputs to the decryption proofs, which completes the TaR check and E2E-V. Note that this last step of performing homomorphic addition is deliberately not implemented in another tool accompanying *Helverify*. The voter would have to trust this implementation the same way she must trust the original implementation for calculating the results. Thus, the only way to verify TaR is to have third parties implement homomorphic addition to verify the decryption proofs. *Helverify* supports this option by storing the required evidence on the PBB. Finally, EV works the same as in the current Swiss RPV procedures [42] by requiring the voter to sign the VSC, which election officials verify before registering the ballot choices.

7.4 Requirements

Chapter 5 lists the requirements for building an RPV system suitable for use in Switzerland. At this point, the question is whether *Helverify* fulfills said requirements. In terms of functional requirements (A), the main goal was to achieve E2E-V, which *Helverify* fulfills by allowing the voter to use the Voter frontend to perform the necessary verification steps (A1). Moreover, verification is optional for voters, allowing them to choose whether they want to verify election procedures (A2). Furthermore, using *Helverify*, the VA can manage the life-cycle of an election effortlessly (A3, A4). Finally, the voter can cast her ballot by postal mail, as in the current system (A5).

From a quality standpoint (B), *Helverify* also fulfills the specified requirements. It scales for national elections and votes by adapting the principles of the current Swiss RPV procedures, where municipalities and cantons execute the elections and report the results to the federal level [42]. Regarding usability, both the Voter- and the VA frontend are based on the Material Design principles [32, 49] to ensure an adequate user experience (B2). Lastly, by applying HE for ballots, the voter's privacy is protected (B3). Furthermore, the association between paper ballots and voters is not stored anywhere, which allows storing the paper ballots in the VA database without violating BP.

In terms of constraints (C), *Helverify* is fully compatible with current election procedures in Switzerland (C1), including EV (C2). Furthermore, *Helverify* augments the current Swiss RPV procedures while leaving the current mechanisms as they used to be (C4). Thus, during system outages, election procedures can continue according to the current

procedures (C3). Ultimately, *Helverify* uses an Ethereum PoA BC and IPFS for hosting the PBB of an election (C5), fulfilling decentralization requirements.

In summary, *Helverify* fulfills all requirements specified in section 5.1.

7.5 Projection to Real-World Votes

After discussing the performance and scalability through benchmarking, this section explores the hypothetical performance of *Helverify* for two real-world votes. These votes took place in Switzerland in 2021 and are related to the Swiss Covid 19 law [21]. The first vote took place on June 13th, 2021, with the subject of approving a new law on Covid 19 measures in Switzerland [12]. The second vote occurred on November 28th, 2021, regarding changes to the same Covid 19 law [9]. For one thing, the reason for choosing these two options lies in the high voter participation [13] due to intense discussions and campaigns at the forefront of both votes [57, 30]. Secondly, both votes were executed on a national level, which allows using these numbers to project the ballot creation and tallying durations in a realistic, high-volume (in proportion to other Swiss elections and votes) nationwide scenario.

The projection is constructed as follows: The measurements, as shown in Tables 7.1 and 7.2, were used to deduce interpolation functions for ballot creation and tallying. For both real-world votes, the time to create the required number of ballots (*i.e.,* number of eligible voters) and the time to tally the number of cast ballots was calculated for each Swiss canton using the interpolation functions. Thus, the assumption is that each canton would run an instance of *Helverify* (*i.e.,* the canton is the VA). From a methodological perspective, the interpolation polynomials have been calculated using [65] for ballot creation[1] and [66] tallying[2], yielding the polynomials used for calculating the estimated durations in both scenarios.

The results for the first vote [12] (*cf.* Table 7.3) indicate that the ballot creation would take between 4 (AI, 12'157 eligible voters) and 480.9 minutes (ZH, 954'327 eligible voters). Furthermore, tallying would take between 30 seconds (AI, 7'919 cast ballots) and 38.9 minutes (ZH, 566'730 cast ballots). Moreover, under the assumption that one federal instance of *Helverify* was used for all of Switzerland, the roughly estimated durations for creating the ballots (*i.e.,* 5'507'117 ballots) and tallying (*i.e.,* 3'285'326 ballots) amount to 2795.8 minutes and 210.4 minutes, respectively. Note, however, that this total estimation is only a rough indicator for real-world ballot creation and tallying durations, as performance penalties of some degree are likely for large amounts of ballots.

For the second vote [9] (*cf.* Table 7.4), the results are similar to the first vote: The ballot creation would last between 4 (AI, 12'175 eligible voters) and 480.8 minutes (ZH, 956'792 eligible voters). In this case, the tallying process would take between 30 seconds (AI, 8'615 cast ballots) and 44 minutes (ZH, 643'227 cast ballots). Again, assuming one federal instance for all of Switzerland would yield roughly estimated durations of 2807.2

[1] $-0.0357181 + 0.000321505 \cdot x + 9.143035291183397 \cdot 10^{-10} \cdot x^2 - 7.5777236271063045 \cdot 10^{-16} x^3$
[2] $0.0206561 + 0.0000563001 \cdot x + 4.38172 \cdot 10^{-11} \cdot x^2 - 3.88379 \cdot 10^{-17} \cdot x^3$

Canton	Eligible Voters	Cast Ballots	Participation	Create Ballots [min]	Tally [min]
AG	435'285	256'371	58.9%	250.6	16.7
AI	12'157	7'919	65.1%	4.0	0.5
AR	38'912	25'665	66.0%	13.8	1.5
BE	743'476	462'982	62.3%	433.0	31.6
BL	189'815	112'652	59.3%	88.8	6.9
BS	114'147	66'301	58.1%	47.4	3.9
FR	209'893	129'670	61.8%	100.7	8.0
GE	271'111	137'841	50.8%	139.2	8.5
GL	26'580	15'774	59.3%	9.1	0.9
GR	140'933	84'068	59.7%	61.3	5.0
JU	54'016	31'610	58.5%	19.9	1.8
LU	281'961	183'604	65.1%	146.3	11.6
NE	113'266	61'157	54.0%	47.0	3.6
NW	31'659	21'605	68.2%	11.0	1.3
OW	27'058	19'328	71.4%	9.3	1.1
SG	327'307	194'989	59.6%	176.6	12.4
SH	52'945	38'565	72.8%	19.4	2.3
SO	181'363	107'135	59.1%	83.8	6.5
SZ	106'765	73'070	68.4%	43.8	4.4
TG	176'200	107'938	61.3%	80.9	6.6
TI	223'768	108'809	48.6%	109.2	6.6
UR	26'880	17'077	63.5%	9.3	1.0
VD	460'172	265'969	57.8%	267.7	17.4
VS	229'071	135'932	59.3%	112.5	8.4
ZG	78'050	52'565	67.3%	30.3	3.1
ZH	954'327	566'730	59.4%	480.9	38.9
Total	5'507'117	3'285'326	59.7%	2795.8	210.4

Table 7.3: Projected durations: Vote on Covid 19 law (13.06.2021) [11]

minutes for creating ballots (*i.e.,* 5'528'244 ballots) and 233.8 minutes for tallying (*i.e.,* 3'633'801 ballots).

Overall, these projected durations indicate that *Helverify* could handle the scalability and performance requirements regarding the Swiss RPV scenario.

7.6 Discussion

The evaluation reveals that, while performance and scalability are acceptable for municipal and cantonal elections in Switzerland, there is room for improvement regarding these aspects. Notably, the evaluation revealed the main influences and bottlenecks when it comes to the performance and scalability of *Helverify*. In terms of performance, cryptographic operations greatly influence overall system performance. While at the moment, 2048-bit key lengths are considered secure [4], this might change in the future. *Helverify* allows using longer keys, but this comes at a performance penalty to the point where cryptography can become a system bottleneck. One option to anticipate this issue would

Canton	Eligible Voters	Cast Ballots	Participation	Create Ballots [min]	Tally [min]
AG	437'516	287'482	65.7%	252.2	18.9
AI	12'175	8'615	70.8%	4.0	0.5
AR	39'012	28'425	72.9%	13.9	1.7
BE	744'859	496'811	66.7%	433.6	34.0
BL	190'760	127'723	67.0%	89.3	7.8
BS	113'956	73'682	64.7%	47.4	4.4
FR	211'226	133'306	63.1%	101.5	8.2
GE	273'057	147'460	54.0%	140.5	9.2
GL	26'637	17'616	66.1%	9.2	1.0
GR	141'383	93'112	65.9%	61.6	5.6
JU	54'022	32'780	60.7%	19.9	1.9
LU	282'643	200'131	70.8%	146.8	12.7
NE	113'589	65'787	57.9%	47.2	3.9
NW	31'789	23'889	75.1%	11.1	1.4
OW	27'332	20'600	75.4%	9.4	1.2
SG	328'286	226'148	68.9%	177.2	14.5
SH	53'170	40'372	75.9%	19.5	2.4
SO	182'004	120'868	66.4%	84.2	7.4
SZ	107'265	79'656	74.3%	44.0	4.8
TG	177'080	122'364	69.1%	81.4	7.5
TI	224'027	133'645	59.7%	109.4	8.2
UR	26'864	18'915	70.4%	9.2	1.1
VD	464'365	283'944	61.1%	270.5	18.7
VS	230'387	150'666	65.4%	113.3	9.4
ZG	78'048	56'577	72.5%	30.3	3.3
ZH	956'792	643'227	67.2%	480.8	44.0
Total	5'528'244	3'633'801	65.7%	2807.2	233.8

Table 7.4: Projected durations: Vote on changes to Covid 19 law (28.11.2021) [10]

be to refactor the cryptographic primitives to use Elliptic Curve Cryptography (ECC), allowing shorter keys while assuring the same security level (*cf.* security strength comparison of [3]). Another limiting factor is the throughput of the PoA BC: During the evaluation, the block period of 8 seconds proved stable while allowing a decent number of transactions with a block gas limit of 180'000'000. With more fine-tuning, the transactions per second could probably be increased a bit. Finally, the most limiting factor is the number of options/candidates per election, where the duration to create ballots is subject to polynomial growth.

The conclusion of this evaluation leads to the following recommendation. *Helverify* should be run on the premises of municipalities or cantons to get the most out of the system. By following this strategy, *Helverify* achieves nationwide scalability.

Chapter 8

Summary, Conclusion, and Future Work

This chapter begins with a summary of this thesis, followed by the conclusions drawn. Resulting the conclusions, potential areas of future work to improve *Helverify* are presented and briefly discussed.

8.1 Summary

This thesis introduces the design and implementation of an RPV system for use in Switzerland. First, the fundamentals of the Swiss RPV procedures have been highlighted, as they serve as a basis for the protocol design in this work. In addition, relevant cryptographic primitives were explained, particularly the concepts of HE, NIZKPs, and threshold cryptography. Furthermore, relevant related work has been highlighted and compared, leading to the selection of a protocol (*i.e.,* STROBE-Voting [5]) serving as a basis for the design of *Helverify*. Moreover, a prototype of *Helverify* has been implemented: It leverages an Ethereum PoA BC in combination with a private IPFS swarm as a PBB for storing election evidence publicly. The *Helverify* prototype also uses HE, threshold cryptography, and multiple variants of NIZKPs for fulfilling selected privacy and verifiability requirements. Further, the prototype has been evaluated regarding performance, scalability, security, RPV properties, and requirements fulfillment. Additionally, the evaluation results have been interpolated to real-world votes, indicating the real-world performance for votes and elections in Switzerland.

8.2 Conclusion

This thesis' main objective was to design and implement a verifiable RPV system usable in the context of Swiss RPV procedures. In conclusion, the *Helverify* prototype designed and implemented during this thesis meets the requirements regarding verifiability by providing E2E-V. Moreover, the prototype proved to meet the scalability requirements for use in Switzerland by scaling horizontally (*i.e.,* run *Helverify* per municipality or canton), which

is supported by the projection of the evaluation results to real-world elections in Switzerland. More precisely, in the largest canton of Switzerland (*i.e.,* Zürich), the projected duration for creating and tallying ballots for a real-world scenario amounts to 8 hours for creating 954'327 ballots and 38.9 minutes for tallying 566'730 ballots [11]. Further optimization is recommended for larger countries, despite these results, especially concerning vertical scalability. In terms of security, *Helverify* distributes the trust amongst voters, the VA, Consensus Nodes, and the PBB. This trust distribution lowers the trust assumptions of each entity. Consequently, if the trust assumptions hold, *Helverify* can be considered secure, meaning that it fulfills the privacy (*i.e.,* BP) and verifiability (*i.e.,* E2E-V) properties as specified in the requirements.

The limitations of *Helverify* are primarily influenced by the type of cryptography used: The computational effort of finite-field cryptography is a limiting factor for scalability. Furthermore, the number of NIZKPs per ballot grows polynomially with the number of candidates/options of an election and is, thus, another limiting factor in terms of performance and scalability. Finally, the PoA BC is also a limiting factor, as the number of transactions per second is determined by the block period and the block gas limit. These two factors proved challenging to balance, forming a trade-off between performance and stability.

In the end, voter acceptance probably remains one of the biggest challenges of RPV: Due to the complexity of verifiable RPV protocols, the average voter will struggle to understand the principles that guarantee verifiability and privacy in such a system. This difficulty primarily stems from using cryptography, BC, and distributed systems, which are complicated concepts to grasp for laypeople. Consequently, the voter has to trust the judgment of experts who certify the correctness and the guarantees of a verifiable RPV system. The same issue also concerns REV systems. In the case of *Helverify*, the protocol augments the existing Swiss RPV procedures while still allowing for the existing process to coexist. Thus, a fallback to the existing procedures is always an option during a system outage.

8.3 Future Work

Thus, for tackling said limitations, future work might consist of the following topics:

- **Optimizing Cryptography**: Refactoring the cryptographic primitives to use ECC instead of finite-field cryptography, which promises performance and storage space advantages. Using ECC might also positively affect scalability regarding the number of options/candidates because of performance improvements for generating NIZKPs.
- **Ballot Scanning**: The ballot scanning process could be optimized by implementing an OCR scanning mechanism, replacing the manual inputs by election officials, yielding an increase in ballot processing speed. Additionally, this reduces the manual work an election official must do to register ballot choices.
- **Write-In Elections**: Support for write-in elections could be another area of future work, as the current prototype only supports predefined lists of candidates/options.

- **Authentication & Authorization**: Authentication and authorization have been largely omitted in the *Helverify* prototype. However, these aspects would have to be implemented for real-world usage of the VA application.

- **IPFS Cluster**: In addition to the private IPFS swarm, an IPFS cluster could be configured to provide more resilience by allowing data replication across the entire IPFS swarm.

Bibliography

[1] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4), July 2018.

[2] Dölf Barben. Moutier-Abstimmung: Folgt das nächste Desaster?, March 2021. `https://www.derbund.ch/brisantes-papier-weckt-unbehagen-677279356631` accessed 13 May 2022.

[3] Elaine Barker. Recommendation for Key Management: Part 1 - General. Technical report, National Institute of Standards and Technology, May 2020.

[4] Elaine Barker and Quynh Dang. Recommendation for Key Management Part 3: Application-Specific Key Management Guidance. Technical report, National Institute of Standards and Technology, January 2015.

[5] Josh Benaloh. STROBE-Voting: Send Two, Receive One Ballot Encoding. In Robert Krimmer, Melanie Volkamer, David Duenas-Cid, Oksana Kulyk, Peter Rønne, Mihkel Solvak, and Micha Germann, editors, *Electronic Voting*, pages 33–46, Cham, 2021. Springer International Publishing.

[6] Josh Benaloh, Peter Y. A. Ryan, and Vanessa Teague. Verifiable Postal Voting. In Bruce Christianson, James Malcolm, Frank Stajano, Jonathan Anderson, and Joseph Bonneau, editors, *Security Protocols XXI*, pages 54–65, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[7] Josh Benaloh and Dwight Tuinstra. Receipt-free secret-ballot elections (extended abstract). In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 544–553, New York, NY, USA, 1994. Association for Computing Machinery.

[8] David Bernhard and Bogdan Warinschi. Cryptographic Voting - A Gentle Introduction, 2016. `https://eprint.iacr.org/2016/765.pdf` accessed 13 May 2022.

[9] Bundesamt für Statistik. Änderung des Covid-19-Gesetzes. `https://www.bfs.admin.ch/bfs/de/home/statistiken/politik/abstimmungen/jahr-2021/2021-11-28/aenderung-covid19-gesetz.html` accessed 15 September 2022.

[10] Bundesamt für Statistik. Änderung des Covid-19-Gesetzes, nach Kanton. https://www.bfs.admin.ch/bfs/de/home/statistiken/politik/abstimmungen/jahr-2021/2021-11-28/aenderung-covid19-gesetz.assetdetail.20004979.html accessed 14 September 2022.

[11] Bundesamt für Statistik. Bundesgesetz über die gesetzlichen Grundlagen für Verordnungen des Bundesrates zur Bewältigung der Covid-19-Epidemie (Covid-19-Gesetz), nach Kanton. https://www.bfs.admin.ch/bfs/de/home/statistiken/politik/abstimmungen/jahr-2021/2021-06-13/covid19-gesetz.assetdetail.17484063.html accessed 14 September 2022.

[12] Bundesamt für Statistik. Covid-19-Gesetz. https://www.bfs.admin.ch/bfs/de/home/statistiken/politik/abstimmungen/jahr-2021/2021-06-13/covid19-gesetz.html accessed 15 September 2022.

[13] Bundesamt für Statistik. Stimmbeteiligung. https://www.bfs.admin.ch/bfs/de/home/statistiken/politik/abstimmungen/stimmbeteiligung.html accessed 15 September 2022.

[14] Bundeskanzlei Sektion Kommunikation. Abstimmungsunterlagen. https://www.ch.ch/de/abstimmungen-und-wahlen/abstimmungen/abstimmungsunterlagen/ accessed 12 May 2022.

[15] Bundeskanzlei Sektion Kommunikation. Stimm- und Wahlrecht. https://www.ch.ch/de/abstimmungen-und-wahlen/abstimmungen/stimm-und-wahlrecht/ accessed 12 May 2022.

[16] Canonical Ltd. Ubuntu 22.04.1 LTS (Jammy Jellyfish). https://releases.ubuntu.com/22.04/ accessed 6 September 2022.

[17] David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *Advances in Cryptology — CRYPTO' 92*, pages 89–105, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[18] David L. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Commun. ACM*, 24(2):84–90, February 1981. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[19] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo G. Desmedt, editor, *Advances in Cryptology — CRYPTO '94*, pages 174–187, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[20] Braden L. Crimmins, Marshall Rhea, and J. Alex Halderman. RemoteVote and SAFE Vote: Towards Usable End-to-End Verification for Vote-by-Mail, 2021.

[21] Die Bundesversammlung der Schweizerischen Eidgenossenschaft. Bundesgesetz über die gesetzlichen Grundlagen für Verordnungen des Bundesrates zur Bewältigung der Covid-19-Epidemie. https://www.fedlex.admin.ch/eli/cc/2020/711/de accessed 13 October 2022.

[22] Die Bundesversammlung der Schweizerischen Eidgenossenschaft. SR 101 Bundesverfassung der Schweizerischen Eidgenossenschaft vom 18. April 1999 (Stand am 13. Februar 2022).

[23] Die Bundesversammlung der Schweizerischen Eidgenossenschaft. SR 161.1 Bundesgesetz vom 17. Dezember 1976 über die politischen Rechte (BPR), January 2015. `https://www.fedlex.admin.ch/eli/cc/1978/688_688_688/de` accessed 12 May 2022.

[24] Docker Inc. Home | Docker. `https://www.docker.com/` accessed 6 September 2022.

[25] Docker Inc., IPFS GitHub Contributors. ipfs/go-ipfs - Docker Image | Docker Hub, 2022. `https://hub.docker.com/r/ipfs/go-ipfs/` accessed 06 August 2022.

[26] e-Estonia. e-Democracy & open data, 2022. `https://e-estonia.com/solutions/e-governance/e-democracy/` accessed 10 October 2022.

[27] Taher Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[28] Ethereum Foundation. Home | ethereum.org, 2022. `https://ethereum.org/en/` accessed 08 August 2022.

[29] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[30] Jonas Glatthard. SWI swissinfo.ch - "Deutliches Ja in der Abstimmung zum Covid-Gesetz", November 2021. `https://www.swissinfo.ch/ger/wirtschaft/resultat-covid-gesetz-zertifikat/47141410` accessed 15 September 2022.

[31] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[32] Google. Material Design. `https://material.io/` accessed 15 September 2022.

[33] IPFS GitHub contributors. How IPFS works | IPFS Docs, 2022. `https://docs.ipfs.tech/concepts/how-ipfs-works/` accessed 06 August 2022.

[34] IPFS GitHub contributors. Immutability | IPFS Docs, 2022. `https://docs.ipfs.tech/concepts/immutability/` accessed 06 August 2022.

[35] IPFS GitHub contributors. IPFS Documentation | IPFS Docs, 2022. `https://docs.ipfs.tech/` accessed 06 August 2022.

[36] IPFS GitHub contributors. Run Kubo IPFS inside Docker | IPFS Docs, 2022. `https://docs.ipfs.tech/how-to/run-ipfs-inside-docker/#private-swarms-inside-docker` accessed 06 August 2022.

[37] Hugo Jonker, Sjouke Mauw, and Jun Pang. Privacy and verifiability in voting systems: Methods, developments and trends. *Computer Science Review*, 10:1–30, 2013.

[38] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-Resistant Electronic Elections. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, WPES '05, pages 61–70, New York, NY, USA, 2005. Association for Computing Machinery.

[39] Keystone-SDA. SWI swissinfo.ch - Untersuchung wegen Wahlfälschung und Stimmenfang in Pruntrut JU, December 2012. `https://www.swissinfo.ch/ger/untersuchung-wegen-wahlfaelschung-und-stimmenfang-in-pruntrut-ju/34129884` accessed 13 May 2022.

[40] Keystone-SDA. SWI swissinfo.ch - Wallis leitet nach Wahlbetrug Administrativuntersuchung ein, September 2018. `https://www.swissinfo.ch/ger/wallis-leitet-nach-wahlbetrug-administrativuntersuchung-ein/44414482` accessed 13 May 2022.

[41] Christian Killer, Bruno Rodrigues, Eder John Scheid, Muriel Franco, Moritz Eck, Nik Zaugg, Alex Scheitlin, and Burkhard Stiller. Provotum: A blockchain-based and end-to-end verifiable remote electronic voting system. In *2020 IEEE 45th Conference on Local Computer Networks (LCN)*, pages 172–183, 2020.

[42] Christian Killer and Burkhard Stiller. The Swiss Postal Voting Process and Its System and Security Analysis. In *4th International Joint Conference, E-Vote-ID 2019, Bregenz, Austria, October 1-4, 2019, Proceedings*, pages 134–149. Springer, 09 2019.

[43] Christian Killer, Lucas Thorbecke, Bruno Rodrigues, Eder J. Scheid, Muriel Figueredo Franco, and Burkhard Stiller. Proverum: A hybrid public verifiability and decentralized identity management. *CoRR*, abs/2008.09841, 2020.

[44] Steve Kremer, Mark Ryan, and Ben Smyth. Election verifiability in electronic voting protocols. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security – ESORICS 2010*, pages 389–404, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[45] Niels Kübler. Survey and Comparison of Blockchain-based Voting Systems, February 2022.

[46] Renat Kuenzi. SWI swissinfo.ch - "Genfer Fall ist nur die Spitze des Eisbergs", May 2019. `https://www.swissinfo.ch/ger/direktedemokratie/wahlbetrug-schweiz-direkte-demokratie/44959280` accessed 13 May 2022.

[47] Renat Kuenzi. SWI swissinfo.ch - "Aus der Apfelrepublik Thurgau ist eine Bananenrepublik geworden", April 2020. `https://www.swissinfo.ch/ger/direktedemokratie/wahlmanipulation_-aus-der-apfelrepublik-thurgau-ist-eine-bananenrepublik-geworden-/45718620` accessed 13 May 2022.

[48] Legion of the Bouncy Castle Inc. bouncycastle.org, 2022. `https://www.bouncycastle.org/` accessed 08 August 2022.

[49] Material UI SAS. MUI: The React component library you always wanted. https://mui.com/ accessed 15 September 2022.

[50] Gary McGraw. Software security. *IEEE Security Privacy*, 2(2):80–83, 2004.

[51] Eleanor McMurtry, Xavier Boyen, Chris Culnane, Kristian Gjøsteen, Thomas Haines, and Vanessa Teague. Towards Verifiable Remote Voting with Paper Assurance, 2021.

[52] Meta Platforms, Inc. React - A JavaScript library for building user interfaces, 2022. https://reactjs.org/ accessed 08 August 2022.

[53] Microsoft. TypeScript: JavaScript With Syntax For Types., 2022. https://www.typescriptlang.org/ accessed 08 August 2022.

[54] Tal Moran and Moni Naor. Receipt-Free Universally-Verifiable Voting with Everlasting Privacy. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006*, pages 373–392, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[55] Domhnall O'Sullivan. SWI swissinfo.ch - Die Schweiz wird zum Briefwahl-Paradies, October 2020. https://www.swissinfo.ch/ger/direkte-demokratie_die-schweiz-wird-zum-briefwahl-paradies/46073018 accessed 01 October 2022.

[56] Péter Szilágyi. EIP-225: Clique proof-of-authority consensus protocol, 03 2017. https://eips.ethereum.org/EIPS/eip-225 accessed 08 August 2022.

[57] Christian Raaflaub. SWI swissinfo.ch - "Stimmvolk heisst Covid-19-Gesetz gut", June 2021. https://www.swissinfo.ch/ger/abstimmung-schweiz-13--juni-2021_-covid-19-gesetz_resultat/46697724 accessed 15 September 2022.

[58] Ronald L. Rivest and Michael L. Dertouzos. On data banks and privacy homomorphisms, 1978.

[59] Daniel Roth, Rick Anderson, and Shaun Luttin. Overview of ASP.NET Core | Microsoft Docs, 2022. https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0 accessed 08 August 2022.

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

[61] Adam Shostack. *Threat Modeling: Designing for Security.* John Wiley & Sons, Inc., 2014.

[62] Solidity Team. Solidity Programming Language, 2022. https://soliditylang.org/ accessed 08 August 2022.

[63] Swiss Post Ltd. Swiss Post's e-voting solution, 2022. https://www.post.ch/en/business-solutions/e-voting/the-e-voting-solution-for-cantons accessed 10 October 2022.

[64] The go-ethereum Authors. Go Ethereum, 2022. https://geth.ethereum.org/ accessed 08 August 2022.

[65] Wolfram Alpha LLC. WolframAlpha - computational intelligence. https://www.wolframalpha.com/input?i=interpolate+%5B%281000%2C+0.2867%29%2C%2810000%2C+3.27%29%2C%28100000%2C+40.5%29%2C%281000000%2C+478%29%5D accessed 14 September 2022.

[66] Wolfram Alpha LLC. WolframAlpha - computational intelligence. https://www.wolframalpha.com/input?i=interpolate+%5B%281000%2C+0.077%29%2C%2810000%2C+0.588%29%2C%28100000%2C+6.05%29%2C%281000000%2C+61.3%29%5D accessed 14 September 2022.

Abbreviations

BC	Blockchain
BP	Ballot Privacy
CaC	Counted-as-Cast
CaI	Cast-as-Intended
CD	Cooperative Decryption
CID	Content Identifier
CR	Coercion Resistance
CS	Commitment Scheme
DCP	Disjunctive Chaum-Pedersen Proof
DKG	Distributed Key Generation
E2E-V	End-to-End Verifiability
ECC	Elliptic Curve Cryptography
EP	Everlasting Privacy
ER	Electoral Register
EV	Eligibility Verifiability
GETH	Go Ethereum
HE	Homomorphic Encryption
IPFS	Interplanetary File System
IV	Individual Verifiability
PB	Paper Ballot
PBB	Public Bulletin Board
PBE	Paper Ballot Envelope
PoA	Proof-of-Authority
MN	Mix Network
NIZKP	Non-Interactive Zero Knowledge Proof
RaC	Recorded-as-Cast
RaI	Recorded-as-Intended
REV	Remote Electronic Voting
RF	Receipt-Freeness
RPV	Remote Postal Voting
SC	Smart Contract
TaR	Tallied-as-Recorded
UV	Universal Verifiability
VA	Voting Authority
VE	Two-Way Voting Envelope
VSC	Voting Signature Card

ZKP Zero-Knowledge Proof

List of Figures

2.1 Process flow of Remote Postal Voting in Switzerland [42] 5

4.1 Contents of the Public Bulletin Board [6] 18

4.2 Encryption of ballot options in STROBE-Voting [5] 19

4.3 Example of PB in STROBE-Voting [5] 20

4.4 Example of PB in RemoteVote [20] 20

4.5 Example of PB in SAFE Vote [20] 22

5.1 Use Cases of *Helverify* 25

5.2 *Helverify* Protocol Phases 27

5.3 Process of running an election in *Helverify* 27

5.4 *Helverify* Protocol Setup Phase 28

5.5 *Helverify* Protocol Ballot Creation Phase 29

5.6 *Helverify* Protocol Delivery Phase 29

5.7 *Helverify* Protocol Casting Phase 30

5.8 *Helverify* Protocol Storage Phase 30

5.9 *Helverify* Protocol Tallying, Verification, and Destruction Phases 31

5.10 Architecture of *Helverify* 32

6.1 Consensus Node Backend REST API 37

6.2 Sequence diagram showing the BC setup orchestrated by the VA backend. 39

6.3 Sequence diagram showing the setup process of an election. 40

6.4 Class diagram showing the structure and composition of ballots in plaintext and encrypted states. 43

6.5 Sequence diagram showing the ballot generation process of an election. . . 44

6.6 Sequence diagram showing the ballot recording process of an election. . . . 45

6.7 Sequence diagram showing the tallying and evidence publishing process of an election. 46

6.8 Voting Authority Backend REST API 47

6.9 Voting Authority UI 48

6.10 Printed Ballot 49

6.11 Ballot Authenticity Verification 51

6.12 Choice and Spoilt Ballot Verification 52

6.13 Verification Summary 52

7.1 Ballot creation duration for different key lengths (3 options) 58

7.2 Number of NIZKPs required for number of options/candidates 59

7.3 Ballot creation duration for different number of options (2048-bit key length) 59

7.4 Tallying duration for different key lengths (3 options) 60

7.5 Tallying duration for different number of options (2048-bit key length) . . 61

7.6 *Helverify*'s trust boundaries (notation inspired by [61]) 62

List of Tables

4.1 Comparison of RPV protocols 23

7.1 Duration measurements in minutes for creating ballots 58

7.2 Duration measurements in minutes for tallying ballots 60

7.3 Projected durations: Vote on Covid 19 law (13.06.2021) [11] 66

7.4 Projected durations: Vote on changes to Covid 19 law (28.11.2021) [10] . . 67

Appendix A

Installation Instructions

The installation instructions for *Helverify* are straightforward, as it is built using Docker containers. Thus, to simply start and run *Helverify*, run the following scripts inside the root folder of the *Helverify* repository:

1. `cd docker`
2. `docker compose up`

For further instructions and details concerning application start up and development, please consult the `README.md` file in the root of the repository.