Reverse Engineering IVXV: A Specification and Security Analysis of Estonia's Internet Voting System

Tingfeng Yu
Australian National University
tingfeng.yu@anu.edu.au

Thomas Haines
Australian National University
thomas.haines@anu.edu.au

October 2026

Abstract

Estonia's IVXV system is one of the world's longest-running and largest-scale deployments of internet voting, yet it lacks a precise public specification of the protocol as implemented or security claims, which has left it difficult to audit independently. We address this as follows:

First, we reverse engineer a detailed specification of IVXV as implemented—covering the setup, voting, tallying, and audit phases—comprising protocol-flow diagrams, pseudocode algorithms, and the core data structures, reconstructed from the public source code, official documentation, and configuration examples.

Second, we complement the specification with an executable TLA+ model of the setup, voting, and tally phases, which we check with TLC for basic executability and correctness and which serves as an extensible basis for modelling concrete attack scenarios.

Third, using the reconstructed specification, we carry out a systematic security analysis under a broad range of threat models spanning individual verifiability, universal verifiability, privacy, coercion resistance, and eligibility verifiability, and we reassess previously known attacks against the current protocol.

Fourth, we compare IVXV against two of the most extensively studied deployed systems, Belenios and Swiss Post, across these properties in order to situate its guarantees.

Fifth, we discover numerous previously unknown, vulnerabilities—spanning ballot privacy, individual verifiability, and universal verifiability, many of them arising from inconsistent validation logic across components, encoding discrepancies, and unchecked cryptographic parameters—and validate selected findings with proof-of-concept tests that exercise the real implementation.

Finally, we propose concrete mitigations for the attacks we identify.

Contents

1 Introduction **4**
- 1.1 Motivation . . . 5
- 1.2 Contributions . . . 5
- 1.3 Summary of New Findings . . . 6
- 1.4 Limitations . . . 7
- 1.5 Responsible Disclosure and Ethics . . . 7
- 1.6 Outline . . . 8

2 Background and Related Work **10**
- 2.1 Related Work . . . 10
- 2.2 Cryptographic Primitives . . . 12

3 Methodology **15**
- 3.1 Protocol Reconstruction . . . 15
- 3.2 Vulnerability Categorisation . . . 15
- 3.3 Experimental Validation . . . 16

4 Protocol Specification **17**
- 4.1 Participants . . . 17
- 4.2 Setup Phase . . . 20
- 4.3 Voting Phase . . . 30
- 4.4 Tally Phase . . . 67
- 4.5 Audit Phase . . . 94
- 4.6 Channel Security . . . 107
- 4.7 TLA Model . . . 110

5 Security Analysis **111**
- 5.1 Known Attacks . . . 112
- 5.2 Attack Annotations . . . 116
- 5.3 Trust Boundary . . . 118
- 5.4 Individual Verifiability . . . 119
- 5.5 Universal Verifiability . . . 122
- 5.6 Privacy . . . 124
- 5.7 Coercion Resistance . . . 126
- 5.8 Eligibility Verifiability . . . 128

6 Comparison with Other Systems **129**
- 6.1 Deployment, Auditability, and Maturity . . . 130
- 6.2 Architectural Asymmetries . . . 131
- 6.3 Overview of Security Differences . . . 131
- 6.4 Belenios . . . 132
- 6.5 Swiss Post . . . 134
- 6.6 Synthesis . . . 137

7 Findings **139**
- 7.1 New Attacks . . . 139
- 7.2 Other Security Findings . . . 144

8 Possible Mitigations **149**
8.1 IV Mitigations . . . 149
8.2 UV Mitigations . . . 150
8.3 Privacy Mitigations . . . 150

9 Conclusion **152**

Appendices **157**

A Notes on the codebase **157**
A.1 Trust Anchors . . . 157

B Cryptographic and Other Primitives **160**
B.1 Symmetric Encryption . . . 160
B.2 ElGamal . . . 160

C Notation Conventions **164**

D Lists **165**
D.1 Algorithms . . . 165
D.2 Cryptographic and Other Primitives . . . 166
D.3 Data Structures . . . 166

E Data Structures **167**
E.1 BDOC File . . . 167

F Helper Functions **180**

1 Introduction

Estonia has used internet voting in elections since the early 2000s, making it one of the longest-running internet voting deployments in the world. Verifiability mechanisms, however, were incorporated more recently. In 2013, a mechanism was introduced to allow individual voters to check their vote was correctly received [22] though this was quickly shown to be flawed [31]. The current system, IVXV, was introduced in 2017 in collaboration with the vendor Smartmatic - Cybernetica C.O.E for Internet Voting [21]. In recent elections, roughly half of the votes have been cast online.

Despite the long history and prominence of internet voting in Estonia, IVXV remains difficult to audit. IVXV provides public documentation that describes the system architecture, interfaces, and operational workflows, but lacks a complete algorithmic specification of the implemented protocol; important security-relevant details, including validation logic, data dependencies, and trust anchors, are distributed across the source code, configuration files, and operational assumptions. A comprehensive analysis of IVXV therefore requires reconstructing the protocol from the implementation. This reconstruction is further complicated by the number of protocol participants involved. Some of these components are not open source, most notably the vote-casting device (VCD), which makes end-to-end protocol analysis more challenging.

Previous work has identified vulnerabilities in IVXV and related Estonian internet voting deployments. However, prior analyses have generally focused on specific attack mechanisms in an ad-hoc manner [26, 28, 32, 38, 36, 37], or on simplified models of the protocol [4]. As a result, several protocol phases, especially setup, tallying, and auditing, remain under-analysed. Moreover, evaluating an e-voting protocol across standard security properties (e.g., individual verifiability, universal verifiability) requires more than analysing isolated protocol steps. Such an assessment requires an end-to-end understanding of the protocol, including inter-phase behaviour and data-channel security. Additionally, some low-level properties, such as encoding consistency and cryptographic-parameter validation, are often idealised and omitted from high-level protocol analyses.

Property	Result
Clear claims	✗
Thorough documentation	✗
Minimality	✗
Buildable	✗
Executable	✗
Exportable	✗
Consistent documentation and source	✗
Regularly updated	✗
Minimal restriction on disclosure	✓

Table 1: Sutopo et al.'s summary of IVXV with respect to requirements listed in [18].

This lack of auditability has also been observed in prior work. In 2021, Haines and Roenne [18] proposed nine criteria upon which to assess the auditability of an e-voting system. Sutopo et al. [32] later applied these criteria to the IVXV system, and the results, as summarised in Table 1, show that IVXV satisfies only one of the nine criteria. While this observation does not by itself imply that the system is insecure, an under-documented system with no clear security claims is likely to contain unnoticed vulnerabilities. Without a precise protocol specification, explicit trust assumptions, and clear security goals, developers and auditors have no common reference point for deciding which checks are required and which participants are trusted for which properties. In

such a setting, vulnerabilities are more likely to arise from inconsistent validation logic, implicit data-channel assumptions, and gaps between intended and implemented behaviours.

1.1 Motivation

While various works have uncovered vulnerabilities in the Estonian e-voting system IVXV, the protocol remains under-documented. Official specifications provide only high-level architectural/protocol overviews and lack formalism and technical detail.

The formal modelling of IVXV by Baloglu et al. [4] represents the most systematic investigation of the full IVXV protocol. The model captures the sequence of key checks and protocol flows across the voting stage. However, the abstraction level (especially for Setup and Tally) constrains the scope of vulnerabilities that can be explored. The model simplifies ballots, registration, and collector interactions, and the automated tool itself cannot capture attacks exploiting underlying mathematical properties (e.g., ElGamal homomorphism, re-randomisation equations). As a result, a significant portion of the potential attack surface remains unexplored.

Motivated by the auditability gap and limitations of prior analyses, our work aims to reconstruct a detailed protocol specification of IVXV from the current implementation, and use the resulting specification as the basis for a systematic security analysis. This addresses several currently unsatisfied auditability criteria, most notably *clear claims*, *thorough documentation*, and *consistent documentation and source*. Realistically, such aspects of auditability need to be supported by living documents managed by the stakeholders, and as such our report is only a very interim solution unless it becomes the basis for an ongoing process.

To bridge this auditability gap, we take a reverse direction for protocol analysis: rather than checking an implementation against a pre-existing formal specification and security model, we reconstruct the protocol and infer the relevant security assumptions from the implementation, public documentation, and comparable deployed systems. This shapes our analysis: because IVXV does not provide explicit trust assumptions for each security property, we model a broad range of corruption scenarios, then distinguish between trivial attacks, known attacks, and new findings.

1.2 Contributions

This work makes six main contributions.

Protocol Specification (Sec. 4) We reconstruct a detailed protocol specification of IVXV as implemented, based on the public IVXV repositories, official documentation, and configuration examples. The specification covers the setup, voting, tallying, and audit phases, and includes process descriptions, pseudocode algorithms, and the core data structures used by the protocol.

Formal Model (Sec. 4.7) We developed a TLA+ model of the protocol specification, covering the main data structures, algorithms, and the three core protocol phases (Setup, Voting, Tally). We check each protocol phase model with TLC to demonstrate basic executability and correctness properties of our specification. This model also provides an extensible basis for future analysis of concrete attack scenarios.

Security Analysis (Sec. 5) We perform systematic analysis of the reverse-engineered specification under a range of threat models, including individual verifiability, universal verifiability, privacy, coercion resistance, and eligibility verifiability.

Comparison with Other Systems (Sec. 6) We compare IVXV against two of the most studied deployed internet voting systems, Belenios and Swiss Post, across the five security properties.

This situates IVXV's guarantees, and the new findings reported here, relative to comparators that have published formal security analyses, and identifies where the gap between IVXV and these systems is largest.

New Findings (Sec. 7) We identify new security vulnerabilities in IVXV across these threat models, and validate selected findings through proof-of-concept tests that exercise the real implementation logic of the relevant participants.

Possible Mitigations (Sec. 8) We propose concrete mitigations for the vulnerabilities we identify, spanning trivial, previously known, and newly discovered attacks across individual verifiability, universal verifiability, and privacy.

1.3 Summary of New Findings

Table 2 summarises the ten new vulnerabilities reported in this work: six affecting individual verifiability, two affecting universal verifiability, and two affecting ballot privacy. Five are supported by proof-of-concept tests that exercise the real implementation logic of the affected participants (Section 3.3); the remaining five are argued from code analysis alone, either because exploitation requires a live multi-session voting flow driven by a compromised device, or because the attack depends on behaviour of the closed-source vote-casting device that we can only infer. Alongside these vulnerabilities we catalogue fourteen further implementation observations (OB-1 to OB-14 in Section 7), which we treat as bugs or trust-boundary concerns rather than as directly exploitable attacks.

ID	Attack	Corrupt party	Labels	PoC	Details
Individual verifiability					
IV-V1	Sign revoting	Sign	LI, RV	–	§7.1.1
IV-V2	CS ballot dropping (TSP digest mismatch)	CS	BD, D	✓	§7.1.2
IV-V3	Reg reordering (timestamp manipulation)	Reg	LC, LT, R	–	§7.1.3
IV-V4	VCD identity substitution	VCD	LI, SUB	–	§7.1.4
IV-V5	VCD encoding (ciphertext parsing mismatch)	VCD	LI, BD	✓	§7.1.5
IV-V6	VCD ballot dropping via BDOC replay	VCD	LI, BD, D	–	§7.1.6
Universal verifiability					
UV-V1	PA ballot dropping	PA	BD	✓	§7.1.7
UV-V2	KA ballot dropping via invalid-vote marking	KA	LC, BD	✓	§7.1.8
Privacy					
P-V1	Weak ElGamal generator	EA or KA	–	✓	§7.1.9
P-V2	Passive recovery from randomness reuse	none (passive)	LC	–	§7.1.10

Table 2: The ten new vulnerabilities reported in this work. *Corrupt party* gives the minimal corruption the attack requires; the full corruption scenarios appear in the threat-model tables of Section 5. *Labels* are the mechanism, impact, and detectability annotations defined in Section 5.2. ✓ in the *PoC* column indicates that the finding is covered by a proof-of-concept test (Section 3.3); – indicates that it is argued from code analysis only, not that it was tested and failed. All ten were disclosed to the IVXV maintainers (Section 1.5).

1.4 Limitations

We state the main limitations of our work to clarify the scope of our claims:

No stated security model to analyse against IVXV publishes no explicit security model, trust assumptions, or security claims. Our analysis therefore speaks into a void: with no stated guarantees, there is no authoritative reference against which a given behaviour can be judged an intended design choice or a vulnerability. We necessarily infer the intended trust assumptions and security goals from the implementation, public documentation, and comparable deployed systems. Our findings should accordingly be read against those inferred assumptions rather than against claims made by the system's designers, and some of what we report as a weakness may reflect a mismatch between our inferred goals and the designers' unstated intent.

No formal proofs: upper bound without a lower bound We do not formally prove any security property of IVXV. Our results are exclusively negative: the vulnerabilities we enumerate establish an *upper* bound on the security the system provides, but we give no formal *lower* bound. The absence of a demonstrated attack against a given property or corruption scenario should therefore not be read as evidence that the property holds; it may equally reflect a limit of our analysis. Establishing positive guarantees would require a formal security model and proofs, which IVXV lacks and which are beyond the scope of this work.

Specification does not map one-to-one to the code IVXV is a large implemented system, and not every implementation detail is relevant at the protocol level. A fully faithful specification would essentially be as large as the codebase itself. In reconstructing the protocol, we therefore made abstraction choices about which details to model explicitly and which to abstract away. We aimed to include details about data flow, validation logic, trust assumptions, and inter-component consistency, while omitting details that do not appear security-relevant.

Property framing chosen for comparability Our security analysis is shaped by the goal of enabling easy comparison with the SwissPost system. We therefore use the same high-level security properties in our analysis of IVXV, such as individual verifiability, universal verifiability, privacy, coercion resistance, and eligibility verifiability. This approach is useful for comparison, but it has limitations. In particular, individual and universal verifiability do not, in general, imply end-to-end verifiability [25]. Moreover, IVXV does not provide a public bulletin board, so the term "universal verifiability" should be read as referring to the auditability of the election result by authorised auditors, rather than by arbitrary public observers.

1.5 Responsible Disclosure and Ethics

This work reports vulnerabilities in the protocol of the IVXV system; if these could be exploited in the deployed system cannot be assessed based on the information available to us. We have no reason to believe that they were exploited. Moreover, the version we analysed will not be used in future elections. The new version which is currently in development is planned for public release two weeks before the system goes live in 2027.

Disclosure We disclosed almost all vulnerabilities reported here to the IVXV maintainers by 22 May 2026; the few that were subsequently discovered were shared later on the 17th of September. P-V2 (§7.1.10), which we judged to carry the most immediate privacy impact, was found and disclosed earlier, on 26 September 2025. The maintainers have indicated that past elections were not affected by P-V2, because it requires election configurations that were not used. At the time of writing we

are not aware of deployed fixes for the issues reported here but the system is not due to run until March of 2027; we understand that all vulnerabilities we reported will be triaged and the more important ones fixed before that election.

Stakeholders This work may affect Estonian voters, the Estonian electoral management body, the IVXV maintainers and operators, election auditors, and the wider e-voting research community.

Potential harms and mitigations We identified three principal risks and mitigated each as follows.

Disruption of elections or live services Testing proof-of-concept attacks against deployed IVXV services could disrupt election infrastructure or affect real voters. All of our experiments were therefore performed locally, against patched copies of the public IVXV repositories. We did not test any attack against a live election service, and we used no real voter data.

Misuse of findings Releasing exploit-ready code could assist a malicious actor. However, the proof-of-concept code could be easily recreated using LLMs since the source code is available.

Premature disclosure Publicly revealing protocol weaknesses before the maintainers can respond could increase the risk to future elections. We therefore disclosed privately, through a coordinated process, and delayed public release accordingly.

Assessment Guided by the principles of *beneficence* and *respect for persons* set out in the Menlo Report, we weighed the benefit of publishing a reconstructed specification and security analysis of IVXV against the risks of disclosure. We judge the work justified: the reconstruction addresses a documented auditability gap that affects every party relying on the system, and the residual risk is reduced by coordinated disclosure. No human subjects were involved in this research, and no ethics-board review was therefore required.

1.6 Outline

Section 2 describes the cryptographic background needed for this paper as well as more information on related work. Section 3 describes our methodology for reconstructing the IVXV protocol from the available documentation, source code, and configuration examples. Section 4 presents the reconstructed IVXV protocol as implemented. Section 5 presents the Security Analysis of IVXV as specified using the threat models for individual verifiability, universal verifiability, privacy, coercion resistance, and eligibility verifiability, and reassesses known vulnerabilities identified in previous work against the current protocol. Section 6 compares IVXV with other comparable internet voting systems. Section 7 discusses selected new attack findings, and Section 8 proposes mitigations. Finally, Section 9 concludes the paper.

1.6.1 Suggested Reading Order

This paper is intended both as a security analysis and as a reference specification. Different readers may therefore want to approach it in different ways.

Security analysis Readers who are primarily interested in the security implications can skim the heavy algorithmic details of Section 4, and focus on Section 5, Section 7, and Section 8. Specific protocol details in Section 4 can be consulted when needed.

Protocol implementation Readers who are particularly interested in the IVXV protocol as implemented should read Section 4. This section serves more as reference material than as a linear narrative. We have included various cross-references to data structures (App. E) and primitives (App. B) used to allow easy navigation of the implementation details.

2 Background and Related Work

In this section we recap the related work and the various cryptographic primitives which will be relevant to the IVXV system.

2.1 Related Work

Estonia's internet voting stands out for coupling decades of practical use with sparse public information on its security. To our knowledge, no prior work in the e-voting literature reconstructs a deployed internet-voting protocol at the level of detail attempted in our work. The rest of this section therefore focuses on two bodies of related work: formal security analyses of e-voting protocols, and vulnerabilities found in deployed e-voting systems.

2.1.1 Vulnerabilities in e-voting systems

We briefly recall some significant related works in this subsection; later in Section 5.1, we will discuss the current status of the vulnerabilities previously found in the IVXV system.

2.1.1.1 Historical and Known Vulnerabilities in the IVXV system We will cover the known attacks in detail in Section 5.1, where we assess their status against the current implementation and classify them under our threat models. There have been several papers noting vulnerabilities in the system, including: insufficient ElGamal ciphertext validation that undermined cast-as-intended verification [32] (Sec. 5.1), a revoting attack [28], and shifting and Pfitzmann attacks [26].

More recently, a formal model of IVXV [4], focused on the vote casting process, rediscovered the revoting attack and identified further issues, including a ballot reordering attack on individual verifiability and a ballot copying attack on privacy.

Treier and Düüna [38] examined the processing stage and found that a dishonest insider could replace ballots during anonymisation without detection, since the audit at the time verified only signed checksums of PA's input files; the set-consistency check they proposed was incorporated into the Audit App in 2024. Treier [36] subsequently exercised the deployed checker with fault-seeded variants and found one that escapes it, and separately showed that national eID transaction logs reveal whether a voter re-voted [37]. We assess the first two in Section 5.1.7 and discuss the third when scoping our coercion-resistance adversary (Section 5.7).

2.1.1.2 Norwegian Internet Voting System During the 2013 Norwegian Internet voting, usage of a faulty pseudo-random generator caused the encryption randomness to be a constant, which made the ciphertexts deterministic and enabled trivial plaintext recovery attacks [7]. In this case, the risk was partly mitigated as the bulletin board was not publicly accessible; the IVXV "bulletin board" is also not publicly accessible, which mitigates some privacy issues at the cost of weakening verifiability.

2.1.1.3 Moscow Internet Voting System Research revealed two main attacks on the Moscow Internet voting deployment [15]. The protocol originally used a custom "multi-level" ElGamal to encrypt votes. The purpose of this multi-level design was unclear, and the underlying groups were the full multiplicative groups F_p^* of order $p-1 = 2q$, which is vulnerable to a small subgroup attack, as it has a small subgroup of order 2. More critically, the primes p used in these groups were less than 256 bits, making the discrete logarithm problem easily solvable by modern computers.

In the modified version, the protocol replaced the custom scheme with standard ElGamal $(a, b) = (g^r, m \cdot y^r)$, with the generator g (and hence $y = g^x$) chosen inside the subgroup Q_p of quadratic residues modulo p to eliminate the order-2 subgroup weakness. However, this introduced a plaintext encoding flaw that leaks one bit of information per ballot. The encoding would map voters' choices to arbitrary elements of F_p^* rather than to Q_p. As $g, y \in Q_p$, and $y^r \in Q_p$ for all r, the residuosity of b equals the residuosity of the underlying plaintext choice m. An attacker can therefore compute the Legendre symbol on b to learn whether the encoded choice is a quadratic residue.

2.1.1.4 SwissPost Voting System 2019 In 2019, during a public review of the SwissPost–Scytl (sVote) system, Haines, Lewis, Pereira, and Teague identified critical soundness flaws in the system's zero-knowledge proofs that broke verifiability [16]. The mixnet's proof of correct shuffle relied on commitment parameters generated without a verifiable procedure: an entity that knew the corresponding trapdoor could produce a shuffle proof that verified correctly while the mixer altered, removed, or substituted the underlying votes, defeating universal verifiability. A closely related weakness stemmed from a flawed, non-adaptive use of the Fiat–Shamir transform in the decryption proofs, which allowed a malicious decryption service to generate proofs that verified but attested to an incorrect plaintext. Either flaw permitted undetectable manipulation of the tally, and the authors demonstrated the issues with concrete cheating proof transcripts.

In a more recent study [19], Haines and Rose identified eight new weaknesses in SwissPost. Four of these vulnerabilities can violate the main security properties of the system, including one privacy vulnerability and three vulnerabilities that break universal verifiability (UV):

- Privacy: A tally component can be triggered early, causing only a small subset of votes to be mixed and decrypted;
- UV: inconsistent parsing of shuffle-verification data, allowing the verifier to check the wrong chain of ballot transformations;
- UV: inadequate election configuration validation that allows a malicious setup component to spoof the election, allowing ballotboxes to be silently dropped;
- UV: auditor-side filtering of invalid ballots during tallying is not enforced, allowing invalid ballots to be tallied.

2.1.1.5 New South Wales iVote system The iVote system for the New South Wales state election was found vulnerable to active attacks [20] due to weak transport security. Its HTTPS servers supported weak (export-grade) ciphersuites, enabling man-in-the-middle attacks on voter sessions, while the election gateway used plain HTTP, allowing an attacker who could misdirect voter registration to substitute the voter's ballot with their own.

2.1.2 Security definitions and analyses of e-voting protocols

E-voting protocols have motivated a substantial body of work on formal security analysis. Prior work has formalised a range of security properties for analysing such protocols. Kremer et al. [24] give an early formal classification of election verifiability into individual, universal, and eligibility verifiability (IV, UV, and EV); Cortier et al. [10] develop strong verifiability definitions for Helios-style systems; Bernhard et al. [6] introduce the BPRIV framework for ballot privacy; and Juels et al. [23] formalise coercion resistance (CR) for electronic elections. Security analyses of e-voting protocols are commonly framed around verifiability, particularly IV and UV, and privacy. Other

properties, such as EV and CR, are also studied, but are less consistently targeted by deployed systems.

IV allows a voter to check that her vote was recorded as intended.

UV allows observers to check that the tallied result matches the recorded votes.

EV requires that only eligible voters can cast votes that will be included in the election outcome.

Privacy requires that an adversary learns nothing about honest voters' choices beyond the election result.

CR requires that a coerced voter has a way to evade the coercer while producing evidence that is indistinguishable from compliance.

The exact definition of each property may vary depending on the specific protocol and protocol participants. Several deployed e-voting systems have been analysed under such formal models, including Belenios [11] and the Swiss Post voting system [35], both of which are prominent comparators for IVXV in the European e-voting context. Belenios has been studied for both verifiability and privacy, including machine-checked analyses under explicit trust assumptions [9]. Swiss Post has a substantially documented protocol and formal security analyses aligned with its published specification and regulatory setting. These works provide the definitions and comparison points used later in our security comparison (Sec. 6).

2.2 Cryptographic Primitives

We now recap the core cryptographic primitives used throughout the IVXV protocol and its security analysis.

2.2.1 ElGamal Encryption

ElGamal [14] is a public-key encryption scheme built directly on top of the Diffie–Hellman problem. In a group G of order q with generator g, a user's key pair is

$$\text{private key: } x \in \mathbb{Z}_q, \text{ public key: } h = g^x$$

To encrypt a message $m \in G$, the sender chooses a random value $r \in \mathbb{Z}_q$ and computes the ciphertext via the receiver's public key h:

$$(c_1, c_2) = (g^r, m \cdot h^r)$$

Decryption reveals the message by raising c_1 to the secret exponent x to cancel the common factor:

$$m = c_2 \left(c_1^x\right)^{-1} = m \cdot h^r \cdot (g^r)^{-x} = m \cdot g^{xr} g^{-xr} = m$$

ElGamal is semantically secure under the Decisional Diffie-Hellman (DDH) assumption and it is homomorphic under multiplications:

$$(c_1, c_2) \times (c_1', c_2') = (c_1 c_1', c_2 c_2') = \left(g^{r+r'}, m m' h^{r+r'}\right)$$

This natural homomorphism of ElGamal allows products of plaintexts to be computed in the ciphertext domain, making ElGamal a building block for privacy-preserving primitives, such as mixnets and threshold decryption, where the encrypted data needs to be processed without being revealed.

2.2.2 Digital Signatures

A digital signature scheme provides message authenticity, integrity, and non-repudiation. Formally, a signature scheme $\mathsf{Sig} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ consists of three efficient algorithms:

- The key generation algorithm $(sk, pk) \leftarrow \mathsf{KeyGen}()$ generates a key pair: a private key sk for signing, and a public key pk for verification of signatures.
- The signing algorithm $\sigma \leftarrow \mathsf{Sign}(sk, m)$ produces a signature σ on message m using sk.
- The verification algorithm $\{\texttt{accept}, \texttt{reject}\} \leftarrow \mathsf{Verify}(pk, m, \sigma)$ checks that σ is a valid signature on m under pk.

A secure signature scheme is existentially unforgeable under adaptive chosen-message attack (EUF–CMA): even if an adversary can obtain signatures on any messages of its choice, it remains infeasible to produce a valid signature on any new message not previously queried.

In e-voting, digital signatures are often used to bind ballots to voter identities and protect the integrity of the ballots. Each voter possesses a signing key pair certified by a trusted certificate authority (CA), e.g., a national PKI. When casting a vote, the voter signs the ballot container. The election authority that collects the vote can then verify the signature against the public key in the voter's certificate, ensuring that the ballot is produced by an authenticated voter and the contents have not been tampered with after signing.

2.2.3 Mixnets

A mix network (mixnet) is a cryptographic system for anonymising communications by obscuring the link between inputs and outputs.

A mixnet consists of a sequence of servers (mix nodes) $n_1, \ldots, n_k$. Given input as a batch of elements $M = (m_1, \ldots, m_n)$, each node n_i takes as input M'_{i-1} (with $M_0 = M$), applies a random permutation π_i over the input, and transforms it with a cryptographic operation f_i (e.g., re-encryption, decryption). The output $M'_i = f_i(\pi_i(M'_{i-1}))$ is passed to the next node.

The final node n_k produces the mixed batch M'_k. By applying independent permutations and transformations, the mixnet provides input-output unlinkability, meaning that it is computationally infeasible for an adversary to link any input element m_j to its corresponding output in M'_k, under standard cryptographic assumptions.

In e-voting systems, a mixnet is often used to anonymise the ballots collected by the election authority during the voting phase before decryption and tallying. This ensures that individual votes remain unlinkable to voters.

2.2.4 Threshold Decryption

Threshold decryption schemes split a decryption key into shares such that only multiple parties (each with a share) can jointly perform decryption. In a (t, n)-threshold scheme, the key is divided among n parties, and any subset of $\geq t$ parties can perform decryption, while $< t$ shares reveal no information.

A commonly used scheme is Shamir's Secret Sharing [29], which relies on polynomial interpolation. Let $f(x)$ be a random polynomial of degree $t - 1$, where the secret key s is embedded as the constant term $f(0) = s$. Each share is a distinct point $(i, f(i))$ for $i \in \{1, \ldots, n\}$. Given any $\geq t$ shares, the polynomial $f(x)$ can be uniquely interpolated, allowing recovery of s. With fewer than t points, the polynomial remains underdetermined; crucially, this interpolation can be computed in the exponent, allowing decryption of ElGamal ciphertexts without revealing the secret key.

2.2.5 Zero Knowledge Proofs

Zero Knowledge Proofs (ZKP) allow one party (prover) to prove to another (verifier) the validity of a statement without revealing any additional information. A ZKP must satisfy three properties:

- Completeness: if a statement is true and both parties follow the protocol, the verifier accepts.
- Soundness: if a statement is false, no prover can convince the verifier to accept with nonnegligible probability.
- Zero Knowledge: the verifier learns nothing beyond the truth of the statement

In e-voting systems, ZKPs are often used to ensure verifiability without breaking ballot secrecy. For example, e-voting systems often use mixnets to make the input ballotbox unlinkable from the output ballotbox for tallying. The mixnet might use a proof of shuffle to show that the output batch is a shuffle of the input batch without revealing any correlation (e.g., input slot i maps to output slot j)

3 Methodology

3.1 Protocol Reconstruction

Our analysis of the IVXV protocol is primarily based on its partially open-source implementation:

- `ivxv` (commit 2785872): the core infrastructure of deployed IVXV, including the server-side code for CS, PA, and application code for KA and AA. The repository also contains demo election configurations, logs, and official documentations that we used to help infer the intended protocol participants, security properties, and protocol workflows that are unspecified in the publicly accessible implementation.
- `ivotingverification` (commit 53ea2d4): the VVD source code used for Android devices to perform self-verification.
- `ios-ivotingverification` (commit 4230518): the VVD source code used for iOS devices.
- `ivxv-mixnet-adapter` (commit d40a9f3): the wrapper code for each Verificatum mixer; handles encoding conversion from IVXV ballots to Verificatum group elements.

Analysed version These commits correspond to the IVXV release (version 1.9.1, published under the tag KOV2025) deployed in the October 2025 Estonian local government elections, which was the most recent public release at the time of our analysis. The next release, intended for the March 2027 Riigikogu elections, was still under development and not publicly available when this report was written. All protocol descriptions, findings, and reassessments of known attacks in this report therefore refer to the 2025 release; whether a given issue persists in the 2027 release can only be determined once that code is published.

We additionally performed light-audit of public libraries used by different protocol participants for parsing, encoding, and serialisation, to ensure consistency:

- `digidoc4j` (commit b74d2da): used by PA for BDOC container parsing and validation.
- `Bouncy Castle` 1.70 and `Spongy Castle` 1.56: used by PA and VVD, respectively, to handle ciphertext encoding.

3.2 Vulnerability Categorisation

Because the public IVXV documentation does not provide a complete trust model, our security analysis starts from the reconstructed protocol under a set of standard election-security properties: individual verifiability, universal verifiability, privacy, coercion resistance, and eligibility verifiability. For each property, we define the adversarial goal, the default adversarial capabilities, and the relevant corruption scenarios. This exposes a large number of possible violations, not all of which are equally meaningful in practice. We therefore categorise attacks as follows.

Trivial attacks. Attacks correspond to corruptions that would normally be outside the intended trust model, e.g. participants that a property should inherently trust, are considered trivial. These cases are recorded for completeness but are not treated as substantive vulnerabilities.

Known attacks. Attacks previously identified in the literature or public analyses. We reassess these attacks against the current implementation where possible.

New findings. Non-trivial attacks we identified that have not previously been reported.

We further annotate each attack with labels corresponding to different metrics. For instance, capability labels identify additional assumptions required by the attack beyond the default adversarial model; impact labels describe the practical impact of the attack, such as whether it affects only a small set of voter, or whether the exploitability depends on specific election configurations outside the adversary's control. These labels are used to make the analysis comparable across different security properties without implying that every listed violation has the same severity. Details of the annotations are presented in Section 5.2.

3.3 Experimental Validation

We created proof-of-concept (PoC) tests to validate and demonstrate selected new security findings discussed in Section 7. The PoCs are organised by vulnerability identifier. The current artefact covers security findings: IV-V2, IV-V5, UV-V1, UV-V2, and P-V1 (discussed in Sec. 5).

Running the PoCs requires three repositories: The `ivxv-poc` repository contains fixtures for each vulnerability, scripts for setting up the test environment; the `ivxv` and `ivotingverification` repositories are private mirrors of the upstream repositories, with test cases added. Each test invokes the real implementation logic of the relevant participant without building or running the full participant code.

See `README.md` in `ivxv-poc` for details on test implementation and instructions for running tests.

4 Protocol Specification

The protocol algorithms and data structures use the variable naming conventions described in Appendix C.

Our description of the protocol is structured as follows:

- We begin by introducing the protocol participants and their interactions at a high level in Subsection 4.1.
- Subsections 4.2, 4.3, 4.4, and 4.5 present the setup, voting, tally, and audit phases, respectively.
- In Subsection 4.6, we discuss the mechanisms which ensure that messages are not forged.
- In Subsection 4.7, we describe the accompanying TLA+ model of the protocol.
- In Appendix B we introduce the core cryptographic algorithms used in the protocol.
- In Appendix E we introduce the core data structures used in the protocol.

4.1 Participants

Table 3 defines the protocol participants and their abbreviations used throughout the specification, and Figure 1 shows the participants clustered by functionality. Detailed message flows between protocol participants are given in the phase descriptions in the following sections and summarised in Subsection 4.6.

Abbrev.	Participant	Role	Main phase(s)
AA	Audit Application	Verifies tally and audit artefacts produced by PA, Mixer, and KA.	Audit
CS	Collector Service	Runs the online voting services, exposing APIs used by VCD and VVD for authentication, signing relay, vote submission, verification, and related requests.	Voting
EA	Election Authority	Configures the election, distributes configuration and phase inputs, and publishes election artefacts.	Setup, tally, audit
IDP	ID Provider	Relays authentication and signing requests between CS and voters' Sign devices.	Voting
KA	Key Application	Generates election keys and key shares; later reconstructs the election secret key, decrypts mixed ballots, and computes the election result.	Setup, tally
KH	Key Holder	Holds a threshold key share used by KA during key generation or decryption.	Setup, tally
Mixer	Verificatum Mixer	Shuffles the anonymised encrypted ballotbox and produces shuffle proofs.	Tally
PA	Processor Application	Checks exported ballots, applies revocation and filtering, and produces the anonymised ballotbox.	Tally
Reg	Registration Service	Provides qualification responses, such as TSP/TSPReg and OCSP responses, for submitted ballots.	Voting
Sign	Signing Device	Produces authentication and ballot signatures upon voter approval.	Voting
VCD	Vote-Casting Device	Presents choices to the voter, constructs the encrypted ballot, and submits the ballot to CS.	Voting
Voter	Human voter	Selects voting choices, approves signing operations, and optionally performs self-verification.	Voting
VVD	Vote-Verification Device	Retrieves the stored ballot from CS and helps the voter verify the vote.	Voting

Table 3: Protocol participants

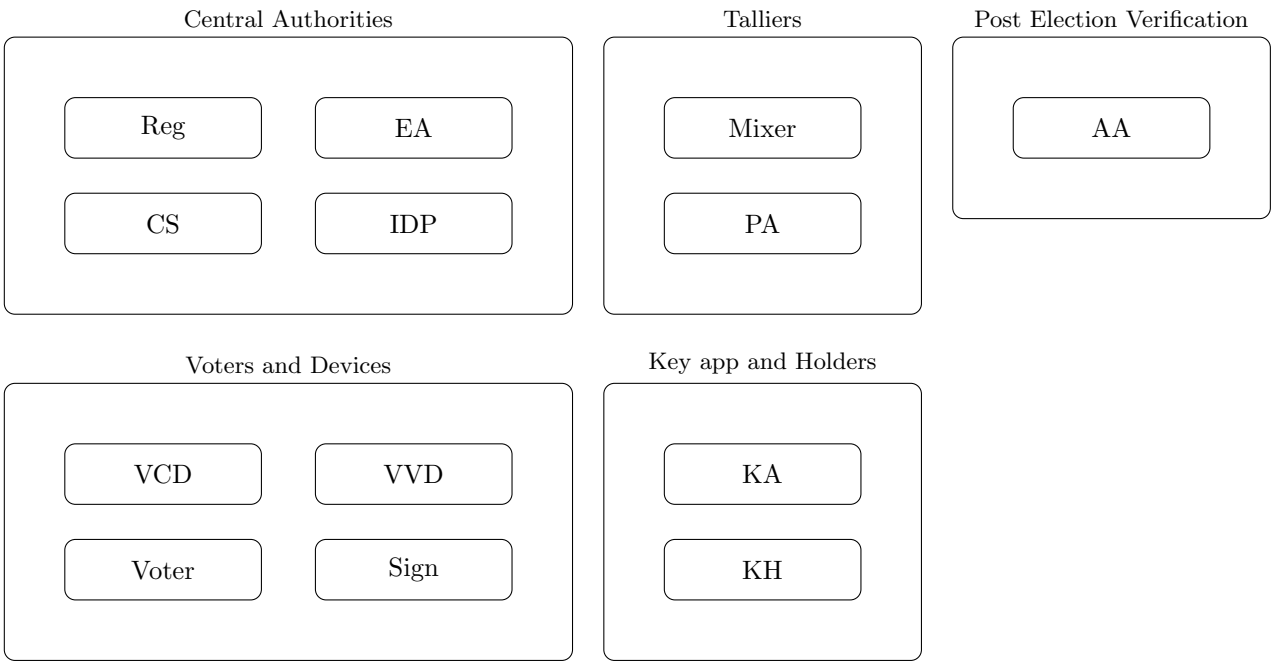

Figure 1: Protocol participants

4.2 Setup Phase

Table 4 shows the subprotocols and algorithms involved in the setup phase.

Subprotocol / Algorithm	Actor
InitElection	
KeyGen	
KeyAppInit	KA
SetupElection	

Table 4: Setup Phase Subprotocols

4.2.1 InitElection

EA starts the InitElection subprotocol by releasing the Key App (KA), Processor App (PA), Audit App (AA), and the client applications for Mixer, VCD, and VVD. Each application has a locally configured trust root ($\texttt{conf}_{\texttt{trust}}$: `TrustConfig`) that defines the trusted CA certificate and the certificates authorised for the OCSP and TSP services (run by Reg).

Algorithm 1: TrustConfig - Link to Code

Struct

- $\texttt{crt}_{\texttt{CA}}$: `X509Certificate`. PEM-encoded authentication (CA) certificate
- $\texttt{crt}_{\texttt{OCSP}}$: `X509Certificate`. PEM-encoded OCSP certificate
- $\texttt{crt}_{\texttt{TSP}}$: `X509Certificate`. PEM-encoded TSP certificate

Each of the following participants produces or owns certificates issued by the CA specified in $\texttt{conf}_{\texttt{trust}}$:

- CS: $\texttt{crt}_{\texttt{CS}}$ and $\texttt{crt}_{\texttt{tspClient}}$
- IDP: $\texttt{crt}_{\texttt{MID}}$
- EA: $\texttt{crt}_{\texttt{voterlist}}$

These certificates are later distributed by EA as part of the configuration package for certain applications and services.

4.2.2 KeyGen

Figure 2 summarises this subprotocol for generating election public keys and threshold key shares. EA provides cryptographic parameters for the election to KA; Key Holders (KH) insert SmartCards for KA to write the key shares; KA then generates the election key materials using threshold algorithms. The election decryption key is generated using threshold ElGamal, which follows the original Pedersen method [27]; the election signing key is produced using threshold RSA, following Shoup's construction [30]. KA writes key shares to the smartcard of each KH, and sends EA the corresponding election public key certificates.

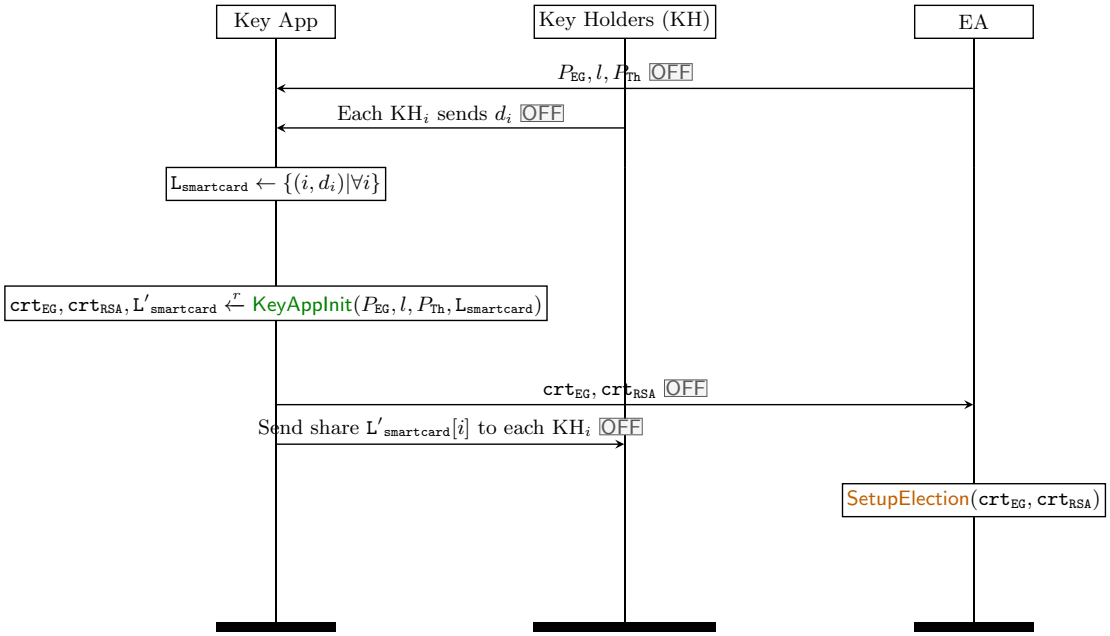

Figure 2: Setup phase. KeyGen subprotocol

4.2.2.1 KeyAppInit takes as input the ElGamal parameters, the RSA modulus length, threshold key sharing parameters, and the Key Holders' smart cards. It generates threshold key shares for ElGamal decryption and RSA signing, writes the private shares to the smart cards, and outputs the corresponding public keys as signed certificates.

Algorithm 2: KeyAppInit - Link to Code

Input:

- $P_{\texttt{EG}}$: `ElGamalParameters`. With group order q and generator g
- l : `Int`. RSA mod length
- $P_{\texttt{Th}}$: `Tuple<Int, Int>` $:= (t, n)$. Threshold Decryption setting: t: threshold, n: number of shares
- $\texttt{L}_{\texttt{smartcard}}$: `List<SmartCardData>`. A list of SmartCardData

Output: $\texttt{crt}_{\texttt{EG}}$: `X509Certificate`, $\texttt{crt}_{\texttt{RSA}}$: `X509Certificate`, and $\texttt{L}'_{\texttt{smartcard}}$: `List<SmartCardData>` containing key shares for each trustee.

Operation:

1. $\texttt{L}'_{\texttt{smartcard}} \leftarrow []$
2. $epk_{\texttt{EG}}, K \leftarrow \mathsf{GenerateKeyShares}(P_{\texttt{EG}}, (t, n))$
3. $epk_{\texttt{RSA}}, \texttt{M}_{\texttt{signShare}} \leftarrow \mathsf{GenerateSignKeyShares}(l, (t, n))$
4. **for** $i \leftarrow 1$ *to* n **do**
5. $\quad d \leftarrow \texttt{L}_{\texttt{smartcard}}[i]$
6. $\quad d.idx \leftarrow i$
7. $\quad d.P_{\texttt{EG}} \leftarrow P_{\texttt{EG}}$
8. $\quad d.k \leftarrow K[i]$
9. $\quad d.P_{\texttt{RSA}} \leftarrow epk_{\texttt{RSA}}$
10. $\quad d.\texttt{k}_{\texttt{sign}} \leftarrow \texttt{M}_{\texttt{signShare}}[i]$
11. $\quad \texttt{L}'_{\texttt{smartcard}}.\mathsf{append}(d)$
12. $\sigma \leftarrow \mathsf{ShoupSign}((t, n), epk_{\texttt{RSA}}, \texttt{M}_{\texttt{signShare}}, epk_{\texttt{EG}})$
13. $\texttt{crt}_{\texttt{EG}} \leftarrow (epk_{\texttt{EG}}, \sigma)$
14. $sig_{\texttt{sign}} \leftarrow \mathsf{ShoupSign}((t, n), epk_{\texttt{RSA}}, \texttt{M}_{\texttt{signShare}}, epk_{\texttt{RSA}})$
15. $\texttt{crt}_{\texttt{RSA}} \leftarrow (epk_{\texttt{RSA}}, sig_{\texttt{sign}})$
16. **return** $(\texttt{crt}_{\texttt{EG}}, \texttt{crt}_{\texttt{RSA}}, \texttt{L}'_{\texttt{smartcard}})$

GenerateKeyShares generates and outputs the public key and secret shares for threshold ElGamal given the ElGamal parameters and threshold key sharing parameter. This follows the original Pedersen method [27] without any of the verification features which are now common [5] but these are not needed under the assumption of a trusted dealer.

Algorithm 3: GenerateKeyShares - Link to Code

Input:
- $P_{\texttt{EG}}$: `ElGamalParameters`. With group order q and generator g
- $P_{\texttt{Th}}$: `Tuple<Int, Int>` $:= (t, n)$. Threshold Decryption setting: t: threshold, n: number of shares

Output: $epk_{\texttt{EG}}$: `Int` and K : `List<Int>` containing the key share for each trustee.

Operation:

$K \leftarrow []$
$A \leftarrow []$
$sk \xleftarrow{r} \mathbb{Z}_q$
for $k \leftarrow 1$ *to* $t-1$ **do** ▷ `DesmedtGeneration.java#L94`
  $a \xleftarrow{r} \mathbb{Z}_q$
  A.append(a)
for $i \leftarrow 1$ *to* n **do** ▷ `ProtocolUtil.java#L17`
  $s_i \leftarrow sk$
  for $k \leftarrow 1$ *to* $t-1$ **do**
    $s_i \leftarrow (s_i + A[k] \cdot i^k) \bmod q$
  K.append(s_i)
$epk_{\texttt{EG}} \leftarrow g^{sk}$
return $epk_{\texttt{EG}}, K$

GenerateSignKeyShares generates an RSA key pair using the fixed public exponent $e = 65537$ from the RSA modulus length and threshold key sharing parameter, outputs the public key (N, e), and produces secret shares of the private exponent for threshold signing.

Algorithm 4: GenerateSignKeyShares - Link to Code

Input:
- l : `Int`. RSA mod length
- P_{Th} : `Tuple<Int, Int>` $:= (t, n)$. Threshold setting: t: threshold, n: number of shares

Output: epk_{RSA} : `Tuple<Int, Int>` and K : `List<Int>` containing the key share for each trustee.

Operation:

$K \leftarrow \emptyset$
$A \leftarrow []$
$e \leftarrow 65537$
repeat
  $p \xleftarrow{r} \mathsf{Prime}(\lfloor \frac{l}{2} \rfloor)$
  $q \xleftarrow{r} \mathsf{Prime}(\lfloor \frac{l}{2} \rfloor)$
  $N \leftarrow p \cdot q$
  $\varphi \leftarrow (p-1) \cdot (q-1)$ ▷ MathUtil.java#L134
until $\mathsf{gcd}(e, \varphi) = 1$
$sk \leftarrow e^{-1} \bmod \varphi$
/* Random coefficient */
for $k \leftarrow 1$ *to* $t-1$ **do**
  $a \xleftarrow{r} \mathbb{Z}_{\varphi}$
  $A.\mathsf{append}(a)$
for $i \leftarrow 1$ *to* n **do**
  $s_i \leftarrow sk$
  for $k \leftarrow 1$ *to* $t-1$ **do**
    $s_i \leftarrow (s_i + A[k] \cdot i^k) \bmod \varphi$ ▷ Polynomial.java#L59
  $K[i] \leftarrow s_i$
$epk_{\text{RSA}} \leftarrow (N, e)$
return epk_{RSA}, K

ShoupSign implements threshold signing following Shoup's construction [30]. Given a message m, an RSA public key (N, e), and private key shares under a (t, n)-threshold scheme, the algorithm generates a list of partial signatures of m using the key shares, then combines them to produce an RSA signature without ever reconstructing the private key.

Algorithm 5: ShoupSign - Link to Code

Context:

$\mathsf{PSSEncode}(m, N, s) \in \mathbb{Z}_N^*$. Returns the EMSA-PSS encoding of message m for modulus N, using a secure hash function and salt s

Input:

$P_{\texttt{Th}}$: `Tuple<Int, Int>` $:= (t, n)$. Threshold setting

$epk_{\texttt{RSA}}$: `Tuple<Int, Int>` $:= (N, e)$. RSA public key

K : `Map<Int, Int>`. Key shares, $K[i]$ denotes the share for key holder i. If key holder i did not provide a share, then $K[i] =$ **null**

m : `Byte[]`. Message to sign

Output: σ : `Byte[]` signature

Operation:

1 $U \leftarrow \{i \in [1, n] \mid K[i] \neq \textbf{null}\}$

2 **if** $|U| < t$ **then return** $\perp$

3 $s \xleftarrow{r} \mathsf{Rand}(32)$

4 $c \leftarrow \mathsf{PSSEncode}(m, N, s)$ ▷ SignatureUtil.java#L60

5 $\texttt{share} \leftarrow 1 \in \mathbb{Z}_N^*$

6 **for** $i \in U$ **do** ▷ ShoupSigning.java#L156

7 $\quad \sigma_i \leftarrow c^{K[i]} \bmod N$

8 $\quad \lambda_i \leftarrow (\prod_{j \in U, j \neq i} \frac{j}{(j-i)}) \cdot n!$

9 $\quad \texttt{share} \leftarrow \texttt{share} \cdot \sigma_i^{\lambda_i} \bmod N$

10 find $\mathbf{a}, \mathbf{b}$ s.t. $\mathbf{a}e + \mathbf{b}n! = 1$ ▷ ShoupSigning.java#L166

11 $\sigma \leftarrow \texttt{share}^b \cdot c^a \bmod N$

12 **return** σ

Correctness. Let $c = \mathsf{PSSEncode}(m, N, s)$ and $\varphi = \varphi(N)$, and recall that the shares satisfy $K[i] = f(i) \bmod \varphi$ for a degree-$(t-1)$ polynomial f with $f(0) = d = e^{-1} \bmod \varphi$ (GenerateSignKeyShares). The weights $\lambda_i = n! \cdot \prod_{j \in U, j \neq i} \frac{j}{j-i}$ are integers, because $\prod_{j \neq i}(i - j)$ divides $n!$, and Lagrange interpolation gives the integer identity $\sum_{i \in U} \lambda_i f(i) = n!\, d$. Hence, for some integer M,

$$\texttt{share} = \prod_{i \in U} c^{K[i]\lambda_i} = c^{\sum_{i \in U} \lambda_i K[i]} = c^{\,n!\, d - \varphi M} \equiv c^{\,n!\, d} \pmod N,$$

where the last step uses $c^{\varphi} \equiv 1 \pmod N$ (Euler), which simultaneously absorbs the reduction of the shares modulo φ. With Bézout coefficients a, b satisfying $ae + b\, n! = 1$, the output is $\sigma = \texttt{share}^b\, c^a = c^{\,n!\, db + a}$, and therefore

$$\sigma^e = c^{\,e(n!\, db + a)} \equiv c^{\,n!\, b + ea} = c^1 = c \pmod N,$$

using $ed \equiv 1 \pmod{\varphi}$ and $ea + n!\,b = 1$. Thus σ verifies as a standard RSA-PSS signature. The argument requires that c be invertible modulo N, that $\gcd(e, n!) = 1$ (immediate, since $e = 65537$ is prime and $n < e$), and that N be a product of two distinct primes with totient $\varphi(N)$; notably it does *not* require N to be a product of safe primes.

4.2.3 SetupElection

During the SetupElection subprotocol, EA prepares the election configuration, which contains the certificates of the election public keys and other election parameters, and then shares it with the relevant protocol participants. Some distribution channels are visible in the implementation (VVD); others are closed source (VCD) or run over deployment-specific out-of-band channels.

Figure 3 summarises the subprotocol; in it, we group all out-of-band configuration receivers into one entity, and group VCD and VVD as client applications. We note that this is a logical grouping rather than an exact runtime order: the VVD retrieves its configuration when the verification app is run during the voting phase; the VCD is closed source, so we model it as receiving client configuration analogous to the VVD's.

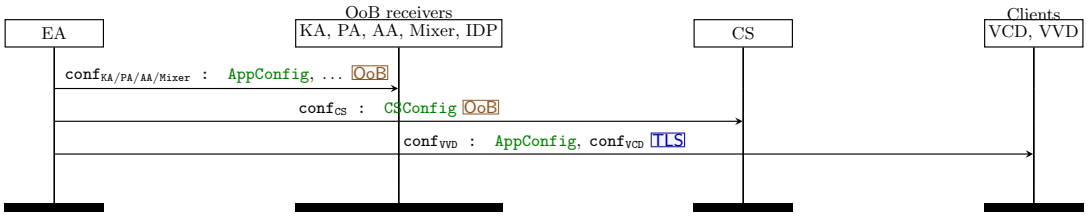

Figure 3: Setup phase. SetupElection subprotocol

Each protocol participant receives a role-specific version of the election configuration. Here we use two configuration data structures to represent all of them:

- CS receives CSConfig, which will be referenced numerous times during the voting phase.
- Other participants receive AppConfig, a single abstract data structure that we use to summarise the configuration files for non-CS participants. We annotate AppConfig with the fields common to all these participants and with participant-specific fields. For instance, the VVD additionally receives the TSP client certificate; PA receives a voter list; Mixer receives the Verificatum proof metadata; AA receives audit flags.

Algorithm 6: CSConfig

Struct

$\texttt{conf}_{\texttt{trust}}$: TrustConfig
/* Received from EA */

eid : String. Election identifier

Q : List<String>. A list of question IDs

$\texttt{crt}_{\texttt{voterlist}}$: X509Certificate. voter list cert

$\texttt{conf}_{\texttt{mid}} := \{\texttt{url}, \texttt{crt}_{\texttt{MID}}, \dots\}$. A record containing ID provider (IDP) data for mobile ID authentication, e.g., url and certificate

$P_{\texttt{EG}}$: ElGamalParameters. ElGamal group parameters used for ciphertext validation, obtained from the collector election configuration's ballot-encryption public key or group identifier ▷ conf.go#L88

$\texttt{L}_{\texttt{time}} := \{t_{\texttt{SStart}}, t_{\texttt{EStart}}, t_{\texttt{EEnd}}, t_{\texttt{SEnd}}, t_{\texttt{VEnd}}\}$. A list of time, marking the start, end of the election, service, and verification

$\texttt{idType} \in \{\texttt{commonname}, \texttt{serialnumber}, \texttt{pnoee}\}$. Defines the field in X509Certificate to be used as the voter ID (see VerifyBallot). In the demo config, "pnoee" is used

$\texttt{conf}_{\texttt{age}} := \{\texttt{timeZone}, \texttt{limit}\}$. A record that stores the age limit info

$\texttt{cnt}_{\texttt{verif}}$: Int. Number of times a vote can be verified

latestonly : Bool. Determines whether a voter can only verify her latest cast during Voting phase

$t_{\texttt{ver}} = 30$ min. Number of minutes a vote can be verified after cast

$\texttt{TTL}\{\texttt{auth}, \texttt{choice}, \texttt{vote}, \texttt{verify}$: Duration$\}$. Session timeout intervals for authentication, choice selection, vote signing, and verification respectively.
▷ example.technical.yaml

$\texttt{M}_{\texttt{DC}}$: Map<String, CandidateList>. Maps district to candidate list

$\texttt{M}_{\texttt{voterlist}}$: Map<String, Voter>. Maps a voter ID to a voter
/* Service state */

$sk_{\texttt{tspClient}}, pk_{\texttt{tspClient}}$. Key pair for TSP REG client to sign a TSP request

$\texttt{k}_{\texttt{ticket}}$: Byte[]. Symmetric key for authentication and data tokens, loaded from the service's shared ticket-key file

$\texttt{n}_{\texttt{ticket}}$: Byte[12]. Per-ticket-manager nonce counter, randomly initialized when the manager is created and shared by authentication- and data-token creation. It is incremented as a fixed-width big-endian value with byte carry

Algorithm 7: AppConfig

Struct

`/* Common election material */`

eid : `String`. Election identifier; for PA, this is taken from the loaded district list.

$\texttt{crt}_{\texttt{EG}}$: `X509Certificate`. Election ElGamal public key certificate; its public key is $epk_{\texttt{EG}}$

$P_{\texttt{EG}}$: `ElGamalParameters`. ElGamal parameters

`/* Election list material used by several apps */`

Q : `List<String>`. Election questions

$\texttt{M}_{\texttt{DC}}$: `Map<String, CandidateList>`. Maps districts to candidate lists

$\texttt{conf}_{\texttt{trust}}$: `TrustConfig`. Trust root used for BDOC and qualification-response checks

`/* VVD, VCD */`

$\texttt{url}_{\texttt{Conf}}$: `String` := `www.valimised.ee`. URL used to retrieve client configuration

$\texttt{url}_{\texttt{CS}}$: `String` := `verification.ivxv.valimised.ee`. Verification service URL

$\texttt{crt}_{\texttt{CS}}, \texttt{crt}_{\texttt{tspClient}}$: `X509Certificate`. Certificates used for TLS, BDOC, and qualification-response validation

`candidates` : `List<String>`. Candidate data received from CS during voting

`/* Processor App */`

$\texttt{crt}_{\texttt{voterlist}}, \texttt{crt}_{\texttt{tspClient}}$: `X509Certificate`. Voter-list signing key and TSP client certificate

$\texttt{L}_{\texttt{time}} := \{t_{\texttt{SStart}}, t_{\texttt{EStart}}, t_{\texttt{EEnd}}, t_{\texttt{SEnd}}, t_{\texttt{VEnd}}\}$. Election, service, and verification time bounds

`idType` $\in \{$`commonname`, `serialnumber`, `pnoee`$\}$. Certificate field used as voter ID (see VerifyBallot)

$\texttt{M}_{\texttt{voterlist}}$: `Map<String, Voter>`. Maps voter ID to voter data

`/* Mixer */`

`protInfo`, `proofDir` : `String`. Verificatum protocol information and proof directory

`threaded` : `Bool`. Whether shuffle verification uses the threaded verifier

`/* Audit App */`

$\texttt{crt}_{\texttt{RSA}}$: `X509Certificate`. KA's self-signed result-signing certificate, supplied out of band for the modelled verification of KA output signatures. The current Audit App does not load this certificate or perform that verification itself

n_v : `Int`. Shuffle-verification challenge length

`verifyInvalid` : `Bool`. Whether invalid ciphertext proofs are verified

`/* Key App */`

$\texttt{M}_{\texttt{DP}}$: `Map<String, List<String>>`. Maps each configured district to its parish IDs, from the loaded district list

$P_{\texttt{Th}}$. Threshold parameters

`checkDecodable`, `withProof`, `proveInvalid` : `Bool`. Decryption and proof-output flags

4.3 Voting Phase

Table 5 shows the subprotocols and algorithms involved in the voting phase.

Subprotocol / Algorithm	Actor
SendBallot	
VerifyAndUpdateSession	CS
CreateAuthToken	CS
VoterChoices	CS
CreateBDOC	VCD
ConfirmBallot	
VerifyBallot	CS
RegisterBallot	CS
VerifyRegistration	CS
StoreRegisteredBallot	CS
VerifyCast	
VerifyRequestedVote	CS
ValidateBallot	VVD
DecryptVotes	VVD
ExportVotingData	
ExportBallots	CS

Table 5: Voting Phase Subprotocols

4.3.1 SendBallot

SendBallot is the first subprotocol of the voting phase. It involves the voter, VCD, CS, IDP, and the voter's Sign device. The concrete signing device depends on the signing method. IVXV supports Mobile-ID, Smart-ID, and web-based flows; our protocol specification uses Mobile-ID as the representative example. The SendBallot subprotocol is further divided into multiple steps: Auth, Choices, GetCert, and Sign.

4.3.1.1 Auth Figure 4 shows the Auth step, which creates a voting session with CS and authenticates the voter through the configured IDP. The voter initiates the protocol by entering her voter profile data in the VCD client app; the VCD asks CS to start authentication using the voter-provided profile; CS generates an authentication challenge r and forwards it through IDP to the voter's Sign device for signing. CS returns the same challenge, a session identifier $sid_{\texttt{CS}}$, a polling identifier $sid_{\texttt{auth}}$, and a data token to the VCD; the voter compares the challenge displayed on the VCD and on the Sign device, and approves the signing request on the Sign device; the Sign device then signs the challenge and returns the signature through IDP.

The VCD subsequently polls CS for completion by sending $sid_{\texttt{CS}}$ and $sid_{\texttt{auth}}$. CS checks the voting-session state, forwards the polling identifier to IDP, receives the corresponding signed authentication result, verifies the returned signature, and issues an authentication token `authToken` for the session. Subsequent requests from the VCD (or VVD) to CS in the SendBallot and VerifyCast protocols must carry a valid $sid_{\texttt{CS}}$, and certain steps additionally require `authToken`.

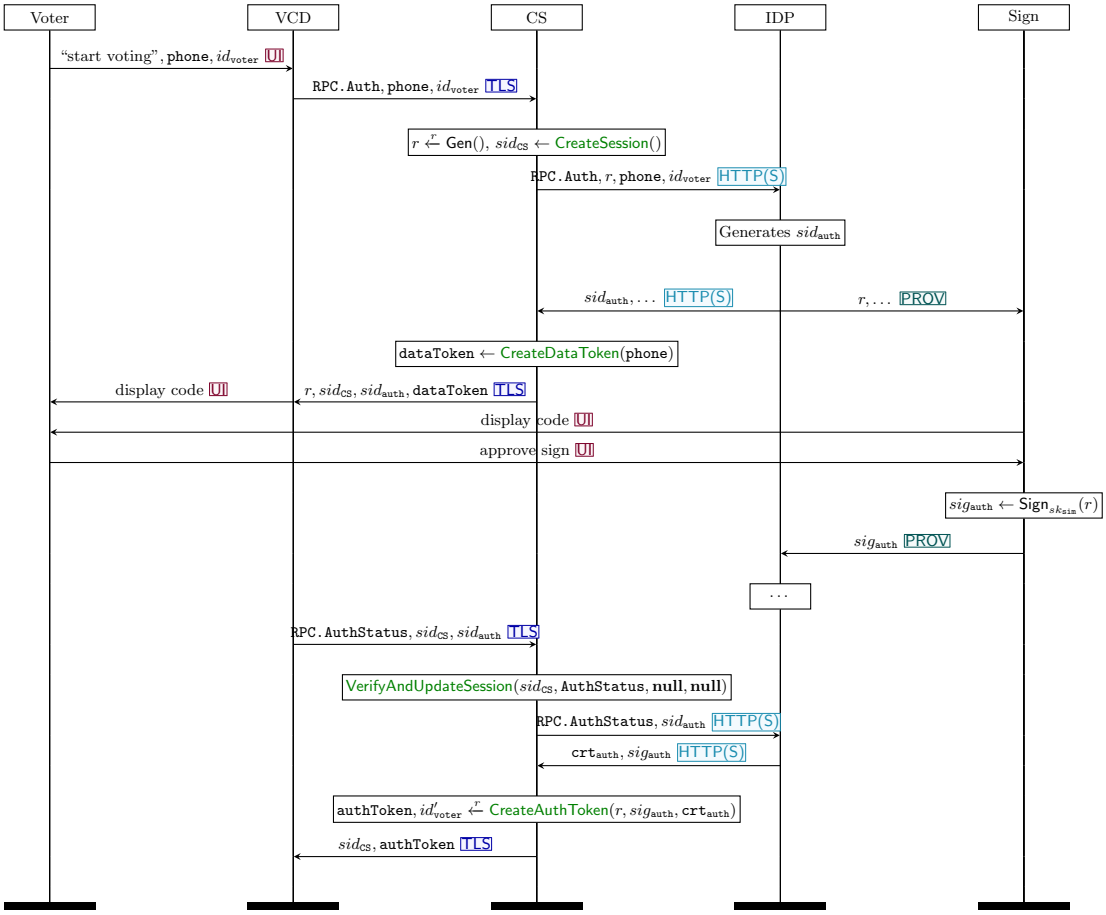

Figure 4: Auth

4.3.1.2 Choices Figure 5 shows the Choices step, where VCD requests the election candidate list for the authenticated voter. CS validates the session and authentication token, resolves the voter's district from the voter list, and returns the corresponding district-specific choice list to VCD for display.

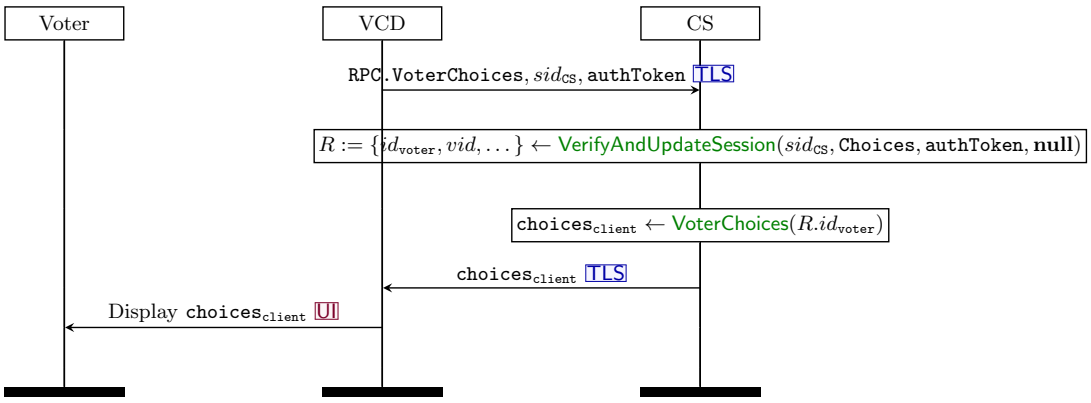

Figure 5: Choices

4.3.1.3 GetCert Figure 6 shows the GetCert step, where VCD obtains the voter's signing certificate and prepares a BDOC for voter's intent. The voter selects a candidate from the received choice list; the VCD asks CS to obtain the voter's signing certificate from IDP; CS validates the session, authentication token, and data token, then requests the certificate from IDP using the authenticated voter identity and the stored provider-specific data; after receiving the voter's certificate, the VCD prepares a ballot BDOC that contains the voter's intent and the signing certificate.

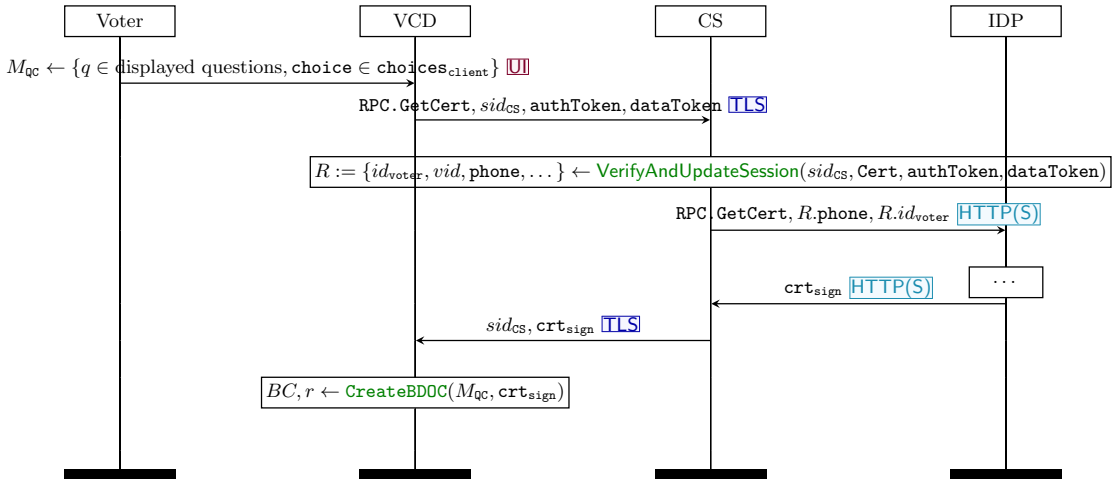

Figure 6: GetCert

4.3.1.4 Sign Figure 7 shows the Sign step, where VCD obtains the voter's signature on the ballot BDOC and prepares a signed BDOC ready for submission. The VCD sends CS the digest to be signed for the constructed BDOC; CS validates the session and relays the request to IDP, which obtains a signature from the Sign device. The VCD then retrieves the signature from CS through a polling process similar to the Auth step and appends it to the BDOC container.

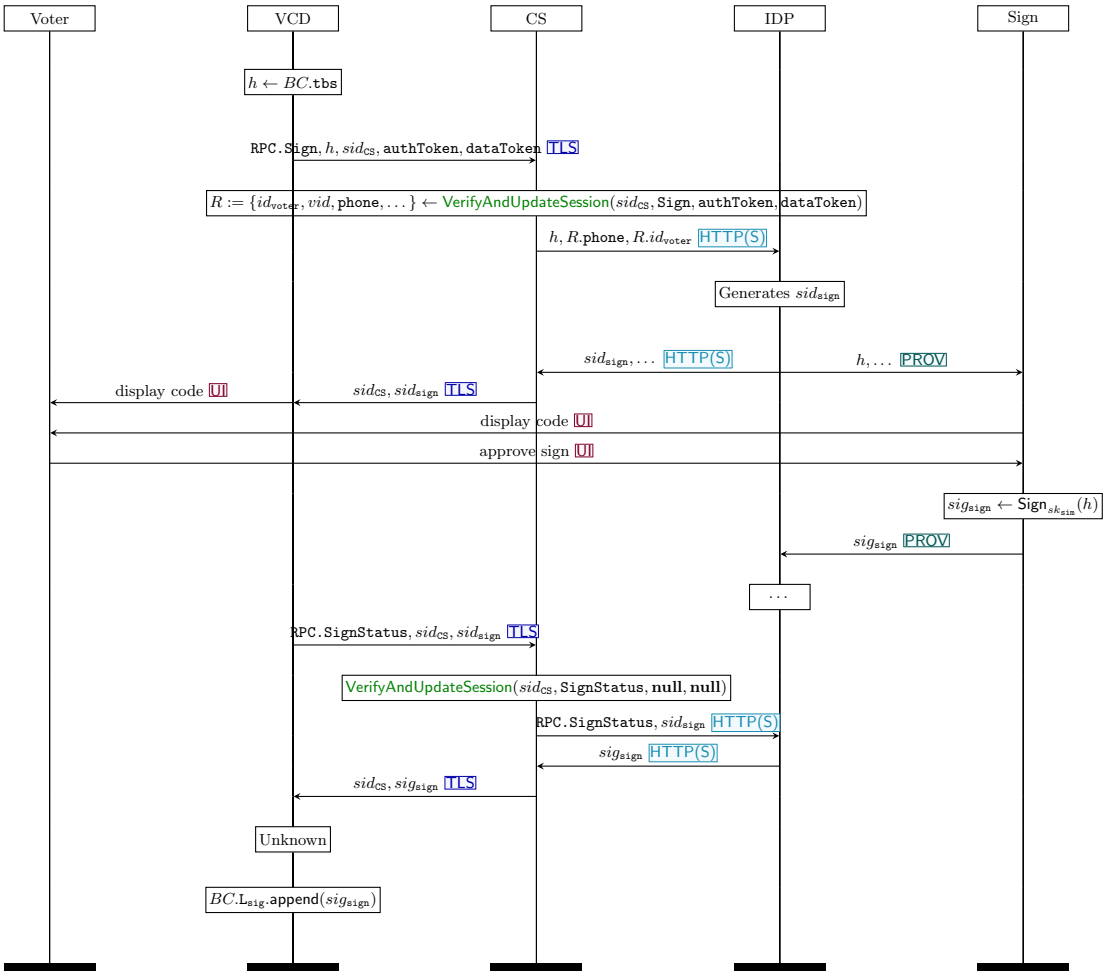

Figure 7: Sign

4.3.1.5 Session Management Helpers

This paragraph details the session-management logic involved in the SendBallot subprotocol.

CreateSession creates a voting-session record and a session ID that the voter's VCD/VVD uses for the subsequent authentication, certificate, signing, voting, and verification requests.

Algorithm 8: CreateSession - Link to Code

Context:

- $\texttt{conf}_{\texttt{CS}}$: CSConfig
- DB : StorageDB: storage database

Input: –

Output: $sid_{\texttt{CS}}$: String

Operation:

1. $sid_{\texttt{CS}} \xleftarrow{r} \mathsf{Rand}(16)$
2. $t_{\texttt{exp}} \leftarrow \mathsf{Now}() + \texttt{conf}_{\texttt{CS}}.\texttt{TTL.auth}$
3. $R \leftarrow \mathsf{SessionData}(\texttt{stage} = \texttt{Auth}, t_{\texttt{exp}} = t_{\texttt{exp}})$
4. $\texttt{DB.SessionStore.put}(sid_{\texttt{CS}}, R)$
5. **return** $sid_{\texttt{CS}}$

VerifyAndUpdateSession checks that the supplied session identifier $sid_{\texttt{CS}}$ refers to a live session in the expected protocol stage, validates any required authentication or data token, and advances the session stage before serving the request.

NeedAuth lists the protocol steps that require an authentication token.

Algorithm 9: VerifyAndUpdateSession

Context:

- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`
- `DB` : `StorageDB`: storage database

Input:

- $sid_{\texttt{CS}}$: `String`
- `nextStage` : `String`
- `authToken` : `Byte[]` or **null**
- `dataToken` : `Byte[]` or **null**

Output:

- R' : `SessionData` or $\perp$. For authenticated requests, R' additionally contains the voter ID and vote ID extracted from `authToken`; for Mobile-ID certificate and signing requests, it also contains the phone number extracted from `dataToken`.

Operation:

1 $R \leftarrow \texttt{DB.SessionStore}[sid_{\texttt{CS}}]$ ▷ `repository.go`

2 **if** $R = \textbf{null} \vee \mathsf{Now}() > R.t_{\texttt{exp}}$ **then return** $\perp$ ▷ `filter.go`

3 **if** NeedAuth(`nextStage`) **then**

4 $A \leftarrow$ OpenAuthToken(`authToken`)

5 **if** $A = \perp$ **then return** $\perp$

6 $id_{\texttt{voter}} \leftarrow A.id_{\texttt{voter}}$

7 $vid \leftarrow A.vid$

8 **if** NeedDataToken(`nextStage`) **then**

9 `phone` $\leftarrow$ OpenDataToken(`dataToken`)

10 **if** `phone` $= \perp$ **then return** $\perp$

11 **if** $\neg$CanAdvance($R.\texttt{stage} \rightarrow \texttt{nextStage}$) **then return** $\perp$

12 `ttl` $\leftarrow$ SelectTTL(`nextStage`)

13 **if** `ttl` $= \perp$ **then return** $\perp$

14 **if** `nextStage` $\in \{\texttt{Choices}, \texttt{Cert}, \texttt{Vote}\}$ **then**

15 $t \leftarrow \mathsf{Now}() + \texttt{ttl}$

16 **else**

17 $t \leftarrow R.t_{\texttt{exp}}$

18 $S' \leftarrow$ SessionData($\texttt{stage} = \texttt{nextStage}, t_{\texttt{exp}} = t$)

19 $\texttt{DB.SessionStore.put}(sid_{\texttt{CS}}, S')$

20 $R' \leftarrow S'$

21 **if** NeedAuth(`nextStage`) **then** $R'.id_{\texttt{voter}} \leftarrow id_{\texttt{voter}}$; $R'.vid \leftarrow vid$

22 **if** NeedDataToken(`nextStage`) **then**

$R'.\texttt{phone} \leftarrow \texttt{phone}$

23 **return** R'

Algorithm 10: NeedAuth

Input:
`nextStage : String`

Output:
b : `Bool`: true if the request must carry an authentication token, false otherwise

Operation:
1 **if** `nextStage` $\in \{$`Choices`, `Cert`, `Sign`, `Vote`$\}$ **then return** true
2 **return** false

NeedDataToken lists the protocol steps that require the data token generated during the Auth step.

Algorithm 11: NeedDataToken

Input:
`nextStage : String`

Output:
b : `Bool`: true if the request must carry a data token, false otherwise

Operation:
1 **if** `nextStage` $\in \{$`Cert`, `Sign`$\}$ **then return** true
2 **return** false

SelectTTL returns a time-to-live (TTL) value for the given SendBallot step. The implementation has different TTL values for each step. For simplicity, we represent them using a single abstract variable.

Algorithm 12: SelectTTL

Context:
- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`

Input:
- `nextStage` : `String`

Output:
- `ttl` : `Duration`. The expiry interval assigned to the session after advancing to `nextStage`

Operation:

if $\texttt{nextStage} \in \{\texttt{Auth}, \texttt{AuthStatus}\}$ **then return** $\texttt{conf}_{\texttt{CS}}.\texttt{TTL.auth}$
if $\texttt{nextStage} = \texttt{Choices}$ **then return** $\texttt{conf}_{\texttt{CS}}.\texttt{TTL.choice}$
if $\texttt{nextStage} \in \{\texttt{Cert}, \texttt{Sign}, \texttt{SignStatus}\}$ **then return** $\texttt{conf}_{\texttt{CS}}.\texttt{TTL.vote}$
if $\texttt{nextStage} \in \{\texttt{Vote}, \texttt{Verify}\}$ **then return** $\texttt{conf}_{\texttt{CS}}.\texttt{TTL.verify}$
return $\perp$

CanAdvance validates that a voting session may advance from its current step to the requested next step.

Algorithm 13: CanAdvance

Input:
- `currentStage` : `String`, `nextStage` : `String`

Output:
- b : `Bool`: true if the session may advance from `currentStage` to `nextStage`, false otherwise

Operation:

$$\texttt{allowed} \leftarrow \left\{ \begin{array}{l} (\texttt{Auth}, \texttt{AuthStatus}), (\texttt{AuthStatus}, \texttt{AuthStatus} \vee \texttt{Choices}), \\ (\texttt{Choices}, \texttt{Cert}), (\texttt{Cert}, \texttt{Sign}), \\ (\texttt{Sign}, \texttt{SignStatus}), (\texttt{SignStatus}, \texttt{SignStatus} \vee \texttt{Vote}), \\ (\texttt{Vote}, \texttt{Verify}), (\texttt{Verify}, \texttt{Verify}) \end{array} \right\}$$

if $(\texttt{currentStage}, \texttt{nextStage}) \in \texttt{allowed}$ **then return** true
return false

4.3.1.6 Choice Lookup Helper

VoterChoices returns the candidate list for the district of the specified voter.

Algorithm 14: VoterChoices - Link to Code

Context:

- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`
- `DB` : `StorageDB`. Storage Database

Input:

- $id_{\texttt{voter}}$: `String`

Output:

- `choices` : `CandidateList`; candidate list for the voter

Operation:

▷ `storage.go#L522`

1 $\texttt{district} \leftarrow \texttt{conf}_{\texttt{CS}}.\texttt{M}_{\texttt{voterlist}}[id_{\texttt{voter}}].d$

2 **return** $\texttt{choices} \leftarrow \texttt{conf}_{\texttt{CS}}.\texttt{M}_{\texttt{DC}}[\texttt{district}]$

4.3.1.7 Token Helpers The authentication and data-token creators use the same ticket manager. Each creation atomically reserves the current nonce and increments the shared counter before encryption.

CreateAuthToken validates the returned certificate and signature, extracts the authenticated voter ID from that certificate, and then generates an authentication token containing the authenticated certificate subject and a fresh vote identifier.

Algorithm 15: CreateAuthToken - Link to Code

Context:

$\texttt{conf}_{\texttt{CS}}$: `CSConfig`

Input:

r : `Byte[]`. Challenge

σ : `XAdESSignature`. Signature

`crt` : `X509Certificate`. Certificate associated with the auth challenge

Output:

(`authToken` : `Byte[]`, $id_{\texttt{voter}}$: `String`) if the certificate and signature verify, otherwise $\perp$

Operation:

1 **if** $\neg\mathsf{Verify}(\texttt{conf}_{\texttt{CS}}.\texttt{conf}_{\texttt{trust}}.\texttt{crt}_{\texttt{CA}}.pk, \texttt{crt.tbs}, \texttt{crt}.\sigma)$ ▷ mid/authenticate.go#L65
then return $\perp$

2 **if** $\neg\mathsf{Verify}(\texttt{crt}.pk, r, \sigma)$ ▷ /mid/service/mid/main.go#L286
then return $\perp$

3 $id_{\texttt{voter}} \leftarrow \texttt{crt}[\texttt{conf}_{\texttt{CS}}.\texttt{idType}]$ ▷ mid/main.go#L295

4 $vid \xleftarrow{r} \mathsf{Rand}(16)$ ▷ ticket.go#L83

5 $\texttt{idFields} \leftarrow \{\texttt{commonname} := \texttt{crt.commonname}, \texttt{serialnumber} := \texttt{crt.serialnumber}, \texttt{pnoee} := \texttt{crt.pnoee}\}$

6 $m \leftarrow \{\texttt{idFields}, vid\}$
▷ cookie.go#L54

7 **atomically** $\texttt{n}_{\texttt{used}} \leftarrow \texttt{conf}_{\texttt{CS}}.\texttt{n}_{\texttt{ticket}}$ and $\texttt{conf}_{\texttt{CS}}.\texttt{n}_{\texttt{ticket}} \leftarrow \texttt{conf}_{\texttt{CS}}.\texttt{n}_{\texttt{ticket}} + 1$

8 $c \leftarrow \mathsf{AEADEnc}(\texttt{conf}_{\texttt{CS}}.\texttt{k}_{\texttt{ticket}}, m, \texttt{n}_{\texttt{used}})$

9 $\texttt{authToken} \leftarrow \texttt{n}_{\texttt{used}} \| c$

10 **return** $\texttt{authToken}, id_{\texttt{voter}}$

OpenAuthToken decrypts the auth token using AEAD to reveal the plaintext data.

Algorithm 16: OpenAuthToken - Link to Code

Context:
- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`

Input:
- `authToken` : `Byte[]`

Output:
- $\{vid$: `String`, $id_{\texttt{voter}}$: `String`$\}$. A record containing vote ID and voter ID, or $\perp$ if opening or decoding fails.

Operation:

1 **if** `authToken` *cannot be split as* $\texttt{n}_{\texttt{used}} \| c$ *with* $|\texttt{n}_{\texttt{used}}| = 12$ **then return** $\perp$
2 $\texttt{n}_{\texttt{used}} \| c \leftarrow \texttt{authToken}$
3 $m \leftarrow \mathsf{AEADDec}(\texttt{conf}_{\texttt{CS}}.\texttt{k}_{\texttt{ticket}}, c, \texttt{n}_{\texttt{used}})$
4 **if** $m = \perp$ **then return** $\perp$
5 **if** m *is not a well-formed record* $\{\texttt{idFields}, vid\}$ **then return** $\perp$
6 $\{\texttt{idFields}, vid\} \leftarrow m$
7 $id_{\texttt{voter}} \leftarrow \texttt{idFields}[\texttt{conf}_{\texttt{CS}}.\texttt{idType}]$
8 **return** $\{vid, id_{\texttt{voter}}\}$

CreateDataToken generates a data token containing the Mobile-ID phone number so later certificate and signing requests do not rely on a fresh client-supplied phone number.

Algorithm 17: CreateDataToken - Link to Code

Context:
- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`

Input:
- `phone` : `String`

Output:
- `dataToken` : `Byte[]`

Operation:

▷ cookie.go#L54
1 **atomically** $\texttt{n}_{\texttt{used}} \leftarrow \texttt{conf}_{\texttt{CS}}.\texttt{n}_{\texttt{ticket}}$ and $\texttt{conf}_{\texttt{CS}}.\texttt{n}_{\texttt{ticket}} \leftarrow \texttt{conf}_{\texttt{CS}}.\texttt{n}_{\texttt{ticket}} + 1$
2 $c \leftarrow \mathsf{AEADEnc}(\texttt{conf}_{\texttt{CS}}.\texttt{k}_{\texttt{ticket}}, \texttt{phone}, \texttt{n}_{\texttt{used}})$
3 $\texttt{dataToken} \leftarrow \texttt{n}_{\texttt{used}} \| c$
4 **return** `dataToken`

OpenDataToken decrypts the data token using AEAD to recover the Mobile-ID phone number.

Algorithm 18: OpenDataToken - Link to Code

Context:
- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`

Input:
- `dataToken` : `Byte[]`

Output:
- `phone` : `String` if the token opens successfully, or $\perp$ otherwise

Operation:

1. **if** `dataToken` *cannot be split as* $\texttt{n}_{\texttt{used}} \| c$ *with* $|\texttt{n}_{\texttt{used}}| = 12$ **then return** $\perp$
2. $\texttt{n}_{\texttt{used}} \| c \leftarrow \texttt{dataToken}$
3. $\texttt{phone} \leftarrow \mathsf{AEADDec}(\texttt{conf}_{\texttt{CS}}.\texttt{k}_{\texttt{ticket}}, c, \texttt{n}_{\texttt{used}})$
4. **if** $\texttt{phone} = \perp$ **then return** $\perp$
5. **return** `phone`

4.3.1.8 Opaque VCD Helper

Estonian law prohibits disclosure of the VCD source code, so its exact implementation details are unknown. We infer from the protocol that the VCD must, at minimum, implement a function that constructs the voter's ballot BDOC.

CreateBDOC denotes this helper function, which constructs the BDOC from the voter's intent.

Algorithm 19: CreateBDOC

Context:
- $\texttt{conf}_{\texttt{VCD}}$: `AppConfig`

Input:
- $M_{\texttt{QC}}$: `Map<String, String>`. Maps election questions to voter-selected choices
- $\texttt{crt}_{\texttt{sign}}$: `X509Certificate`. Signer certificate returned for the authenticated voter

Output:
- BC : `BDOC`. Unsigned or partially constructed ballot container
- r : `Int`. Encryption randomness used for vote verification

Operation:

1. $(\texttt{M}_{\texttt{vote}}, r) \xleftarrow{r} \mathsf{EncryptChoices}(\texttt{conf}_{\texttt{VCD}}.P_{\texttt{EG}}, \texttt{conf}_{\texttt{VCD}}.\texttt{crt}_{\texttt{EG}}.pk, M_{\texttt{QC}})$
2. $\texttt{L}_{\texttt{ballot}} \leftarrow []$
3. **foreach** $(q, v) \in \texttt{M}_{\texttt{vote}}$ **do**
4. $\quad k \leftarrow \texttt{conf}_{\texttt{VCD}}.eid \,\|\, \text{“.”} \,\|\, q \,\|\, \text{“.ballot”}$
5. $\quad \texttt{L}_{\texttt{ballot}}.\mathsf{append}(\mathsf{BallotFile}(k, v))$
6. $BC \leftarrow \mathsf{BDOC}(\texttt{L}_{\texttt{ballot}}, \texttt{crt}_{\texttt{sign}})$
7. **return** BC, r

EncryptChoices denotes the VCD-side encryption helper. Given the election ElGamal parameters, an ElGamal public key, and a map of election questions to choices, it returns a map from questions to encoded vote ciphertexts $\texttt{M}_{\texttt{vote}}$: `Map<String, Byte[]>` and the randomness r used

for vote verification. Because the VCD source code is unavailable, this helper is modelled abstractly; later participants treat each payload as bytes and parse or validate it using their own ciphertext checks.

4.3.2 ConfirmBallot

Figure 8 shows the ConfirmBallot subprotocol, which covers everything that happens after the Collector Service (CS) receives a signed ballot from the vote-casting device (VCD), up to the point where the CS completes qualification, stores the registered ballot, and returns the qualification response to the voter. IVXV uses two qualification services during this phase, both run by the Registration Service (Reg):

- **TSP/TSPReg:** CS creates a timestamping request containing data derived from the ballot and sends it to the TSP service, which returns a signed timestamp token that cryptographically binds the request data and assigns an authoritative registration time. IVXV uses this timestamp as proof that a ballot was registered.
- **OCSP:** CS sends an OCSP request containing the voter/signer certificate and its issuing certificate embedded in the BDOC. The OCSP service returns a signed status response attesting to the validity of the signer certificate at the moment of registration.

The implementation details of the TSP and OCSP authorities are unknown except through their returned certificate-status and timestamp responses, and hence we treat the Reg participant as a black box.

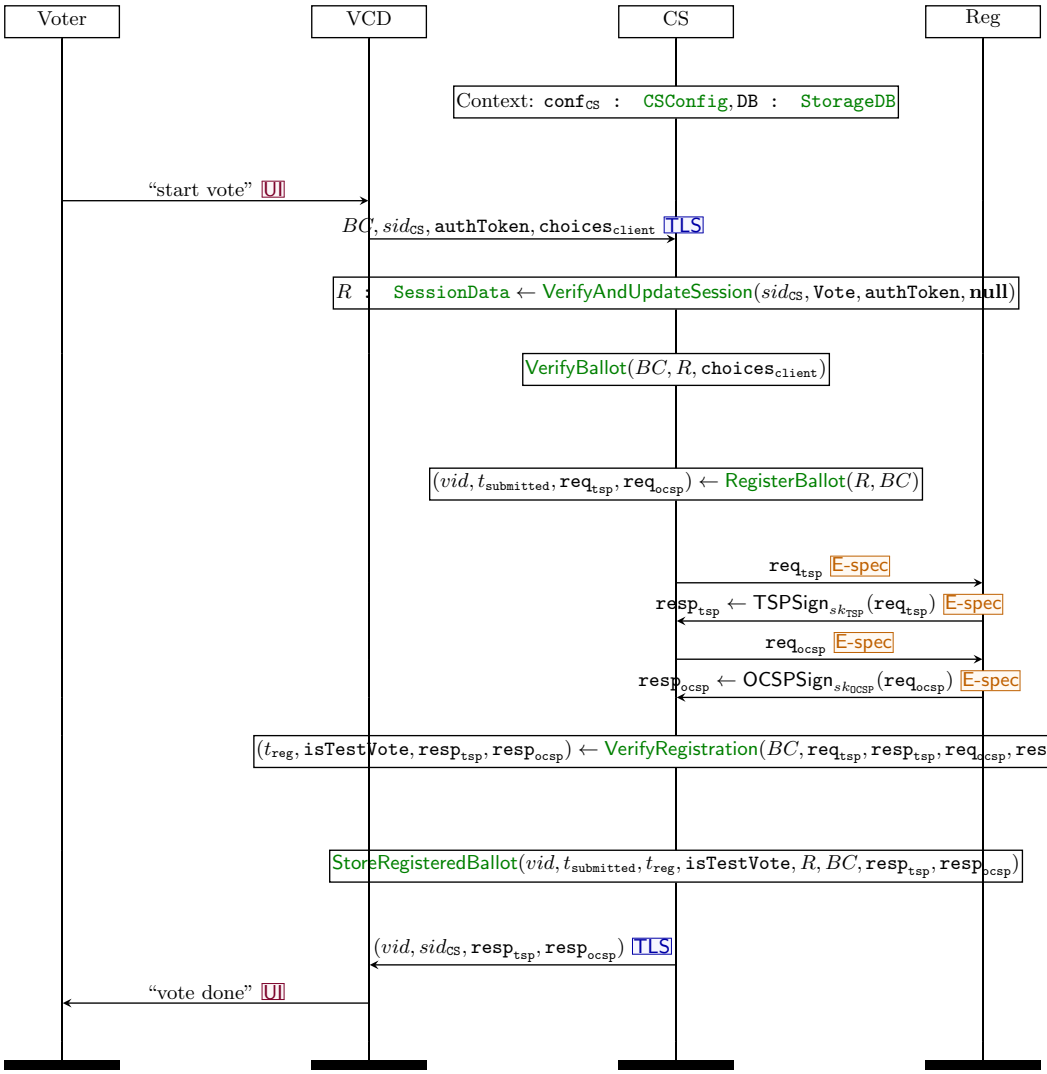

Figure 8: Voting phase. ConfirmBallot subprotocol

4.3.2.1 `VerifyBallot` performs the CS-side checks on a submitted ballot BDOC:

- validates the voter's certificate against the configured CA and verifies the voter's signature;
- checks that the BDOC contains the expected number of signatures;
- extracts the voter's identity from the signer certificate and uses it to check voter eligibility and choice-list correctness;
- checks the vote file names carried by the BDOC ballot-file list;
- checks ciphertext validity for the encrypted vote(s);
- checks that the voter identity matches the identity stored in the session data record.

Algorithm 20: VerifyBallot - Link to Code

Context:

- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`
- `DB` : `StorageDB`. Storage Database

Input:

- BC : `BDOC`. Signed ballot container
- R : `SessionData`
- $\texttt{choices}_{\texttt{client}}$: `CandidateList`. Candidate list provided by the vote casting device

Output: b : `Bool`: true if valid, false otherwise

Operation:

▷ /voting/main.go#L406

1 **if** $\neg$VerifyBDOC$(BC, \texttt{conf}_{\texttt{CS}}.\texttt{conf}_{\texttt{trust}}.\texttt{crt}_{\texttt{CA}})$ **then return** false

2 **if** $|BC.\text{L}_{\texttt{sig}}| \neq 1$ **then return** false ▷ /voting/main.go#L432

3 $\texttt{signer}$: `X509Certificate` $\leftarrow BC.\text{L}_{\texttt{sig}}[0].\texttt{crt}_{\texttt{signer}}$ ▷ /voting/main.go#L437

4 $id \leftarrow \texttt{signer}[\texttt{conf}_{\texttt{CS}}.\texttt{idType}]$

5 **if** $id =$ **null then return** false

6 **if** $\texttt{choices}_{\texttt{client}} \neq$ VoterChoices(id) **then return** false ▷ /voting/main.go#L469

7 $B' \leftarrow \emptyset$

8 **foreach** $bf \in BC.\text{L}_{\texttt{ballot}}$ **do** ▷ /voting/main.go#L482

9 $k \leftarrow bf.\texttt{filename}$

10 **if** $\exists q \in \texttt{conf}_{\texttt{CS}}.Q : k = \texttt{conf}_{\texttt{CS}}.eid \,\|\, \text{“.”} \,\|\, q \,\|\, \text{“.ballot”}$ **then** $B'.\text{put}(q, bf.v)$

11 **else return** false

12 **if** $|B'| \neq |Q|$ **then return** false

13 **foreach** $(q, v) \in B'$ **do** ▷ /voting/main.go#L147

14 **if** $\neg$IsValidCiphertextCS$(\texttt{conf}_{\texttt{CS}}.P_{\texttt{EG}}, v)$ **then return** false

15 **if** $R.id_{\texttt{voter}} \neq id$ **then return** false ▷ /voting/main.go#L160

16 **return** true

VerifyBDOC validates the voter's certificate against the configured CA and verifies the voter's signature.

Algorithm 21: VerifyBDOC - Link to Code, - Link to Code

Input:
- BC : `BDOC`
- $\texttt{crt}_{\texttt{CA}}$: `X509Certificate`: trusted CA

Output: b : `Bool`: true if verified, false otherwise

Operation:

1. $s \leftarrow BC.\texttt{L}_{\texttt{sig}}[0]$
2. $\texttt{crt}_{\texttt{voter}} \leftarrow s.\texttt{crt}_{\texttt{signer}}$
3. **if** $\neg\mathsf{Verify}(\texttt{crt}_{\texttt{CA}}.pk, \texttt{crt}_{\texttt{voter}}.\texttt{tbs}, \texttt{crt}_{\texttt{voter}}.\sigma)$ ▷ bdoc.go#L273
   then return false
4. **if** $\neg\mathsf{Verify}(\texttt{crt}_{\texttt{voter}}.pk, BC.\texttt{tbs}, s.\sigma)$ ▷ bdoc.go#L304
   then return false
5. $BC.\texttt{L}_{\texttt{sig}}[0].\texttt{crt}_{\texttt{issuer}} \leftarrow \texttt{crt}_{\texttt{CA}}$
6. **return** true

IsValidCiphertextCS checks ciphertext correctness: it parses and verifies the ASN.1-encoded vote ciphertext(s). DecodeASN1CiphertextCS is CS's helper function for interpreting ASN.1 encoded vote ciphtext, which is omitted at the abstarction level of this specification.

Algorithm 22: IsValidCiphertextCS - Link to Code, - Link to Code

Input:

$P_{\texttt{EG}}$: `ElGamalParameters`. ElGamal parameters

v : `Byte[]`. Vote ciphertext, with ASN.1 encoding

Output:

b : `Bool`: true if valid, false otherwise

Operation:

$c \leftarrow$ DecodeASN1CiphertextCS$(v, P_{\texttt{EG}})$

if $c = \perp$ **then return** false

$(c_0, c_1) \leftarrow c$

if $P_{\texttt{EG}}$.`oid` $\neq$ `oid` **then return** false

if *n has extra children beyond the already parsed structure* **then return** false

▷ /core/crypto/elgamal/ciphertext.go#L72

$p, q \leftarrow P_{\texttt{EG}}$

for $i \leftarrow 0$ *to* 1 ▷ /core/crypto/elgamal/ciphertext.go#L92

do

  if $\neg(c_i^q \equiv 1 \bmod p)$ **then return** false

  if $\neg(0 < c_i < p)$ **then return** false

  if $\neg(c_i^{(p-1)/2} \equiv 1 \bmod p)$ **then return** false

return true

4.3.2.2 `RegisterBallot` generates a unique vote identifier (vid) for the ballot and constructs registration requests for the TSP (Timestamping) and OCSP (Online Certificate Status Protocol) services after CS verifies a voting request. In this model, both services are run by Reg.

The TSP request payload contains a MessageImprint m and a nonce field n:

$$m = (\mathsf{alg}_m, \mathsf{Digest}^{\mathsf{alg}_m}(s)),\ n = (\mathsf{alg}_n, \mathsf{Sign}^{\mathsf{alg}_n}_{sk_{\mathsf{CS}}}(s)).$$

Here alg_m is the digest algorithm used in the TSP message imprint, while alg_n is the signature algorithm used for CS's nonce signature. For example, alg_n = `sha256WithRSAEncryption` means that the nonce signature is computed over the SHA-256 digest of s.

The OCSP request contains a OCSPCertID derived from the signer certificate and issuer certificate fields already present in the ballot signature. In this model, the identifier has form $\{\texttt{alg}, \texttt{id}\}$, where `alg` = "SHA-1" and `id` abstracts the implementation-level OCSP certificate identifier.

Algorithm 23: RegisterBallot - Link to Code

Context:

- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`
- `DB` : `StorageDB`. Storage Database

Input:

- $R := \{id_{\texttt{voter}}, vid, \dots\}$: `SessionData`
- BC : `BDOC`. signed ballot container

Output:

- vid : `String`. Vote ID
- $t_{\texttt{submitted}}$: `Time`. Submission time
- $\texttt{req}_{\texttt{tsp}}$: `TSPReq`. TSP request
- $\texttt{req}_{\texttt{ocsp}}$: `OCSPReq`. OCSP request

Operation:

1 $t_{\texttt{submitted}} = \mathsf{Now}()$ ▷ /voting/main.go#L127

▷ /voting/main.go#L171

2 $vid \leftarrow R.vid$

3 **if** $vid =$ **null then**

  $vid \xleftarrow{r} \mathsf{Rand}(16)$

▷ /voting/main.go#L218

4 $h \leftarrow \mathsf{Digest}(\text{“SHA-256”}, BC.\texttt{L}_{\texttt{sig}}[0].\sigma)$

5 $\texttt{m}_{\texttt{tsp}} \leftarrow \mathsf{MessageImprint}(\text{“SHA-256”}, h)$

6 $\sigma \leftarrow \mathsf{Sign}(\texttt{conf}_{\texttt{CS}}.sk_{\texttt{tspClient}}, h)$

7 $\texttt{n}_{\texttt{tsp}} \leftarrow \mathsf{TSPNonce}(\text{“SHA-256”}, \sigma)$ ▷ /q11n/tsp/tsp.go#L120

8 $\texttt{req}_{\texttt{tsp}} \leftarrow \mathsf{TSPReq}(\texttt{m}_{\texttt{tsp}}, \texttt{n}_{\texttt{tsp}})$ ▷ /tsp/tsp.go#L224

9 $\texttt{crt}_{\texttt{voter}} \leftarrow BC.\texttt{L}_{\texttt{sig}}[0].\texttt{crt}_{\texttt{signer}}$

10 $\texttt{crt}_{\texttt{issuer}} \leftarrow BC.\texttt{L}_{\texttt{sig}}[0].\texttt{crt}_{\texttt{issuer}}$

11 $\texttt{h}_{\texttt{ocsp}} \leftarrow \mathsf{Digest}(\text{“SHA-1”}, \texttt{crt}_{\texttt{voter}}.\texttt{issuer} \| \texttt{crt}_{\texttt{issuer}}.pk \| \texttt{crt}_{\texttt{voter}}.\texttt{serialnumber})$

12 $\texttt{certID} \leftarrow \mathsf{OCSPCertID}(\text{“SHA-1”}, \texttt{h}_{\texttt{ocsp}})$

13 $\texttt{req}_{\texttt{ocsp}} \leftarrow \mathsf{OCSPReq}(\texttt{certID})$ ▷ /q11n/ocsp/ocsp.go#L49

14 **return** $(vid, t_{\texttt{submitted}}, \texttt{req}_{\texttt{tsp}}, \texttt{req}_{\texttt{ocsp}})$

4.3.2.3 `VerifyRegistration` verifies the qualification responses after CS sends the requests and receives the corresponding responses. If both responses verify, CS derives the ballot's registration time and whether the ballot should be treated as a test vote.

Algorithm 24: VerifyRegistration - Link to Code

Context:

- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`

Input:

- BC : `BDOC`. Signed BDOC container
- $\texttt{req}_{\texttt{tsp}}$: `TSPReq`. TSP request
- $\texttt{resp}_{\texttt{tsp}}$: `TSPResponse` Registration response from TSP service
- $\texttt{req}_{\texttt{ocsp}}$: `OCSPReq`. OCSP request
- $\texttt{resp}_{\texttt{ocsp}}$: `OCSPResponse`. Registration response from OCSP service

Output:

registration result
$(t_{\texttt{reg}} : \texttt{Time}, \texttt{isTestVote} : \texttt{Bool}, \texttt{resp}_{\texttt{tsp}} : \texttt{TSPResponse}, \texttt{resp}_{\texttt{ocsp}} : \texttt{OCSPResponse})$
if verified successfully, $\perp$ otherwise

Operation:

1. s : `XAdESSignature` $\leftarrow BC.\texttt{L}_{\texttt{sig}}[0]$
2. $sig_{\texttt{voter}} \leftarrow s.\sigma$
3. $\texttt{crt}_{\texttt{issuer}} \leftarrow s.\texttt{crt}_{\texttt{issuer}}$
4. $t_{\texttt{reg}} \leftarrow 0$
5. **if** $\neg$VerifyTSPCS$(\texttt{req}_{\texttt{tsp}}, \texttt{resp}_{\texttt{tsp}}, sig_{\texttt{voter}})$ **then return** $\perp$
6. $t_{\texttt{reg}} \leftarrow \texttt{resp}_{\texttt{tsp}}.t$
7. **if** $\texttt{resp}_{\texttt{ocsp}}.\texttt{status} \neq \texttt{true}$ **then return** $\perp$
8. **if** $\neg$VerifyOCSPCS$(\texttt{req}_{\texttt{ocsp}}, \texttt{crt}_{\texttt{issuer}}, \texttt{resp}_{\texttt{ocsp}})$ **then return** $\perp$
9. **if** $\texttt{resp}_{\texttt{ocsp}}.t < t_{\texttt{reg}}$ **then return** $\perp$
10. $\texttt{isTestVote} : \texttt{Bool} = t_{\texttt{reg}} < \texttt{conf}_{\texttt{CS}}.\texttt{L}_{\texttt{time}}.t_{\texttt{EStart}}$
11. **return** $(t_{\texttt{reg}}, \texttt{isTestVote}, \texttt{resp}_{\texttt{tsp}}, \texttt{resp}_{\texttt{ocsp}})$

VerifyTSPCS verifies the TSP response returned by Reg. The response contains the same message imprint m and nonce n as the corresponding TSP request. CS first recomputes the digest of the voter's signature s using the digest algorithm recorded in $m.\texttt{alg}$, and checks that the result matches $m.h$. It then checks that the nonce n matches the nonce in the original request, and finally verifies the TSP signature over the response:

Algorithm 25: VerifyTSPCS - Link to Code

Context:

- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`

Input:

- $\texttt{req}_{\texttt{tsp}}$: `TSPReq`
- r : `TSPResponse` Registration response from TSP service
- s : `Byte[]`. Voter signature value

Output: b : `Bool`: true if valid, false otherwise

Operation:

$m, n, t, \sigma, \texttt{crt} \leftarrow r.m, r.n, r.t, r.\sigma, r.\texttt{crt}$

▷ /tsp/tsp.go#L352

if $m.h \neq \mathsf{Digest}(m.\texttt{alg}, s)$ **then return** false

if $n \neq \texttt{req}_{\texttt{tsp}}.n$ **then return** false

▷ /tsp/tsp.go#L408

if $\neg(-60s \leq t - \mathsf{Now}() \leq 2s)$ **then return** false

▷ /tsp/cms.go#L83

if $\texttt{crt} \neq \texttt{conf}_{\texttt{CS}}.\texttt{conf}_{\texttt{trust}}.\texttt{crt}_{\texttt{TSP}}$ **then return** false

$m' \leftarrow m \| n \| t$

if $\neg\mathsf{Verify}(\texttt{crt}.pk, m', \sigma)$ **then return** false

return true

VerifyOCSPCS verifies the OCSP response returned by Reg. An OCSP request contains a OCSPCertID `certID`, modelled as $\{\texttt{alg}, \texttt{id}\}$. When CS creates the request, it sets `alg` = "SHA-1" and derives `id` from the modelled signer-certificate and issuer-certificate fields:

$$\texttt{id} = \mathsf{Digest}(\text{"SHA-1"}, \texttt{crt}_{\texttt{voter}}.\texttt{issuer} \| \texttt{crt}_{\texttt{issuer}}.pk \| \texttt{crt}_{\texttt{voter}}.\texttt{serialnumber}).$$

This abstracts the implementation-level OCSP CertID fields, namely the hash algorithm, issuer-name hash, issuer-key hash, and serial number, without adding those lower-level fields to X509Certificate. VerifyOCSPCS checks that the requested certificate identifier matches the response, validates the responder certificate against the issuer certificate carried in the ballot signature, and verifies the signature. The status of an OCSP response is checked earlier in VerifyRegistration.

Algorithm 26: VerifyOCSPCS - Link to Code

Context:
- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`

Input:
- $\texttt{req}_{\texttt{ocsp}}$: `OCSPReq`. OCSP request
- $\texttt{crt}_{\texttt{issuer}}$: `X509Certificate`. Issuer certificate of the ballot's signature, used as the trusted CA
- r : `OCSPResponse`. Registration response from OCSP service

Output: b : `Bool`: true if valid, false otherwise

Operation:

1 $r := \{\texttt{certID}, \texttt{status}, t, \sigma, \texttt{crt}_{\texttt{OCSP}}\}$
2 **if** $\texttt{certID} \neq \texttt{req}_{\texttt{ocsp}}.\texttt{certID}$ ▷ /ocsp/ocsp.go#L541
then return false
3 **if** $\neg\mathsf{Verify}(\texttt{crt}_{\texttt{issuer}}.pk, \texttt{crt}_{\texttt{OCSP}}.\texttt{tbs}, \texttt{crt}_{\texttt{OCSP}}.\sigma)$ ▷ /ocsp/ocsp.go#L566
then return false
▷ /ocsp/ocsp.go#L579
4 $m \leftarrow \texttt{certID} \| \texttt{status} \| t$
5 **if** $\neg\mathsf{Verify}(\texttt{crt}_{\texttt{OCSP}}.pk, m, \sigma)$ **then return** false
6 **return** true

4.3.2.4 `StoreRegisteredBallot` stores the registered ballot after successful registration as a single record containing the original ballot, both qualification responses, and the relevant timing metadata. The latest vote of a voter is derived from the stored registered ballots.

Algorithm 27: StoreRegisteredBallot - Link to Code

Context:

- `DB` : `StorageDB`. Storage Database

Input:

- vid : `String`. Vote ID
- $t_{\texttt{submitted}}$: `Time`. Submission time
- $t_{\texttt{reg}}$: `Time`. Registration time
- `isTestVote` : `Bool`
- R : `SessionData`
- BC : `BDOC`. Signed ballot container
- $\texttt{resp}_{\texttt{tsp}}$: `TSPResponse`
- $\texttt{resp}_{\texttt{ocsp}}$: `OCSPResponse`

Output:

- b : `Bool`: true if stored, false otherwise

Operation:

1. $rb \leftarrow \mathsf{RegisteredBallot}(vid, t_{\texttt{submitted}}, t_{\texttt{reg}}, BC, R.id_{\texttt{voter}}, 0, \texttt{resp}_{\texttt{tsp}}, \texttt{resp}_{\texttt{ocsp}}, \texttt{isTestVote})$
2. $\texttt{DB.BallotStore.put}(vid, rb)$
3. **return** true

4.3.3 VerifyCast

The VerifyCast subprotocol corresponds to the self-verification of a submitted ballot. It covers the steps from the Collector Service (CS) returning the registration result to the Vote-Casting Device (VCD), through to the Vote-Verification Device (VVD) decrypting the ballot for the voter to check.

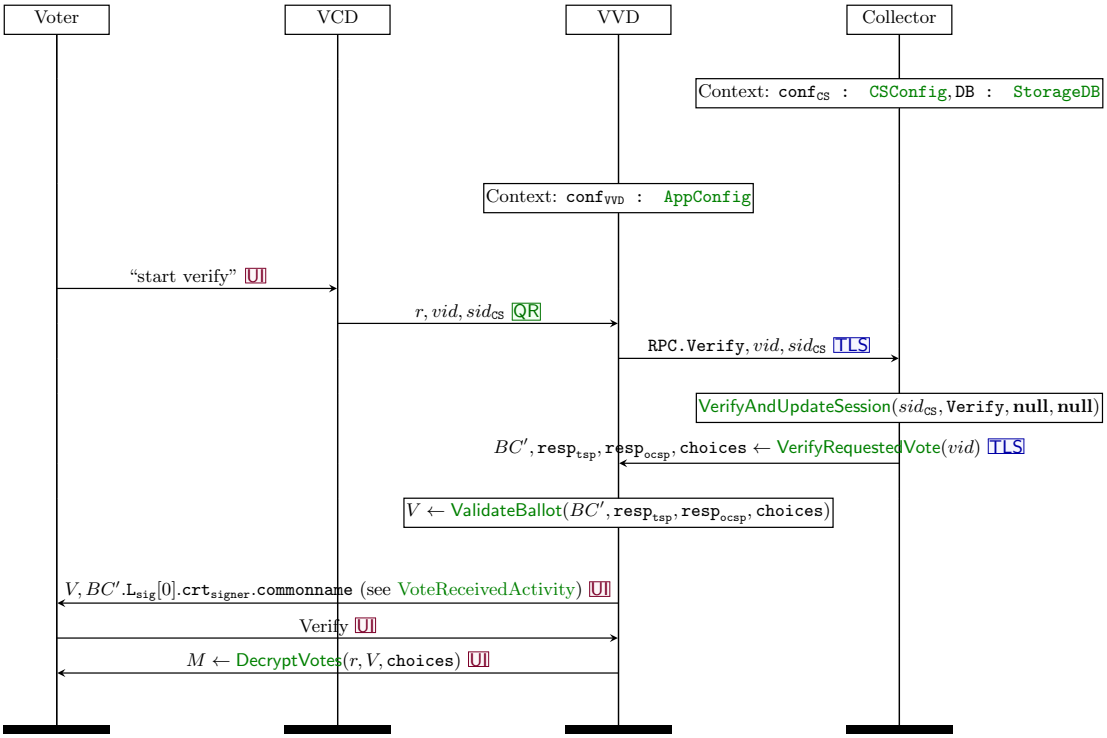

Figure 9: Voting phase. VerifyCast subprotocol

4.3.3.1 VerifyRequestedVote checks a vote-verification request received by CS from VVD, including the session validity and whether the requested vote ID matches the voter's latest vote in the storage database.

Algorithm 28: VerifyRequestedVote - Link to Code

Context:

- $\texttt{conf}_{\texttt{CS}}$: CSConfig
- DB : StorageDB. Storage Database

Input:

- vid : String

Output:

If verifies, return the following record containing the requested vote, qualification responses, and choice list; otherwise return $\perp$:

- BC : BDOC
- $\texttt{resp}_{\texttt{tsp}}$: TSPResponse
- $\texttt{resp}_{\texttt{ocsp}}$: OCSPResponse
- choices : CandidateList

Operation:

1 $res \leftarrow \mathsf{GetVerificationStats}(vid)$
2 **if** $res =$ **null then return** $\perp$
3 $\{cnt, t_{\texttt{submitted}}, \texttt{latest}, \texttt{choices}\} \leftarrow res$
4 **if** $cnt \geq \texttt{conf}_{\texttt{CS}}.\texttt{cnt}_{\texttt{verif}}$ **then return** $\perp$
5 $t_{\texttt{now}} \leftarrow \mathsf{Now}()$
6 **if** $t_{\texttt{now}} > \texttt{conf}_{\texttt{CS}}.t_{\texttt{ver}} + t_{\texttt{submitted}}$ **then return** $\perp$
7 **if** $\neg\texttt{latest} \wedge \texttt{conf}_{\texttt{CS}}.\texttt{latestonly}$ **then return** $\perp$
8 $\texttt{voteData} \leftarrow \mathsf{GetVerification}(vid, cnt)$
9 **if** $\texttt{voteData} =$ **null then return** $\perp$
10 $(BC, \texttt{resp}_{\texttt{tsp}}, \texttt{resp}_{\texttt{ocsp}}) \leftarrow \texttt{voteData}$
11 **return** $BC, \texttt{resp}_{\texttt{tsp}}, \texttt{resp}_{\texttt{ocsp}}, \texttt{choices}$

GetVerificationStats retrieves the registered ballot record from the storage database using the requested vid as the key. It also computes a flag `latest` indicating whether the requested ballot is the voter's latest registered ballot.

Algorithm 29: GetVerificationStats - Link to Code

Context:

- $\texttt{conf}_{\texttt{CS}}$: CSConfig
- DB : StorageDB. Storage Database

Input:

- vid : String

Output:

- $R_{\texttt{stats}}$: Record if a vote record is found, **null** otherwise:
- cnt : Int
- $t_{\texttt{submitted}}$: Time
- latest : Bool
- choices : CandidateList

Operation:

1. rb : RegisteredBallot $\leftarrow$ DB.BallotStore$[vid]$
2. **if** $rb =$ **null then return null**
3. $\{vid, t_{\texttt{submitted}}, t_{\texttt{reg}}, v, id_{\texttt{voter}}, cnt, \texttt{resp}_{\texttt{tsp}}, \texttt{resp}_{\texttt{ocsp}}, \texttt{isTestVote}\} \leftarrow rb$
4. choices $\leftarrow$ VoterChoices$(id_{\texttt{voter}})$
5. $\texttt{rb}_{\texttt{latest}}$: RegisteredBallot $\leftarrow$ LatestRegisteredBallot(DB.BallotStore, $id_{\texttt{voter}}$)
6. latest : Bool $\leftarrow \texttt{rb}_{\texttt{latest}}.vid = vid$
7. **return** $\{cnt, t_{\texttt{submitted}}, \texttt{latest}, \texttt{choices}\}$

LatestRegisteredBallot returns the registered ballot with the greatest registration time among ballots attributed to the requested voter identifier.

Algorithm 30: LatestRegisteredBallot

Input:
- `BallotStore` : `Map<String, RegisteredBallot>`
- $id_{\texttt{voter}}$: `String`

Output:
- rb : `RegisteredBallot` determined by $t_{\texttt{reg}}$ for $id_{\texttt{voter}}$, **null** if none exists

Operation:

1 $\texttt{rb}_{\texttt{latest}} \leftarrow$ **null**
2 **foreach** $vid \in \mathsf{Keys}(\texttt{BallotStore})$ **do**
3 rb : `RegisteredBallot` $\leftarrow \texttt{BallotStore}[vid]$
4 **if** $rb.id_{\texttt{voter}} = id_{\texttt{voter}} \wedge (\texttt{rb}_{\texttt{latest}} = \textbf{null} \vee rb.t_{\texttt{reg}} > \texttt{rb}_{\texttt{latest}}.t_{\texttt{reg}})$ **then**
5 $\texttt{rb}_{\texttt{latest}} \leftarrow rb$
6 **return** $\texttt{rb}_{\texttt{latest}}$

GetVerification retrieves the registered ballot and its qualification responses from storage, increments the ballot's verification counter in the storage, and returns the data needed by VVD for self-verification.

Algorithm 31: GetVerification - Link to Code

Context:
- $\texttt{conf}_{\texttt{CS}}$: `CSConfig`
- `DB` : `StorageDB`. Storage Database

Input:
- vid : `String`, cnt : `Int`

Output:
- BC : `BDOC`, $\texttt{resp}_{\texttt{tsp}}$: `TSPResponse`, $\texttt{resp}_{\texttt{ocsp}}$: `OCSPResponse` if a finalised vote is found, **null** otherwise

Operation:

1 rb : `RegisteredBallot` $\leftarrow \texttt{DB.BallotStore}[vid]$
2 **if** $rb =$ **null then return null**
3 $\{vid, t_{\texttt{submitted}}, t_{\texttt{reg}}, v, id_{\texttt{voter}}, cnt, \texttt{resp}_{\texttt{tsp}}, \texttt{resp}_{\texttt{ocsp}}, \texttt{isTestVote}\} \leftarrow rb$
4 $\texttt{DB.BallotStore.put}(vid, \{vid, t_{\texttt{submitted}}, t_{\texttt{reg}}, v, id_{\texttt{voter}}, cnt + 1, \texttt{resp}_{\texttt{tsp}}, \texttt{resp}_{\texttt{ocsp}}, \texttt{isTestVote}\})$
5 **return** $v, \texttt{resp}_{\texttt{tsp}}, \texttt{resp}_{\texttt{ocsp}}$

4.3.3.2 ValidateBallot validates the ballot requested by VVD from CS using the session and vote identifiers supplied by VCD. The VVD verifies the integrity and authenticity of the BDOC and the validity of the accompanying qualification responses. For a successfully validated ballot, the VVD returns the list of encrypted votes contained in the BDOC.

The VVD's BDOC verification checks that all digest values match their corresponding data objects (not modelled at the report's abstraction level), verifies that the signer's certificate is issued by a trusted authority, and confirms the correctness of the digital signature using the public key embedded in the BDOC. We do not introduce a separate algorithm for it; instead, we reuse VerifyBDOC, which omits the digest-value checks at our level of abstraction.

Algorithm 32: ValidateBallot - Link to Code

Context:

- $\texttt{conf}_{\texttt{VVD}}$: `AppConfig`

Input:

- BC : `BDOC`
- $\texttt{resp}_{\texttt{tsp}}$: `TSPResponse`
- $\texttt{resp}_{\texttt{ocsp}}$: `OCSPResponse`
- `choices` : `CandidateList`

Output:

- V : `List<Byte[]>`

Operation:

1. $V \leftarrow []$
2. **if** `choices` $=$ **null then return** $\perp$
3. **if** $\neg\mathsf{VerifyBDOC}(BC, \texttt{conf}_{\texttt{VVD}}.\texttt{conf}_{\texttt{trust}}.\texttt{crt}_{\texttt{CA}})$ **then return** $\perp$
4. **if** $\neg\mathsf{VerifyTSPVVD}(BC, \texttt{resp}_{\texttt{tsp}})$ **then return** $\perp$
5. **if** $\neg\mathsf{VerifyOCSPVVD}(BC, \texttt{resp}_{\texttt{ocsp}})$ **then return** $\perp$
6. **foreach** $bf \in BC.\texttt{L}_{\texttt{ballot}}$ **do**
7. $\quad V.\mathsf{append}(bf.v)$
8. **return** V

VerifyTSPVVD verifies the TSP response. The VVD uses the two algorithm identifiers independently: it checks that the message imprint value $m.h$ matches s under alg_m, and that the nonce signature $n.\sigma$ verifies under alg_n using the TSP client public key in its local config:

$$\mathsf{Digest}^{\mathsf{alg}_m}(s) = m.h \wedge \mathsf{Verify}^{\mathsf{alg}_n}_{pk_{\mathsf{CS}}}(s, n.\sigma).$$

Algorithm 33: VerifyTSPVVD - Link to Code - Link to Code

Context:

$\texttt{conf}_{\texttt{VVD}}$: `AppConfig`

Input:

BC : `BDOC`

$\texttt{resp}_{\texttt{tsp}}$: `TSPResponse` Registration response from TSP service

Output: b : `Bool`: true if verified, false otherwise

Operation:

1 $m, n, t, \sigma, \texttt{crt} \leftarrow \texttt{resp}_{\texttt{tsp}}.m, \texttt{resp}_{\texttt{tsp}}.n, \texttt{resp}_{\texttt{tsp}}.t, \texttt{resp}_{\texttt{tsp}}.\sigma, \texttt{resp}_{\texttt{tsp}}.\texttt{crt}$

2 **if** $\texttt{crt} \neq \texttt{conf}_{\texttt{VVD}}.\texttt{crt}_{\texttt{TSP}}$ **then return** false

3 $s \leftarrow BC.\texttt{L}_{\texttt{sig}}[0].\sigma$

4 **if** $m.h \neq \mathsf{Digest}(m.\texttt{alg}, s)$ **then return** false

5 $pk_{\texttt{tspClient}} \leftarrow \texttt{conf}_{\texttt{VVD}}.\texttt{crt}_{\texttt{tspClient}}.pk$

6 $n' \leftarrow \mathsf{Digest}(n.\texttt{alg}, s)$

7 **if** $\neg\mathsf{Verify}(pk_{\texttt{tspClient}}, n', n.\sigma)$ **then return** false

8 $m' \leftarrow m \| n \| t$

9 **if** $\neg\mathsf{Verify}(\texttt{crt}.pk, m', \sigma)$ **then return** false

10 **return** true

VerifyOCSPVVD checks the OCSP response status, verifies the correctness of the OCSP responder's signature, and confirms that the certificate identifier in the response matches the identifier reconstructed from the downloaded ballot.

Algorithm 34: VerifyOCSPVVD - Link to Code

Context:

$\texttt{conf}_{\texttt{VVD}}$: AppConfig

Input:

BC : BDOC

r : OCSPResponse

Output: b : Bool: true if verified, false otherwise

Operation:

1 $\texttt{certID}, \texttt{status}, t, \sigma \leftarrow r.\texttt{certID}, r.\texttt{status}, r.t, r.\sigma$

2 $\texttt{crt}_{\texttt{OCSP}} \leftarrow \texttt{conf}_{\texttt{VVD}}.\texttt{crt}_{\texttt{OCSP}}$

3 $\texttt{crt}_{\texttt{voter}} \leftarrow BC.\text{L}_{\texttt{sig}}[0].\texttt{crt}_{\texttt{signer}}$

4 $\texttt{crt}_{\texttt{issuer}} \leftarrow BC.\text{L}_{\texttt{sig}}[0].\texttt{crt}_{\texttt{issuer}}$

5 $\texttt{h}_{\texttt{ocsp}} \leftarrow \text{Digest}(\text{“SHA-1”}, \texttt{crt}_{\texttt{voter}}.\texttt{issuer} \| \texttt{crt}_{\texttt{issuer}}.pk \| \texttt{crt}_{\texttt{voter}}.\texttt{serialnumber})$

6 $\texttt{certID}_{\texttt{expected}} \leftarrow \text{OCSPCertID}(\text{“SHA-1”}, \texttt{h}_{\texttt{ocsp}})$

7 $m \leftarrow \texttt{certID} \| \texttt{status} \| t$

8 **if** $\neg\text{Verify}(\texttt{crt}_{\texttt{OCSP}}.pk, m, \sigma)$ **then return** false

9 **if** $r.\texttt{status} \neq$ true **then return** false

10 **if** $\texttt{certID} \neq \texttt{certID}_{\texttt{expected}}$ **then return** false

11 **return** true

4.3.3.3 DecryptVotes decrypts the encrypted vote(s) extracted from a successfully validated BDOC container. Before decrypting each vote, VVD verifies that the blinding component c_0 of the ElGamal ciphertext corresponds to the randomness r provided by VCD, i.e., checks that $c_0 \stackrel{?}{=} g^r$. It then decrypts each ciphertext and verifies the correctness of the plaintext vote format and choice. If all checks succeed, the VVD returns a list of decrypted votes for voter verification.

Algorithm 35: DecryptVotes - Link to Code

Context:

- $\texttt{conf}_{\texttt{VVD}}$: `AppConfig`

Input:

- r : `Int`
- V : `List<Byte[]>`
- `choices` : `CandidateList`

Output:

- M : `List<String>` (plaintext choices) if decryption successful, $\perp$ otherwise

Operation:

1 $M \leftarrow []$
2 $P_{\texttt{EG}} \leftarrow \texttt{conf}_{\texttt{VVD}}.P_{\texttt{EG}}$
3 **foreach** $v \in V$ **do**
4 $v, ok \leftarrow \mathsf{IsValidCiphertextVVD}(\texttt{conf}_{\texttt{VVD}}.P_{\texttt{EG}}, r, v)$
5 **if** $\neg ok$ **then return** $\perp$
6 $(c_0, c_1) \leftarrow v$
7 $e \leftarrow \frac{c_1}{c_0^r}$
8 $m \leftarrow \mathsf{DecodeModPElement}(P_{\texttt{EG}}, e)$
9 **if** $|m| = 0$ **then**
10 M.append(`NO_CHOICE`)
continue
11 `cand` $\leftarrow$ `choices`$[m]$
12 **if** `cand` $=$ **null then return** $\perp$
13 M.append(`cand.number`$\|$`cand.party`$\|$`cand.name`)
14 **return** M

IsValidCiphertextVVD parses and verifies the ASN.1-encoded vote ciphertext. In our model, this is done similarly to IsValidCiphertextCS, except that the VVD additionally verifies the randomness binding $c_0 = g^r$.

Algorithm 36: IsValidCiphertextVVD - Link to Code

Input:

- P_{EG} : `ElGamalParameters`. ElGamal parameters
- r : `Int`. Randomness provided by VCD
- v : `Byte[]`. Vote ciphertext, with ASN.1 encoding

Output:

- c : `Tuple<Int, Int>` $:= (c_0, c_1)$ or **null**
- ok : `Bool`: true if valid, false otherwise

Operation:

1 $c \leftarrow$ DecodeASN1CiphertextVVD(v, P_{EG})
2 **if** $c = \perp$ **then return null**, false
3 $(c_0, c_1) \leftarrow c$
4 $g, p, q \leftarrow P_{\text{EG}}.g, P_{\text{EG}}.p, P_{\text{EG}}.q$
5 **for** $i \leftarrow 0$ *to* 1 ▷ `ModPGroup.java#L86`
do
6 **if** $\neg(c_i^q \equiv 1 \bmod p)$ **then return null**, false
7 **if** $\neg(0 < c_i < p)$ **then return null**, false
8 **if** $\neg(c_i^{(p-1)/2} \equiv 1 \bmod p)$ **then return null**, false
9 **if** $c_0 \neq g^r$ ▷ `ElGamalPublicKey.java#L86`
then return null, false
10 **return** c, true

DecodeModPElement decodes a ModP group element into a plaintext string.

Algorithm 37: DecodeModPElement - Link to Code

Input:
- $P_{\texttt{EG}}$: `ElGamalParameters`.
- e : `Int` $\in \mathbb{Z}_p^*$

Output: m : `String` if valid, $\perp$ otherwise

Operation:

1 **if** $e > P_{\texttt{EG}}.q$ **then**
  $e \leftarrow P_{\texttt{EG}}.p - e$
2 $m \leftarrow \mathsf{String}(\mathsf{PKCSDecode}(\mathsf{Bytes}(e)))$
3 **if** $m =$ **null then return** $\perp$
4 **return** m

4.3.4 ExportVotingData

The ExportVotingData subprotocol bridges the voting-phase output to the tally-phase input. CS exports ballots and embedded registration responses from its storage and passes them to EA. Reg passes the registration request log to EA.

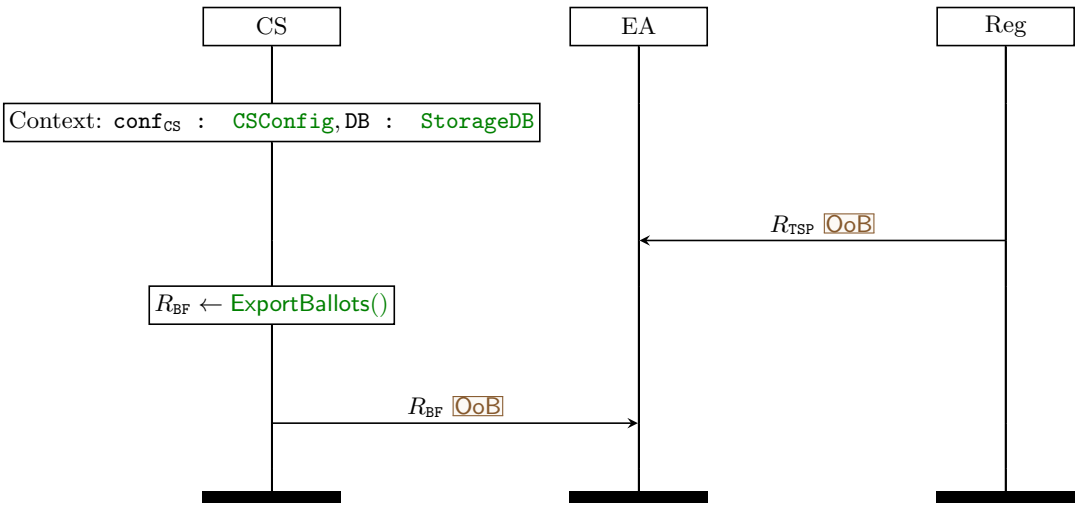

Figure 10: Voting phase. ExportVotingData subprotocol

4.3.4.1 ExportBallots exports a list of ballot records.

Algorithm 38: ExportBallots - Link to Code

Context:

`DB` : `StorageDB`. Storage Database

Output:

$R_{\texttt{BF}}$: `List<BallotFileRecord>`. Exported votes as records

Operation:

$R_{\texttt{BF}} \leftarrow []$
foreach $vid \in \mathsf{Keys}(\texttt{DB.BallotStore})$ **do**
  $rb \leftarrow \texttt{DB.BallotStore}[vid]$
  $id_{\texttt{ballot}}$: `String` $\leftarrow \mathsf{FormatTime}(rb.t_{\texttt{submitted}})$
  $ref \leftarrow \mathsf{BallotRef}(rb.id_{\texttt{voter}}, id_{\texttt{ballot}})$
  $r \leftarrow \mathsf{BallotFileRecord}(ref, rb.\texttt{resp}_{\texttt{tsp}}, rb.\texttt{resp}_{\texttt{ocsp}}, rb.v)$
  $R_{\texttt{BF}}.\mathsf{Append}(r)$
return $R_{\texttt{BF}}$

4.4 Tally Phase

Table 6 shows the subprotocols and algorithms involved in the tally phase.

Subprotocol / Algorithm	Actor
ProcessBallotbox	
CheckTool	PA
FilterBallotbox	PA
AnonymizeBallotbox	PA
MixBallotbox	
BallotboxAsBT	Mixer
Mix	Mixer
BTAsBallotbox	Mixer
DecryptAndTally	
DecryptAndTally	Key App

Table 6: Tally Phase Subprotocols

4.4.1 ProcessBallotbox

The ProcessBallotbox subprotocol marks the start of the tally phase. During this subprotocol, the Processor Application (PA) receives and processes input files from the Election Authority (EA). These inputs correspond to the files exported by the Collector Service (CS) and the Registration Service (Reg) during the ExportVotingData subprotocol of the voting phase, together with the revocation list supplied by EA.

PA parses the received ballotbox from CS ($R_{\texttt{BF}}$), the registration log from Reg ($R_{\texttt{TSP}}$), and the revocation list from EA (`revokeList`). It then validates the ballots and registration data, removes duplicates, applies revocations, and outputs an anonymised ballotbox for mixing.

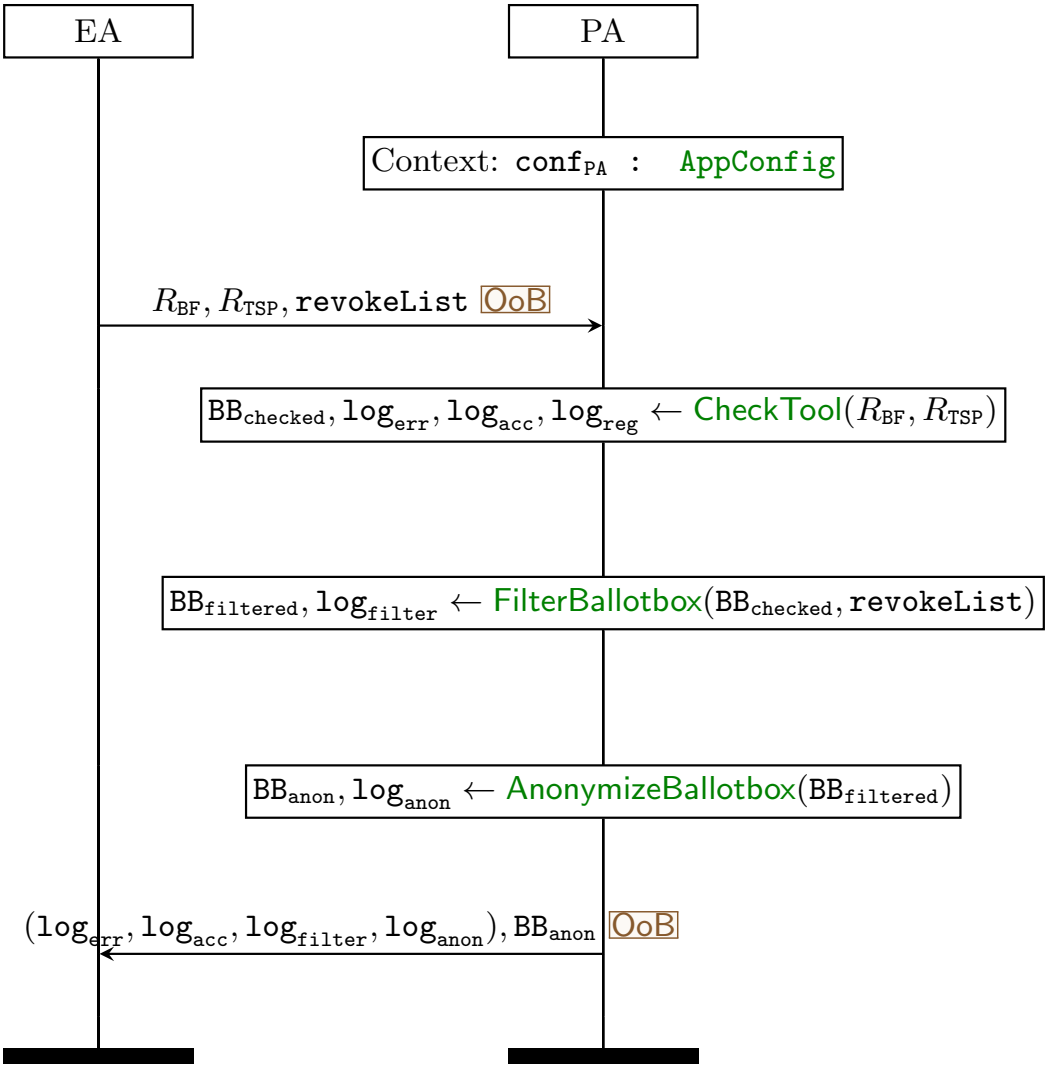

Figure 11: Tally phase. ProcessBallotbox subprotocol

4.4.1.1 CheckTool validates the received ballots (carried in CS's export) and cross-checks each ballot and its associated TSP response (also carried in CS's export) against the TSP request log (carried in Reg's export). PA's CheckTool accounts for various error types (see CheckBallotResult); detected errors are written to the corresponding error log, together with data binding them to the affected ballots for audit purposes. CheckTool outputs a checked ballotbox together with the error logs.

Algorithm 39: CheckTool - Link to Code

Context:

- $\texttt{conf}_{\texttt{PA}}$: `AppConfig`
- $\texttt{log}_{\texttt{err}}$: `List<PAErrorLogEntry>`. Error log for ballots rejected by CheckTool.
- $\texttt{log}_{\texttt{acc}}$: `List<PALogEntry>`. Log for ballots accepted by CheckTool.
- $\texttt{log}_{\texttt{reg}}$: `Map<MessageImprint, CheckBallotResult>`. Error log for reg data

Input:

- R : `List<BallotFileRecord>`. Exported votes as records
- $R_{\texttt{TSP}}$: `List<TSPReq>`. Registration data log by TSP responder

Output:

- $\texttt{BB}_{\texttt{checked}}$: `CheckedBallotbox`. Its M field maps voter ID to `VoterBallot`.
- $\texttt{log}_{\texttt{err}}$: `List<PAErrorLogEntry>`
- $\texttt{log}_{\texttt{acc}}$: `List<PALogEntry>`
- $\texttt{log}_{\texttt{reg}}$: `Map<MessageImprint, CheckBallotResult>`

Operation:

1 $\texttt{BB}_{\texttt{checked}} \leftarrow \mathsf{CheckedBallotbox}(\texttt{conf}_{\texttt{PA}}.eid, \emptyset)$
2 $\texttt{log}_{\texttt{err}} \leftarrow \emptyset$
3 $\texttt{log}_{\texttt{acc}} \leftarrow []$
/* getCheckedBallots */
4 $\texttt{M}_{\texttt{BR}} \leftarrow \mathsf{LoadBallotFileRecord}(R)$
/* getRegData */
5 $\texttt{log}_{\texttt{TSP}}$: `Map<MessageImprint, TSPReq>` $\leftarrow \mathsf{LoadRegData}(R_{\texttt{TSP}})$
/* checkRegData */
6 $\texttt{M}'_{\texttt{BR}} \leftarrow \mathsf{CheckCollectorRegConsistency}(\texttt{M}_{\texttt{BR}}, \texttt{log}_{\texttt{TSP}})$
/* getBallotBox */
7 **foreach** $(k, br) \in \texttt{M}'_{\texttt{BR}}$ ▷ IvxvBboxLoader.java#L445 **do**
8 $id_{\texttt{voter}} \leftarrow k.id_{\texttt{voter}}$
9 **if** $\texttt{BB}_{\texttt{checked}}.M[id_{\texttt{voter}}] =$ **null then**
$\texttt{BB}_{\texttt{checked}}.M.\mathsf{put}(id_{\texttt{voter}}, \mathsf{VoterBallot}(id_{\texttt{voter}}, \{\}, \mathsf{false}))$
10 $\texttt{BB}_{\texttt{checked}}.M[id_{\texttt{voter}}].\texttt{ballots}.\mathsf{append}(br.b)$
11 $\texttt{log}_{\texttt{acc}} \leftarrow \mathsf{CreatePALogEntry}(\texttt{log}_{\texttt{acc}}, br.b)$
12 **foreach** $(id_{\texttt{voter}}, vb) \in \texttt{BB}_{\texttt{checked}}.M$ **do**
13 stably sort $vb.\texttt{ballots}$ by $b.t$ in ascending order
14 $\texttt{latest} \leftarrow vb.\texttt{ballots}[-1]$
15 **foreach** $b \in vb.\texttt{ballots}$ **do**
16 **if** $b \neq \texttt{latest} \wedge b.t = \texttt{latest}.t$ **then**
17 $ref \leftarrow \mathsf{BallotRef}(id_{\texttt{voter}}, b.id)$
18 $\texttt{log}_{\texttt{err}}.\mathsf{append}(\mathsf{CreatePAErrorLogEntry}(ref, \texttt{SAME_TIME_AS_LATEST}))$
19 **return** $\texttt{BB}_{\texttt{checked}}, \texttt{log}_{\texttt{err}}, \texttt{log}_{\texttt{acc}}, \texttt{log}_{\texttt{reg}}$

LoadBallotFileRecord parses CS's exported ballot-file records into ballot responses keyed by ballot reference. It also filters ballots with recurrent TSP responses: records are grouped by their TSP message imprint; the earliest record in each group is retained, and the rest are removed and logged.

Algorithm 40: LoadBallotFileRecord - Link to Code

Context:

- $\texttt{conf}_{\texttt{PA}}$: AppConfig
- $\texttt{log}_{\texttt{err}}$: List<PAErrorLogEntry>. Error log for ballots rejected by CheckTool.

Input:

- R : List<BallotFileRecord>. Exported votes as records

Output:

- $\texttt{M}_{\texttt{BR}}$: Map<BallotRef,BallotResponse>. Stores ballot response keyed by ballot reference

Operation:

$\texttt{M}_{\texttt{BR}} \leftarrow \emptyset$
foreach $r \in R$ **do**
  $br \leftarrow \mathsf{CreateBallotResponse}(r)$
  if $br \neq$ **null then**
    $\texttt{M}_{\texttt{BR}}.\mathsf{put}(r.ref, br)$
▷ removeRecurrentResponses
G : Map<MessageImprint, (BallotRef, BallotResponse)[]> $\leftarrow \emptyset$
foreach $(ref, br) \in \texttt{M}_{\texttt{BR}}$ **do**
  $k_{reg} \leftarrow br.\texttt{resp}_{\texttt{tsp}}.m$
  $G[k_{reg}].\mathsf{append}((ref, br))$
foreach $k_{reg} \in \mathsf{Keys}(G)$ **do**
  if $|G[k_{reg}]| > 1$ **then**
    sort $G[k_{reg}]$ by $(e.br.b.t, e.br.b.id)$ (asc) for $e \in G[k_{reg}]$
    for $i \leftarrow 1$ *to* $|G[k_{reg}]| - 1$ **do**
▷ Keeping the earliest one
      $(ref, br) \leftarrow G[k_{reg}][i]$
      $\texttt{M}_{\texttt{BR}}.\mathsf{remove}(ref)$
      $\texttt{log}_{\texttt{err}}.\mathsf{append}(\mathsf{CreatePAErrorLogEntry}(ref, \texttt{REG_RESP_NOT_UNIQUE}))$
return $\texttt{M}_{\texttt{BR}}$

LoadRegData parses Reg's exported TSP request log into a map keyed by message imprint. Duplicate request imprints are logged as `REG_REQ_NOT_UNIQUE`, so the later CS–Reg consistency check operates over unique registration-request keys.

Algorithm 41: LoadRegData - Link to Code

Context:

- $\texttt{conf}_{\texttt{PA}}$: `AppConfig`
- $\texttt{log}_{\texttt{reg}}$: `Map<MessageImprint, CheckBallotResult>`. Error log for reg data

Input:

- $R_{\texttt{TSP}}$: `List<TSPReq>`. Registration data log by TSP responder; each entry is a TSP request

Output:

- $\texttt{log}_{\texttt{TSP}}$: `Map<MessageImprint, TSPReq>`

Operation:

1 $\texttt{log}_{\texttt{TSP}} \leftarrow \emptyset$
2 **foreach** `req` $\in R_{\texttt{TSP}}$ **do**
3 $k \leftarrow \texttt{req}.m$
4 **if** $k \in \mathsf{Keys}(\texttt{log}_{\texttt{TSP}})$ **then**
$\texttt{log}_{\texttt{reg}}.\mathsf{put}(k, \texttt{REG_REQ_NOT_UNIQUE})$
5 **else**
$\texttt{log}_{\texttt{TSP}}.\mathsf{put}(k, \texttt{req})$
6 **return** $\texttt{log}_{\texttt{TSP}}$

CheckCollectorRegConsistency cross-checks CS's ballot export against Reg's TSP request log by matching the TSP message imprint in each ballot response to a request entry.

Algorithm 42: CheckCollectorRegConsistency - Link to Code

Context:

- $\texttt{conf}_{\texttt{PA}}$: AppConfig
- $\texttt{log}_{\texttt{err}}$: List<PAErrorLogEntry>. Error log for ballots rejected by CheckTool.
- $\texttt{log}_{\texttt{reg}}$: Map<MessageImprint, CheckBallotResult>. Error log for reg data

Input:

- $\texttt{M}_{\texttt{BR}}$: Map<BallotRef, BallotResponse>
- $\texttt{log}_{\texttt{TSP}}$: Map<MessageImprint, TSPReq>

Output:

- $\texttt{M}'_{\texttt{BR}}$: Map<BallotRef, BallotResponse>

Operation:

1 $\texttt{M}'_{\texttt{BR}} \leftarrow \emptyset$

2 **foreach** $(ref, br) \in \texttt{M}_{\texttt{BR}}$ **do**

3 $k \leftarrow br.\texttt{resp}_{\texttt{tsp}}.m$

4 $\texttt{req} \leftarrow \texttt{log}_{\texttt{TSP}}.\texttt{remove}(k)$

5 **if** $\texttt{req} = \textbf{null}$ **then**

$\texttt{log}_{\texttt{err}}.\texttt{append}(\texttt{CreatePAErrorLogEntry}(ref, \texttt{BALLOT_WITHOUT_REG_REQ}))$

▷ IvxvBboxLoader.java#L376

6 **else if** $k \neq \texttt{req}.m$ **then**

$\texttt{log}_{\texttt{err}}.\texttt{append}(\texttt{CreatePAErrorLogEntry}(ref, \texttt{REG_RESP_REQ_UNMATCH}))$

▷ IvxvBboxLoader.java#L379

continue

7 $\texttt{M}'_{\texttt{BR}}.\texttt{put}(ref, br)$

8 **foreach** $\texttt{regKey} \in \texttt{Keys}(\texttt{log}_{\texttt{TSP}})$ ▷ IvxvBboxLoader.java#L423 **do**

$\texttt{log}_{\texttt{reg}}.\texttt{put}(\texttt{regKey}, \texttt{REG_REQ_WITHOUT_BALLOT})$

9 **return** $\texttt{M}'_{\texttt{BR}}$

CreateBallotResponse constructs a checked ballot response from a ballot-file record. It resolves the voter in PA's voter list, validates the BDOC together with its registration response, creates a ballot object, and rejects ballots whose ballot time is before the election start.

Algorithm 43: CreateBallotResponse - Link to Code

Context:

- $\texttt{conf}_{\texttt{PA}}$: AppConfig
- $\texttt{log}_{\texttt{err}}$: List<PAErrorLogEntry>. Error log for ballots rejected by CheckTool.

Input:

- r : BallotFileRecord

Output:

- br : BallotResponse

Operation:

1 $id_{\texttt{voter}} \leftarrow r.ref.id_{\texttt{voter}}$
▷ TspProfile.java#L171
2 voter : Voter $\leftarrow \texttt{conf}_{\texttt{PA}}.\texttt{M}_{\texttt{voterlist}}[id_{\texttt{voter}}]$
3 **if** voter $=$ **null then**
4 $\texttt{log}_{\texttt{err}}$.append(CreatePAErrorLogEntry($r.ref$, VOTER_NOT_FOUND))
return
5 $BC, \texttt{err} \leftarrow$ GetTrustedBDOC($r.\texttt{resp}_{\texttt{tsp}}, r.\texttt{resp}_{\texttt{ocsp}}, r.BC$)
6 **if** err $\neq$ **null then**
7 $\texttt{log}_{\texttt{err}}$.append(CreatePAErrorLogEntry($r.ref$, err))
return
8 $b \leftarrow$ CreateBallot(r, BC, voter)
9 **if** $b =$ **null then return**
10 **if** $b.t < \texttt{conf}_{\texttt{PA}}.\texttt{L}_{\texttt{time}}.t_{\texttt{EStart}}$ **then**
11 $\texttt{log}_{\texttt{err}}$.append(CreatePAErrorLogEntry($r.ref$, TIME_BEFORE_START))
return
12 $br \leftarrow$ BallotResponse($b, r.\texttt{resp}_{\texttt{tsp}}$)
▷ IvxvBboxLoader.java#L282
13 **return** br

GetTrustedBDOC parses and validates BDOC; combines the original BDOC with its validated TSP timestamp and OCSP response.

Algorithm 44: GetTrustedBDOC - Link to Code

Input:

- $\texttt{resp}_{\texttt{tsp}}$: TSPResponse
- $\texttt{resp}_{\texttt{ocsp}}$: OCSPResponse
- b : BDOC

Output:

- b' : BDOC
- err : CheckBallotResult

Operation:

1 $\texttt{err} \leftarrow \textbf{null}$

/* combine $r.BC$ with the TSP/OCSP response */

2 $b' \leftarrow b$

3 $b'.\texttt{resp}_{\texttt{tsp}} \leftarrow \texttt{resp}_{\texttt{tsp}}$

4 $b'.\texttt{resp}_{\texttt{ocsp}} \leftarrow \texttt{resp}_{\texttt{ocsp}}$

/* validate BDOC via digidoc4j library */

▷ BdocContainerReader.java#L142

5 $err \leftarrow \text{ValidateCombinedBDOCPA}(\texttt{conf}_{\texttt{PA}}.\texttt{conf}_{\texttt{trust}}, b')$

6 **if** $\texttt{err} \neq \textbf{null}$ **then return** $b', \texttt{err}$

7 $\texttt{err} \leftarrow \text{VerifyTSPPA}(b'.\texttt{resp}_{\texttt{tsp}})$

8 **if** $\texttt{err} \neq \textbf{null}$ **then return** $b', \texttt{err}$

9 **return** $b', \texttt{err}$

ValidateCombinedBDOCPA abstract helper that verifies the combined BDOC under PA's trusted certificates using the digidoc4j library. It validates the voter signature, return `INVALID_BALLOT_SIGNATURE` on on failed checks; validates the TSP and OCSP responses and returns `INVALID_SIGNATURE_PROFILE` on failed checks. It returns **null** if all checks passed.

VerifyTSPPA validates the TSP response using PA's configured TSP client public key.

Algorithm 45: VerifyTSPPA - Link to Code

Context:

- $\texttt{conf}_{\texttt{PA}}$: `AppConfig`

Input:

- r : `TSPResponse` Registration response from TSP service

Output:

- `err` : `CheckBallotResult` or **null** if verified

Operation:

1. $m, n \leftarrow r.m, r.n$
2. $pk_{\texttt{tspClient}} \leftarrow \texttt{conf}_{\texttt{PA}}.\texttt{crt}_{\texttt{tspClient}}.pk$
3. **if** $n =$ **null then return** `REG_NO_NONCE`
4. **if** n *is not* `tn` *:* *TSPNonce* **then return** `REG_NONCE_NOT_SIG`
5. **if** $m.\texttt{alg} \neq n.\texttt{alg}$ **then return** `REG_NONCE_ALG_MISMATCH`
6. **if** $\neg\mathsf{Verify}(pk_{\texttt{tspClient}}, m.h, n.\sigma)$ **then return** `REG_NONCE_SIG_INVALID`
7. **return null**

CreateBallot creates a `Ballot` from BDOC container data and voter information. The downstream ballot key `qKey` is the full filename stem after removing `.ballot`; for a CS-accepted filename `E.Q.ballot`, it is `E.Q`. This differs from the bare question ID q used during voting. PA checks only the suffix here and retains the stem as an opaque key, without checking it against the configured election or question IDs.

Algorithm 46: CreateBallot - Link to Code

Context:

- $\texttt{conf}_{\texttt{PA}}$: AppConfig
- $\texttt{log}_{\texttt{err}}$: List<PAErrorLogEntry>. Error log for ballots rejected by CheckTool.

Input:

- r : BallotFileRecord
- BC : BDOC
- voter : Voter

Output:

- b : Ballot

Operation:

1 signer : X509Certificate $\leftarrow BC.\texttt{L}_{\texttt{sig}}[0].\texttt{crt}_{\texttt{signer}}$
2 **if** $\neg(\texttt{signer} \neq \textbf{null} \wedge \texttt{signer}[\texttt{conf}_{\texttt{PA}}.\texttt{idType}] = r.ref.id_{\texttt{voter}})$ ▷ TspProfile.java#L58 **then**
3 $\texttt{log}_{\texttt{err}}$.append(CreatePAErrorLogEntry($r.ref$, MISSING_VOTER_SIGNATURE))
return
4 **if** $\exists bf \in BC.\texttt{L}_{\texttt{ballot}}$, $\neg bf.\texttt{filename}$.EndsWith(*".ballot"*) **then**
5 $\texttt{log}_{\texttt{err}}$.append(CreatePAErrorLogEntry($r.ref$, UNKNOWN_FILE_IN_VOTE_CONTAINER))
return
6 $V \leftarrow \emptyset$
7 **foreach** $bf \in BC.\texttt{L}_{\texttt{ballot}}$ **do**
8 qKey $\leftarrow bf.\texttt{filename}$.RemoveSuffix(".ballot")
9 V.put(qKey, $bf.v$)
10 $b \leftarrow$ Ballot($r.ref.id_{\texttt{ballot}}, BC.\texttt{resp}_{\texttt{tsp}}.t$, voter, V)
11 **return** b

4.4.1.2 FilterBallotbox

performs deduplication and revocation of ballots. In the checked ballotbox, each voter ID may map to multiple ballots, as IVXV supports revoting. To filter the ballotbox, only the most recent valid ballot for each voter is retained, and all ballots associated with voters appearing in the revocation list are removed. The resulting filtered ballotbox is then returned.

Algorithm 47: FilterBallotbox - Link to Code, - Link to Code

Context:

- $\texttt{conf}_{\texttt{PA}}$: AppConfig

Input:

- $\texttt{BB}_{\texttt{checked}}$: CheckedBallotbox
- `revokeList` : `List<String>`. A list of voter ID for their votes to be revoked

Output:

- $\texttt{BB}_{\texttt{filtered}}$: FilteredBallotbox. Its M field maps voter ID to the voter's latest ballot
- $\texttt{log}_{\texttt{filter}}$: `List<PALogEntry>`. Log for ballots being filtered

Operation:

1 $\texttt{BB}_{\texttt{filtered}} \leftarrow \mathsf{FilteredBallotbox}(\texttt{BB}_{\texttt{checked}}.eid, \emptyset)$
2 $\texttt{log}_{\texttt{filter}} \leftarrow []$
3 **foreach** $(id_{\texttt{voter}}, vb) \in \texttt{BB}_{\texttt{checked}}.M$ **do** ▷ SquashTool.java#L59
4 $t \leftarrow -1$
5 $\texttt{latest} \leftarrow \textbf{null}$
6 **foreach** $b \in vb.\texttt{ballots}$ **do**
7 **if** $b.t \geq t$ **then**
8 $t \leftarrow b.t$
9 $\texttt{latest} \leftarrow b$
10 **foreach** $b \in vb.\texttt{ballots}$ **do**
11 **if** $b \neq \texttt{latest}$ **then**
12 $\texttt{log}_{\texttt{filter}} \leftarrow \mathsf{CreatePALogEntry}(\texttt{log}_{\texttt{filter}}, b)$
13 $\texttt{valid} \leftarrow \mathsf{true}$
14 **foreach** $v \in \texttt{latest.votes}$ **do** ▷ SquashTool.java#L60
if $\neg\mathsf{IsValidCiphertextPA}(\texttt{conf}_{\texttt{PA}}.P_{\texttt{EG}}, \texttt{conf}_{\texttt{PA}}.\texttt{crt}_{\texttt{EG}}.pk, v)$ **then** $\texttt{valid} \leftarrow \mathsf{false}$
15 **if** $id_{\texttt{voter}} \in \texttt{revokeList}$ **then**
$\texttt{valid} \leftarrow \mathsf{false}$
16 **if** $\neg\texttt{valid}$ **then**
17 $\texttt{log}_{\texttt{filter}} \leftarrow \mathsf{CreatePALogEntry}(\texttt{log}_{\texttt{filter}}, \texttt{latest})$
18 **else**
$\texttt{BB}_{\texttt{filtered}}.M.\mathsf{put}(id_{\texttt{voter}}, \texttt{latest})$
19 **return** $\texttt{BB}_{\texttt{filtered}}, \texttt{log}_{\texttt{filter}}$

IsValidCiphertextPA parses and verifies an ASN.1-encoded vote ciphertext. The validation logic is similar to IsValidCiphertextCS; the main difference is that the PA check has an extra check that rejects non-canonical data.

Algorithm 48: IsValidCiphertextPA - Link to Code

Input:
- P_{EG} : `ElGamalParameters`. ElGamal parameters
- pk : `Int`. ElGamal public key
- v : `Byte[]`. Vote ciphertext, with ASN.1 encoding

Output:
- b : `Bool`: true if valid, false otherwise

Operation:

1 $c \leftarrow \text{DecodeASN1CiphertextPA}(v, P_{\text{EG}})$
2 **if** $c = \perp$ **then return** false
3 **for** $i \leftarrow 0$ *to* 1 **do**
4 **if** $\neg(c[i]^q \equiv 1 \bmod p)$ **then return** false
5 **if** $\neg(0 < c[i] < p)$ **then return** false
6 **if** $\neg(c[i]^{(p-1)/2} \equiv 1 \bmod p)$ **then return** false
7 **return** true

4.4.1.3 AnonymizeBallotbox

is the last stage in PA's workflow. It creates an anonymised ballotbox that excludes the identifiable information of each ballot, such as submission time and voter ID, retaining only minimal geographical metadata (district and parish).

Algorithm 49: AnonymizeBallotbox - Link to Code

Input:

$\texttt{BB}_{\texttt{filtered}}$: FilteredBallotbox

Output:

$\texttt{BB}_{\texttt{anon}}$: AnonBallotbox

$\texttt{log}_{\texttt{anon}}$: List<PALogEntry>. Log for ballots being anonymised

Operation:

$\texttt{BB}_{\texttt{anon}} \leftarrow \text{AnonBallotbox}(\texttt{BB}_{\texttt{filtered}}.eid, \emptyset)$

$\texttt{log}_{\texttt{anon}} \leftarrow []$

foreach $(id_{\texttt{voter}}, b) \in \texttt{BB}_{\texttt{filtered}}.M$ **do**

  $\texttt{log}_{\texttt{anon}} \leftarrow \text{CreatePALogEntry}(\texttt{log}_{\texttt{anon}}, b)$

  $d \leftarrow b.\texttt{voter}.d$

  $p \leftarrow b.\texttt{voter}.p$

  foreach $(\texttt{qKey}, v) \in b.\texttt{votes}$ **do**

    if $\texttt{BB}_{\texttt{anon}}.M[d][p][\texttt{qKey}] =$ **null then**

      $\texttt{BB}_{\texttt{anon}}.M[d][p].\text{put}(\texttt{qKey}, [])$

    $\texttt{BB}_{\texttt{anon}}.M[d][p][\texttt{qKey}].\text{append}(v)$

return $\texttt{BB}_{\texttt{anon}}, \texttt{log}_{\texttt{anon}}$

4.4.2 MixBallotbox

The MixBallotbox subprotocol describes the cryptographic mixing process applied to the anonymised ballotbox. The Mixer converts PA's anonymised ballotbox from the IVXV ballotbox format into Verificatum's ByteTree format, invokes the Verificatum mixnet, and converts the resulting shuffled output back into the IVXV format.

We treat the internal Verificatum mixing operation, denoted Mix, as an ideal black box, since the Verificatum shuffle protocol itself is outside the scope of this IVXV protocol specification. Instead, we model the IVXV-side wrapper logic, including the data conversion between the IVXV and Verificatum formats and the input and output artefacts.

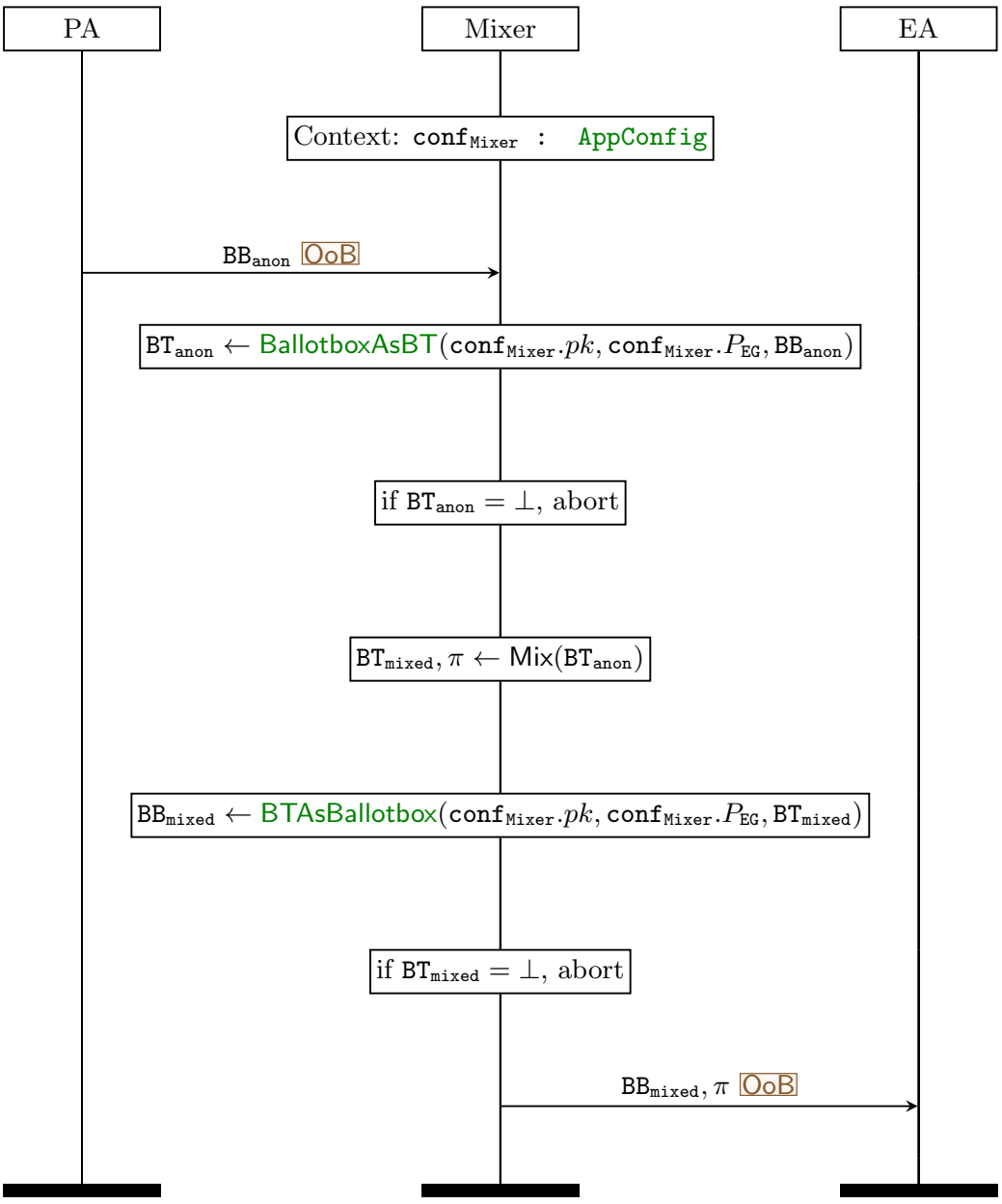

Figure 12: Tally phase. MixBallotbox subprotocol

4.4.2.1 BallotboxAsBT

encodes each ballot into a product group element suitable for Verificatum after the mixnet adapter receives the anonymised ballotbox.

Algorithm 50: BallotboxAsBT - Link to Code - Link to Code

Input:

- pk : `Int`. ElGamal public key
- $P_{\texttt{EG}}$: `ElGamalParameters`. ElGamal parameters
- $\texttt{BB}_{\texttt{anon}}$: `AnonBallotbox`

Output:

- $\texttt{BT}_{\texttt{anon}}$: `List<BTElement>` or $\perp$ if ciphertext decoding fails

Operation:

```
1  BT_anon ← []
2  bt_pk ← PublicKeyAsBT(P_EG, pk)
3  e_e ← (1, S2GE(P_EG, bt_pk, BB_anon.eid))
4  foreach (d, P) ∈ BB_anon.M do
5      e_d ← (1, S2GE(P_EG, bt_pk, d))
6      foreach (p, Q) ∈ P do
7          e_p ← (1, S2GE(P_EG, bt_pk, p))
8          foreach (qKey, V) ∈ Q do
9              e_q ← (1, S2GE(P_EG, bt_pk, qKey))
10             foreach v ∈ V do
11                 e_c ← DecodeASN1CiphertextMixer(v, P_EG)
                   if e_c = ⊥ then return ⊥
12                 e : BTElement ← (e_e, e_d, e_p, e_q, e_c)
13                 BT_anon.append(e)
14 return BT_anon
```

4.4.2.2 BTAsBallotbox

decodes the mixed ByteTree representation back into an anonymised ballotbox.

Algorithm 51: BTAsBallotbox

Input:

- pk : `Int`. ElGamal public key
- $P_{\texttt{EG}}$: `ElGamalParameters`. ElGamal parameters
- $\texttt{BT}_{\texttt{mixed}}$: `List<BTElement>`

Output:

- $\texttt{BB}_{\texttt{mixed}}$: `AnonBallotbox` mixed ballotbox, or $\perp$ if eid mismatches or encoding fails

Operation:

1. $\texttt{BB}_{\texttt{mixed}} \leftarrow \mathsf{AnonBallotbox}(\textbf{null}, \emptyset)$
2. **foreach** $e \in \texttt{BT}_{\texttt{mixed}}$ **do**
3. $\quad eid \leftarrow \mathsf{GE2S}(P_{\texttt{EG}}, pk, e[0][1])$
4. $\quad$ **if** $\texttt{BB}_{\texttt{mixed}}.eid = \textbf{null}$ **then** $\texttt{BB}_{\texttt{mixed}}.eid = eid$
5. $\quad$ **else if** $\texttt{BB}_{\texttt{mixed}}.eid \neq eid$ **then return** $\perp$
6. $\quad d \leftarrow \mathsf{GE2S}(P_{\texttt{EG}}, pk, e[1][1])$
7. $\quad p \leftarrow \mathsf{GE2S}(P_{\texttt{EG}}, pk, e[2][1])$
8. $\quad \texttt{qKey} \leftarrow \mathsf{GE2S}(P_{\texttt{EG}}, pk, e[3][1])$
9. $\quad e_c \leftarrow e[4]$
10. $\quad n \leftarrow \mathsf{ASN1Node}([P_{\texttt{EG}}.\texttt{oid}, [e_c[0], e_c[1]]])$
11. $\quad c$: `Byte[]` $\leftarrow \mathsf{ASN1Encode}(n)$
12. $\quad$ **if** $c = \perp$ **then return** $\perp$
13. $\quad$ **if** $\texttt{BB}_{\texttt{mixed}}.M[d][p][\texttt{qKey}] = \textbf{null}$ **then** $\texttt{BB}_{\texttt{mixed}}.M[d][p].\texttt{put}(\texttt{qKey}, [])$
14. $\quad \texttt{BB}_{\texttt{mixed}}.M[d][p][\texttt{qKey}].\texttt{append}(c)$
15. **return** $\texttt{BB}_{\texttt{mixed}}$

PublicKeyAsBT converts the election ElGamal public key into Verificatum's `ByteTree` encoding.

Algorithm 52: PublicKeyAsBT

Context:

- $\texttt{conf}_{\texttt{Mixer}}$: `AppConfig`

Output:

- bt_{pk} : `BTElement`

Operation:

1. $g1 \leftarrow (g, 1)$
2. $pk' := (1, \texttt{conf}_{\texttt{Mixer}}.\texttt{crt}_{\texttt{EG}}.pk)$
3. **return** $bt_{pk} \leftarrow (g1, g1, g1, g1, pk')$

4.4.3 DecryptAndTally

The DecryptAndTally subprotocol covers the final step of the tally phase, in which the Key Application (KA) decrypts the mixed ballotbox and produces the election result and audit artefacts.

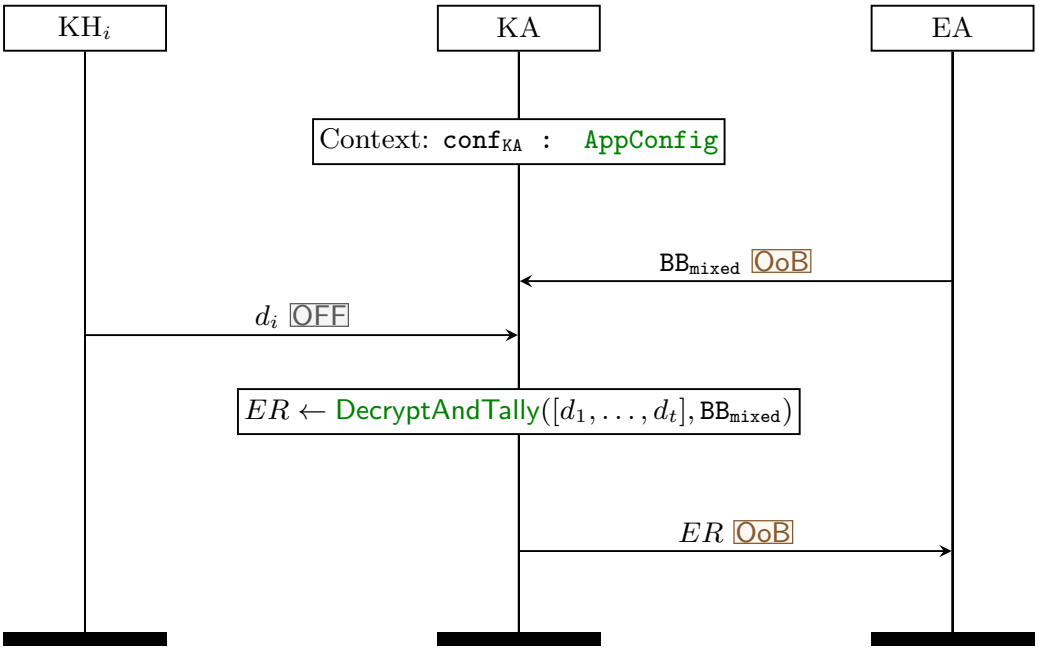

Figure 13: Tally phase. DecryptAndTally subprotocol

4.4.3.1 DecryptAndTally gathers partial shares from the Key Holders' (KH) smart cards to reconstruct the election ElGamal private key, decrypts the mixed ballotbox, computes the election result, and then collectively signs the decryption outputs (the tally and the plaintext ballotbox).

Algorithm 53: DecryptAndTally - Link to Code

Context:

- $\texttt{conf}_{\texttt{KA}}$: `AppConfig`

Input:

- $\texttt{L}'_{\texttt{smartcard}}$: `List<SmartCardData>`. A list of SmartCardData
- $\texttt{BB}_{\texttt{mixed}}$: `AnonBallotbox`. Mixed ballotbox

Output: ER : `ElectionResult` if successful, $\perp$ otherwise

Operation:

1 $(t, n) \leftarrow \texttt{conf}_{\texttt{KA}}.P_{\texttt{Th}}$
2 $P_{\texttt{EG}} \leftarrow \texttt{conf}_{\texttt{KA}}.P_{\texttt{EG}}$
3 $P_{\texttt{RSA}}$: `Tuple<Int, Int>` $\leftarrow$ **null**
4 K : `Map<Int, Int>` $\leftarrow \emptyset$
5 $\texttt{M}_{\texttt{signShare}}$: `Map<Int, Int>` $\leftarrow \emptyset$
6 **foreach** $d \in \texttt{L}'_{\texttt{smartcard}}$ **do**
7 **if** $d.P_{\texttt{EG}} \neq \texttt{conf}_{\texttt{KA}}.P_{\texttt{EG}}$ **then return** $\perp$
8 **if** $P_{\texttt{RSA}} =$ **null then** $P_{\texttt{RSA}} \leftarrow d.P_{\texttt{RSA}}$
9 **else if** $d.P_{\texttt{RSA}} \neq P_{\texttt{RSA}}$ **then return** $\perp$ ▷ ShoupSigning.java#L112
10 $i \leftarrow d.idx$
11 $K.\texttt{put}(i, d.k)$
12 $\texttt{M}_{\texttt{signShare}}.\texttt{put}(i, d.\texttt{k}_{\texttt{sign}})$
13 $esk_{\texttt{EG}} \leftarrow \texttt{RecoverDecryption}((t, n), P_{\texttt{EG}}, K)$
14 **if** $esk_{\texttt{EG}} = \perp$ **then return** $\perp$
15 $\texttt{BB}_{\texttt{dec}} \leftarrow \texttt{DecryptBallotbox}(\texttt{BB}_{\texttt{mixed}}.eid, (t, n), P_{\texttt{EG}}, \texttt{BB}_{\texttt{mixed}}, esk_{\texttt{EG}}, \texttt{conf}_{\texttt{KA}}.\texttt{checkDecodable}, \texttt{conf}_{\texttt{KA}}.\texttt{withProof})$
16 $ER \leftarrow \texttt{ComputeElectionResult}(P_{\texttt{EG}}, \texttt{conf}_{\texttt{KA}}.\texttt{M}_{\texttt{DC}}, \texttt{conf}_{\texttt{KA}}.\texttt{M}_{\texttt{DP}}, \texttt{BB}_{\texttt{dec}}, \texttt{conf}_{\texttt{KA}}.\texttt{withProof}, \texttt{conf}_{\texttt{KA}}.\texttt{proveInvalid})$
17 **foreach** $\texttt{qKey} \in \texttt{Keys}(ER.\texttt{M}_{\texttt{plainResult}})$ **do**
18 $e \leftarrow ER.\texttt{M}_{\texttt{plainResult}}[\texttt{qKey}]$
19 $\sigma \leftarrow \texttt{ShoupSign}((t, n), P_{\texttt{RSA}}, \texttt{M}_{\texttt{signShare}}, e)$
20 $ER.\texttt{M}_{\texttt{plainSig}}.\texttt{put}(\texttt{qKey}, \sigma)$
21 **foreach** $\texttt{qKey} \in \texttt{Keys}(ER.\texttt{M}_{\texttt{tallyResult}})$ **do**
22 $e \leftarrow ER.\texttt{M}_{\texttt{tallyResult}}[\texttt{qKey}]$
23 $\sigma \leftarrow \texttt{ShoupSign}((t, n), P_{\texttt{RSA}}, \texttt{M}_{\texttt{signShare}}, e)$
24 $ER.\texttt{M}_{\texttt{tallySig}}.\texttt{put}(\texttt{qKey}, \sigma)$
25 **return** ER

RecoverDecryption takes as input the threshold decryption shares obtained from the key holders' smart cards and reconstructs the election decryption key.

Algorithm 54: RecoverDecryption - Link to Code

Input:
- (t, n) : `Tuple<Int, Int>`. Threshold Decryption setting: t: threshold, n: number of shares
- $P_{\texttt{EG}}$: `ElGamalParameters`
- K : `Map<Int, Int>`. A map that maps key holders' indices to decryption shares

Output: sk : `Int` if reconstructed successfully, $\perp$ otherwise

Operation:

1 $sk \leftarrow 0$
   `/* set of share indices */`
2 $I \leftarrow \mathsf{Keys}(K)$
3 **if** $|I| < t$ **then return** $\perp$
4 **foreach** $i \in I$ **do**
5 $\quad \lambda_i \leftarrow \prod_{j \in I, j \neq i} \frac{-j}{(i-j)} \bmod P_{\texttt{EG}}.q$
6 $\quad sk \leftarrow (sk + K[i] \cdot \lambda_i) \bmod P_{\texttt{EG}}.q$
7 **return** sk

DecryptBallotbox validates each ciphertext in the mixed ballotbox and decrypts valid ciphertexts, optionally producing a decryption proof for each decrypted vote.

Algorithm 55: DecryptBallotbox - Link to Code

Input:

- eid : `String`. Election ID.
- (t, n) : `Tuple<Int, Int>`. Threshold Decryption setting: t: threshold, n: number of shares
- P_{EG} : `ElGamalParameters`
- BB_{mixed} : `AnonBallotbox`. Mixed, anonymised ballotbox
- sk : `Int`. Reconstructed decryption key
- `checkDecodable` : `Bool`: indicates whether validation on ballot ciphertext needs to be performed.
- `withProof` : `Bool`: indicates whether outputting decryption proof of valid votes.

Output:

- BB_{dec} : `DecryptedBallotbox`. Decrypted ballotbox

Operation:

```
1  pk ← (P_EG.g)^sk
2  BB_dec ← DecryptedBallotbox(eid, ∅)
3  foreach (d, P) ∈ BB_mixed.M do
4      foreach (p, Q) ∈ P do
5          foreach (qKey, B) ∈ Q do
6              BB_dec.M[d][p].put(qKey, [])
7              foreach b : Byte[] ∈ B do
8                  ok ← true
9                  v : DecryptedVote ← DecryptedVote(d, p, qKey, b, null)
10                 if checkDecodable then
                       ok ← IsValidCiphertextPA(P_EG, pk, b)
11                 if ok then
12                     dp ← ElGamalDecrypt(sk, b, withProof)
13                     if dp ≠ ⊥ then
                           v.dp ← dp
14                 BB_dec.M[d][p][qKey].append(v)
15 return BB_dec
```

ComputeElectionResult produces the final election result. It checks decrypted plaintext ballots against the configured district and candidate lists, optionally records invalid candidate choices, counts the valid votes, and outputs both the tally result and the plaintext ballotbox.

For each ballot key first encountered in a decrypted vote, this algorithm constructs a fresh Tally and PlainBallotbox, both carrying $\text{BB}_{\text{dec}}.eid$. The helpers InitTallyMap($\text{M}_{\text{DC}}, \text{M}_{\text{DP}}$) and InitPlainMap($\text{M}_{\text{DP}}$) initialize their inner M maps over every configured district and parish in M_{DP}, including parishes with no votes. Each tally bucket starts with `INVALID_VOTE_ID` at zero and, if the district is present in M_{DC}, a zero counter for each of its candidate IDs; each plaintext bucket starts as an empty list. Empty

ballot-key buckets create no result objects.

▷ Tally.java#L25
▷ PlainBallotBox.java#L27

Algorithm 56: ComputeElectionResult - Link to Code

Input:

- $P_{\texttt{EG}}$: `ElGamalParameters`
- $\texttt{M}_{\texttt{DC}}$: `Map<String, CandidateList>`. Maps district to candidate list
- $\texttt{M}_{\texttt{DP}}$: `Map<String, List<String>>`. Configured districts and their parish IDs
- $\texttt{BB}_{\texttt{dec}}$: `DecryptedBallotbox`. In a decrypted ballotbox, each element is a `DecryptedVote` object
- `withProof` : `Bool`: indicates whether outputting decryption proof of valid votes.
- `proveInvalid` : `Bool`: indicates whether outputting decryption proof of invalid votes.

Output:

- ER : `ElectionResult` if successful, $\perp$ otherwise

Operation:

```
1  TallySet : Map<String, Tally> ← ∅
2  PlainSet : Map<String, PlainBallotbox> ← ∅
3  Proof, ProofInvalid, Invalid ← []
4  foreach (d, P) ∈ BB_dec.M do
5      foreach (p, Q) ∈ P do
6          foreach (qKey, V) ∈ Q do
7              foreach v : DecryptedVote ∈ V do
8                  if qKey ∉ Keys(TallySet) then
9                      TallySet.put(qKey, Tally(BB_dec.eid, InitTallyMap(M_DC, M_DP)))
10                     PlainSet.put(qKey, PlainBallotbox(BB_dec.eid, InitPlainMap(M_DP)))
11                 choice ← INVALID_VOTE_ID
12                 dp : DecryptionProof ← v.dp
13                 if dp ≠ null then
14                     candidate ← IsValidChoice(P_EG, v, M_DC)
15                     if candidate ≠ ⊥ then choice ← candidate
                                                    /* collect valid proof */
16                 if withProof ∧ choice ≠ INVALID_VOTE_ID then
                       Proof.append(dp)
                                /* collect invalid votes and potentially proofs */
17                 if choice = INVALID_VOTE_ID then
18                     Invalid.append(v)
19                     if withProof ∧ proveInvalid ∧ dp ≠ null then
                           ProofInvalid.append(dp)
20                 TallySet[qKey].M[d][p].put(choice, TallySet[qKey].M[d][p][choice] + 1)
21                 PlainSet[qKey].M[d][p].append(choice)
22 ER ← ElectionResult(TallySet, PlainSet, Proof, ProofInvalid, Invalid, ∅, ∅)
23 return ER
```

IsValidChoice checks whether a decrypted plaintext vote encodes a valid candidate choice for the corresponding district.

Algorithm 57: IsValidChoice - Link to Code

Input:

- P_{EG} : `ElGamalParameters`
- v : `DecryptedVote`. Decrypted vote containing geographic data and proof
- M_{DC} : `Map<String, CandidateList>`. Maps district to candidate list

Output:

- `choice` : `String` if valid, $\perp$ otherwise.

Operation:

1 `choice` $\leftarrow$ DecodeModPElement($P_{\text{EG}}, v.dp.m$)
2 **if** $v.d \notin \text{Keys}(\text{M}_{\text{DC}})$ **then return** $\perp$
3 **if** `choice` $\notin \text{CandidateIDs}(\text{M}_{\text{DC}}[v.d])$ **then return** $\perp$
4 **return** `choice`

4.5 Audit Phase

The audit phase covers the post-election audit process. We infer IVXV's audit process from the Audit App (AA) source code and the audit-run demo log in the IVXV codebase, which shows a sequence of commands run over various audit artefacts.

Table 7 shows the subprotocol and algorithms involved in the audit phase.

Subprotocol / Algorithm	Actor
RunAuditTool	
VerifyAuditInput	AA
ConvertTool	AA
MixerTool	AA
DecryptTool	AA
IntegrityTool	AA

Table 7: Audit Phase Subprotocols

4.5.1 RunAuditTool

This subprotocol covers the entire audit phase. It consists of audit-input validation followed by four verification stages: IVXV–Verificatum data conversion validation, shuffle-proof verification, decryption-proof verification, and Processor App (PA) log consistency checking.

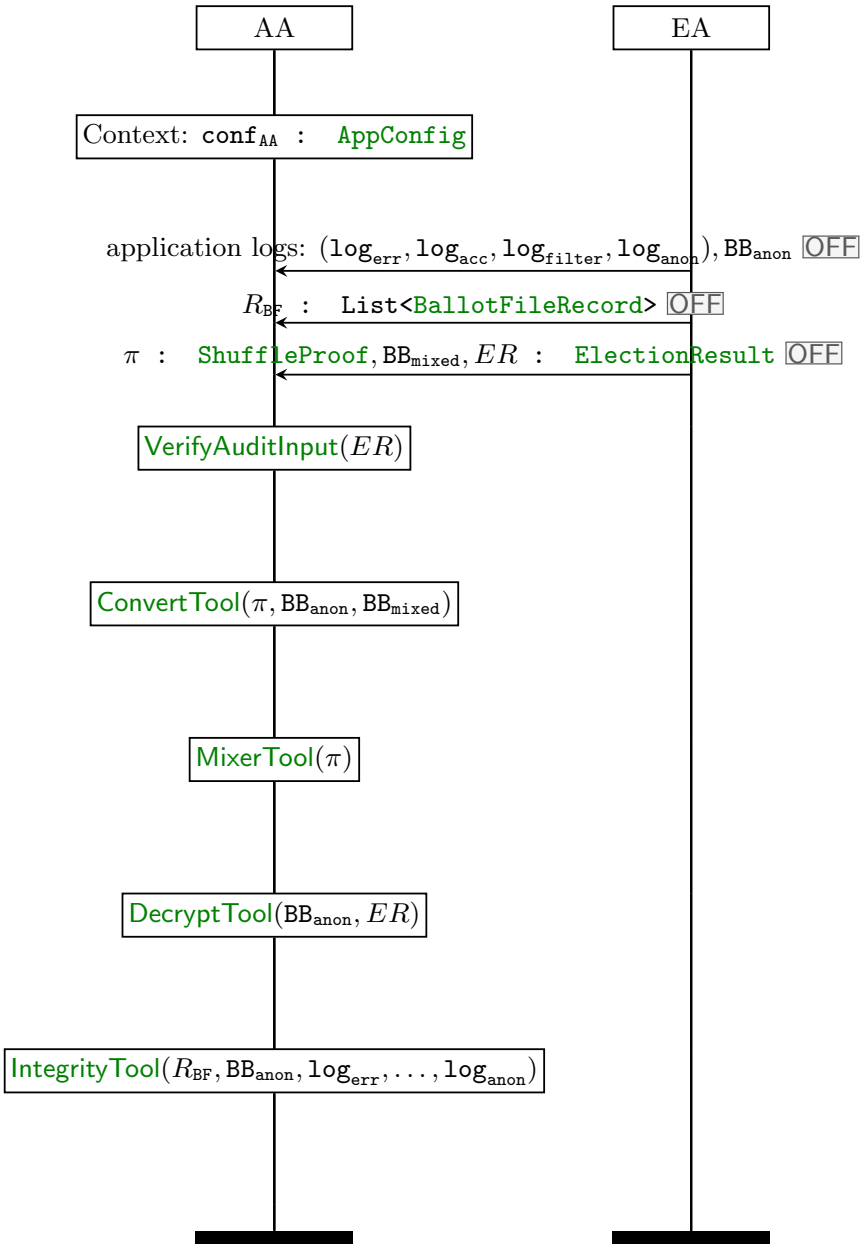

Figure 14: Audit phase. RunAuditTool subprotocol

4.5.1.1 VerifyAuditInput checks that the election result is well-formed and verifies its signatures using the configured election signing public key. Each signature covers the whole tally or plaintext record, including its *eid*.

Algorithm 58: VerifyAuditInput - Link to Code

Context:

- $\texttt{conf}_{\texttt{AA}}$: AppConfig

Input:

- ER : ElectionResult

Output:

- b : Bool: true if verifies, false otherwise

Operation:

```
                                                    /* verify election signing cert */
1   if ¬Verify(conf_AA.crt_RSA.pk, conf_AA.crt_RSA.tbs, conf_AA.crt_RSA.σ) then return false
                                                  /* verify election encryption cert */
2   if ¬Verify(conf_AA.crt_RSA.pk, conf_AA.crt_EG.tbs, conf_AA.crt_EG.σ) then return false
                                                /* verify election result signatures */
3   M_tallyResult ← ER.M_tallyResult
4   M_plainResult ← ER.M_plainResult
5   M_tallySig ← ER.M_tallySig
6   M_plainSig ← ER.M_plainSig
7   if Keys(M_tallyResult) ≠ Keys(M_tallySig) then return false
8   if Keys(M_plainResult) ≠ Keys(M_plainSig) then return false
9   foreach qKey ∈ Keys(M_tallyResult) do
10      if ¬Verify(conf_AA.crt_RSA.pk, M_tallyResult[qKey], M_tallySig[qKey]) then return false
11  foreach qKey ∈ Keys(M_plainResult) do
12      if ¬Verify(conf_AA.crt_RSA.pk, M_plainResult[qKey], M_plainSig[qKey]) then return false
13  return true
```

4.5.1.2 ConvertTool verifies the correctness of the data conversions $\texttt{BB}_{\texttt{anon}} \mapsto \pi.\texttt{BT}_{\texttt{in}}$ and $\pi.\texttt{BT}_{\texttt{out}} \mapsto \texttt{BB}_{\texttt{mixed}}$. The remaining link, $\pi.\texttt{BT}_{\texttt{in}} \mapsto \pi.\texttt{BT}_{\texttt{out}}$, is verified by MixerTool.

Algorithm 59: ConvertTool - Link to Code

Context:

- $\texttt{conf}_{\texttt{AA}}$: `AppConfig`. contains n_v (challenge length), `verifyInvalid`

Input:

- π : `ShuffleProof`. Shuffle proof
- $\texttt{BB}_{\texttt{anon}}$: `AnonBallotbox`. Input ballotbox
- $\texttt{BB}_{\texttt{mixed}}$: `AnonBallotbox`. Output ballotbox

Output:

- b : `Bool`: true if verifies, false otherwise

Operation:

```
                                                    /* verify Public key */
1   pk ← conf_AA.crt_EG.pk
2   if π.pk ≠ pk then return false
                                        /* verify Ballotbox To Bytetree Conversion */
3   ciphs_BB ← BallotboxAsBT(pk, conf_AA.P_EG, BB_anon)
4   if ciphs_BB = ⊥ then return false
5   if |ciphs_BB| ≠ |π.BT_in| then return false
6   for i ← 0 to |ciphs_BB| − 1 do
7       if ciphs_BB[i] ≠ π.BT_in[i] then return false
                                        /* verify Bytetree To Ballotbox Conversion */
8   BB_proof ← BTAsBallotbox(pk, conf_AA.P_EG, π.BT_out)
9   if BB_proof = ⊥ then return false
10  if BB_proof.eid ≠ BB_mixed.eid then return false
11  if Keys(BB_proof.M) ≠ Keys(BB_mixed.M) then return false
12  foreach (d, P) ∈ BB_proof.M do
13      if Keys(P) ≠ Keys(BB_mixed.M[d]) then return false
14      foreach (p, Q) ∈ P do
15          if Keys(Q) ≠ Keys(BB_mixed.M[d][p]) then return false
16          foreach (qKey, B_proof) ∈ Q do
17              B_mixed ← BB_mixed.M[d][p][qKey]
18              if |B_mixed| ≠ |B_proof| then return false
19              for i ← 0 to |B_proof| − 1 do
20                  if B_proof[i] ≠ B_mixed[i] then return false
21  return true
```

4.5.1.3 MixerTool verifies the Verificatum shuffle proof using the serial or threaded IVXV verifier selected by the audit configuration. We do not model the verifier details.

Algorithm 60: MixerTool - Link to Code

Context:

- $\texttt{conf}_{\texttt{AA}}$: `AppConfig`

Input:

- π : `ShuffleProof`. Shuffle proof

Output:

- b : `Bool`: true if valid, false otherwise

Operation:

1 $\rho \leftarrow \mathsf{ComputeRho}(\pi.\texttt{prot})$
2 $H \leftarrow \mathsf{GetGenerators}(\rho, \pi.\texttt{prot}, |\pi.\texttt{BT}_{\texttt{in}}|)$
3 $s \leftarrow \mathsf{ComputeROSeed}(\pi, \rho, H)$
4 $\texttt{L}_{\texttt{batch}} \leftarrow \mathsf{ComputeBatchVector}(s, \pi.\texttt{prot}, |\pi.\texttt{BT}_{\texttt{in}}|)$
5 $v \leftarrow \mathsf{ComputeChallenge}(\pi, \rho, s)$
6 **if** $\neg\mathsf{VerifyPermutation}(\pi, \texttt{L}_{\texttt{batch}}, H, v, \texttt{conf}_{\texttt{AA}}.\texttt{threaded})$ **then return** false
7 **if** $\neg\mathsf{VerifyReRand}(\pi, \texttt{L}_{\texttt{batch}}, v, \texttt{conf}_{\texttt{AA}}.\texttt{threaded})$ **then return** false
8 **return** true

4.5.1.4 DecryptTool verifies the relationship between the ballotbox and the election result produced by KA, covering the decryption proofs, the discarded-ciphertext list, the plaintext ballotbox, and the tally result. The algorithm also verifies decryption-proof and plaintext correctness.

AA performs multiset checks involving the following multisets: Let $C_{\texttt{mixed}}$ be the multiset of ciphertexts in $\texttt{BB}_{\texttt{mixed}}$, $C_{\texttt{valid}} = \{dp.c \mid dp \in \Pi_{\texttt{valid}}\}$ be the ciphertexts appearing in valid decryption proofs, and $C_{\texttt{invalid}} = \{v.v \mid v \in V_{\texttt{invalid}}\}$ be the ciphertexts declared invalid in the election result. If $\texttt{verifyInvalid} = \mathsf{true}$, let $C^{\pi}_{\texttt{invalid}} = \{dp.c \mid dp \in \Pi_{\texttt{invalid}}\}$ be the ciphertexts appearing in invalid decryption proofs. AA then checks the following invariants:

- **Valid decryption correctness:** every proof in $\Pi_{\texttt{valid}}$ verifies cryptographically, and its plaintext decodes to a valid choice.
- **Invalid decryption consistency:** if $\texttt{verifyInvalid} = \mathsf{true}$, every proof in $\Pi_{\texttt{invalid}}$ verifies cryptographically, its plaintext decodes to an invalid choice, and $C^{\pi}_{\texttt{invalid}} = C_{\texttt{invalid}}$. If $\texttt{verifyInvalid} = \mathsf{false}$, the tool checks only that the discarded ciphertexts in $C_{\texttt{invalid}}$ occur in $\texttt{BB}_{\texttt{mixed}}$.
- **Ciphertext partitioning:** no ciphertext is declared both valid and invalid, and every ciphertext in the ballotbox is accounted for:

$$C_{\texttt{valid}} \cap C_{\texttt{invalid}} = \emptyset \wedge C_{\texttt{mixed}} = C_{\texttt{valid}} \cup C_{\texttt{invalid}}.$$

- **Plaintext and tally consistency:** the plaintext ballotbox is consistent with the valid decryption proofs, up to the number of discarded invalid votes, and the published tally equals the count computed from the plaintext ballotbox.

The implementation audits one supplied plaintext/tally pair per invocation. The loop below presents the corresponding supplied result pairs using the downstream ballot keys `qKey` in $\texttt{M}_{\texttt{plainResult}}$. We treat this iteration as operational handling of the `ElectionResult` maps, rather than a code-level loop over configured question IDs; the corresponding tally is assumed to be supplied under the same key. The working ballotbox map's IsEmpty operation checks that every ciphertext leaf list is empty.

Algorithm 61: DecryptTool - Link to Code

Context:

$\texttt{conf}_{\texttt{AA}}$: AppConfig

Input:

$\texttt{BB}_{\texttt{mixed}}$: AnonBallotbox

ER : ElectionResult

Output:

b : Bool: true if verifies, false otherwise

Operation:

$\texttt{log}_{\texttt{err}}, \texttt{S}_{\texttt{invalidProofCiph}}, \texttt{S}_{\texttt{validCiph}}, \texttt{S}_{\texttt{droppedCiph}} \leftarrow \emptyset$

$\texttt{M}_{\texttt{tallyResult}}, \texttt{M}_{\texttt{plainResult}}, \Pi_{\texttt{valid}}, \Pi_{\texttt{invalid}}, V_{\texttt{invalid}} \leftarrow ER.\texttt{M}_{\texttt{tallyResult}}, ER.\texttt{M}_{\texttt{plainResult}}, ER.\Pi_{\texttt{valid}}, ER.\Pi_{\texttt{invalid}}, ER.V_{\texttt{invalid}}$

if $\texttt{conf}_{\texttt{AA}}.\texttt{verifyInvalid}$ **then**

  foreach $dp \in \Pi_{\texttt{invalid}}$ **do**

    $\texttt{S}_{\texttt{invalidProofCiph}}.\text{add}(dp.c)$

    if $\neg$VerifyElGamalDecrypt$(\texttt{conf}_{\texttt{AA}}.\texttt{crt}_{\texttt{EG}}.pk, dp)$ **then** $\texttt{log}_{\texttt{err}}.\text{add}(dp)$

  if $\texttt{log}_{\texttt{err}} \neq \emptyset$ **then return** false

  if $\neg$VerifyPlaintexts$(\texttt{BB}_{\texttt{mixed}}, \Pi_{\texttt{invalid}}, \text{false})$ **then return** false

$\texttt{log}_{\texttt{err}} \leftarrow \emptyset$

foreach $dp \in \Pi_{\texttt{valid}}$ **do**

  $\texttt{S}_{\texttt{validCiph}}.\text{add}(dp.c)$

  if $\neg$VerifyElGamalDecrypt$(\texttt{conf}_{\texttt{AA}}.\texttt{crt}_{\texttt{EG}}.pk, dp)$ **then** $\texttt{log}_{\texttt{err}}.\text{add}(dp)$

if $\texttt{log}_{\texttt{err}} \neq \emptyset$ **then return** false

/* mutates the input ballotbox inplace */

if $\neg$VerifyPlaintexts$(\texttt{BB}_{\texttt{mixed}}, \Pi_{\texttt{valid}}, \text{true})$ **then return** false

foreach v : *DecryptedVote* $\in V_{\texttt{invalid}}$ **do**

  $\texttt{S}_{\texttt{droppedCiph}}.\text{add}(v.v)$

if $\texttt{conf}_{\texttt{AA}}.\texttt{verifyInvalid}$ **then**

  if $\texttt{S}_{\texttt{invalidProofCiph}} \neq \texttt{S}_{\texttt{droppedCiph}}$ **then return** false

else

  if $\neg$VerifyCiphersInBB$(\texttt{BB}_{\texttt{mixed}}, \texttt{S}_{\texttt{droppedCiph}})$ **then return** false

if $\texttt{S}_{\texttt{validCiph}} \cap \texttt{S}_{\texttt{droppedCiph}} \neq \emptyset$ **then return** false

if $\neg \texttt{BB}_{\texttt{mixed}}.M.\text{IsEmpty}()$ **then return** false

foreach qKey $\in$ Keys$(\texttt{M}_{\texttt{plainResult}})$ **do**

  if $\neg$VerifyValidConsistency$(\texttt{conf}_{\texttt{AA}}.\texttt{crt}_{\texttt{EG}}.pk, \Pi_{\texttt{valid}}, \texttt{M}_{\texttt{plainResult}}[\text{qKey}], |V_{\texttt{invalid}}|)$ **then return** false

  if $\neg$VerifyTally$(\texttt{M}_{\texttt{plainResult}}[\text{qKey}], \texttt{M}_{\texttt{tallyResult}}[\text{qKey}])$ **then return** false

return true

VerifyCiphersInBB checks that every supplied ciphertext occurrence appears in the working ballotbox. Each matched occurrence is removed from the working ballotbox.

Algorithm 62: VerifyCiphersInBB - Link to Code

Input:

- $\texttt{BB}_{\texttt{work}}$: `AnonBallotbox`. The ballotbox is mutated in place
- $\texttt{S}_{\texttt{ciph}}$: `Multiset<Byte[]>`. Ciphertext occurrences to consume

Output:

- b : `Bool`: `true` iff every occurrence in $\texttt{S}_{\texttt{ciph}}$ was found

Operation:

```
1  D : Map<Byte[],Int> ← ∅
2  foreach occurrence c ∈ S_ciph do
3      if c ∉ D then
           D.put(c, 1)
4      else
           D.put(c, D[c] + 1)
5  foreach (d, P) ∈ BB_work.M do
6      foreach (p, Q) ∈ P do
7          foreach (qKey, B) ∈ Q do
8              i ← 0
9              while i < |B| do
10                 c ← B[i]
11                 if c ∉ D then
12                     i ← i + 1
13                 else
14                     D.put(c, D[c] − 1)
15                     if D[c] = 0 then D.remove(c)
16                     B.removeAt(i)
17 return D = ∅
```

VerifyPlaintexts checks that each plaintext vote in the decrypted ballotbox is correctly encoded and corresponds to a valid candidate choice under its district and question.

Algorithm 63: VerifyPlaintexts - Link to Code

Context:

`conf`$_{\texttt{AA}}$: AppConfig. contains n_v (challenge length), `verifyInvalid` (flag for verifying invalid ciphertext), $P_{\texttt{EG}}, \texttt{M}_{\texttt{DC}}, \texttt{conf}_{\texttt{AA}}.\texttt{crt}_{\texttt{EG}}.pk$

Input:

`BB`$_{\texttt{work}}$: AnonBallotbox
DP : List<DecryptionProof>
`expect` : Bool

Output:

b : Bool: true if verifies, false otherwise

Operation:

```
                                                          /* maps ct to pt */
1   V : Map<Byte[], Byte[]> ← ∅
                                        /* maps ct to number of occurrence */
2   N : Map<Byte[], Int> ← ∅
3   foreach dp ∈ DP do
4       old ← V.put(dp.c, dp.m)
5       if old ≠ null then
6           if dp.c ∉ N then
                N.put(dp.c, 2)
7           else
                N.put(dp.c, N[dp.c] + 1)
8   ok ← true
9   foreach (d, P) ∈ BB_work.M do
10      foreach (p, Q) ∈ P do
11          foreach (qKey, B) ∈ Q do
12              i ← 0
13              while i < |B| do
14                  b ← B[i]
15                  m ← V.remove(b)
16                  N.remove(b)
17                  if m = null then
                        i ← i + 1
18                  else
19                      actual ← IsValidPlaintextChoice(conf_AA.P_EG, m, d, conf_AA.M_DC)
20                      ok ← ok ∧ (expect = actual)
21                      B.removeAt(i)
22  return ok ∧ V = ∅ ∧ N = ∅
```

IsValidPlaintextChoice checks whether a plaintext byte string decoded by AA is a valid candidate choice for the supplied district.

Algorithm 64: IsValidPlaintextChoice - Link to Code

Input:

- $P_{\texttt{EG}}$: `ElGamalParameters`
- m : `Byte[]`. Decrypted plaintext value
- d : `String`. District identifier
- $\texttt{M}_{\texttt{DC}}$: `Map<String, CandidateList>`. Maps district to candidate list

Output:

- b : `Bool`: true if the plaintext is a valid choice in district d, false otherwise

Operation:

```
1 choice ← DecodeModPElement(P_EG, Int(m))
2 if d ∉ Keys(M_DC) then return false
3 if choice ∉ CandidateIDs(M_DC[d]) then return false
4 return true
```

VerifyValidConsistency checks consistency between the set of valid decrypted plaintexts and the tally input used to compute the election result. On the working map M, removeFirst removes one matching choice from its leaf lists, and size counts all remaining choices.

Algorithm 65: VerifyValidConsistency - Link to Code

Context:

- $\texttt{conf}_{\texttt{AA}}$: `AppConfig`

Input:

- pk : `Int`
- DP : `List<DecryptionProof>`
- $\texttt{BB}_{\texttt{plain}}$: `PlainBallotbox`
- n : `Int`

Output:

- b : `Bool`: true if verifies, false otherwise

Operation:

```
1 BB'_plain ← Clone(BB_plain)
2 foreach dp ∈ DP do
3     m ← dp.m
4     candidate ← DecodeModPElement(conf_AA.P_EG, m)
5     if BB'_plain.M.removeFirst(candidate) = null then return false
6 return BB'_plain.M.size() = n
```

VerifyTally checks that the published tally and plaintext ballotbox have the same election ID and that the tally equals the count recomputed from the plaintext ballotbox.

Algorithm 66: VerifyTally - Link to Code

Input:
- $\texttt{BB}_{\texttt{plain}}$: PlainBallotbox
- T : Tally

Output:
- b : Bool: true if verifies, false otherwise

Operation:

```
1  if T.eid ≠ BB_plain.eid then return false
2  if Keys(T.M) ≠ Keys(BB_plain.M) then return false
3  foreach (d, P) ∈ T.M do
4      if Keys(P) ≠ Keys(BB_plain.M[d]) then return false
5      foreach (p, C) ∈ P do
6          foreach (candidate, cnt) ∈ C do
7              L_candidate ← BB_plain.M[d][p].removeAll(candidate)
8              if cnt ≠ |L_candidate| then return false
9  foreach (d, P) ∈ BB_plain.M do
10     foreach (p, C) ∈ P do
11         if |C| ≠ 0 then return false
12 return true
```

4.5.1.5 IntegrityTool checks PA-stage consistency by comparing the input and output of each PA ProcessBallotbox stage. The tool originates from the set-consistency check proposed by Treier and Düüna [38], which IVXV incorporated into the Audit App in 2024. It verifies the following audit trail:

1. invalid ballots (log) + valid ballots (log) = original BB
2. anon BB matches with anon log
3. anon ballots (log) + squashed ballots (log) + revoked ballots (log) = valid ballots (log). It follows that the anonymised ballots are part of the original ballotbox.

Algorithm 67: IntegrityTool - Link to Code

Context:

$\mathsf{Sum}(L) = \sum_{i=0}^{|L|-1} L_i$: computes the sum of all list elements

$\mathsf{SumHashes}(L$: `List<RawBallotEntry>`$) = \sum_{i=0}^{|L|-1} \mathsf{Hash}(L_i.c)$: computes the sum of all `RawBallotEntrys`' ciphertext's hash digest

Input:

$R_{\mathtt{BF}}$: `List<BallotFileRecord>`: a list of BallotFileRecord provided by collector

$\mathtt{BB}_{\mathtt{anon}}$: `AnonBallotbox` the post-processing ballotbox (output of Processor)

$\mathtt{log}_{\mathtt{err}}$: `List<PAErrorLogEntry>`: log of all rejected ballots

$\mathtt{log}_{\mathtt{acc}}$: `List<PALogEntry>`: log of all accepted/valid received ballots

$\mathtt{log}_{\mathtt{filter}}$: `List<PALogEntry>`: log of all squashed and revoked ballots

$\mathtt{log}_{\mathtt{anon}}$: `List<PALogEntry>`: log of all anonymised phase

Output:

b : `Bool`: true if verifies, false otherwise

Operation:

1 $\mathtt{L}_{\mathtt{BB}}$: `List<RawBallotEntry>` is CS's export $R_{\mathtt{BF}}$ parsed as a list of `RawBallotEntry`.

2 $\mathtt{L}_{\mathtt{BBAnon}}$: `List<RawBallotEntry>` is obtained by flattening the ciphertext entries in $\mathtt{BB}_{\mathtt{anon}}.M$ into `RawBallotEntry` records. Since $\mathtt{BB}_{\mathtt{anon}}$ contains no $id_{\mathtt{ballot}}$, that field is null

3 $\mathtt{sumBb}_{\mathtt{all}} \leftarrow \mathsf{SumHashes}(\mathtt{L}_{\mathtt{BB}})$

4 $\mathtt{sumBb}_{\mathtt{anon}} \leftarrow \mathsf{SumHashes}(\mathtt{L}_{\mathtt{BBAnon}})$

5 $\mathtt{sum}_{\mathtt{acc}} \leftarrow \mathsf{Sum}([e.h \mid e \in \mathtt{log}_{\mathtt{acc}}])$

6 do the same to obtain $\mathtt{sum}_{\mathtt{filter}}, \mathtt{sum}_{\mathtt{anon}}$ from corresponding logs

7 $\mathtt{sum}_{\mathtt{vi}} \leftarrow \mathsf{SumValidInvalid}(\mathtt{L}_{\mathtt{BB}}, \mathtt{log}_{\mathtt{acc}}, \mathtt{log}_{\mathtt{err}})$

8 **if** $\mathtt{sum}_{\mathtt{vi}} = \perp$ **then return** false

9 $(h_{\mathtt{valid}}, h_{\mathtt{invalid}}) \leftarrow \mathtt{sum}_{\mathtt{vi}}$

10 **if** $h_{\mathtt{valid}} \neq \mathtt{sum}_{\mathtt{acc}}$ **then return** false

11 **if** $h_{\mathtt{valid}} + h_{\mathtt{invalid}} \neq \mathtt{sumBb}_{\mathtt{all}}$ ▷ 1
then return false

12 **if** $\mathtt{sum}_{\mathtt{anon}} \neq \mathtt{sumBb}_{\mathtt{anon}}$ ▷ 2
then return false

13 **if** $\mathtt{sum}_{\mathtt{anon}} + \mathtt{sum}_{\mathtt{filter}} \neq \mathtt{sum}_{\mathtt{acc}}$ ▷ 3
then return false

14 **return** true

SumValidInvalid computes hash sums for the accepted and rejected ballots, then compares their total with the same aggregate over PA's input ballotbox (CS's export).

Algorithm 68: SumValidInvalid - Link to Code

Input:
- $\mathsf{L}_{\mathtt{BB}}$: `List<RawBallotEntry>` is CS's export $R_{\mathtt{BF}}$ parsed as a list of `RawBallotEntry`.
- $\mathtt{log}_{\mathtt{acc}}$: `List<PALogEntry>`: log of all accepted/valid received ballots
- $\mathtt{log}_{\mathtt{err}}$: `List<PAErrorLogEntry>`: log of all rejected ballots

Output: $(h_{\mathtt{valid}} : \mathtt{Int}, h_{\mathtt{invalid}} : \mathtt{Int})$ sum of the valid and invalid digests, or $\perp$ on failure

Operation:

1 $h_{\mathtt{invalid}}, h_{\mathtt{valid}} \leftarrow 0$
2 $H_{\mathtt{acc}} \leftarrow [e.h \mid e \in \mathtt{log}_{\mathtt{acc}}]$
3 **if** $|H_{\mathtt{acc}}| \neq |\mathsf{Set}(H_{\mathtt{acc}})|$ **then** show a warning of recurrences
4 $\mathsf{S}_{\mathtt{invalidID}} \leftarrow \{e.id_{\mathtt{ballot}} \mid e \in \mathtt{log}_{\mathtt{err}}\}$
5 **foreach** $b \in \mathsf{L}_{\mathtt{BB}}$ **do**
6 $h \leftarrow \mathsf{Hash}(b.c)$
7 $\mathtt{seen} \leftarrow 0$
8 **if** $\mathsf{S}_{\mathtt{invalidID}}.\mathsf{remove}(b.id_{\mathtt{ballot}}) \neq$ **null then**
9 $h_{\mathtt{invalid}} \leftarrow h_{\mathtt{invalid}} + h$
10 $\mathtt{seen} \leftarrow \mathtt{seen} + 1$
11 **if** $H_{\mathtt{acc}}.\mathsf{removeFirst}(h) \neq$ **null then**
12 $h_{\mathtt{valid}} \leftarrow h_{\mathtt{valid}} + h$
13 $\mathtt{seen} \leftarrow \mathtt{seen} + 1$
14 **if** $\mathtt{seen} \neq 1$ **then return** $\perp$
15 **if** $H_{\mathtt{acc}} \neq \emptyset$ *or* $\mathsf{S}_{\mathtt{invalidID}} \neq \emptyset$ **then return** $\perp$
16 **return** $(h_{\mathtt{valid}}, h_{\mathtt{invalid}})$

4.6 Channel Security

This section summarises the data flow between protocol participants according to our reconstructed specification, providing a basis for assessment of data-channel security.

Since IVXV does not provide a public specification of data-channel security, ambiguities concerning inter-phase channels remain. For instance, data produced during the voting phase, such as collector service exports and registration logs, are later used as tally-phase inputs, but the public documentation does not fully specify which party transfers these artefacts, or by which channel. The relevant tally-phase implementations load these inputs from local file paths and do not appear to have any network-transfer capability between participants. We therefore treat inter-phase channels as deployment-specific, and, unless otherwise stated, assume that an authorised operational party such as the EA relays these artefacts between participants. We annotate these channels as OoB (Out-of-band).

Tables 8, 9, and 10 present the observed data flow for each protocol phase. We exclude application and code releases for server-side applications/services (CS, PA, KA, AA) and client-side applications (VCD, VVD) from the modelled setup-phase data channels. We assume all participants have already been deployed and therefore only model the communication between them.

Due to the remaining deployment-specific ambiguity and the lack of public security claims, it is difficult to state precise security guarantees for each IVXV channel. Nonetheless, two main general observations follow:

- For OoB channels, the public specification does not define a uniform channel-security mechanism. In many cases, the transferred artefacts are not protected by any application-level security checks, e.g., a signature binding the artefact to its sender and its content. The relaying party, which we model as EA, can therefore modify or replace these artefacts.

- For several online channels, transport protection is deployment-specific. In particular, whether HTTPS is used depends on the configured URL, and the implementation generally relies on the platform's default trust anchors rather than election-specific trust anchors. The implications of these observations are discussed in the security analysis.

Message Name	Sender	Receiver	Message Content	Context Data	Channel
InitElection					
`CSCert`	CS	EA	Self-signed certificates CS uses for TLS and microservices	–	OoB
`RegCert`	Reg	EA	CA-issued certificates for Reg response verification	–	OoB
`TrustRoot`	EA	CS, KA, PA, AA, VCD, VVD	`CACert` and `RegCert` used for vote and registration response validation	–	OoB
KeyGen					
`KeyGenParams`	EA	KA	$P_{\texttt{EG}}$, l, $P_{\texttt{Th}}$	–	Offline
$\texttt{KeyHolderData}_i$	KH_i	KA	Smartcard containing index i	`KeyGenParams`	Offline
`ElectionCerts`	KA	EA	$\texttt{crt}_{\texttt{EG}}$, $\texttt{crt}_{\texttt{RSA}}$	`KeyGenParams`	Offline
$\texttt{KeyShares}_i$	KA	KH_i	$D[i]$ decryption and signing shares	$\texttt{KeyHolderData}_i$, `KeyGenParams`	Offline
SetupElection					
`CSConfig`	EA	CS	`CSCert`, ElGamal ciphertext-validation parameters, and collector-specific configuration data	–	OoB
`AppConfig`	EA	KA, PA, AA, Mixer	Relevant election artefacts and role-specific application configuration data; $\texttt{crt}_{\texttt{RSA}}$ is supplied to AA for the modelled result-signature check	–	OoB
`AppConfig`	EA	VCD, VVD	Client configuration retrieved before or during voting	–	HTTPS

Table 8: Setup phase data channels

Message Name	Sender	Receiver	Message Content	Channel / Protection
SendBallot				
VoterProfile	Voter	VCD	phone, id_{voter}	UI
AuthReq	VCD	CS	RPC.Auth, phone, id_{voter}	TLS
AuthReq'	CS	IDP	RPC.Auth, phone, id_{voter}, r	HTTP(S)
AuthChallenge	IDP	Sign	RPC.Auth, r	Provider-specific
AuthResp	IDP	CS	sid_{auth}	HTTP(S)
AuthResp'	CS	VCD	sid_{CS}, sid_{auth}, r, dataToken	TLS
AuthDisplay	VCD, Sign	Voter	RPC.Auth, r	UI
AuthApproval	Voter	Sign	Approval of signing	UI
AuthSig	Sign	IDP	sig_{auth}	Provider-specific
AuthStatusReq	VCD	CS	RPC.AuthStatus, sid_{CS}, sid_{auth}	TLS
AuthStatusReq'	CS	IDP	RPC.AuthStatus, sid_{auth}	HTTP(S)
AuthResult	IDP	CS	crt_{auth}, sig_{auth}	HTTP(S)
AuthStatusResp	CS	VCD	sid_{CS}, authToken	TLS
ChoicesReq	VCD	CS	RPC.VoterChoices, sid_{CS}, authToken	TLS
ChoicesResp	CS	VCD	choices	TLS
ChoicesDisplay'	VCD	Voter	choices	UI
ChoiceSelection	Voter	VCD	Selected choices M	UI
GetCertReq	VCD	CS	RPC.GetCert, sid_{CS}, authToken	TLS
GetCertReq'	CS	IDP	RPC.GetCert, phone, id_{voter}	HTTP(S)
SignCert	IDP	CS	crt_{sign}	HTTP(S)
GetCertResp	CS	VCD	sid_{CS}, crt_{sign}	TLS
SignReq	VCD	CS	RPC.Sign, sid_{CS}, authToken, h	TLS
SignReq'	CS	IDP	RPC.Sign, phone, id_{voter}, h	HTTP(S)
SignChallenge	IDP	Sign	RPC.Sign, h	Provider-specific
SignResp	IDP	CS	sid_{sign}	HTTP(S)
SignResp'	CS	VCD	sid_{CS}, sid_{sign}	TLS
SignDisplay	VCD, Sign	Voter	RPC.Sign, h	UI
SignApproval	Voter	Sign	Approval of signing	UI
BallotSig	Sign	IDP	sig_{sign}	Provider-specific
SignStatusReq	VCD	CS	RPC.SignStatus, sid_{CS}, sid_{sign}	TLS
SignStatusReq'	CS	IDP	RPC.SignStatus, sid_{sign}	HTTP(S)
SignResult	IDP	CS	sig_{sign}	HTTP(S)
SignStatusResp	CS	VCD	sid_{CS}, sig_{sign}	TLS
ConfirmBallot				
VoteReq	VCD	CS	RPC.Vote, sid_{CS}, authToken, BC, choices	TLS
RegReq	CS	Reg	req_{tsp}, req_{ocsp}	Election-specific
RegResp	Reg	CS	resp_{tsp}, $\text{resp}_{\text{ocsp}}$	Election-specific
VoteResp	CS	VCD	vid, sid_{CS}, resp_{tsp}, $\text{resp}_{\text{ocsp}}$	TLS
VerifyCast				
VerifTransfer	VCD	VVD	r, vid, sid_{CS}	QR
VerifReq	VVD	CS	vid, sid_{CS}	TLS
VerifResp	CS	VVD	BC', resp_{tsp}, $\text{resp}_{\text{ocsp}}$, choices	TLS
BallotDisplay	VVD	Voter	signerCN, V	UI
ChoicesDisplay	VVD	Voter	M	UI
ExportVotingData				
CSStore	CS	EA	R_{BF}	OoB
RegStore	Reg	EA	R_{TSP}	OoB

Table 9: Voting phase data channels

Message Name	Sender	Receiver	Message Content	Context Data	Channel
ProcessBallotbox					
`TallyPhaseInput`	EA	PA	`CSStore`, `RegStore`, `revokeList`	–	OoB
`PAOutput`	PA	EA	application logs, $\texttt{BB}_{\texttt{anon}}$	–	OoB
MixBallotbox					
`AnonBallotbox`	PA	Mixer	$\texttt{BB}_{\texttt{anon}}$	–	deployment-specific
`MixerOutput`	Mixer	EA	$\texttt{BB}_{\texttt{mixed}}, \pi$	–	OoB
DecryptAndTally					
$\texttt{KeyShares}_i$	KH_i	KA	$D[i]$	–	Offline
`MixedBallotbox`	EA	KA	$\texttt{BB}_{\texttt{mixed}}$	–	OoB
`ElectionResult`	KA	EA	ER	–	OoB
RunAuditTool					
`AuditPhaseInput`	EA	AA	`CSStore`, `TallyPhaseInput`, `PAOutput`,`MixerOutput`, `ElectionResult`	–	Offline

Table 10: Tally (and Audit) phase data channels

4.7 TLA Model

We also provide a TLA+ model of the reconstructed IVXV protocol, available at `ivxv-model`. The model covers setup, voting, and tallying at a similar abstraction level to the specification. Cryptographic operations are symbolic, while security-relevant data flow, data structures, algorithms, and states are modelled explicitly (see `scope.md` in the repository for modelling scope details). The protocol participants are modelled faithfully.

We use TLC to check executability and consistency invariants for each phase. For example, in an honest run, the voter verifies and accepts the latest registered ballot retained by CS, and the e-votes of voters in the revocation list are filtered out during tally. The model also includes a concrete election configuration and phase scenario. The voting-phase scenario specifies which voters vote for which choices and in what order. We define corresponding invariants that compare the protocol output with the expected outcome, for example that Alice wins the election in the modelled scenario.

The model is not intended as a formal proof of IVXV security, but as a sanity check of the specified protocol and as a basis for modelling the specific attack scenarios presented in the next section.

5 Security Analysis

In analysing the security of the IVXV system we have used the security properties in the Swiss regulations [33]; we have not used the threat models in the regulations, as the IVXV system differs in the protocol participants, rather we have analysed vulnerabilities across a range of threat models. We have made this choice for three reasons:

1. There is no protocol security model for the IVXV system so it was necessary to choose an external one.
2. Both the Swiss and Estonian systems forgo a public bulletin board which is assumed by most of the academic definitions.
3. By choosing the same security properties we ease comparison between the two most prominent examples of internet voting in Europe.

The Swiss regulations consider four security properties; of these, only privacy is defined similarly to the academic literature. The other three are all significantly impacted by the absence of a public bulletin board; without this, the properties are based on the notion of the honest (online) control component. We will expand on the meaning of the properties in the relevant sections. We will also include an analysis of coercion-resistance.

The remainder of this section is structured as follows. We first summarise currently known attacks and their status (Sec. 5.1), then define the vulnerability categories and annotations used in our analysis (Sec. 5.2). We then state our trust assumptions (Sec. 5.3). The following five subsections introduce the security properties considered in this paper and analyse the threat models under which each property holds. Section 6 then compares IVXV with other e-voting systems. We do not construct security proofs for the system in any formal and systematic way, though our TLA+ formalisation captures some security properties. Given the unclear security requirements and breadth of current issues, formalising the security of the IVXV seems premature.

5.1 Known Attacks

We first assess the current exploitability of previously identified attacks in light of our protocol analysis.

5.1.1 Insufficient Ciphertext Validation [32]

During the Voting phase of IVXV, VCD (Vote Casting Device) encrypts the voter's choice v under the election public key y using ElGamal: $(c_0, c_1) \leftarrow (g^r, y^r v)$. During the Verify Cast phase, VVD (Vote Verification Device) receives randomness r from VCD, and (c_0, c_1) from CS (Collector Service), to check ciphertext validity and decryption correctness. However, IVXV did not verify whether $g^r = c_0$. This allows a malicious VCD to modify a voter's intended choice while passing self-verification. Sutopo et al. discovered 3 concrete attacks:

- vote discarding attack: VCD uses different randomness values in the vote ciphertext, e.g., $(g^r, y^{r'} v)$, sends it to CS, then sends r' to VVD. VVD decrypts c_1 correctly and recovers its intended vote $v \leftarrow y^{r'} v / y^{r'}$, yet the actual vote sent to CS would be decrypt to an invalid group element during tallying, leading to it being silently discarded.
- Vote substitution attacks (with secret key): If the VCD knows the election secret key, it can construct ciphertexts that decrypt to a different vote v' while verifying as v.
- Vote substitution attacks (without secret key): Without knowing the election secret key, if the VCD knows a discrete logarithm relation between vote choices ($v/v' = y^s$), it can adjust the randomness sent to the VVD (e.g., $r' = r - s$) so that verification outputs v while the tally decrypts to v'.

5.1.1.1 Current Impact

This vulnerability has been patched as IVXV has introduced an extra ElGamal ciphertext validation logic in VVD. During the IV process, VVD verifies whether $c_0^r = c_1$, where c_0, r are provided by VCD and c_1 is obtained from CS

5.1.2 Revoting Attack [28]

IVXV provides a self-verification service by which a voter checks their cast vote, as described in Section 4.3.3. This individual-verifiability mechanism was first introduced in the 2013 Estonian system [22] and revised in the 2016–2017 IVXV redesign [21].

Pereira discovered a revoting attack in 2021 [28]. The attack is performed by a malicious VCD as follows:

- VCD submits a vote of voter's intent v. After receiving the *vid* from CS, and before displaying it to the voter, it crashes (or otherwise reports that voting failed), so that the voter proceeds to revote.
- VCD submits its own intent v' for the second vote, but when the voter performs self-verification, the VCD sends VVD the *vid* of the first vote. VVD successfully retrieves the first vote of the voter and verifies, despite the latest vote that will be tallied is v'.

5.1.2.1 Current Impact

Since commit 4916080 (2021, IVXV 1.7.7), each election configuration has a `latestonly` field. When set to `true`, CS returns a vote during the self-verification step only if its *vid* matches the voter's latest vote (see Alg. 28), so the specific variant above — in which the voter is made to verify an already-overwritten ballot — is no longer exploitable. Pereira notes,

however, that this mitigation can still be bypassed: a corrupt CS can withhold registration of the second ballot until after the verification window, and a VCD that captures the voter's eID PIN can silently submit the second ballot once the window has closed (the "Ghost Click" variant), so that the verified ballot genuinely was the latest at verification time. These configuration-independent variants correspond to the reordering attack discussed in Section 5.1.3.

5.1.3 Reordering Attack [28, 4]

This reordering variant — delaying a ballot so that the tallied vote differs from the verified one, independently of the `latestonly` setting — was already described by Pereira for a corrupt collector [28] (Section 5.1.2), and was independently identified as a reordering attack by Baloglu et al. [4]. It remains exploitable under any election configuration.

The attack can be performed by a corrupt VCD, CS, or a Network attacker who can delay messages between VCD and CS. The attack assumes a voter who was coerced to sign a vote v_1, the attacker-controlled component then delays v_1. Later, after the voter revotes for a different candidate v_2 and completes self-verification, the attacker submits the delayed v_1 and overwrites v_2.

5.1.3.1 Current Impact IVXV has not addressed this vulnerability. However, a malicious network or VCD cannot indefinitely delay a signed vote. Voting requires the VCD to maintain an active, authenticated session with CS, as CS would reject a vote request if the corresponding voting session has expired (see $t_{\texttt{TTL}}$ in Dat. 6). The default value is 60 seconds, which is significantly smaller than the 30-minute verification window (see $t_{\texttt{ver}}$ in Dat. 6). Therefore, a malicious CS would make a stronger reordering attack adversary as this timeout is enforced by itself.

5.1.4 Shifting [26]

Müller discovered a privacy attack on IVXV that allows an adversary who controls one voter to reveal information of another voter's vote, given that the adversary can also see the encrypted ballot $c \leftarrow \mathsf{Enc}(y, v)$ of the victim. The adversary submits a ballot with the structure $c' \leftarrow c \cdot \mathsf{Enc}(y, v_a^{-1} \cdot v_b)$. If vote c encrypts v_a, then the adversary's vote c' becomes a valid ElGamal ciphertext that encrypts v_b:

$$\begin{aligned} c' &= c \cdot \mathsf{Enc}(y, v_a^{-1} \cdot v_b) \\ &= (g^r, y^r v_a) \cdot (g^{r'}, y^{r'} v_a^{-1} v_b) \\ &= (g^{r''}, y^{r''} v_b) \end{aligned}$$

Otherwise, if $v \neq v_a$, then c' will encrypt a random group element and be discarded in tally with overwhelming probability. Therefore, the adversary can infer whether the target voter voted for v_a based on the number of v_b in election result.

5.1.4.1 Current Impact This vulnerability is not mitigated. However, the one-vote bias is likely small in a state-wise election even if there exists an unpopular choice v_b.

5.1.5 Pfitzmann [26]

Müller also discovered a Pfitzmann attack that allows an adversary who controls one voter; sees the raw election result; and sees the encrypted ballots of voters to learn the underlying choices of multiple voters.

To learn the choice of a set of voters' ballots: $c_1, \ldots, c_n$ the adversary chooses a set of random coefficients $a_1, \ldots, a_n$, and create a vote $c' = \prod_{i=1}^{n} c_i^{a_i}$. During tally, c' will be decrypted into $v' = \prod_{i=1}^{n} v_i^{a_i}$, which is likely an invalid choice. By using the same set of coefficients, the adversary can try different sets of possible choices $m_1, \ldots, m_n$ until finding one combination such that $v' = \prod_{i=1}^{n} m_i^{a_i}$.

5.1.5.1 Current Impact The attack assumes an adversary who can observe invalid votes in the raw election result. However, the raw election result is only visible to KA and Audit. Therefore, to mount the attack, the adversary must control a protocol participant that knows the correlation between voters and encrypted votes (CS or PA), and a protocol participant that sees the raw election result (KA or Audit).

5.1.6 Ballot Copying [4]

Baloglu et al. found a statistical privacy attack that allows an adversary with voting power and the ability to observe one encrypted ballot $b = \{c, \sigma\}$ (simplified notation of a ballot that only contains a ciphertext and a signature component) to infer its underlying choice with non-negligible probability. The adversary submits $b' = \{c, \sigma'\}$ by copying c and re-signing it with the adversary's voter certificate. This attack would hence increase the number of the underlying choice in ballot b in the election result by 1.

5.1.6.1 Current Impact This vulnerability is not mitigated; its practical impact is limited as the bias would be small in a state-wise election.

5.1.7 Ballot Substitution During Anonymisation [38, 36]

Treier and Düüna [38] observed that, once PA strips signatures and registration data during AnonymizeBallotbox, nothing in the remaining ballotbox binds its contents to the voting-phase records: anyone with the election public key can produce a well-formed ciphertext, so a dishonest PA could replace ballots in the anonymised ballotbox. At the time, the audit verified only the signed checksums of PA's input files, which say nothing about the correctness of its outputs. They proposed checking that the input ballots, less those rejected as improperly encrypted, equal the union of the anonymised output and the ballots invalidated by squashing and revocation, and the 2024 IVXV release added an Audit App tool based on their script; this is the IntegrityTool (Alg. 67) analysed under UV-V1.

Treier [36] then tested the deployed tool against fault-seeded variants of a processing run. One variant (F26-2), in which a new ballot is added to the anonymised output and the stage logs and aggregate counters are edited so that the published digest totals remain consistent, was not flagged. We attribute this to the granularity of IntegrityTool's checks: only the CheckTool stage output is matched per ballot against CS's export (check 1 in Alg. 67), whereas the subsequent filter and anonymisation stages are checked only through aggregate digest sums (checks 2 and 3), $\texttt{sum}_{\texttt{anon}} + \texttt{sum}_{\texttt{filter}} = \texttt{sum}_{\texttt{acc}}$, in which $\texttt{sum}_{\texttt{acc}}$ is checked against CS's export and $\texttt{sum}_{\texttt{anon}}$ against PA's anonymised ballotbox, but $\texttt{sum}_{\texttt{filter}}$ is derived from $\texttt{log}_{\texttt{filter}}$ alone. A corrupt PA that edits the anonymised ballotbox together with $\texttt{log}_{\texttt{anon}}$ and $\texttt{log}_{\texttt{filter}}$ therefore preserves both equations.

5.1.7.1 Current Impact The original substitution attack is mitigated by IntegrityTool; the F26-2 variant is not mitigated in the version we analysed. It shares its root cause with UV-V1 (Section 7.1.7): IntegrityTool binds aggregates rather than individual ballots across PA's pipeline stages.

5.1.8 Summary

One interesting takeaway from the current status of the known attacks is the implicit threat model it shows. Two of the attacks the designers have addressed — insufficient ciphertext validation (fully, Section 5.1.1) and the revoting attack (in its basic form, Section 5.1.2) — target individual verifiability against a corrupt vote-casting device, indicating that IVXV treats the voter's own device as untrusted for IV and responds to concrete breaks of that kind. The third, ballot substitution by PA (Section 5.1.7), was addressed by adding IntegrityTool to the Audit App, which shows that the processing application is likewise not fully trusted; the fact that the added check binds aggregates rather than individual ballots, and so leaves the F26-2 variant and UV-V1 open, suggests the fix targeted the reported attack rather than the underlying trust gap.

The reordering attack (Section 5.1.3), which achieves a similar effect by exploiting the gap between when a ballot is submitted and when it is registered, remains unmitigated, and is strongest for a corrupt collector or registration service that can evade the session timeout bounding a merely network- or client-level attacker. The three privacy attacks — shifting, Pfitzmann, and ballot copying (Sections 5.1.4–5.1.6) — are likewise left unaddressed, and all share one precondition: the adversary must be able to observe the submitted ciphertexts. Their acceptance reflects a reliance on the confidentiality of the ballotbox, which IVXV keeps non-public, as an implicit privacy assumption, rather than on cryptographic protection against an adversary who does see the ballots. Taken together, the fixes and the deliberate non-fixes sketch a consistent, if unstated, threat model: the client and, to a lesser extent, the processing application are distrusted for verifiability, while the confidentiality of the ballotbox is trusted for privacy and server-side reordering is left out of scope.

5.2 Attack Annotations

Section 3.2 defines the general categorisation used in this paper: attacks are classified as trivial, known, or new, and annotated with additional labels. This section defines those labels and explains how they should be interpreted. Table 11 summarises the labels. The labels make the analysis comparable across security properties, without implying that every listed violation has the same severity.

Category	Labels and interpretation
Mechanism	**R**: reordering; the voter's perceived submission order differs from the order used to select the effective latest ballot. **BD**: ballot dropping; a ballot accepted during voting or verification is later removed from the tally-eligible set. **RV**: revoting; the adversary abuses IVXV's override mechanism so that the voter signs, misses, or misunderstands an additional vote. **SUB**: substitution; the voter verifies a plaintext choice without an enforced binding to the ballot signer identity.
Impact	**LI**: limited impact, e.g., limited tally bias or only impact specific voters. **LC**: limited condition; exploitable only under specific conditions outside the adversary's control. **LT**: limited time; exploitable only within a small timing window.
Detectability	**D**: detectable after the fact by some check, warning, or abnormal trace.

Table 11: Vulnerability categorisation labels used in the threat-model tables and findings.

5.2.1 Impact Labels

- Limited Impact (LI): The attack can only achieve a limited impact. For example, the attack can only introduce a small bias to the tally result; the attack can only affect a small set of voters, e.g., a corrupt VCD can only affect its owner's votes.
- Limited Time (LT): The attack can only be exploited within a small window after its prerequisite conditions are met.
- Limited Condition (LC): The vulnerability is only exploitable under specific election configurations, e.g., multi-vote, `latestonly` = `false`.

5.2.2 Mechanism Labels

- Reordering (R): To define reordering attack, it is important to clarify two notions: ballot submission time (when VCD sends signed ballot to CS) and registration time (when CS register a received signed ballot to Reg). In tally phase, for each voter, only the ballot with the latest registration time is retained. A reordering attack is performed by an adversary who manipulates some of a voter's ballots ($[b_1, b_2]$) such that $t_1^{\texttt{sub}} < t_2^{\texttt{sub}}$, $t_1^{\texttt{reg}} > t_2^{\texttt{reg}}$, while t_2 remains self-verifiable. As a result, the voter believes that b_2 is the ballot that will be tallied, since it was submitted later and passes self-verification, even though b_1 is ultimately tallied because it has a later registration timestamp.
- Ballot Drop (BD): Ballot Dropping attacks might sound similar to Reordering attacks intuitively, as dropping the latest ballot allows an older ballot to be tallied. However, their mechanisms differ: In reordering attacks, the adversary changes the effective ordering of a voter's ballots by causing the registration time order to differ from the submission time order; in ballot dropping attacks, the adversary does not manipulate this ordering. Instead, it

removes a ballot from the set of ballots that survive to tally, either directly or indirectly (e.g., tampering with a ballot such that it passes all voting phase checks but will be rejected during tally phase). Consequently, for all ballots that remain in the system, the natural ordering holds.

- Revoting (RV): Revoting attacks differ from Reordering attacks in the underlying mechanisms. In revoting attacks, the natural ordering of a voter's ballots ($[b_1, b_2]$) is preserved: $t_1^{\texttt{sub}} < t_2^{\texttt{sub}}$ implies $t_1^{\texttt{reg}} < t_2^{\texttt{reg}}$. A Revoting attack adversary does not tamper with the ordering of ballots, but exploits the revoting feature itself: the attacker tricks the victim voter into signing two ballots $[b_1, b_2]$ while the voter believes that they have only signed one. After the ballot the voter believes they submitted has been self-verified (e.g., after the verification interval has passed), the adversary silently submits the later ballot b_2, overwriting the voter's latest vote.
- Substitution (SUB): Substitution attacks break the identity binding between a voter and the ballot they believe they submitted, creating a "whose vote is being verified" problem. A voter under such attacks would believe that they have successfully cast a vote and that it is cast as intended, even though the ballot's cryptographic materials attribute it to a different voter's identity. In that case, the voter has not actually voted (no vote in the system binds to their voter identity), despite getting a successful verification outcome.

5.2.3 Detectability Labels

- Detectable (D): the attack is detectable afterwards, for example because it causes an explicit system check to fail (after the privacy breach occurs) or because it generates a significant amount of unusual system behaviour; in reality just because an attack could be detected does not mean it would be identified as such.

  Detectability may weaken the relevance of an attack depending on the property it targets:

  IV an attack that is detectable by self-verification is not a successful IV break; an attack that passes self-verification but may be detected by Tally/Audit participants is a successful IV break.

  UV an attack that will be reliably detected by the prescribed audit process (implied by the Audit App implementation and demo audit log) is not a successful UV break; an attack that passes the prescribed audit workflow, but generates unusual trace that could be detected through additional manual investigation is considered a valid UV break.

  Privacy any attack that causes privacy loss is a successful privacy attack, regardless of its detectability.

5.3 Trust Boundary

Across all properties, we allow some voters to be corrupt. We assume the threshold-key assumption holds: at least t key holders (KH) remain honest, while fewer than t corrupt KHs may collude with the adversary. We therefore do not include KH as a separate column in the threat model tables.

For IV, we exclude PA, Mixers, KA, and AA from the threat model, as they only run after the voting phase, and corrupt tally-phase behaviour is analysed under UV.

For UV, we model the participants that produce or verify tally/audit artefacts: EA, CS, Reg, PA, Mixers, KA, and AA. We therefore exclude VCD, VVD, Sign, and IDP because they only participate in the voting phase.

For privacy, we exclude the Sign device and IDP from the collusion table because, in the normal protocol flow, they do not receive ballot plaintext or ciphertext.

5.4 Individual Verifiability

We first analyse individual verifiability which roughly corresponds to recorded-as-cast and cast-as-intended in the academic literature. The basic idea is that if the voter performs the verification check successfully using their second device then the ballot recorded by (honest part of) the system matches their intent. In this case the system receiving votes is distributed between the Registration service (Reg) and the Collector service (CS). Before the analysis, we clarify how we separate IV and UV failures in IVXV.

Invalid Ballots One attack to drop ballots is to cause them to be filtered out as invalid. If a ballot accepted by honest voting phase participant will be later rejected by honest tally participant, we classify this as an IV failure. By contrast, ballot dropping from a tally-phase adversary deviating from the protocol to reject or remove an otherwise valid ballot, is considered a UV failure (Sec. 5.5).

Revoting and Reordering We treat revoting or reordering attacks as IV failures when the revoting or reordering of ballots involves a voter's participation, even if the voter is misled about the action's purpose or result. By contrast, if the counted ballot is submitted without the claimed voter's involvement, we treat it as an eligibility-verifiability failure (Sec. 5.8).

EA	CS	Reg	VCD	VVD	Sign	IDP	Result	Violation
✗	✓	✓	✓	✓	✓	✓	✗	Trivial
✓	✗	✗	✓	✓	✓	✓	✗	Trivial
✓	✓	✓	✗	✗	✓	✓	✗	Trivial (LI)
✓	✓	✓	✓	✓	✗	✓	✗	Trivial (LI, RV), IV-V1 (LI, RV)
✓	✗	✓	✓	✓	✓	✓	✗	Reorder [4] (R), IV-V2 (BD, D)
✓	✓	✗	✓	✓	✓	✓	✗	IV-V3 (LC, LT, R)
✓	✓	✓	✗	✓	✓	✓	✗	Reorder [4] (LT, LI, R), Revote [28] (LI, LC, RV), IV-V4 (LI, SUB), IV-V5 (LI, BD), IV-V6 (LI, BD, D)
✓	✓	✓	✓	✗	✓	✗	✓	

Table 12: IV Results

Table 12 summarises the security analysis result for IV based on our threat model. We will now sketch the violations of the security properties starting with trivial and known attacks before moving into new findings.

5.4.1 Trivial Attacks

5.4.1.1 Corrupt EA The EA provides the certificates identifying the participants to each other. A corrupt EA allows a trivial spoofing attack. One could imagine operational measures which would limit or prevent this kind of attack but these are not specified in the public information.

5.4.1.2 Corrupt CS and Reg

If both the collector and registration service are corrupt there is no honest part of the system keeping track of ballots and so the basic premise of this kind of verifiability is unmet.

5.4.1.3 Corrupt VCD and VVD

If both the vote casting device and vote verification device are corrupt, the voter has no independent means to check what vote was submitted.

5.4.2 Known Attacks

5.4.2.1 Corrupt CS

A corrupt CS can break IV through the Reordering attack discussed in §5.1.3.

5.4.2.2 Corrupt VCD

A corrupt VCD can potentially break IV through the Revoting attack discussed in §5.1.2, if the election config allows verification of non-latest votes; or through the Reordering attack discussed in §5.1.3, if the delayed vote is submitted within the session TTL. See the referred sections for details.

5.4.3 New Findings

5.4.3.1 IV-V1 An adversary controlling an honest voter's Sign device can perform a Revoting attack. See Section 7.1.1 for details.

5.4.3.2 IV-V2 A malicious CS can create a TSP response that is accepted by VVD but rejected by PA due to discrepancies in TSP response verification between VVD and PA (see Alg. 23 for TSP response construction). See Section 7.1.2 for details.

5.4.3.3 IV-V3 The Reg service (Timestamping Authority) can perform ballot re-ordering attacks by exploiting weakness in CS's TSP timestamp validation. See Section 7.1.3 for details.

5.4.3.4 IV-V4 An adversary controlling a corrupt VCD for some honest voter A, and controlling some voter (e.g., voter B), can trick voter A into verifying a ballot signed by B. This attack exploits the fact that the self-verification protocol in IVXV does not enforce an identity binding between the person performing verification and the signer of the retrieved ballot. See Section 7.1.4 for details.

5.4.3.5 IV-V5 A corrupt VCD can break IV by exploiting inconsistent parsing of `ModP` ElGamal ciphertext elements across CS, VVD, and PA. See Section 7.1.5 for details.

5.4.3.6 IV-V6 An adversary controlling a corrupt VCD for an honest voter can perform a ballot-dropping attack by replaying a previously signed BDOC as a fresh vote. The attack exploits an additional tally phase check on recurring TSP response key, allowing the adversary to produce a ballot that gets accepted by CS and passes self-verification, but would later be rejected during tally by PA. See Section 7.1.6 for details.

5.5 Universal Verifiability

Universal verifiability, also called counted-as-recorded, captures that the result of the election matches the ballot records stored by the honest part of the system; in this case, either the collector service or the registration service. Recall that in this case universal verifiability is a misleading term since only the auditors actually see the verification data.

The analysis of universal verifiability is complicated by the way in which the audit process works. Unlike most e-voting systems, the audit in the Estonian system does not abort on most errors but rather reports them for further investigation. We are assured that these errors are all investigated and discussed in the post election reports.

We will describe attacks which do not cause an abort, but the reader should check the details to see what errors/warnings are reported. One might be tempted to suggest that because a warning is given these should not be regarded as vulnerabilities; however, we are studying the IVXV system as a protocol and any area in which it relies on an external system for security whose behaviour is not defined presents a possibly promising area for an attacker. We denote violations as either UV or OB depending on how silent or noisy the attack is.

EA	CS	Reg	PA	Mixers	AA	Result	Violation
✗	✓	✓	✓	✓	✓	✗	Trivial
✓	✗	✗	✓	✓	✓	✗	Trivial
✓	✓	✓	✗	✓	✗	✗	Trivial
✓	✓	✓	✓	✗	✗	✗	Trivial
✓	✗	✓	✓	✓	✓	✗	OB-1
✓	✓	✗	✓	✓	✓	✓	OB-2
✓	✓	✓	✗	✓	✓	✗	Substitution [36], UV-V1 (BD)
✗	✓	✓	✓	✗	✓	✓	OB-10
✓	✓	✓	✓	✓	✓	✗	UV-V2 (LC, BD)

Table 13: UV Results

5.5.1 Trivial Attacks

5.5.1.1 Corrupt EA A corrupt EA can launch multiple trivial attacks that break UV. For example, EA provides the revocation list used by PA to revoke ballots of voters who have voted offline. A corrupt EA can thereby drop any voter's ballots by adding them to the revocation list. More broadly, EA seems to occupy a privileged position as a data channel adversary. In particular, the exact inter-phase (voting → tally → audit) dataflow between protocol participants is not fully specified in IVXV, but the current architecture suggests that the CS and Reg voting phase outputs are collected by EA and later supplied to PA as tally phase input. If so, since there is no integrity mechanism for various phase output files (e.g., CS and Reg service's exports), EA can modify or delete entries before they reach PA.

5.5.1.2 Corrupt CS and Reg If both CS and Reg are corrupt, then PA has no independent source of truth for a voter's submitted ballots. In this case, the adversary can omit a voter's ballots by dropping the corresponding entry from both the CS's and Reg's exports. It can also cause an older valid ballot of the voter to be tallied as the latest ballot by generating a fresh TSP request/response pair for that ballot and updating the entries accordingly (the same effect can be achieved by dropping

the latest ballot). However, they cannot forge a new ballot choice that will pass PA checks and get tallied as PA performs adequate checks over the voter's signature and certificate.

5.5.1.3 Corrupt PA and Audit If Audit and PA are corrupt then PA can change the ballots going into tally without detection.

5.5.1.4 Corrupt Mixers and Audit PA does not check the mix proofs so if Audit is corrupt the mixer can change ballots.

5.5.2 Known Attacks

5.5.2.1 Corrupt PA A corrupt PA can add a ballot to the anonymised ballotbox without detection by editing the filter and anonymisation logs consistently, because `IntegrityTool` checks those stages only through an aggregate digest-sum equation [36]. See Section 5.1.7.

5.5.3 New Findings

5.5.3.1 UV-V1 PA can silently drop ballots in various ways due to inadequate checks on tally integrity in the Audit App. See Section 7.1.7 for details.

5.5.3.2 UV-V2 Under certain election configurations, a corrupt KA can drop ballots silently by marking them as invalid. See Section 7.1.8 for details.

5.5.4 Security Concerns

5.5.4.1 OB-1 A corrupt CS could make a ballot ineligible for tallying by corrupting its content, and making it ineligible for tally, rather than dropping the entry. Whether this is a vulnerability depends on whether ballot-content errors trigger manual investigation. PA's `CheckTool` workflow appears enough to detect and attribute CS misbehaviour, yet certain errors do not seem to be manually checked. See Section 7.2.1 for details.

5.5.4.2 OB-2 An error branch in PA appears unreachable in the current implementation of IVXV. Whether this is a vulnerability depends on whether other `CheckTool` errors trigger manual investigation. See Section 7.2.2 for details.

5.5.4.3 OB-10 Corrupt EA and Mixers may be able to break the Verificatum's shuffle proof soundness in an IVXV election. The potential attack surface arises because several protocol participants do not validate the supplied ModP ElGamal parameters. See Section 7.2.10 for details.

5.6 Privacy

Analysing privacy for the IVXV system is complicated because the process is not based around a public bulletin board; indeed, the rationale for not defending against various well known attacks [26] is based on the inability of an external adversary to see the data which would be required.

EA	VCD	VVD	CS	Reg	PA	Mixers	KA	AA	Result	Violation
✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	P-V1
✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	Trivial (LI)
✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	Trivial (LI)
✓	✓	✓	✗	✗	✓	✓	✓	✓	✗	Trivial (LI,BD)
✓	✓	✓	✗	✓	✗	✓	✗	✓	✗	Trivial
✓	✓	✓	✗	✓	✗	✓	✓	✗	✗	Trivial (BD)
✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	Trivial
✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	Shifting [26] (LI), Copying [4] (LI), P-V2 (LC)
✓	✓	✓	$✗_1$	✓	$✗_1$	✓	$✗_2$	$✗_2$	✗	Pfitzmann [26] (LI)
✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	

Table 14: Privacy Results (✗: required corrupt actor; ✗: disjunctive corruption requirement)

5.6.1 Trivial Attacks

5.6.1.1 Corrupt VCD If the vote casting device is corrupt it trivially learns the votes it casts.

5.6.1.2 Corrupt VVD If the vote verification device is corrupt it trivially learns the votes it verifies.

5.6.1.3 Corrupt CS and Reg If both CS and Reg are corrupt, they can drop ballots and their corresponding registration request entries to cause only a small subset of ballots being tallied, thereby breaking privacy.

5.6.1.4 Corrupt CS/PA and KA If both CS/PA and KA are corrupt, the Privacy is trivially broken, as CS/PA knows the correlation between voters and their encrypted ballots and KA knows the election secret key for decrypting each ballot.

5.6.1.5 Corrupt CS/PA and Audit If Audit is corrupt, any participant that knows the correlation between voters and their encrypted ballots can perform privacy attacks which the corrupted audit will not report.

5.6.1.6 Corrupt PA and Mixers PA can correlate voters and their votes in the anonymous ballotbox; Mixers can collectively correlate shuffle input and output. Together, they can map decrypted votes back to voters.

5.6.2 Known Vulnerabilities

5.6.2.1 Corrupt CS A corrupt CS can perform the Shifting attack described in §5.1.4 or the Copying attack in §5.1.6, to introduce a one-vote bias in the final election result.

5.6.2.2 Corrupt CS/PA (and KA/Audit) A corrupt CS or PA can perform Pfitzmann attack to create a related ciphertext that tracks a set of honest votes, as discussed in §5.1.5, provided that the adversary also has access to the raw election result.

In the current IVXV implementation, the raw result is only available to KA and Audit. Therefore, a corrupt CS or PA must collude with either KA or Audit to carry out the attack. This limits the practical relevance of the attack, because both CS/PA + KA and CS/PA + Audit corruption patterns already imply stronger privacy attacks (see previous section).

5.6.3 New Findings

5.6.3.1 P-V1 Inadequate checks on group parameters allows a corrupt EA or KA to set up an election with weak parameters and break privacy. See Section 7.1.9 for details.

5.6.3.2 P-V2 Reuse of ElGamal randomness for each election question in a multi-question election allows a passive adversary to recover plaintext choices from an encrypted ballot. See Section 7.1.10 for details.

5.7 Coercion Resistance

Coercion-resistance is the ability of the voter to evade coercion; this is normally achieved by arguing that the voter has a means to evade doing what the coercer wants which is indistinguishable to the coercer from compliance. The IVXV aims for a measure of coercion-resistance through re-voting.

5.7.1 Compliance Metric

We model the coercer as an entity who instructs and monitors the target voter to cast a designated electronic ballot. The evidence of compliance is whether this coerced ballot is the voter's final effective ballot. A voter evades coercion if she can later replace that ballot with another vote without the coercer's detection. We assume for the purpose of this analysis that the voter cannot use outside evidence to prove (or disprove) that they voted again.

We make the following assumptions of a CR adversary's capabilities:

- Can only coerce electronic votes (a voter can make at most one physical vote, and it invalidates any electronic votes of the voter).
- The adversary can observe enough of the coerced voting ceremony to be convinced that the designated electronic ballot was cast, the attacker coerces the voter to provide her self-verification materials (displayed as QR code on VCD) to download and verify the cast vote himself.

Under this baseline, an attack succeeds if the adversary learns whether the voter later replaced the coerced electronic ballot with another electronic vote. Some stronger attacks also reveal whether the coerced ballot was invalidated by a physical vote; we annotate the violation scenario with (Phy). We do not model a coercer who obtains the voter's eID transaction history after the election. Treier [37] shows that such a coercer learns whether the voter re-voted from the signing log alone, since every signing transaction is logged and shown to the credential holder; this is a channel in the national eID infrastructure rather than in the IVXV protocol, and lies outside the capabilities we grant our CR adversary.

EA	CS	Reg	IDP	Sign	VCD	VVD	PA	KA	Mixers	AA	Result	Violation
✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	Trivial (Phy)
✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	Trivial
✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	Trivial
✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	Trivial
✓	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	Trivial
✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	
✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	Trivial (Phy)

Table 15: CR Results (✗: required corrupt actor; ✗: disjunctive corruption requirement)

5.7.1.1 EA/PA/Audit (Phy) learns whether a voter revoted by observing the pre-mixed ballotbox and revoke list.

5.7.1.2 CS learns whether a voter revoted from voter ID in the request.

5.7.1.3 Reg learns whether a voter revoted from incoming OCSP requests. Since the coercer obtains the self-verification material, it can download the coerced ballot's BDOC and learn the signer certificate. The OCSP request contains a certificate identifier derived from that certificate, and this identifier is static because the same voter certificate is used throughout the election.

5.7.1.4 IDP learns whether a voter revoted from the voter ID in the relayed message.

5.7.1.5 VCD/VVD/Sign learns whether a voter revoted from device usage. Corrupt VVD is weaker than VCD or Sign as a voter might not verify the revote.

5.7.1.6 Mixer (Phy) learns whether a voter revoted by comparing vote ciphertext in the coerced vote's BDOC with ciphertext in the anonymised ballotbox.

5.8 Eligibility Verifiability

Normally, when we define eligibility verifiability we have a public bulletin board and a public list voters. Neither of these is true in the case of the IVXV system. In IVXV, the EA provides a list of eligible voters (Dat. 88), which stores a map of voter ID → Voter Information (Dat. 87), to the collector service (CS) and processor application (PA). Ballot eligibility is checked internally by CS and PA, but there are no publicly checkable credentials.

We define eligibility verifiability (EV), along the lines of Effective Authentication in the Swiss regulations, as the ability to cast votes without the involvement of the claimed voter. EV is a voting phase property, where the adversary model only include actors who participated in the protocol prior to the Tally phase, because:

- EV attacks can only be launched by participants that process non-anonymous ballots, as there will be no voter attribution to each vote past that point.
- PA should be honest, otherwise verifiability is undefined, as PA is the last participant who verifies eligibility/voter identity binding.

Based on the publicly available code of IVXV, the eligibility mechanism is enforced by CS and PA: When CS receives a vote request (Alg. 20), it checks that the voter ID bound to the voter's authenticated session matches the voter ID derived from signed fields of the voter's certificate, and that the voter ID is in the list of eligible voters. PA performs the same checks in Alg. 43 (checks voter ID eligibility) and Alg. 46 (checks voter ID matches voter's certificate). CS and PA also verify that the voter's certificate is issued by a trusted CA and that the voter's signature verifies.

Since PA verifies voter certificate issuance, signature correctness, and that the attributed voter ID matches the voter certificate, EV attacks require the adversary to either

- forge a valid signature of the voter without the signing key or,
- cause a ballot to be signed in a way that binds it to the target voter's identity without any participation by that voter.

The first case is excluded as we assume secure cryptographic primitives. The second case requires the adversary to possess the signing key corresponding to a valid voter certificate (of the target voter) issued by a trusted CA.

Concrete attack scenarios may include:

Corrupt Sign Whether a corrupt Sign device can break EV depends on the capability of the signing environment. If a Sign Device can use SIM as a signing oracle, then the attack is trivial; If a Sign device can only sign one message within a short window after physical approval by its owner, then even a corrupt sign device requires voter's involvement to produce a signature.

Corrupt CA A corrupt CA can trivially break EV by issuing an additional certificate that binds to the target voter and revealing the corresponding signing key to the adversary, allowing the adversary to produce signatures that attribute to that voter.

6 Comparison with Other Systems

In this section we compare the IVXV system with two of the most studied deployed internet voting systems: Belenios [11] and the Swiss Post voting system [35].[1] We choose these two systems because they are the most prominent comparators in the European context, they have both received substantial formal analysis, and they sit at instructive structural distances from IVXV: Belenios is built around a public bulletin board and academic verifiability definitions, while Swiss Post — like IVXV — forgoes a public bulletin board and is designed against the same Swiss regulations [33] that we use as our property framework.

We organise the comparison as follows:

- We begin by contrasting the deployment stakes, auditability of the systems, and maturity of current analysis in Section 6.1.
- We then discuss differences in architecture in Section 6.2, in particular the consequences of the bulletin board asymmetry between the three systems.
- We do a detailed comparison of the security properties starting in Section 6.3 before comparing to each system individually. Section 6.4 compares IVXV with Belenios across the five properties, and Section 6.5 does the same for Swiss Post.
- Finally we synthesise the key lessons in Section 6.6.

Belenios Overview

Belenios [11] is a remote internet voting system descended from Helios [1]. Voters cast encrypted ballots through a browser-based voting client; ballots are encrypted under a threshold ElGamal public key whose corresponding private key is shared among trustees, and accompanied by non-interactive zero-knowledge proofs of well-formedness. The principal architectural difference from Helios is the introduction of an explicit *registrar* (also called a credential authority): for each election the registrar generates per-voter signing key pairs and issues them through a private channel. Each cast ballot must be signed under one of these per-voter keys. The set of public credentials is published on the bulletin board, so spurious votes are publicly attributable. The tally is performed via a verifiable mixnet or homomorphic tabulation followed by threshold decryption, with all intermediate artefacts and ZK proofs posted to the bulletin board. Belenios has machine-checked privacy and verifiability proofs in EasyCrypt [9] and is deployed for many low-stakes elections every year.

Swiss Post System Overview

The Swiss Post voting system [35] is the system currently deployed in several Swiss cantons. Its design follows the Federal Chancellery's regulations [33] and centres on a group of four control components in addition to the voting server and the voting client. Each of the four control components acts in both the return-code (CCR) and mixing (CCM) roles, and the security assumptions throughout require only that at least one of the four is honest. The system uses paper-based return codes for cast-as-intended verification: each voter receives a postal voting card containing per-candidate Choice Return Codes, a Ballot Casting Key, a Vote Cast Return Code, and a Start Voting Key. Authentication additionally requires an extended authentication factor, which is not printed on the card but supplied by the voter. After encrypting the voter's selection, the voting client interacts with the CCRs to derive return codes that the voter can compare against the printed sheet. The

[1] For the purpose of this claim we take Belenios as an enhanced descendant of Helios to subsume the latter.

CCMs perform a verifiable mix and partial decryption; the canton's auditor runs an independent verifier against signed CCM outputs. The system has a published computational proof of complete verifiability and privacy [35] and a corresponding ProVerif symbolic analysis.

6.1 Deployment, Auditability, and Maturity

The three systems differ first in what is at stake and in how much of each system is open to independent scrutiny, before any property-by-property comparison.

Deployment IVXV is used for binding national elections in Estonia at large scale and has been in continuous use since 2017, succeeding an earlier system that had carried Estonian internet voting since 2005; Swiss Post is deployed for binding elections in Switzerland; Belenios is used for a far larger number of lower-stakes elections—more than 1,400 in each of 2020 and 2021—typically run by academic institutions and associations [12]. The consequence of a failure, and therefore the threat model each system must plausibly withstand, differs accordingly: a nation-state-level adversary is a realistic concern for IVXV and Swiss Post in a way that it is not for a typical Belenios election.

Auditability The extent to which an independent party can determine what the system does and check that it did it correctly—separates the systems even more sharply than deployment context does. Table 1 assessed IVXV against the nine auditability criteria of Haines and Rønne [18], of which it satisfies only one. Table 16 extends that assessment to the two comparators. Swiss Post publishes a detailed protocol specification, the corresponding source code, a computational security proof, and a prescribed verifier that a cantonal auditor runs against signed outputs. Belenios is open source, is described in peer-reviewed publications, and exposes its entire election transcript on a public bulletin board that anyone may re-verify. IVXV sits at the opposite end: it is only partially open source, publishes no complete protocol specification and no explicit security model, and exposes no public election transcript. The reconstruction in this paper exists precisely because this information is otherwise unavailable; every statement we make about IVXV protocol rests on that reverse engineering rather an existing specification.

Auditability criterion	**IVXV**	**Belenios**	**Swiss Post**
Clear claims	✗	✓	✓
Thorough documentation	✗	✓	✓
Minimality	✗	✗	✓
Buildable	✗	✓	✓
Executable	✗	✓	✓
Exportable	✗	✓	✓
Consistent documentation and source	✗	✓	?
Regularly updated	✗	✓	✓
Minimal restriction on disclosure	✓	✓	✓

Table 16: Auditability of IVXV, Belenios, and Swiss Post against the nine criteria of Haines and Rønne [18]. The IVXV column reproduces Sutopo et al.'s assessment (Table 1); the Belenios and Swiss Post columns are our own assessment.

We note that while IVXV taken as a whole does not satisfy the buildable, executable, and exportable criteria, specific sub-components do.

While both Belenios and the Swiss Post system are vastly better than IVXV system in terms of auditability, they are not perfect. Most notable at the time of writing the Belenios source code includes group parameters which would cause the system to have a passive privacy vulnerability if used, these parameters are not used in the standard deployment. We treat this as a violation of the minimality criteria. Likewise the Swiss Post system includes numerous security checks in the source which are not documented in the specification; these offset the additional complexity of the real system in terms of batching, application invocation, and concurrent elections which are not captured in the security analysis. We treat these as a partial failure of the consistent documentation and source property.

Maturity of Analysis This asymmetry extends to the maturity of formal analysis. Belenios has machine-checked proofs of privacy and verifiability in EasyCrypt [9]; Swiss Post publishes a computational proof of complete verifiability and privacy, refreshed per protocol version, together with a ProVerif symbolic analysis [35]; IVXV has neither. Two caveats apply evenly to all three. First, a proof binds only the model it is stated over, and each system has had attacks reported against its deployed implementation that lay outside the original model [3, 16]. Second, the IVXV side of every comparison below is grounded in our own reconstruction and analysis (Sections 5.5–5.8), whereas the Belenios and Swiss Post sides draw on analyses produced and reviewed by their design teams and external auditors.

6.2 Architectural Asymmetries

The single most consequential structural difference between the three systems is the presence or absence of a public bulletin board. Belenios publishes encrypted ballots, credentials, and zero-knowledge proofs on a bulletin board that any party may read and re-verify; IVXV and Swiss Post do not. This asymmetry shapes the comparison of every property except coercion resistance. For universal verifiability it is decisive: Belenios's UV is universally checkable in the literal sense, whereas IVXV's and Swiss Post's UV is checkable only by designated auditors. Several authors have noted the resulting inconsistency in the use of the term "universal verifiability" in the non-bulletin-board setting [17]; we adopt the regulation's usage for IVXV and Swiss Post but make the distinction explicit when contrasting with Belenios. The bulletin board likewise reshapes eligibility verifiability, since spurious ballots are publicly visible in Belenios but visible only to internal participants in the other two systems.

The three systems also distribute trust differently across their tallying and verification components. Both Belenios and Swiss Post system share decryption across a set of trustees. Belenios goes further by using two different credentials one controlled by the voting server and one by the registrar. The Swiss Post has distribution of credentials but retains a single point of failure in the print office and voting channel. IVXV concentrates far more authority in a small number of participants—most notably the Election Authority, and, for individual properties, the collector and key applications. We return to these single points of trust throughout the property-by-property comparison.

6.3 Overview of Security Differences

We compare the three systems along the five security properties used throughout this paper: individual verifiability (IV), universal verifiability (UV), privacy, coercion resistance (CR), and eligibility verifiability (EV). For IVXV and Swiss Post the properties align well by construction, since both are evaluated in terms close to the Swiss Federal Chancellery's Ordinance on Electronic Voting [33].

For Belenios the reference definitions are academic—ballot privacy (BPRIV [6]) and strong verifiability [10]—which differ in detail but coincide on the essential intuitions; where a difference in definition affects the comparison we flag it in the relevant subsection. Table 17 gives the high-level picture, and the following subsections work through Belenios and Swiss Post in turn.

Property	IVXV	Belenios	Swiss Post
IV cast-as-intended + recorded-as-cast	**Method:** two-device verification. **Trust:** EA; VCD or VVD; CS OR Reg; non-collusion of several actors. **[V]** IV-V1 through IV-V6.	**Method:** Benaloh audit-or-cast with public bulletin board. **Trust:** honest registrar OR voting server [9, 10]. Revoting gap closed in Belenios+ [3].	**Method:** postal return codes. **Trust:** $\geq$1 CCR, print component, and setup component. Holds against corrupt voting client.
UV counted-as-recorded	**Method:** auditor-only checks; no proof. **Trust:** EA; PA and Audit; non-collusion of several actors. **[V]** UV-V1, UV-V2.	**Method:** public bulletin board. **Trust:** honest bulletin board. EasyCrypt proof; anyone can recompute the tally.	**Method:** auditor-checkable verification. **Trust:** $\geq$1 CCM and cantonal auditor. Published computational proof [35].
Privacy ballot secrecy	**Method:** encrypted ballots; anonymise before decrypting. **Trust:** VCD; VVD; decryption threshold; non-collusion of several actors. **[V]** P-V1, P-V2, Shifting/Copying attacks.	**Method:** encrypted ballots; anonymise before decrypting. **Trust:** voting client, registrar, decryption threshold. Machine-checked in EasyCrypt.	**Method:** encrypted ballots; anonymise before decrypting. **Trust:** voting client, print component, $\geq$1 CCM, and setup component. Client trust shared with IVXV.
CR coercion resistance	**Method:** revoting, attempted. Many single-actor corruptions detect or prevent override. Only design of the three to attempt CR.	**Method:** not targeted. No JCJ-style fake credentials.	**Method:** not targeted. Out of scope by design [2].
EV eligibility verifiability	**Method:** national PKI signing. **Trust:** CA; Sign device. No public credential list; leverages existing PKI.	**Method:** per-election credentials on public bulletin board. **Trust:** registrar. Public attribution; registrar is a SPOF.	**Method:** postal voting cards. **Trust:** print component; postal channel. **[W]** Postal interception bypasses crypto entirely.

Table 17: Comparison of IVXV, Belenios, and Swiss Post across the five security properties. Each cell gives the high-level mechanism, salient trust assumptions, and the most important strength or weakness. These trust assumptions are not exhaustive corruption tables; full corruption cases, including trivial violations, are analysed in the security analysis. All three systems also rely on the standard cryptographic assumptions for their primitives and election parameters unless a row highlights a parameter-specific issue. **[V]** marks a concrete vulnerability identified in this paper or prior work; **[W]** marks a trust assumption flagged as a notable weakness in the published literature.

6.4 Belenios

6.4.1 Individual Verifiability

Belenios uses Benaloh-style audit-or-cast on the voter's device: the voter may, at each casting attempt, choose either to audit the ballot (revealing the encryption randomness, which makes the ballot unusable) or to cast it. After casting, the voter checks that their ballot's tracking number appears on the public bulletin board. This gives cast-as-intended with probability that grows in the number of audits performed, and recorded-as-cast unconditionally once the ballot appears on the bulletin board, provided either the registrar or the voting server is honest.

Compared with IVXV, the two designs occupy different points in the trade-off space. IVXV's

two-device verification protocol (VCD to VVD via QR code) is intended to provide cast-as-intended deterministically against a corrupt VCD, at the cost of requiring a second device and trusting it (a corrupt VVD trivially breaks IV; see §5.6.1.2 for the analogous trivial privacy attack and the IV table for the corresponding violation). Belenios's Benaloh audit is probabilistic and assumes only the one device, but its IV guarantee against a corrupt voting client is weaker per ballot. Both approaches additionally depend on the voter actually performing the check; this is an out-of-protocol assumption in both cases.

The Belenios IV story is also complicated by revoting. Baloglu, Bursuc, Mauw, and Pang showed [3] that the original formal proof assumed no further ballots were cast after verification, which does not match deployed practice; they proposed Belenios+ to close this gap. IVXV faces a structurally similar problem. We have documented this in IV-V3 §5.4.3.3 and in the discussion of the reordering attack of [4] (see Sec.5.1.3), but no analogous patched protocol has been deployed.

6.4.2 Universal Verifiability

Belenios's bulletin board carries all encrypted ballots, all credentials, and all tally-phase ZK proofs. Any party — not only a designated auditor — can recompute the result from the bulletin board and check every proof. This is universal verifiability in the strong academic sense, holding provided the bulletin board is honest; the bulletin board is itself a single trust point, which the Belenios design treats as a deliberate simplification.

IVXV's UV is qualitatively weaker on two counts. First, it is auditor-only: only the Audit App, with access to data that is not generally published, can perform the checks. Second, our analysis identifies multiple vulnerabilities that allow a corrupt party to break UV without detection by the prescribed audit workflow (UV-V1 §5.5.3.1 and UV-V2 §5.5.3.2). Belenios's UV has been formally proved against the published bulletin-board contents under a specific model; IVXV's UV has not been formalised, and our analysis suggests it would not hold in the form currently implemented.

6.4.3 Privacy

Belenios's ballot privacy is proved in EasyCrypt under the BPRIV definition [9]. The proof established a trust requirement on the registrar that was missing from earlier pen-and-paper analyses: privacy fails if the registrar misbehaves, because the registrar can correlate credentials with voters and use this to attribute ballots. Privacy otherwise holds as long as the threshold of trustees required for decryption is honest and the voting client follows the protocol.

IVXV's privacy story has more failure modes. As discussed in §5.6.2.1, a corrupt CS alone can perform the Shifting [26] or Copying [4] attacks for a one-vote bias, and our finding P-V2 (§5.6.3.2) shows that a passive adversary observing unmixed ciphertexts can recover plaintext choices outright in multi-question elections, owing to ElGamal randomness reuse. Even setting aside our new findings, the IVXV privacy table shows that several pairs of corrupt actors (CS plus KA, PA plus Mixers) trivially break privacy, where Belenios under analogous corruption patterns retains BPRIV. Both systems share the limitation that the voting client is trusted for privacy: in IVXV, a corrupt VCD trivially learns the votes it casts (§5.6.1.2); in Belenios, a corrupt voting client similarly learns the voter's choice before encryption.

6.4.4 Coercion Resistance

Neither system targets coercion resistance in the JCJ [23] sense of indistinguishable fake credentials. Belenios offers no CR mechanism. IVXV attempts CR through revoting: a coerced voter may later cast another vote that overrides the coerced one. As Section 5.7.1.1 onward shows, this defence is

fragile, with many single-actor corruptions (EA, CS, Reg, IDP, the VCD/VVD/Sign cluster, and Mixers) sufficient to detect or prevent the override. Still, IVXV is the only one of the two systems to attempt any form of CR, and this design choice should be acknowledged on its own terms even if the realised guarantee is weak.

6.4.5 Eligibility Verifiability

The two systems take qualitatively different approaches with different weaknesses; neither is uniformly stronger. Belenios's per-voter signing credentials are issued by the registrar specifically for the election, and the bulletin board makes the set of valid credentials and the set of cast ballots publicly checkable, so spurious votes are visible to anyone. The corresponding trust is distributed rather than concentrated: ballot stuffing is prevented as long as the registrar and the voting server do not collude. Casting a ballot requires both a credential from the registrar and authentication to the server, so neither a corrupt registrar (which cannot authenticate as the voter) nor a corrupt server (which cannot forge credentials) can stuff ballots alone; only their collusion breaks eligibility.

IVXV uses national-PKI signing certificates (ID-card or Mobile-ID) issued by an external CA, with the voter list and ballot validity checked internally by CS and PA. The strength of this design is that the trust in the credential issuer is the same trust the Estonian state already places in its national PKI for many other purposes; the weakness, as discussed in §5.8, is that there is no publicly checkable credential list, and a corrupt CA issuing rogue credentials, or corrupt Sign devices with oracle access, can produce signatures binding to voters' identities without their involvement. The two systems thus offload eligibility trust differently: IVXV concentrates it in a single national CA, whose compromise alone breaks EV, whereas Belenios distributes it across the registrar and the voting server, so that stuffing ballots requires the two to collude. The Belenios approach has the audit advantage of public attribution; the IVXV approach has the practical advantage of leveraging an existing, high-assurance national identity system.

6.5 Swiss Post

We emphasise that the comparison is with the current version of the Swiss Post system and not that suggested second version [8].

6.5.1 Individual Verifiability

Swiss Post's IV rests on two components: at least one honest CCR, and an honest code-sheet chain — the setup component, which generates the mapping from voting options to Choice Return Codes, together with the print component, which renders that mapping onto the postal sheet and delivers it confidentially. Given these, cast-as-intended holds even against a fully malicious voting client. The voter compares the Choice Return Codes displayed by the voting client with the codes printed on the postal sheet, and because the return codes are derived from the encrypted ballot via secrets held by the CCRs that the voting client does not possess, a malicious client cannot produce matching codes for a vote different from the one it submitted. Recorded-as-cast is then confirmed at the final step, where the voter enters the Ballot Casting Key and checks the returned Vote Cast Return Code against the printed one.

The voting client is therefore not trusted for correctness at any step, in line with the ordinance's requirement that verifiability not depend on the user platform: a malicious client can withhold the confirmation and cause an abstention (which the voter detects as a missing Vote Cast Return Code), but it cannot cause a verified vote that differs from the voter's selection. What is worth singling out instead is the setup component. Unlike the control components, it is a single trusted party rather

than a distributed one, and it sits outside the "at least one honest control component" guarantee: a setup component (or print component) that mislabels the sheet — printing the return code of one option against the label of another — can, in collusion with a corrupt client that submits the substituted vote, silently record a different choice, since the displayed and printed codes still match. On its own, a mislabelled sheet produces a visible mismatch and is detected; it is the combination with a corrupt client that is silent. This single-party trust is mitigated by offline operation and by examination of the setup and printing processes rather than by cryptographic distribution, and it is the one place where Swiss Post's IV departs from its otherwise distributed trust model.

Compared with IVXV, where IV is guarded by comparing the decrypted choice displayed by an independent VVD with the voter's intent, Swiss Post guards IV by comparing the return codes displayed on the voting client and the printed sheet. Since the return codes are derived using secrets outside the voting client, a corrupt voting client cannot derive matching return codes for a vote different from the one it submitted; in IVXV, a corrupt VCD has more influence over the verification transcript, as the decrypted choice displayed on VVD depends on the VCD-supplied ElGamal randomness. This is illustrated by a historical class of IV attacks that exploited weak VVD validation of ElGamal randomness [32], which has since been mitigated. Therefore, the two systems place the trust boundary for IV on different components. Swiss Post relies on a trusted code-sheet chain (setup and printing) and at least one honest return-code component, whereas IVXV relies on a trusted verification device and on cryptographic checks ensuring that VCD-supplied material cannot meaningfully affect the verification outcome.

Both systems have had IV vulnerabilities reported in deployed implementations: Swiss Post's Issue #1 (insufficient signature validation of the election public key, reported by Teague, Pereira, and Haines, 2024–25) [34] and, historically, the 2019 Scytl/sVote ZK proof soundness flaw [16]; IVXV's new findings IV-V1 through IV-V6 in Section 7.

6.5.2 Universal Verifiability

Both systems define UV as auditor-checkable rather than universally checkable. Swiss Post's UV holds if at least one of the four control components is honest and the canton's auditor faithfully runs the prescribed verifier against the signed CCM outputs; the property has a published computational proof [35].

IVXV's UV is weaker in two distinct senses. First, IVXV has no comparable formal proof; our analysis in Section 5.5 reaches its conclusions by case analysis over corruption patterns rather than by reduction to cryptographic hardness assumptions. Second, and more importantly, our analysis identifies concrete vulnerabilities: UV-V1 (§5.5.3.1) shows that PA can silently drop ballots through gaps in the Audit App's tally-integrity checks, and UV-V2 (§5.5.3.2) shows that a corrupt KA can drop ballots under certain election configurations. These are not artefacts of an unfavourable threat model; they are paths to undetected tally manipulation in the protocol though they will hopefully be caught by undocumented checks. Swiss Post's published UV proof does not preclude implementation gaps but the existence of a proof and enhanced auditability in general make Swiss Post's UV materially stronger than IVXV's in its current state.

6.5.3 Privacy

Swiss Post's privacy holds if the voting client is honest, the code-sheet chain (the setup and print components) is honest, and at least one of the four control components is honest. The reliance on the voting client is a deliberate trade-off: for usability reasons, the voter selects their candidate directly on the device, which therefore necessarily learns the vote. The code-sheet chain is trusted here for the same reason it matters to individual verifiability: the setup component generates the mapping

from each voter's return codes to voting options and the print component renders and delivers it, so corrupting either is a privacy risk as well as an integrity one.

This client trust requirement is shared by IVXV: a corrupt VCD trivially learns the votes it casts (§5.6.1.1). Where they diverge is in their robustness once the voting client and the code-sheet chain are honest. Conditioned on an honest code-sheet chain, Swiss Post requires all four control components to be corrupt before privacy is breached online, and the published computational proof reduces residual privacy to standard cryptographic assumptions. IVXV's privacy table shows multiple smaller corruption sets that break privacy with the voting client honest: a corrupt CS alone enables the Shifting and Copying attacks; CS plus KA gives full decryption; PA plus Mixers gives full deanonymisation; and our P-V2 finding gives a fully passive break under multi-question configurations for an adversary with access to unmixed votes, without requiring any participant corruption. The qualitative gap is therefore in the size of the corruption set required to break the online mixing components, where Swiss Post requires more compromise than IVXV does; both systems, though, retain a trusted setup party — the code-sheet chain (setup and print components) in Swiss Post — whose compromise would break privacy directly. We note also that Swiss Post has had a privacy-relevant issue reported (Issue #11, CCMs not checking the ZKP before mix-decrypting [34]), so this gap reflects the published designs rather than a claim that Swiss Post has never had a privacy bug.

6.5.4 Coercion Resistance

Swiss Post does not claim coercion resistance. External commentary on the Swiss system has flagged this as a deliberate scope decision rather than an oversight: coercion and vote-buying are not currently treated as risks the system mitigates [2]. IVXV's revoting-based approach, although weak (see §5.7.1.2 onward), is at least an attempt to provide a form of CR, and on this dimension IVXV is the more ambitious design. We do not claim IVXV's realised CR is strong; we claim only that it exists as a design goal in IVXV and does not in Swiss Post.

6.5.5 Eligibility Verifiability

Swiss Post authenticates voters using two factors. The Start Voting Key is generated during election configuration and printed on the postal voting card, which the cantonal authority mails to the voter. The extended authentication factor is not printed on the card: it is normally a personal datum the voter already knows, namely their year of birth or full date of birth, depending on the canton. Eligibility holds provided the print component is honest, the postal channel delivers cards to the intended voters without interception or duplication, and the voting server correctly enforces both factors. The print component is a fully trusted single point: it handles the Start Voting Key of every voter, so a corrupt printer can produce duplicate sheets and cast ballots on behalf of voters whose sheets it has copied. The postal channel is a notable weakness in its own right. Nominally, interception yields only the first factor; in practice the second adds little. It is low entropy—a date of birth carries at most about 15 bits, and appreciably less once population demographics are accounted for, while a year of birth carries fewer than seven—and an intercepted card identifies its addressee, so the adversary is guessing the birth date of a known individual rather than an anonymous one. An adversary with access to the postal flow (whether through interception, redirection at the address of record, or insider access at the postal operator) therefore obtains, at modest additional cost, everything needed to cast a ballot for the affected voter, without requiring any compromise of the cryptographic or server-side components of the system. Unlike a cryptographic credential, a birth date also cannot be reissued once exposed [13].

IVXV's EV assumptions concentrate trust differently. The signing credential is an Estonian national PKI certificate (ID-card or Mobile-ID), and the corresponding signing key is held by the voter on a smart card or in a SIM-protected environment rather than on a piece of paper. The trusted parties are the issuing CA and the issuance process, plus the Sign device's policy on producing signatures (§5.8). A corrupt CA can issue rogue certificates; a Sign device usable as an oracle without per-message physical approval can sign on a voter's behalf. The two systems thus both have a single high-trust issuance bottleneck, with Swiss Post's bottleneck being the print component plus the postal channel and IVXV's being the national CA plus Sign device policy. Neither system supports public auditing of the eligibility step in the way Belenios does.

6.6 Synthesis

Pulling the three systems together property by property:

6.6.1 IV

IVXV's two-device verification is deterministic against a corrupt VCD but introduces the VVD as a second corruption point. Belenios's Benaloh audit is probabilistic and single-device. Swiss Post's return codes hold against a corrupt voting client at the cost of trusting the print component. Each design picks a different point in the same trade-off space; IVXV is not uniformly weaker or stronger than the others on IV in the abstract, although our findings IV-V1 through IV-V6 show several concrete weaknesses in the current IVXV implementation that the comparators do not exhibit in their currently deployed versions.

6.6.2 UV

Belenios provides genuine universally checkable UV via the public bulletin board, with a formal proof. Swiss Post provides auditor-checkable UV with a published computational proof and an external cantonal auditor. IVXV provides auditor-checkable UV with no formal proof and with the concrete vulnerabilities UV-V1 and UV-V2 making the property break under simple corruptions of PA or (under certain configurations) KA. IVXV is the weakest of the three on UV both in formal guarantee and in its current implementation.

6.6.3 Privacy

Belenios and Swiss Post both have machine-checked or computational proofs of privacy under their stated trust assumptions; IVXV has no such proof, and our findings include both new and previously known privacy attacks — in particular the passive P-V2 attack, which requires no participant corruption at all under multi-question elections. All three systems require an honest voting client for privacy. With the client honest, however, IVXV breaks under smaller corruption sets than either comparator. IVXV is the weakest of the three on privacy.

6.6.4 CR

Belenios and Swiss Post offer no CR. IVXV attempts CR through revoting; the realised guarantee is weak but the design intent is present. IVXV is the most ambitious of the three on CR even though the realisation is limited.

6.6.5 EV

Belenios offers public attribution of valid credentials via the bulletin board, with trust distributed between the registrar and voting server. Swiss Post offers print-based authentication factors via the postal channel, with both the print component and the postal channel as essentially single points of trust. IVXV offers national PKI-based signing, with the CA and Sign device as single points of trust.

The overall picture is that IVXV occupies a middle ground in design ambition — attempting CR, which neither comparator does, but providing weaker UV and privacy guarantees than either, particularly in light of the new findings reported in this paper. The most actionable take-away from the comparison is on UV and privacy, where the gap between IVXV and the comparators is largest and is driven by concrete vulnerabilities rather than by definitional differences.

7 Findings

7.1 New Attacks

7.1.1 IV-V1 (LI, RV): Sign Revoting Attack

An adversary controlling the Sign device of some honest voter A, and who knows A's district's candidate list, can perform a ballot reordering attack under the following assumptions:

- a Sign device can only sign one message upon the owner's approval (otherwise trivial),
- The voter ID and phone number are the only required materials to start an MID authenticated session with CS
- a voter's Sign device can derive the voter ID and phone number from its locally available data. If not, then this attack also requires collusion with ID Provider.

The adversary waits for A to submit a vote. Instead of signing its digest $h_a \leftarrow \mathsf{Hash}(v_a)$, it signs the digest of a different vote choice $h_b \leftarrow \mathsf{Hash}(v_b)$ then crashes. Voter thinks voting failed, then proceeds to go through the voting process again, submits its vote v'_a and verifies. Sign device later uses the mobile number and voter ID provided by ID provider to start a new voting session with CS on behalf of the voter, and submits the signed BDOC containing v_b.

This attack belongs to the same revoting family as Pereira's attack [28] (Section 5.1.2), but differs in two key respects. First, it corrupts only the Sign device and works with an honest VCD, whereas Pereira's attack requires a corrupt VCD. Second, unlike Pereira's "Ghost Click" variant, it does not assume the adversary can use the eID as a signing oracle: the malicious ballot v_b is signed during the voter's single legitimate approval (by signing h_b in place of h_a), and the subsequent overwrite requires only *submitting* the already-signed BDOC, which needs the non-secret voter ID and phone number to re-establish a CS session rather than a further signature.

7.1.2 IV-V2 (BD, D): CS Ballot Dropping Attack

Discrepancies in TSP response verification between CS, VVD, and PA allows a corrupt CS to create a TSP response that passes self-verification but will be rejected during Tally.

In the voting phase, CS sends Reg a TSP request containing message imprint m and nonce n (see Alg. 23), where m is constructed from the SHA256 digest of the voter signature s, n is constructed from the SHA256 digest of s signed by CS's TSP Client secret key. m and n each has an `alg` field that indicates the digest algorithm used.

When VVD verifies the TSP response $\texttt{resp}_{\texttt{tsp}}$ (Alg. 33), it fetches the voter signature s from the BDOC, re-computes m and n using the specified digest algorithms (extracted from $\texttt{resp}_{\texttt{tsp}}.m.\texttt{alg}$, $\texttt{resp}_{\texttt{tsp}}.n.\texttt{alg}$), then compares the TSP response with the reconstructed values: $\texttt{resp}_{\texttt{tsp}}.m = m$, $\mathsf{Verify}(pk_{\texttt{tspClient}}, n, \texttt{resp}_{\texttt{tsp}}.n.\sigma)$.

When PA verifies the TSP response (Alg. 45), it has an additional assertion that the digest algorithms for m and n matches ($r.m.\texttt{alg} = \texttt{resp}_{\texttt{tsp}}.n.\texttt{alg}$), if the check fails, PA rejects the ballot and logs an `REG_NONCE_ALG_MISMATCH` error.

Therefore, a malicious CS can choose differing digest algorithms for computing m and n, and run the rest of the protocol, causing the ballot to pass the voter's self-verification but later dropped by PA during Tally. The visibility of this attack depends on whether `REG_NONCE_ALG_MISMATCH` is investigated during audit, which is unspecified.

7.1.3 IV-V3 (LC, LT, R): Reg Reordering Attacks

The Reg service (Timestamping Authority) can perform ballot re-ordering attacks by exploiting weakness in CS's TSP timestamp validation. In contrast to the corrupt-collector reordering of Section 5.1.3 (already described by Pereira [28]), this attack requires only a corrupt Reg service while CS remains honest.

For each TSP response received from Reg, CS verifies the signed timestamp ($t_{\texttt{tsp}}$). The timestamp is accepted if it lies within a freshness window relative to the time when CS receives the response ($t_{\texttt{now}}$): $-60s < t_{\texttt{tsp}} - t_{\texttt{now}} < 2s$ (see Alg. 25). In later phases, each voter's latest ballot is selected according to this TSP-authorised timestamp. The timestamp therefore determines which revote is retained and eventually tallied.

There are two weaknesses in this timestamp verification mechanism, each opens up an attack scenario:

- CS does not check that the TSP timestamp in a newly received response is greater than the timestamps of previously registered votes from the same voter. A malicious Reg service can hence backdate/postdate a voter's votes. If a voter made two votes v_1 and v_2 within $62s$, the Reg service can make the timestamp for v_2 smaller than v_1 while passing checks in CS. This would result in v_2 being filtered out during the PA's Squash and Filter stage (see Alg. 47) that retains v_1 instead of v_2. In an election with `latestonly` = `true`, this attack can be detected if the voter attempts to verify v_2; However, if `latestonly` = `false`, this attack is undetectable.

- CS does not check the signed timestamp ($t_{\texttt{tsp}}$) against the time at which it made the TSP request. Instead, it checks $t_{\texttt{tsp}}$ against the time at which the response was received. A corrupt Reg can therefore withhold the TSP response for an earlier vote v_1, wait for the voter to submit and verify v_2, then release a TSP response for v_1 containing a fresh signed timestamp. This attack assumes that the delayed TSP request remains live. The IVXV codebase does not enforce a timeout that would prevent such a delayed response, but it is unknown whether a deployment-level timeout is enforced. It also assumes that the voter does not verify v_1, which matches the assumption for the known reordering attack (see Sec. 5.1.3).

7.1.4 IV-V4 (LI, SUB): VCD Identity Substitution Attack

The self-verification protocol in IVXV does not enforce an identity binding between the person performing verification and the signer of the retrieved ballot. This allows an adversary controlling a corrupt VCD for some honest voter A, and controlling some voter (e.g., voter B), can trick voter A into verifying a ballot signed by B. The attack proceeds as follows:

1. During voting phase, A's corrupt VCD receives the `VoterProfile` of A, but instead of using that, it starts an authenticated session with CS using B's `VoterProfile`.

2. As a result, during Auth step and Sign step, the Sign device of A would not receive the corresponding challenges (as the challenge would be forwarded to B's Sign device). However, it should be somewhat easy for the corrupt VCD to convince the voter that Signing has already completed and prompt the voter to submit the ballot, especially because the voter believes that they can double-check their vote via the self-verification step later.

3. A proceeds to submit the vote. CS accepts the vote as it only verifies that the authenticated session's identity matches the identity inside the signer certificate (both are $id_{\texttt{voter}_\texttt{b}}$).

4. Finally, A starts self-verification using VVD. The self-verification process only uses the CS session ID and the vote ID provided by the VCD to retrieve a vote, there is no binding to the voter's identity, VVD will therefore successfully retrieve the ballot that encrypts A's choice signed by B and binds to $id_{\texttt{voter}_\texttt{b}}$ and verifies.

5. After A's verification window, the adversary makes B revote, replacing the B-signed decoy vote that encrypts A's intended choice. A's intended choice is therefore not tallied.

VVD does display the signer CN field from the ballot's signer certificate before the choice-decryption step. However, this is only a displayed value; VVD does not require the voter to confirm that the signer identity matches their own, and the signer identity is displayed in a separate protocol step from the decrypt choice. Moreover, the official self-verification guideline focuses on checking the decrypted choices, but does not instruct voters to verify the signer identity. Therefore, a voter who only follows the self-verification instruction will fail to notice that the retrieved ballot was signed by a different voter, or fail to understand the implication.

7.1.5 IV-V5 (LI, BD): VCD Encoding Attack

A corrupt VCD can break IV by exploiting inconsistent parsing of `ModP` ElGamal ciphertext elements across CS, VVD, and PA. A ballot ciphertext is ASN.1 encoded as:

```
SEQUENCE {
    SEQUENCE { OID 1.3.6...},
    SEQUENCE { INTEGER A, INTEGER B }
}
```

The second inner sequence contains ElGamal Ciphertext encoded in ASN.1. The discrepancy lies in how the three participants interprets the ciphertext.

- CS parses each ciphertext element as a signed integer first, then reduce it by mod p (ciphertext.go, group.go L220) before performing range and QR (quadratic-residue) checks (group.go L159);
- VVD parses each element as an unsigned integer, then performs range check and QR check (ModPGroup.java L90);
- PA parses each element as a signed integer (ModPGroupElement.java L45) then performs range and QR check (ModPGroup.java L342).

This allows a corrupt VCD to send a ballot with a strategically chosen negative ASN.1 encoded ciphertext element (such that its positive unsigned interpretation matches the voter's intended choice to pass self-verification) for it to be accepted by CS and VVD, but later rejected by PA during range checks.

7.1.6 IV-V6 (LI, BD, D): VCD Ballot-Dropping via BDOC replay

An adversary controlling a corrupt VCD for an honest voter can perform a ballot-dropping attack by replaying a previously signed BDOC as a fresh vote.

The attack exploits the tally phase recurrence checks on TSP response key: CS does not check recurrence of submitted BDOCs or their `MessageImprint` values. Since the `MessageImprint` is derived deterministically from the BDOC signature, replaying the same signed BDOC produces the same `MessageImprint`. However, PA checks recurrence of registration response keys. For TSP/TSPReg, this key is the `MessageImprint` embedded in the TSP response. If multiple ballot records have

the same response key, PA keeps only the earliest one according to the TSP timestamp and rejects the rest. Therefore, a replayed ballot may be accepted by CS and become the latest vote for self-verification, but is later removed by PA during recurrence check.

The attack proceeds as follows. The victim first votes for intent A, producing vote v_A. The victim then revotes for intent B, producing vote v_B. Later, the victim attempts to revote for A again. However, instead of constructing a fresh BDOC, the corrupt VCD replays the old BDOC from v_A, producing v'_A. CS accepts v'_A as a fresh vote and records it under a fresh vote identifier. Since v'_A receives a fresh TSP timestamp, CS treats it as the voter's latest vote. During self-verification, the victim retrieves v'_A and verifies. During tally, PA observes that v_A and v'_A have the same registration response key/`MessageImprint`. PA keeps the earlier occurrence v_A and drops v'_A, the remaining votes for the victim are therefore v_A and v_B. Since v_B has a larger TSP timestamp (submitted after v_A), it becomes the voter's latest vote and gets tallied.

7.1.7 UV-V1 (BD): PA Ballot Dropping Attacks

A corrupt PA can drop ballots silently by exploiting inadequate integrity checks in Audit App.

The Audit App uses IntegrityTool (Alg. 67) to detect misbehaviour of PA. It checks consistency between the input ballot store and output ballotbox of PA. It checks the consistency over the number of ballots and their hash digest sum across all PA pipeline stages, ensuring that the input to PA equals the ballots rejected during each stage plus PA's output ballotbox:

$$\mathtt{PA}_{\mathtt{in}} = \mathtt{PA}_{\mathtt{valid}} + \mathtt{PA}_{\mathtt{invalid}}$$

However, the scope and granularity of IntegrityTool's checks are insufficient to detect various types of UV attacks performed by PA:

7.1.7.1 Direct Ballot Dropping

IntegrityTool does not validate the reasons for PA rejected ballots. A malicious PA can trivially change which/whose ballot gets tallied by dropping voters' ballots during different pipeline stages. For example, during the Squash and Filter stage, where PA is expected to retain the latest ballot of each voter, a malicious PA can choose to retain an older ballot instead of the latest, without triggering any alert in Audit App, as the equation over total ballot counts and hash sum holds.

7.1.7.2 Indirect Ballot Dropping via District Reassignment

Additionally, IntegrityTool's checks do not bind each ballot to its district. This allows a PA to indirectly drop ballots by moving target ballots from their intended district buckets to another district's. Since IVXV uses a unique choice list for each district, these moved ballots will be rejected by the Key App when computing the election result.

The same aggregate-only binding also permits ballot addition rather than dropping: Treier's F26-2 variant [36] inserts a ballot into the anonymised ballotbox and edits the filter and anonymisation logs so that the digest-sum equation still holds (Section 5.1.7).

7.1.8 UV-V2 (LC, BD): KA Ballot Dropping Attack

If the `verifyInvalid` option is not enabled for an election, the Audit App does not verify that invalid ciphertexts in the election result produced by the KA (Key App) are indeed invalid. A malicious KA can therefore exclude a valid ballot from tally by placing it in $V_{\mathtt{invalid}}$ (invalid vote list) and $\Pi_{\mathtt{invalid}}$ (invalid decryption proof list) (see Dat. 110 for data structure of `ElectionResult`). A comment in the codebase further suggests that verification of invalid votes is optional. However, KA cannot

dishonestly decrypt a vote/modify the decrypted choice without detection. IVXV's decryption proof binds vote ciphertext to the plaintext and Audit App always verifies the valid votes in the election result against their decryption proofs.

7.1.9 P-V1: Weak ElGamal Generator Attack

For ModP groups, CS maps the election public key *epk* to a known RFC3526 group based on its bit length, then uses the group parameters (p, q) of the mapped group, instead of using the provided p and g (common/collector/crypto/elgamal/x509.go). CS then perform the following ciphertext validation using the mapped group, without checking its consistency with the provided group:

- $c^q \equiv 1 \bmod p$ (c is in the mapped group),
- $0 < c < p$ (range check),
- $c^{(p-1)/2} \equiv 1 \bmod p$ (whether c is in the quadratic-residue subgroup).

VVD/PA/KA uses the provided group parameters directly, and check membership in the specified group. The ciphertext validation process does the same checks as CS.

The protocol participants do not check properties of the generator g.

A corrupt EA can exploit this vulnerability by choosing a valid RFC3526 modulus p but a weak generator $g = 1$ during the setup phase. So $epk = g^x = 1$ and the ciphertext of any vote v becomes $(1, v)$ regardless of the ElGamal randomness chosen by VCD. Then, EA can see the encrypted ballots in CS's voting phase export and trivially learn voters' choices. The same attack could be performed by KA. We note that this attack would not work if the casting device followed the flow of the CS rather than the other components.

7.1.10 P-V2 (LC): A Passive Privacy Attack

We identified a vulnerability in how the IVXV protocol handles ballot encryption when an election contains multiple questions (`questioncount` > 1). Our analysis was based on the latest commits of the verification app repositories. The vulnerability has been present since at least 2017, based on the source code, and is present in the most recent source code available `https://github.com/valimised`. It allows a passive adversary to recover the plaintext from the encrypted votes under the following conditions:

- The adversary has access to the ciphertexts
- The adversary knows the possible encoded plaintexts
- There was more than one vote in the voteList (the voter was allowed to vote for more than one option in the election)

7.1.10.1 Details of Attack: Randomness Reuse for ElGamal Encryption The IVXV verification apps reuse the same ElGamal randomness r for all encrypted choices within a single ballot. This allows anyone who sees the unmixed ballots to link and recover the voters' underlying selections.

A ballot is represented as a `BDocContainer` that contains a list of votes (`voteList`), where each element corresponds to the voter's choice for one question. In `DecryptionTask` [2], the same public key *pub* and randomness `rnd` are reused for every vote:

[2] DecryptionTask.java

```
for (Vote vote : voteList) {
    String decChoice = pub.getDecryptedChoice(vote.vote, rnd);
    if (decChoice.equals("")) {
        result.add(Candidate.NO_CHOICE);
    } else {
        if (CheckBallot.wrongCandidateNumber(decChoice)
                || CheckBallot.wrongCandidateName(decChoice)
                || CheckBallot.decryptedChoiceNotInCandidateList(decChoice)) {
            return null;
        }
        result.add(new Candidate(decChoice));
    }
}
```

Analysis of the `getDecryptedChoice` method and other class methods of `ElGamalPub` confirms that the only source of entropy in the encryption process is the randomness r, used as the ElGamal exponent. No additional randomisation is applied.

Each choice is deterministically encoded before encryption in the following format with padding:

$$\texttt{candidateNumber}\|\texttt{0x1F}\|\texttt{partyId}\|\texttt{0x1F}\|\texttt{candidateName},$$

then encrypted with y^r. Therefore, in an election with multiple questions, given public parameters $p, y = g^x$ and fixed r for each voter, the ciphertexts from the voter are of the form $c_i = (g^r, y^r \cdot m_i)$, $c_j = (g^r, y^r \cdot m_j)$.

As a result, anyone who has access to the ballot ciphertexts (e.g., `bb-4.json`) can infer the underlying choices made by a participant based on the ratio between the ciphertexts of different questions: $\frac{y^r \cdot m_i}{y^r \cdot m_j} = \frac{m_i}{m_j}$.

While default `questioncount` is 1, both the implementation and the documentation indicate that IVXV is designed to support elections with multiple questions.

7.2 Other Security Findings

We list below a set of code-level bugs and weaknesses that we identified during the analysis; we expect these degrade the security of the system but are still evaluating the best way an adversary could exploit them.

7.2.1 OB-1: CS ballot-content corruption escapes `REG_REQ_WITHOUT_BALLOT`

PA's `CheckTool` workflow loads CS-exported ballot records and validates them (step 1), then cross-checks the resulting set against the Reg-exported TSP log (step 3). The cross-check re-processes ballots that failed step 1 if they still contain enough registration data to be matched against the Reg log. As a result, if a malicious CS corrupts a ballot's content (rather than dropping the entry outright), PA logs only the original ballot-validation error from step 1 and *not* `REG_REQ_WITHOUT_BALLOT`. Only `REG_REQ_WITHOUT_BALLOT` and `REG_RESP_REQ_UNMATCH` are explicitly documented as errors that must be manually investigated (see `Result.java`). If step 1 validation errors are not manually investigated, a corrupt CS can make a ballot ineligible for tally by corrupting its content rather than removing the entry. Exploitability depends on audit procedure rather than code alone.

7.2.2 OB-2: PA CheckTool `REG_RESP_REQ_UNMATCH` branch is unreachable

In PA's `CheckCollectorRegConsistency` routine, $\texttt{log}_{\texttt{tsp}}$ is a map keyed by the message field m of each TSP request entry ($\texttt{req} := \{m, \sigma\}$). Lookups into this map use m as the key, so if the looked-up entry $\texttt{req}'$ is non-null then by construction $\texttt{req}'.m$ equals the lookup key, and if $\texttt{req}' = \textbf{null}$ the execution does not enter the else-if branch that reports `REG_RESP_REQ_UNMATCH`. The error condition is therefore dead code in the current implementation. This is one of only two errors documented as requiring manual investigation (see OB-1), so its unreachability is a defence-in-depth concern.

7.2.3 OB-3: Audit App `ConvertTool` does not validate `ByteTree` arity

The `ConvertTool` in the Audit App checks consistency between the anonymised ballotbox produced by PA and its `ByteTree` encoding produced by the Mixer (for input to Verificatum). The expected encoding has five children per ballot,

$$((1, e), (1, d), (1, p), (1, q), (c_0, c_1)),$$

but `ConvertTool` does not validate the number of children; it only extracts the expected fields when parsing the `ByteTree` into a ballot. A malicious Mixer could therefore append an extra child to tag a ballot, e.g.,

$$((1, e), (1, d), (1, p), (1, q), (c_0, c_1), (1, t)).$$

We believe this is currently *not* exploitable in practice because the Verificatum library (`PPGroup.java#L640`) enforces arity itself and an honest mix node will refuse to shuffle a batch containing a malformed `GroupElement`. The IVXV-side check is nevertheless missing.

7.2.4 OB-4: Audit App `DecryptTool` does not bind Plainset/Tallyset to district/parish

The `DecryptTool` verifies that the flattened Plainset matches $\Pi_{\texttt{valid}}$ and that the Tallyset matches the count of plaintexts, but does not bind these per-question maps to the (district, parish) keys they are reported under. A corrupt KA can therefore reassign decrypted choices across districts and parishes without triggering any consistency error. Whether this constitutes a UV break depends on whether the election semantics require sub-totals to be correct (per-district or per-parish) or whether only the global tally is meaningful.

7.2.5 OB-5: VVD trusts TSP responder certificate directly without chain validation

In the live 2026 configuration retrieved from `https://www.valimised.ee/verify/config.json`, the TSP responder certificate is issued by the intermediate CA `SK_TSA_CA_2023E`, which is in turn issued by the root `SK_ID_Solutions_ROOT_G1E`. However, in the VVD's TSP response verification path (`VoteDownloadActivity.java`), the TSP responder certificate obtained via `GetConfigTask` is used directly to verify TSP responses without any certificate-chain validation against a configured root. The TSP client certificate is similarly treated as a self-signed CA. Trust in the TSP signer therefore reduces to trust in whatever certificate is shipped in the bootstrap config.

7.2.6 OB-6: CS uses no configured trust anchor for the ID-provider channel

CS's HTTPS client for the ID-provider channel (`http.go`, line 66) does not configure any trust anchor; it falls back to Go's default system TLS validation. The channel is HTTPS only if the configured URL happens to start with `https://`, otherwise plain HTTP is used. There is no application-level enforcement that the channel must be TLS-protected.

7.2.7 OB-7: CS uses no configured trust anchor for the Reg channel

The same pattern applies to CS's channel with the Reg (TSP) service (`tsp.go`, line 264): no configured trust anchor, Go default TLS validation, and HTTPS only if the configured URL is HTTPS. Again no application-level enforcement.

7.2.8 OB-8: CS does no chain validation on TSP responder certificates

For application-layer TSP response verification, CS uses a pre-configured list of trusted responder certificates directly without chain validation against a root (`cms.go`, line 112). This mirrors the pattern in OB-5 on the VVD side. The voter signature path, by contrast, does perform chain validation against pre-configured roots and intermediates (`bdoc.go`, line 745).

7.2.9 OB-9: PA's BDOC validation has no explicit certificate-role enforcement

PA validates BDOC via the digidoc4j library (`TspProfile.java`, line 178; `BdocContainerReader.java`, lines 142 and 179). The IVXV code itself does not perform explicit role enforcement on certificates (e.g., that a TSP responder cert can only be used to sign TSP responses, not voter signatures). Role separation relies entirely on digidoc4j inferring roles from certificate content (EKU fields and similar). Whether this is sufficient depends on the digidoc4j implementation, which we have not audited in detail.

7.2.10 OB-10: VVD, PA, KA, and AA accept supplied ModP parameters without validation

P-V1 documents the case where CS silently substitutes RFC 3526 group parameters for the supplied election public key, while the other Java-side components use the supplied ModP parameters directly. VVD, PA, and KA do not validate that p is a safe prime, that $q = (p-1)/2$ is prime, or that $g \in \mathbf{QR}_p \setminus \{1\}$.

The same issue appears in the Audit App (AA)'s standalone verifier for Verificatum shuffle proofs. Although the Mixer-side Verificatum tooling rejects invalid parameters such as $g = 1$, AA does not reuse Verificatum's parameter parser in its independent verifier. Instead, the IVXV parser extracts p and g from the Verificatum group description and constructs IVXV math objects directly (`DataParser.java#L112`), using constructors that do not perform safe-prime or generator-order validation. Therefore, the Audit App verifier itself accepts weak ModP parameters unless they are rejected earlier by the Mixers.

This leaves a separate UV attack surface: a corrupt EA could try to provide different parameters to different participants, e.g. a supported large-modulus key to CS and a small structurally valid key to VVD, PA, KA, AA, and the Mixer. The remaining challenge is constructing ciphertexts that are accepted under both interpretations. We flag this as a bug rather than a concrete finding because we have not constructed a proof-of-concept exploit.

7.2.11 OB-11: AA mishandles multi-question ballots rejected by PA `CheckTool`

Audit App's `IntegrityTool` appears to assume that each ballot container identifier corresponds to at most one raw vote payload. This is correct for single question elections, but fails for multi-question elections when a whole BDOC container is rejected during PA ballotbox validation. The root cause is a granularity mismatch between the audit inputs:

- the CS export is parsed by `IntegrityTool` per question ciphertext/inner `.ballot` payload. If an election has k questions, one BDOC container contributes up to k entries with the same ballot identifier. $L_{\texttt{cs}}$ denotes CS export parsed by AA;
- accepted PA stage logs are also produced per question ciphertext. $L_{\texttt{acc}}$ denotes accepted stage log parsed by AA;
- however, `ballotbox_errors.txt` (an error log produced by PA's `CheckTool`) is produced one entry per ballot container, and `IntegrityTool` loads these identifiers into a set. $S_{\texttt{err}}$ denotes the ballot error log parsed by AA;

In `getValidInvalidSums`, `IntegrityTool` iterates over $L_{\texttt{cs}}$. For a rejected multi-question container, the first question ciphertext consumes the corresponding container-level identifier from $S_{\texttt{err}}$. The remaining ciphertexts from the same container no longer match either $S_{\texttt{err}}$ or $L_{\texttt{acc}}$. Their `seenCount` is therefore zero, causing the function to return `null` and falsely report ballotbox inconsistency. This appears to be a robustness bug in the audit workflow rather than a direct vulnerability.

7.2.12 OB-12: CS stores vote identifier under the wrong key in `filter.go`

CS's ticket authentication mechanism intends to provide a single-use property, where each ticket/auth token can only be used to store one vote. This property is described in `ticket.go` and in the `VoteIdentifier` interface. IVXV aims to achieve this by issuing ticket that encrypts a unique vote identifier (VID) to VCD. When VCD later submits the ballot with that ticket, CS should open the ticket to recover the VID, then store the vote under it. Therefore, even if CS accepts two vote requests with the same ticket, the VID extracted from these requests should collide, causing the second vote to fail when CS tries to `StoreVote`.

In the current implementation, the ticket indeed contains the voter's identity and the vote ID (Alg. 15). However, after opening the ticket, the implementation mistakenly stores the extracted VID under `voterIDKey` rather than `voteIDKey` (`filter.go`). As a result, when CS tries to recover the vote ID from the request context during the `RPC.Vote` stage (`voting/service/voting/main.go`), it reads the unassigned `voteIDKey` and falls back to generating a fresh vote identifier.

This bug breaks the intended single-use property, as each request will be assigned a fresh vote identifier. However, the practical security implication is unclear, as it still requires the adversary to have an active session in the correct session stage for CS to accept an `RPC.Vote` request. We therefore frame this as a bug that weakens replay protection.

7.2.13 OB-13: Shoup Sign robustness relies on unchecked correctness conditions

IVXV's threshold signing (Alg. 5) performs Lagrange recombination over $\mathbb{Z}_N$ and, unlike Shoup's original construction [30], operates in the full group $\mathbb{Z}_N^*$ with exponents reduced modulo $\varphi(N)$ rather than in the quadratic-residue subgroup. As the correctness argument in §19 shows, a produced signature verifies provided that (i) the PSS-encoded message c is invertible modulo N, (ii) $\gcd(e, n!) = 1$, and (iii) N is a product of two distinct primes with $\varphi(N)$ as its totient. The implementation enforces none of these conditions explicitly, and the signer does not re-verify its own output before returning it, contrary to step 7 of its documented procedure. We regard the resulting exposure as minor, for the reasons below.

We distinguish two kinds of deviation:

- **Fail-closed (invertibility).** If c is not invertible modulo N — equivalently, if any partial signature $c^{K[i]}$ is non-invertible — then combining raises such an element to a *negative* Lagrange

weight λ_i, which `BigInteger.modPow` evaluates via a modular inverse and which throws when the base shares a factor with N. Signing therefore aborts rather than emitting an invalid signature. As c is a fresh PSS encoding, this occurs with probability on the order of $2/\sqrt{N}$ (it essentially requires encountering a factor of N) and is of no practical concern. This is the invertibility issue: its effect is an abort, not a bad signature.

- **Silently invalid.** A wrong signature can be *returned* only if condition (ii) or (iii) fails: $n \geq 65537$ key holders, so that $e = 65537$ divides $n!$; a repeated prime $p = q$, so that $\varphi(N)$ is miscomputed; or shares originating from different sharings that happen to agree on (N, e) (`filterKeys` checks only the shared modulus and exponent, not that the shares lie on a common polynomial). None of these arise in any realistic deployment.

Moreover, every Shoup signature covers either the election public-key certificates (KeyGen) or the tally and plaintext results (DecryptAndTally), all of which the Audit App independently re-verifies with standard RSA-PSS in `VerifyAuditInput` (Alg. 58). An undetected invalid signature would therefore be caught at audit time. The only residual robustness gap is the missing self-check in the signer, which could turn an out-of-model misconfiguration into a late (audit-time) failure rather than an immediate one; adding a verification step before returning the signature would close it.

7.2.14 OB-14: Cryptographic parameter and key generation is trusted rather than verifiable

Findings P-V1 (§7.1.9) and OB-10 (§7.2.10) concern the absence of *validation* of supplied cryptographic parameters. A separate and stronger concern is the absence of *verifiable generation*: whether those parameters and keys were produced by a public, trapdoor-free procedure. The two are not interchangeable. Validation can only reject parameters that fail a public predicate (e.g., $g \neq 1$, $\mathrm{ord}(g) = q$, p a safe prime); it cannot detect a parameter set that satisfies every predicate yet was generated with a hidden trapdoor. The canonical example is the 2019 SwissPost–Scytl shuffle, in which the commitment key consisted of valid group elements whose discrete-logarithm relations were known to whoever generated them, allowing forged shuffle proofs [16]. No validation check would have caught this; only verifiable generation of the commitment key does.

IVXV inherits verifiable generation in exactly one place. The mixnet commitment generators $H = [h_1, \ldots, h_n]$ used in the shuffle proof are not trusted as supplied: the Audit App recomputes the instance seed ρ and regenerates H from it via a PRG (Alg. 60), so no party can choose H with a known trapdoor. This is precisely the defence that was absent in [16].

Everywhere else, generation is trusted rather than verifiable. The election ElGamal group (p, g) is an EA-supplied group that consumers accept without any generation transcript (see OB-10); the RSA signing modulus N and the threshold decryption and signing key shares are produced by the Key Application acting as a trusted dealer (Alg. 3, Alg. 4), with no verifiable secret sharing, no proof that N is a well-formed product of two primes, and no proof that the published public keys correspond to correctly distributed shares. Under IVXV's implicit trusted-dealer assumption for the EA and KA this is by design; the observation is that the assumption is load-bearing. A corrupt EA or KA can supply a *valid-but-trapdoored* parameter or key — one that passes every check proposed for P-V1 and OB-10 — so the validation mitigations for those findings do not on their own remove the underlying risk. Closing it would require verifiable (e.g., seed-derived, nothing-up-my-sleeve) group parameters and a verifiable distributed key generation, of the kind IVXV already applies to the mixnet commitment key. We flag this as a structural trust assumption rather than a directly exploitable bug.

8 Possible Mitigations

8.1 IV Mitigations

8.1.1 Trivial Attacks

Corrupt EA The IVXV architecture places substantial trust in EA. A convenient attack vector for EA, as an IV adversary, is election configuration equivocation, e.g., providing CS and PA different voter lists. To mitigate spoofing attacks by EA, the existing protocol may be extended with a configuration validation step: Each protocol participant has a signing certificate issued by a trusted CA, each participant signs the hash of their election config and attach it to its original protocol messages/data for other participants to verify. For example, when CS exports voting phase data, it also signs the digest of its config for other participants to validate. When PA processes CS's export, it validates both signatures to ensure no data channel attacks and that their config matches. Alternatively, EA should commit to a global config that is publicly verifiable.

Corrupt CS and Reg If both CS and Reg are corrupt, IV is trivially broken. However, in practice, the risk can be mitigated through replication of data and cross-checking. For example, CS and Reg could each be deployed as multiple services that maintain redundant records. When PA verifying voting phase data, it checks that the record across all services matches, and aborts upon mismatch.

8.1.2 Known Attacks

Reordering A potential mitigation for reordering attacks (§5.1.3) is to add a consistency check in PA that bounds the allowed difference between a ballot's signing time and its registration time. PA would reject a ballot if the timestamp in the voter's signature differs too much from the timestamp in its Reg response, and report an error. This limits a reordering attack adversary's (VCD/network/CS) ability to delay a voter's older ballot and later submit it to overwrite the voter's choice in tally.

Revoting The revoting attack (§5.1.2) is only exploitable in elections where $\texttt{latestonly} = \texttt{false}$. A simple mitigation is to enforce $\texttt{latestonly} = \texttt{true}$ for all elections.

8.1.3 New Findings

IV-V1 The exploitability of IV-V1 depends on the three assumptions explicitly stated in Section 7.1.1. If all of them hold, IVXV must introduce additional secret material that a voter knows, but a Sign device cannot derive.

IV-V2 Add extra checks to CS and VVD's TSP Response verification logic that checks consistency between the digest algorithm for m and n. For CS, it checks $\texttt{req}.m = \texttt{resp}_{\texttt{tsp}}.m$ (digest algorithm consistency is implied); for VVD, it checks $\texttt{resp}_{\texttt{tsp}}.m.\texttt{alg} = \texttt{resp}_{\texttt{tsp}}.n.\texttt{alg}$ and then $\texttt{resp}_{\texttt{tsp}}.m.h = \mathsf{Digest}(\texttt{resp}_{\texttt{tsp}}.n.\texttt{alg}, s)$.

IV-V3 Extend CS's Registration response verification such that not only the registration time has to fall within the allowed difference window, it must also be larger than the registration time of the voter's latest ballot in CS's local store; additionally, CS should enforce a timeout for the TSP response.

IV-V4 The minimal patch is to update the self-verification instructions to state explicitly that a voter must also verify the displayed signer identity, and make VVD require explicit confirmation that the signer identity belongs to the voter. A stronger mitigation is to extend the VVD workflow so that identity binding is checked by the application automatically. For example, VVD could require the voter to enter their voter ID during self-verification and then compare it with the identity extracted from the ballot signer certificate.

IV-V5 The root cause of IV-V5 is an inconsistency between how CS, VVD, and PA interpret ASN.1 encoded ciphertext differently, with PA applying the strictest checks. A straightforward mitigation is to align the ciphertext interpretation logic for all three participants. In particular, CS and VVD should both adopt PA's interpretation rule that parses ciphertext as signed integer and reject negative values.

8.2 UV Mitigations

8.2.1 Trivial Attacks

Corrupt CS and Reg The mitigation for CS and Reg ballot dropping attack discussed in §8.1 also applies here.

Corrupt Mixers and Audit Currently, Audit `ConvertTool` and `MixerTool` are used to detect misbehaviour of Mixers (during encoding and mixing). One potential mitigation is to let PA receive the Mixer output and run the same checks itself, so that mixing correctness does not sole rely on Audit.

8.2.2 Known Attacks

Ballot Substitution (F26-2) The per-ballot binding proposed for UV-V1 below, extended to every PA pipeline stage rather than only `CheckTool`, also closes the F26-2 variant (§5.1.7).

8.2.3 New Findings

UV-V1 Every ballot rejected/squashed/filtered by PA must be manually investigated, unless Audit `IntegrityTool` is substantially extended to validate the legitimacy of each rejection reason automatically. The `IntegrityTool` must also bind ballots to their districts throughout the audit trail, ensuring that the district key of each ballot in the anonymous ballotbox (PA output) matches the district associated with the same ballot in the CS ballotstore (PA input), to detect district reassignment attack by PA.

UV-V2 Make `verifyInvalid` non-optional, Audit App needs to verify invalid ciphertext for all elections to ensure Key App cannot perform trivial ballot dropping attacks by marking valid ballots as invalid.

8.3 Privacy Mitigations

8.3.1 Trivial Attacks

Corrupt CS and Reg The mitigation for CS and Reg ballot dropping attack discussed in §8.1 also applies here.

8.3.2 New Findings

P-V1 All protocol participants should perform extended checks on cryptographic parameters used for an election, e.g., generator $g \neq 1$ and $\mathsf{ord}(g) = q$.

P-V2 VCD must generate fresh ElGamal randomness values to encrypt each election question in a multi-question election.

9 Conclusion

IVXV is Estonia's deployed internet-voting system, in use since 2017. Despite its long history and prominence, IVXV has a significant auditability gap: it provides no complete public protocol specification or explicit security model.

In this work, we reconstructed IVXV from its partially public implementation and documentation, and used the resulting specification as the basis for a systematic security analysis. We developed a protocol-level description covering setup, voting, tallying, and audit. We used this reconstruction to define threat models for five election-security properties: individual verifiability, universal verifiability, privacy, coercion resistance, and eligibility verifiability. We identified ten new vulnerabilities affecting verifiability and privacy, reassessed known attacks, and documented additional implementation bugs and trust-boundary weaknesses. We validated selected findings with proof-of-concept tests.

The findings reveal recurring weaknesses at component boundaries: inconsistent validation of shared artefacts, incomplete bindings between protocol stages, dependence on deployment-specific data channels, and trust assumptions that are not made explicit by the system. We propose mitigations for the principal findings and compare IVXV with other deployed e-voting systems under the same security properties. The reconstructed specification, threat model, and TLA+ model provide a foundation for independent review and for future analyses of IVXV.

AI Use

The authors used generative AI-based tools to revise the text, improve flow, and correct any typos, grammatical errors, and awkward phrasing. We have manually verified and are responsible for the accuracy, originality, and integrity of the output of all AI-based tools.

AI was not used for the reverse-engineering or analysis but was used to cross-check these results.

References

[1] Ben Adida. Helios: Web-based open-audit voting. In *USENIX Security Symposium 2008*, pages 335–348. USENIX Association, 2008. 6

[2] Andrew W. Appel. Switzerland's e-voting: The threat model. https://blog.citp.princeton.edu/2022/07/01/switzerlands-e-voting-the-threat-model/, July 2022. Accessed: 22 May 2026. 6.3, 6.5.4

[3] Sevdenur Baloglu, Sergiu Bursuc, Sjouke Mauw, and Jun Pang. Provably improving election verifiability in Belenios. In *Electronic Voting (E-Vote-ID) 2021*, volume 12900 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2021. 6.1, 6.3, 6.4.1

[4] Sevdenur Baloglu, Sergiu Bursuc, Sjouke Mauw, and Jun Pang. Formal verification and solutions for Estonian e-voting. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS) 2024*. ACM, 2024. 1, 1.1, 2.1.1.1, 5.1.3, 5.1.6, 5.4, 5.6, 6.4.1, 6.4.3

[5] Josh Benaloh, Michael Naehrig, Olivier Pereira, and Dan S. Wallach. ElectionGuard: a cryptographic toolkit to enable verifiable elections. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024. 16

[6] David Bernhard, Véronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. Sok: A comprehensive analysis of game-based ballot privacy definitions. In *IEEE Symposium on Security and Privacy (S&P) 2015*, pages 499–516. IEEE, 2015. 2.1.2, 6.3

[7] Christian Bull, Kristian Gjøsteen, and Henrik Nore. Faults in Norwegian internet voting. In *E-Vote-ID 2018*, pages 166–169. TUT Press, 2018. 2.1.1.2

[8] Véronique Cortier, Alexandre Debant, Olivier Esseiva, Pierrick Gaudry, Audhild Høgåsen, and Chiara Spadafora. A practical and fully distributed e-voting protocol for the Swiss context. *IACR Cryptol. ePrint Arch.*, 2025:1625, 2025. 6.5

[9] Véronique Cortier, Constantin Catalin Dragan, François Dupressoir, and Bogdan Warinschi. Machine-checked proofs for electronic voting: Privacy and verifiability for Belenios. In *IEEE Computer Security Foundations Symposium (CSF) 2018*, pages 298–312. IEEE, 2018. 2.1.2, 6, 6.1, 6.3, 6.4.3

[10] Véronique Cortier, David Galindo, Stéphane Glondu, and Malika Izabachène. Election verifiability for Helios under weaker trust assumptions. In *European Symposium on Research in Computer Security (ESORICS) 2014*, volume 8713 of *Lecture Notes in Computer Science*, pages 327–344. Springer, 2014. 2.1.2, 6.3

[11] Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. Belenios: A simple private and verifiable electronic voting system. In *Foundations of Security, Protocols, and Equational Reasoning*, volume 11565 of *Lecture Notes in Computer Science*, pages 214–238. Springer, 2019. 2.1.2, 6, 6

[12] Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. Features and usage of Belenios in 2022. In *Electronic Voting (E-Vote-ID) 2022*, 2022. https://inria.hal.science/hal-03791757. 6.1

[13] Aleksander Essex. Rolling re-evaluation of the Swiss Post e-voting system: Scope 1 final report. Technical report, Swiss Federal Chancellery, August 2023. https://www.bk.admin.ch/dam/bk/en/dokumente/pore/E_Voting/Examination_reports_August2023/Scope%201%20Final%20Report%20Aleksander%20Essex%2001.08.2023.pdf.download.pdf. Accessed: 9 September 2026. 6.5.5

[14] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1984. 2.2.1

[15] Pierrick Gaudry and Alexander Golovnev. Breaking the encryption scheme of the Moscow internet voting system. In *Financial Cryptography and Data Security (FC) 2020*, volume 12059 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2020. 2.1.1.3

[16] Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome. In *IEEE Symposium on Security and Privacy (S&P) 2020*, pages 644–660. IEEE, 2020. 2.1.1.4, 6.1, 6.5.1, 7.2.14

[17] Thomas Haines, Olivier Pereira, and Vanessa Teague. Report on the Swiss Post e-voting system. Technical report, Swiss Federal Chancellery, March 2022. https://www.newsd.admin.ch/newsd/message/attachments/71147.pdf. Accessed: 22 May 2026. 6.2

[18] Thomas Haines and Peter B. Rønne. New standards for e-voting systems: Reflections on source code examinations. In *Financial Cryptography and Data Security (FC) Workshops 2021*, volume 12676 of *Lecture Notes in Computer Science*, pages 279–289. Springer, 2021. 1, 6.1, 16

[19] Thomas Haines and Jarrod Rose. Vestigial vulnerabilities in deployed verifiable e-voting systems. Cryptology ePrint Archive, Paper 2025/2033, 2025. https://eprint.iacr.org/2025/2033. Accessed: 22 May 2026. 2.1.1.4

[20] J. Alex Halderman and Vanessa Teague. The New South Wales iVote system: Security failures and verification flaws in a live online election. In *E-Voting and Identity (VoteID) 2015*, volume 9269 of *Lecture Notes in Computer Science*, pages 35–53. Springer, 2015. 2.1.1.5

[21] Sven Heiberg, Tarvi Martens, Priit Vinkel, and Jan Willemson. Improving the verifiability of the Estonian internet voting scheme. In *Electronic Voting (E-Vote-ID) 2016*, volume 10141 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2016. 1, 5.1.2

[22] Sven Heiberg and Jan Willemson. Verifiable internet voting in Estonia. In *6th International Conference on Electronic Voting: Verifying the Vote, EVOTE 2014, Lochau / Bregenz, Austria, October 29-31, 2014*, pages 1–8. IEEE, 2014. 1, 5.1.2

[23] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In *ACM Workshop on Privacy in the Electronic Society (WPES) 2005*, pages 61–70. ACM, 2005. 2.1.2, 6.4.4

[24] Steve Kremer, Mark Ryan, and Ben Smyth. Election verifiability in electronic voting protocols. In *European Symposium on Research in Computer Security (ESORICS) 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 389–404. Springer, 2010. 2.1.2

[25] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Clash attacks on the verifiability of e-voting systems. In *IEEE Symposium on Security and Privacy (S&P) 2012*, pages 395–409. IEEE, 2012. 1.4

[26] Johannes Müller. Breaking and fixing vote privacy of the Estonian e-voting protocol IVXV. In *Financial Cryptography and Data Security (FC) Workshops 2022*, volume 13412 of *Lecture Notes in Computer Science*, pages 325–334. Springer, 2022. 1, 2.1.1.1, 5.1.4, 5.1.5, 5.6, 6.4.3

[27] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology – CRYPTO 1991*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991. 4.2.2, 16

[28] Olivier Pereira. Individual verifiability and revoting in the Estonian internet voting system. In *Financial Cryptography and Data Security (FC) Workshops 2022*, volume 13412 of *Lecture Notes in Computer Science*, pages 315–324. Springer, 2022. 1, 2.1.1.1, 5.1.2, 5.1.3, 5.4, 7.1.1, 7.1.3

[29] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979. 2.2.4

[30] Victor Shoup. Practical threshold signatures. In *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2000. 4.2.2, 19, 7.2.13

[31] Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J. Alex Halderman. Security analysis of the Estonian internet voting system. In *ACM Conference on Computer and Communications Security (CCS) 2014*, pages 703–715. ACM, 2014. 1

[32] Anggrio Sutopo, Thomas Haines, and Peter B. Rønne. On the auditability of the Estonian IVXV system - and an attack on individual verifiability. In *Financial Cryptography and Data Security (FC) Workshops 2023*, volume 13953 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2023. 1, 1, 2.1.1.1, 5.1.1, 6.5.1

[33] Swiss Federal Chancellery. Federal Chancellery ordinance on electronic voting of 25 May 2022. https://www.fedlex.admin.ch/eli/cc/2022/336/en, May 2022. Accessed: 22 May 2026. 5, 6, 6, 6.3

[34] Swiss Post. Continuous improvement of the e-voting system: reports from experts. https://swisspost-digital.ch/en/digital-blog/e-government/new-findings-from-the-public-verification, May 2025. Accessed: 22 May 2026. 6.5.1, 6.5.3

[35] Swiss Post. Protocol of the Swiss Post voting system. https://gitlab.com/swisspost-evoting/e-voting/e-voting-documentation, October 2025. Accessed: 22 May 2026. 2.1.2, 6, 6, 6.1, 6.3, 6.5.2

[36] Tarvo Treier. Beyond the happy path: A safety-critical audit methodology for Estonia's I-Voting processing application. *IEEE Access*, 13:203429–203443, 2025. 1, 2.1.1.1, 5.1.7, 5.5, 5.5.2.1, 7.1.7.2

[37] Tarvo Treier. Re-voting under surveillance: National eID transaction logs as a threat to coercion resistance in Estonian internet voting. In *Electronic Voting (E-Vote-ID) 2025*, volume 16028 of *Lecture Notes in Computer Science*, pages 191–207. Springer, 2025. 1, 2.1.1.1, 5.7.1

[38] Tarvo Treier and Kristjan Düüna. Identifying and solving a vulnerability in the Estonian internet voting process: Subverting ballot integrity without detection. *IEEE Access*, 12:197766–197782, 2024. 1, 2.1.1.1, 4.5.1.5, 5.1.7

A Notes on the codebase

A.1 Trust Anchors

A.1.1 Demo Config

As shown in the demo config (`Documentation/et/example-config/DEMO_trust/` and `Documentation/et/seadistuste` the trust root yaml defines a list of root, intermediate CAs, TSP, and OCSP responders:

```
- roots: EE GovCA, EE Certification Centre Root CA
- intermediates: ESTEID, esteid
- ocsp: SK OCSP
- tsp: SK TSA
```

Inspection of the demo certificates shows that OCSP, TSP, and intermediate CAs (ESTEID, esteid) are all issued by EE Certification Centre Root CA, the EKU field for each cert specifies its role, the X509v3 Basic Constraints field for the ESTEID has pathlen:0, meaning that it is an intermediate CA that's allowed to issue an end-entity cert.

```
EE GovCA
 - EKU: OCSP Signing, TLS Web Client Authentication, E-mail Protection, TLS Web Server Authentication
EE Certification Centre Root CA
 - EKU: TLS Web Client Authentication, TLS Web Server Authentication, Code Signing, E-mail Protection, Time Stamping, OCSP Signing
- SK OCSP
 - EKU: OCSP Signing
 - Issuer: C=EE, O=AS Sertifitseerimiskeskus, CN=EE Certification Centre Root CA, emailAddress=pki@sk.ee
- SK TSA
 - EKU: Time Stamping
 - Issuer: C=EE, O=AS Sertifitseerimiskeskus, CN=EE Certification Centre Root CA, emailAddress=pki@sk.ee
- ESTEID
 - EKU: OCSP Signing, TLS Web Client Authentication, E-mail Protection
    - CA:TRUE, pathlen:0
 - Issuer: Issuer: C=EE, O=AS Sertifitseerimiskeskus, CN=EE Certification Centre Root CA, emailAddress=pki@sk.ee
- esteid
 - EKU: OCSP Signing, TLS Web Client Authentication, E-mail Protection
    - CA:TRUE, pathlen:0
 - Issuer: C=EE, O=SK ID Solutions AS, organizationIdentifier=NTREE-10747013, CN=EE-GovCA2018
```

A.1.2 Live Config (2026)

The live config obtained from https://www.valimised.ee/verify/config.json is used for setting up VVD. The config file contains three certificates:

```
- 2026 TSP
 - EKU: Time Stamping
 - Issuer: CN=SK TSA CA 2023E, organizationIdentifier=NTREE-10747013, O=SK ID Solutions AS, C=EE
- 2026 TSP Client
 - Issuer: C=EE, O=RIA, OU=IVXV Certificates, CN=Collector Registration
 - Subject: C=EE, O=RIA, OU=IVXV Certificates, CN=Collector Registration
 - X509v3 Basic Constraints: CA:TRUE

- 2026 Verification TLS
 - Issuer: C=EE, O=RIA, OU=IVXV Certificates, CN=Service CA
 - Subject: C=EE, O=RIA, OU=IVXV Certificates, CN=Service CA
 - X509v3 Basic Constraints: CA:TRUE
```

The TSP responder certificate is issued by `SK_TSA_CA_2023E`, which is issued by the root CA `SK_ID_Solutions_ROOT_G1E`. It seems that `SK_ID_Solutions_ROOT_G1E` is the successor of EE Certification Centre Root CA, and that an intermediate CA (`SK_TSA_CA_2023E`) has been added to the PKI.

```
- SK_ID_Solutions_ROOT_G1E:
 - Issuer: C=EE, O=SK ID Solutions AS, organizationIdentifier=NTREE-10747013, CN=SK ID Solutions ROOT G1E
 - Subject: C=EE, O=SK ID Solutions AS, organizationIdentifier=NTREE-10747013, CN=SK ID Solutions ROOT G1E

- SK_TSA_CA_2023E:
 - Issuer: C=EE, O=SK ID Solutions AS, organizationIdentifier=NTREE-10747013, CN=SK ID Solutions ROOT G1E
 - Subject: CN=SK TSA CA 2023E, organizationIdentifier=NTREE-10747013, O=SK ID Solutions AS, C=EE
```

Both the TSP Client and the Verification TLS certificates are self-signed CA certificates, and hence certificate chain validation is not applicable.

A.1.3 VVD Trust

A.1.3.1 Channel Security VVD has two main tasks that involve channel security: `GetConfigTask` to obtain bootstrap config from the EA, and `GetVoteTask` to retrieve vote for self-verification from the CS.

- `GetConfigTask`: uses `TrustManagerFactory.init((KeyStore) null)` (HttpRequest.java#L33), so it uses the platform default, instead of custom trust anchors. It enforces HTTPS-only (L40) and `StrictHostnameVerifier` (L46), which verifies the hostname against cert identity fields.
- `GetVoteTask`: the Verification TLS cert (a self-signed CA cert) obtained from the `GetConfigTask` is used to create a truststore for the `GetVoteTask`, then `setTrustManagers()` is applied to the socket factory to use the Verification TLS cert for TLS channel secuity (TlsConnection.java L32-33). `setHostname(socket, hostName)` sets expected server name, and `startHandshake()` enforces cert validation (L75-76).

A.1.3.2 Application Security After obtaining the vote and associated Reg response via the `GetVoteTask`, VVD performs the following verification over the received data:

- TSP Response: verifies the TSP response using the TSP client cert and the TSP responder cert obtained from `GetConfigTask`. Both certificates are trusted straight away. Even though the TSP responder cert is issued by the intermediate CA (`SK_TSA_CA_2023E`), VVD does not seem to do any cert chain validation and uses it directly for verifying TSP response.
- Voter Signature: verifies the voter's signature on the BDOC and validates the voter's certificate. Creates a verification profile using hardcoded issuer certs (ESTEID, esteid) (VoteDownloadActivity.java L43); `getKey(root, issuers))` (BDocContainer.java L159, L178) checks the signer cert against the list of trusted issuer certs.

A.1.4 CS Trust

A.1.4.1 Channel Security

- with ID provider: no trusted cert configured, uses Go default HTTPS validation if the provided URL is HTTPS, otherwise HTTP (http.go L66)
- with Reg: same as for ID provider (tsp.go L264)

A.1.4.2 Application Security

- TSP Response: uses a pre-configured list of trusted responder certs, no chain validation (cms.go L112)
- Voter Signature: cert chain validation using pre-configured roots and intermediates (bdoc.go L745)

A.1.5 PA Trust

A.1.5.1 Channel Security NA

A.1.5.2 Application Security PA validate BDOC via `container.open()` in (TspProfile.java L178), which calls `read()` (ContainerReader.java L86), which then calls `validate()` (BdocContainerReader.java L142, L179).

- TSP Response: cert chain validation using pre-configured root CA, TSP, OCSP certs (BdocContainerReader.java L68). The TSP client signature is checked using a pre-configured TSP client cert, no validation of the cert as it is a self-signed CA cert (TsVerifier.java L38)
- Voter Signature: signature is checked in `validate()` (BdocContainerReader.java L142, L179) using the same configuration as for TSP/OCSP. There is no explicit role enforcement (e.g., TSP cert can only sign TSP request) in the IVXV code, but the digidoc4j library likely infers roles based on their cert content.

  PA also checks that a voter ID matches the voter signature in `createBallot()` (TspProfile.java L55)

B Cryptographic and Other Primitives

B.1 Symmetric Encryption

- $\mathsf{AEADEnc}(k, m, n)$: return the authenticated ciphertext and tag for plaintext m under key k and nonce n; the nonce is not included in the result.
- $\mathsf{AEADDec}(k, c, n)$: return the plaintext authenticated by ciphertext and tag c under key k and nonce n, or $\perp$ if authentication fails.

B.2 ElGamal

B.2.1 Decryption Using Secret Key

Algorithm 69: ElGamalDecrypt

Context:

- `params` : `ElGamalParameters`.
- `modByteLen` : `Int`

Input:

- sk : `Int`: ElGamal secret key
- c : `Tuple<Int, Int>` $:= (c_1, c_2)$: ElGamal ciphertext
- `withProof` : `Bool`: indicates whether decryption proof is needed.

Output: dp : `DecryptionProof`.

Operation:

1. $pk \leftarrow (\texttt{params}.g)^{sk}$
2. $m \leftarrow c_2 \cdot (c_1^{sk})^{-1}$
3. **if** `withProof` **then**
4. $\quad r \xleftarrow{r} \mathsf{Rand}(\mathbb{Z}_q)$
5. $\quad a \leftarrow c_1^r$ ▷ = commit to c_1
6. $\quad b \leftarrow g^r$ ▷ = commit to g
7. $\quad h \leftarrow \mathsf{Hash}(pk \| c \| m \| a \| b)$
8. $\quad s \leftarrow (h \cdot sk + r) \bmod \texttt{params}.q$
9. $\quad$ **if** $|m| \neq$ `modByteLen` **then return** $\perp$
10. $\quad dp \leftarrow (c, m, a, b, s)$
11. $\quad$ **return** dp
12. **return** $(c, m, \textbf{null}, \textbf{null}, \textbf{null})$

B.2.2 Verify ElGamal decryption

Algorithm 70: VerifyElGamalDecrypt

Input:
pk : `Int`: ElGamal public key
decrypted proof π : `DecryptionProof`.

Output: b : `Bool`: true if valid, false otherwise.

Operation:

1 $(c, m, a, b, s) \leftarrow \pi$
2 $h \leftarrow \mathsf{Hash}(pk \| c \| m \| a \| b)$
3 **if** $g^s \neq b \cdot pk^h$ **then return** false ▷ `verifyDecryptionProof`
4 **if** $c[0]^s \neq a \cdot (c[1] \cdot m^{-1})^h$ **then return** false ▷ `verifySecretKeyProof`
5 **return** true

We treat the following primitives and data operations as secure blackbox.

Algorithm 71: Secure Primitives

Struct

- Rand(n). Outputs n bytes from a secure Pseudorandom generator
- Sign(sk, m), Verify(pk, m, σ). Signature scheme
- Hash(x). Hash function
- Now(). Returns the current time
- RegMatch(r, x). true if x matches regex r, false otherwise
- ValidUTF8(x). true if x is valid UTF-8 string, false otherwise
- EndsWith(x, s). true if x ends with s, false otherwise
- SignedInfo(`cert`). Returns the data to be signed
- PKCSDecode(x : `Byte[]`), returns : `Byte[]`
- ASN1Encode(x : `ASN1Node`), returns : `Byte[]` or $\perp$
- ASN1Decode(x : `Byte[]`), returns a `ASN1Node` object or $\perp$
- DecodeASN1Ciphertext$*$(v : `Byte[]`, $P_{\texttt{EG}}$: `ElGamalParameters`). An abstract helper function that parses the ASN.1 encoded v (vote ciphertext) and returns the decoed ciphertext component $c := (c_0, c_1)$ or $\perp$ if decoding fails.
- DEREncode(x : `ASN1Node`), returns the canonical DER encoding of the input or $\perp$ on failure
- S2GE($P_{\texttt{EG}}, pk, x$: `String`) $\rightarrow G$. A function that maps a string x to a Group element in the ElGamal group G defined by pk and $P_{\texttt{EG}}$
- GE2S($P_{\texttt{EG}}, pk, x \in G$) $\rightarrow$ `String`. A function that maps a Group element x in the ElGamal group G defined by pk and $P_{\texttt{EG}}$ to a string
- Prime(x). Samples prime randomly from the interval $[0, 2^x]$
- AEADEnc(k, m, n). Returns the authenticated ciphertext and tag for m under key k and nonce n; the nonce is not included in the result
- AEADDec(k, c, n). Returns the plaintext authenticated by ciphertext and tag c under key k and nonce n, or $\perp$ if authentication fails
- Digest(`alg`, m). Computes digest of m using algoritm `alg`
- CandidateIDs(C). Returns the set of candidate identifiers contained in candidate list C
- FormatTime(t). Returns formatted time string

We define the following data operations for a List

Algorithm 72: List Operations

Struct

- append(x). Appends element x
- removeFirst(x). Returns and removes the first occurrence of x if it exists; otherwise returns **null** without changing the list
- removeAt(i). Returns and removes the value at index i if exists
- removeAll(x). Returns and removes the all occurrences of x if exists

We define the following data operations for Map

Algorithm 73: Map Operations

Struct

- `old` $\leftarrow$ put(k, `new`). stores `new` under k and returns the previous value (null if new k-v pair)
- remove(k). Returns and removes the entry keyed by k if it exists; otherwise returns **null** without changing the map

We define the following data operations for Set

Algorithm 74: Set Operations

Struct

- add(v). Adds a new element
- remove(v). Returns and removes v if it belongs to the set; otherwise returns **null** without changing the set

C Notation Conventions

This appendix records the naming conventions used in the protocol specification. We use structural prefixes only where the collection shape is relevant or where an otherwise short variable name would be ambiguous. Domain-specific names are preferred when the protocol role is clearer than the underlying representation. If a variable is primed and subscripted, the prime is placed before the subscript; if both subscript and superscript are present, the source order is subscript then superscript, e.g., $X'^{y}_{\mathtt{x}}$.

The Boolean values are written as true and false. The value **null** denotes an absent optional value, _ is used only to discard a returned component, and $\perp$ denotes algorithm failure. Empty collections such as [] and $\emptyset$ remain ordinary values rather than absent values.

Prefix	Use	Examples
$M_{\mathtt{name}}$	Map or lookup table.	$M_{\mathtt{voterlist}}$, $M_{\mathtt{ballot}}$, $M_{\mathtt{regReq}}$, $M_{\mathtt{signShare}}$.
$L_{\mathtt{name}}$	Ordered list or sequence.	$L_{\mathtt{sig}}$, $L_{\mathtt{question}}$, $L_{\mathtt{smartcard}}$, $L'_{\mathtt{smartcard}}$, $L_{\mathtt{time}}$.
$S_{\mathtt{name}}$	Set or unordered unique collection.	$S_{\mathtt{invalidID}}$, $S_{\mathtt{invalidProofCiph}}$, $S_{\mathtt{validCiph}}$, $S_{\mathtt{droppedCiph}}$.

Table 18: Structural prefixes used for variables whose collection shape is security-relevant or otherwise ambiguous.

Notation	Reason to keep semantic name
$\mathtt{BB}_{\mathtt{x}}$, $\mathtt{BT}_{\mathtt{x}}$	Ballotboxes and ByteTree values are protocol artefacts; their role is clearer than their nested map/list representation.
$\mathtt{req}_{\mathtt{x}}$, $\mathtt{resp}_{\mathtt{x}}$	Request and response records are protocol messages.
$\mathtt{crt}_{\mathtt{x}}$	Certificates are standard protocol objects, even though they are structured records.
$\mathtt{log}_{\mathtt{x}}$	Logs are named by their protocol origin; if later indexed as maps, the algorithm should state the derived map explicitly.

Table 19: Semantic names retained where the protocol role is clearer than a structural prefix.

D Lists

D.1 Algorithms

2 KeyAppInit . . . 22
3 GenerateKeyShares . . . 23
4 GenerateSignKeyShares . . . 24
5 ShoupSign . . . 25
8 CreateSession . . . 35
9 VerifyAndUpdateSession . . . 36
10 NeedAuth . . . 37
11 NeedDataToken . . . 37
12 SelectTTL . . . 38
13 CanAdvance . . . 38
14 VoterChoices . . . 39
15 CreateAuthToken . . . 40
16 OpenAuthToken . . . 41
17 CreateDataToken . . . 41
18 OpenDataToken . . . 42
19 CreateBDOC . . . 42
20 VerifyBallot . . . 47
21 VerifyBDOC . . . 48
22 IsValidCiphertextCS . . . 49
23 RegisterBallot . . . 51
24 VerifyRegistration . . . 52
25 VerifyTSPCS . . . 53
26 VerifyOCSPCS . . . 54
27 StoreRegisteredBallot . . . 55
28 VerifyRequestedVote . . . 57
29 GetVerificationStats . . . 58
30 LatestRegisteredBallot . . . 59
31 GetVerification . . . 59
32 ValidateBallot . . . 60
33 VerifyTSPVVD . . . 61
34 VerifyOCSPVVD . . . 62
35 DecryptVotes . . . 63
36 IsValidCiphertextVVD . . . 64
37 DecodeModPElement . . . 65
38 ExportBallots . . . 66
39 CheckTool . . . 70
40 LoadBallotFileRecord . . . 71
41 LoadRegData . . . 72
42 CheckCollectorRegConsistency . . . 73
43 CreateBallotResponse . . . 74
44 GetTrustedBDOC . . . 75
45 VerifyTSPPA . . . 76
46 CreateBallot . . . 77
47 FilterBallotbox . . . 78

48 IsValidCiphertextPA . . . 79
49 AnonymizeBallotbox . . . 80
50 BallotboxAsBT . . . 83
51 BTAsBallotbox . . . 84
52 PublicKeyAsBT . . . 85
53 DecryptAndTally . . . 88
54 RecoverDecryption . . . 89
55 DecryptBallotbox . . . 90
56 ComputeElectionResult . . . 92
57 IsValidChoice . . . 93
58 VerifyAuditInput . . . 96
59 ConvertTool . . . 97
60 MixerTool . . . 98
61 DecryptTool . . . 100
62 VerifyCiphersInBB . . . 101
63 VerifyPlaintexts . . . 102
64 IsValidPlaintextChoice . . . 103
65 VerifyValidConsistency . . . 103
66 VerifyTally . . . 104
67 IntegrityTool . . . 105
68 SumValidInvalid . . . 106
115 CreatePALogEntry . . . 180
116 CreatePAErrorLogEntry . . . 180

D.2 Cryptographic and Other Primitives

69 ElGamalDecrypt . . . 160
70 VerifyElGamalDecrypt . . . 161
71 SecurePrim . . . 162
72 ListOp . . . 163
73 MapOp . . . 163
74 SetOp . . . 163

D.3 Data Structures

1 TrustConfig . . . 20
6 CSConfig . . . 28
7 AppConfig . . . 29
75 BDOC . . . 168
76 BallotFile . . . 168
77 XAdESSignature . . . 169
78 X509Certificate . . . 169
79 StorageDB . . . 169
80 SessionData . . . 170
81 CandidateList . . . 170
82 Candidate . . . 170
83 ElGamalParameters . . . 171
84 SmartCardData . . . 171

85 CheckBallotResult . . . 172
86 Ballot . . . 172
87 Voter . . . 173
88 VoterList . . . 173
89 DecryptedVote . . . 173
90 VoterBallot . . . 173
91 CheckedBallotbox . . . 173
92 FilteredBallotbox . . . 173
93 AnonBallotbox . . . 174
94 DecryptedBallotbox . . . 174
95 BTElement . . . 174
96 BallotRef . . . 174
97 BallotFileRecord . . . 174
98 BallotResponse . . . 174
99 DecryptionProof . . . 175
100 PALogEntry . . . 175
101 PAErrorLogEntry . . . 175
102 OCSPCertID . . . 176
103 OCSPReq . . . 176
104 OCSPResponse . . . 176
105 TSPReq . . . 177
106 TSPResponse . . . 177
107 Tally . . . 177
108 PlainBallotbox . . . 178
109 ASN1Node . . . 178
110 ElectionResult . . . 178
111 ShuffleProof . . . 179
112 MessageImprint . . . 179
113 TSPNonce . . . 179
114 RawBallotEntry . . . 180
117 RegisteredBallot . . . 181

E Data Structures

E.1 BDOC File

BDOC shows a simplified BDOC, where BDOC [3] is the envelope format of a voter's signed ballot. It mainly consists of two elements:

- a list of ballot files. We abstract away file-level details by representing each file as a pair containing its filename and encoded vote payload.
- a list of XAdESSignature records, each binds to a ballot and its signing timestamp.

While the BDOC structure supports multiple ballots and signatures, IVXV only allows one ballot and signature per BDOC. The model also omits several lower-level elements present in the real BDOC format, such as:

[3] https://www.id.ee/wp-content/uploads/2021/06/bdoc-spec212-eng.pdf

- XML metadata and manifest files
- Canonicalisation and digest reference details within `<ds:SignedInfo>`. We use a `tbs : Byte[]` field in the model to represent the BDOC content to be signed.
- qualification properties: in Estonian e-voting, qualification services (e.g., TSP, OCSP) are used to register and validate a ballot. This field can be used to store responses from these qualification services. These fields are empty in the BDOC created by the voter, as the qualification process is handled by the Collector Service, and the qualification responses are stored in separate data structures
- the BDOC timestamp: The timestamp field is initially the timestamp inside the voter's first signature. However, if the BDOC contains qualification properties, the signing time becomes the qualification service's signature timestamp. This is omitted as the BDOC timestamp is never used.

Data Structure 75: BDOC

Data: A BDOC container BC parsed by IVXV.

Struct

- `tbs : Byte[]`. To-be-signed data; an abstract field that is assumed to uniquely and completely bind the BDOC content.
- $\texttt{L}_{\texttt{sig}}$ `: List<XAdESSignature>`. A list of voter's signatures. In IVXV, each ballot should only have one voter signature
- $\texttt{L}_{\texttt{ballot}}$ `: List<BallotFile>` : Ballot files carried by the container
- $\texttt{resp}_{\texttt{tsp}}$ `: TSPResponse` : TSP Response
- $\texttt{resp}_{\texttt{ocsp}}$ `: OCSPResponse` : OCSP Response

Data Structure 76: BallotFile

Data: A ballot payload file inside a BDOC container.

Struct

- `filename : String`. Name of the ballot file inside the container
- v `: Byte[]`. Encoded vote ciphertext payload

E.1.1 BDOC Signature

Format of a signature as used in `BDOC` (Sec. E.1), which contains the signature, certificates of the signer and the issuer, and a timestamp.

Data Structure 77: XAdESSignature

Data: Signature in BDOC.

Struct

- σ : : `Byte[]` : Signature Value
- $\texttt{crt}_{\texttt{signer}}$: `X509Certificate`
- $\texttt{crt}_{\texttt{issuer}}$: `X509Certificate or null`. Validated issuer cert of $\texttt{crt}_{\texttt{signer}}$; unset until signer validation succeeds. This field is derived during BDOC verification, rather than trusted certificate supplied by VCD.
- $t_{\texttt{sign}}$: `Time`

In IVXV, the `commonname` field appears to contain the name and ID code of a voter (see - Link to Code), where the ID code matches the `pnoee` field. In - Link to Code, the embedded certificate subject decodes to `CN=KOBRAS,EUROOPA,41602290078` and `serialNumber=PNOEE-41602290078`.

Data Structure 78: X509Certificate

Data: Simplified x509 certificate

Struct

- `tbs` : `Byte[]`. To-be-signed certificate body
- pk : `PublicKey`. Public key
- σ : : `Byte[]`. Issuer's signature
- `issuer` : `String`. Issuer DN
- `commonname` : `String` name and voter ID
- `serialnumber` : `String` voter ID with a "PNOEE-" prefix
- `pnoee` : `String`. voter ID

E.1.2 Storage Service

The `storage` data structure is a simplified representation of the storage sub-service of the Collector service. `BallotStore` stores a mapping of registered ballots keyed by vote ID;

Data Structure 79: StorageDB

Data: Storage Database for Collector

Struct

- `BallotStore` : `Map` $<$ `String`, `RegisteredBallot` $>$. Maps Vote ID to Ballot record
- `SessionStore` : `Map<String, SessionData>`. Maps a session ID ($sid_{\texttt{CS}}$) to SessionData

Data Structure 80: SessionData

Struct

`stage` : `String`. Current session stage
$t_{\texttt{exp}}$: `Time`. Session expiry time
$id_{\texttt{voter}}$: `String`. (Optional) voter identity extracted from a request, if the request contains an `authToken`
vid : `String`. (Optional) vote identifier extracted from a request, if the request contains an `authToken`
`phone` : `String`. (Optional) phone number extracted from the request, if it contains a `dataToken`; used for Mobile-ID certificate and signing requests

Data Structure 81: CandidateList

A `CandidateList` is an abstract data structure that stores a set of `Candidate` entries. It supports lookup of a candidate using any field of the candidate record as a key.

Data Structure 82: Candidate

Struct

`number`
`party`
`name`

E.1.3 ElGamal Parameters

The parameters used by IVXV's ElGamal scheme. The model omits the `oid` (Algorithm Name) and `ec` (Elliptic curve name) fields, assuming a single valid algorithm is in use and that only a multiplicative group (non-ECC) variant of ElGamal is considered.

Data Structure 83: ElGamalParameters

Data: ElGamal parameters structure.

Struct

- `oid`: Algorithm Identifier
- eid: election ID
- p: prime modulus
- q: subgroup order
- g: subgroup generator

E.1.4 Key Holder Smart Card Data

Represents the threshold shares generated by the key application during the setup phase for threshold decryption and signing. Each trustee's smartcard stores one `SmartCardData` record.

Data Structure 84: SmartCardData

Data: SmartCardData of a Key Holder, indexed by party i.

Struct

- idx : Share Index
- $P_{\texttt{EG}}$: ElGamal parameters for the decryption share
- $P_{\texttt{RSA}} := (N, e)$: RSA public parameters stored with the signing share; the implementation fixes $e = 65537$
- k : Integer ▷ `Decryption share as private exponent in` $\mathbb{Z}_q$
- $\texttt{k}_{\texttt{sign}}$: Integer ▷ `Threshold signing share`

E.1.5 Return code for ballot checking

Result codes that could be produced by the processor app (PA) when validating and cross-checking input files (to the tally phase), such as the ballotbox, registration logs from the collector service (CS) and registration service (Reg), and the list of revoked voters from EA. We model PA outcomes that correspond to protocol-level validation, consistency, and ordering checks used in our analysis. IVXV also defines low-level file parsing results, including `INVALID_FILE_NAME`, `MISSING_FILE`, `REPEATED_FILE`, `UNKNOWN_FILE_TYPE`, `INVALID_FILE_SIZE`, and `VOTERLIST_NOT_FOUND`; we omit these because our model abstracts CS and Reg exports as parsed records and does not study low-level voting–tally export-format attacks. `REG_RESP_INVALID` and `REG_REQ_INVALID` denote malformed or unparsable registration responses and requests before IVXV-specific nonce-consistency checks run; we instead model well-formed TSP structures and the nonce-specific failure results explicitly. We also omit `TECHNICAL_ERROR`, which is a catch-all for implementation exceptions rather than a protocol condition. `INVALID_SIGNATURE_PROFILE` is raised when PA expects timestamp-qualified BDOC signatures and encounters another or unknown signature profile; we treat this as part of BDOC validation rather than a separate protocol branch.

Data Structure 85: CheckBallotResult

Data: Result produced from loading Ballotbox.

Struct

- `INVALID_BALLOT_SIGNATURE`
- `MISSING_VOTER_SIGNATURE`
- `VOTER_NOT_FOUND`
- `TIME_BEFORE_START`
- `REG_REQ_NOT_UNIQUE`
- `REG_RESP_NOT_UNIQUE`
- `REG_NO_NONCE`
- `REG_NONCE_NOT_SIG`
- `REG_NONCE_ALG_MISMATCH`
- `REG_NONCE_SIG_INVALID`
- `UNKNOWN_FILE_IN_VOTE_CONTAINER`
- `REG_REQ_WITHOUT_BALLOT`
- `BALLOT_WITHOUT_REG_REQ`
- `REG_RESP_REQ_UNMATCH`
- `SAME_TIME_AS_LATEST`
- `OK`

E.1.6 Ballot and related data structures

Data Structure 86: Ballot

Struct

- id : `String`. Exported ballot identifier copied from BallotRef.$id_{\texttt{ballot}}$.
- t : `Time`.
- `voter` : `Voter` Voter of the ballot
- `votes` : `Map<String, Byte[]>`. Maps ballot key `qKey` to an encoded vote payload copied from the corresponding BallotFile

The district and parish of a Ballot are derived from its voter record as `b.voter.d` and `b.voter.p`; they are not independent ballot fields.

Decrypted Vote ready for Tally

A stage log entry binds to a single vote inside a ballot BDOC. In an election with k questions, PA would create k `PALogEntry`s for each accepted ballot, where their t and $id_{\texttt{voter}}$ values are the same, but has different h values, as h is the hash of each vote ciphertext.

Data Structure 87: Voter

Struct

- id : `String` Voter ID
- n : `String` Name
- d : `String` district
- p : `String` Parish

Data Structure 88: VoterList

Struct

- $\mathtt{M}_{\mathtt{voterlist}}$: `Map<String, Voter>` Voters, keyed by voter ID

Data Structure 89: DecryptedVote

Struct

- d : `String` district
- p : `String` Parish
- q : `String` ballot key `qKey`
- $v := (c_1, c_2)$ Vote ciphertext
- dp : `DecryptionProof` or **null**

Data Structure 90: VoterBallot

Struct

- `voterId` : `String`
- `ballots` : `List<Ballot>`
- `isRevoked` : `Bool`

Data Structure 91: CheckedBallotbox

Struct

- eid : `String`. Election ID
- M : `Map<String, VoterBallot>`. A map of voter ID to `VoterBallot` (which stores all ballots cast by the voter).

Data Structure 92: FilteredBallotbox

Struct

- eid : `String`. Election ID
- M : `Map<String, Ballot>`. Maps voter ID to the voter's latest ballot after deduplication and revocation.

Data Structure 93: AnonBallotbox

Struct

- *eid* : `String`. Election ID
- *M* : `Map<String, Map<String, Map<String, List<Byte[]>>>>`. Maps District → Parish → ballot key`qKey` → ASN.1 encoded votes

Data Structure 94: DecryptedBallotbox

Struct

- *eid* : `String`. Election ID
- *M* : `Map<String, Map<String, Map<String, List<DecryptedVote>>>>`. Maps District → Parish → ballot key`qKey` → decrypted votes.

Data Structure 95: BTElement

Struct

- A `BTElement` := $(G \times G)^5$ is a group element of $(G^2)^5$, used by the Verificatum mixnet

Data Structure 96: BallotRef

Struct

- $id_{\texttt{voter}}$: `String`. Voter ID
- $id_{\texttt{ballot}}$: `String` Exported ballot identifier. Derived by CS this value from the stored ballot submission time.

Data Structure 97: BallotFileRecord

Struct

- `ref` : `BallotRef`. Ballot reference, consisting of voter ID and Ballot ID
- $\texttt{resp}_{\texttt{tsp}}$: `TSPResponse`. Registration TSP response
- $\texttt{resp}_{\texttt{ocsp}}$: `OCSPResponse`. Registration OCSP response
- *BC* : `BDOC`.

Data Structure 98: BallotResponse

Struct

- *b* : `Ballot` : Ballot
- $\texttt{resp}_{\texttt{tsp}}$: `TSPResponse`. Registration TSP response

Data Structure 99: DecryptionProof

Struct

- $c := (c_1, c_2)$: ciphertext
- m: message
- a: message commitment
- b: key commitment
- s: Fiat-Shamir response

Data Structure 100: PALogEntry

Struct

- t : `Time`: timestamp
- h : `Byte[]`: hash digest of the ciphertext field of a ballot
- $id_{\texttt{voter}}$: `String`: voter ID

An error log entry uses $id_{\texttt{ballot}}$ as key, binds to a ballot BDOC.

Data Structure 101: PAErrorLogEntry

Struct

- $id_{\texttt{voter}}$: `String`
- $id_{\texttt{ballot}}$: `String`
- `err` : `CheckBallotResult`: error code

Data Structure 102: OCSPCertID

Struct

`alg` : `String`. Digest algorithm used to derive the certificate identifier
`id` : `Byte[]`. Abstract certificate identifier derived from the signer certificate and issuer certificate fields

Data Structure 103: OCSPReq

Struct

`certID` : `OCSPCertID`. Certificate identifier requested from the OCSP service

Data Structure 104: OCSPResponse

Struct

`certID` : `OCSPCertID`. Certificate identifier whose status is asserted
`status` : `Bool`. Responded certificate's status, where true means that the certificate is valid and false means that it is not
t : `Time`. Response timestamp
σ : `Byte[]`. Responder's signature
$\texttt{crt}_{\texttt{OCSP}}$: `X509Certificate`. Responder's certificate

Data Structure 105: TSPReq

Struct

- m : `MessageImprint`. Message imprint to be timestamped by the TSP provider
- n : `TSPNonce`. Collector/TSP-client signature on m

Data Structure 106: TSPResponse

Struct

- m : `MessageImprint`. Message imprint to be timestamped by the TSP provider
- n : `TSPNonce`. Collector/TSP-client signature on m
- t : `Time`. Response timestamp
- σ : `Byte[]`. TSP provider's signature of the message
- `crt` : `X509Certificate`. TSP provider's certificate

Data Structure 107: Tally

Struct

- eid : `String`. Election ID
- M : `Map<String, Map<String, Map<String, Int>>>`. Maps district to parish to candidate number then to the count.

Data Structure 108: PlainBallotbox

Struct

eid : `String`. Election ID

M : `Map<String, Map<String, List<String>>>`. Maps district to parish to a list of choices, where each choice is a candidate number.

E.1.6.1 ASN1Node

The ASN.1 Tree representation of a vote should be structured as follows:

```
SEQUENCE (2 elem)
  SEQUENCE (1 elem) algorithm
    OBJECT IDENTIFIER 1.3.6.1.4.1.3029.2.1
  SEQUENCE (2 elem) data
    INTEGER (3072 bit) ...
    INTEGER (3069 bit) ...
```

Data Structure 109: ASN1Node

Struct

`ASN1Node` := N : `List<ASN1Node>` | s : `OID` | x : `INT`. An `ASN1Node` represents a node in an ASN.1 parse tree. Each node is either an intermediate node N containing an ordered list of child nodes, or a leaf node which is either an object identifier (`OID`) or integer (`INT`).

Data Structure 110: ElectionResult

Struct

$M_{\texttt{tallyResult}}$: `Map<String, Tally>`. Maps ballot key `qKey` to a tally record

$M_{\texttt{plainResult}}$: `Map<String, PlainBallotbox>`. Maps ballot key `qKey` to a plaintext ballotbox record

$\Pi_{\texttt{valid}}$: `List<DecryptionProof>` (optional). Proofs for valid votes

$\Pi_{\texttt{invalid}}$: `List<DecryptionProof>` (optional). Proofs for invalid votes

$V_{\texttt{invalid}}$: `List<DecryptedVote>`. List of invalid votes

$M_{\texttt{tallySig}}$: `Map<String, Byte[]>` Maps ballot keys `qKey` to signatures on the complete tally records, including *eid*

$M_{\texttt{plainSig}}$: `Map<String, Byte[]>` Maps ballot keys `qKey` to signatures on the complete plaintext ballotbox records, including *eid*

Data Structure 111: ShuffleProof

Data: ShuffleProof

Struct

- `prot` : `ProtocolInformation`. Protocol parameters
- $\texttt{comm}_{\texttt{perm}}$: `PermutationCommitment`.
- $\texttt{comm}_{\texttt{pos}}$: `PoSCommitment`
- $\texttt{reply}_{\texttt{pos}}$: `PoSReply`
- $\texttt{BT}_{\texttt{in}}$: `List<BTElement>`. mixnet input
- $\texttt{BT}_{\texttt{out}}$: `List<BTElement>`. mixnet output
- pk : `Int`

Data Structure 112: MessageImprint

Struct

- `alg` : `String`. Digest algorithm name
- h : `String`. Digest value

Data Structure 113: TSPNonce

Struct

- `alg` : `String`. Digest algorithm name
- σ : `Byte[]`. Signature value

Data Structure 114: RawBallotEntry

Struct

- $id_{\texttt{ballot}}$: `String`: ballot ID
- c : `Byte[]`. Vote ciphertext

F Helper Functions

Data Structure 115: CreatePALogEntry

Input:

- `log` : `List<PALogEntry>`
- b : `Ballot`

Output:

- $\texttt{log}'$: `List<PALogEntry>` Updated log file

Operation:

foreach $(\texttt{qKey}, v) \in b.\texttt{votes}$ **do**

```
                                        /* the hash does not bind to ballot key */
1       h ← Hash(v)
2       e ← PALogEntry(b.t, h, b.voter.id)
3       log.append(e)
    return log
```

Data Structure 116: CreatePAErrorLogEntry

Input:

- `ref` : `BallotRef`
- `err` : `CheckBallotResult`

Output:

- e : `PAErrorLogEntry`

Operation:

```
1   id_voter ← ref.id_voter
2   id_ballot ← ref.id_ballot
3   e ← PAErrorLogEntry(id_voter, id_ballot, err)
    return e
```

Data Structure 117: RegisteredBallot

Data: Data structure of an entry in `BallotStore`

Struct

- *vid* : `String`. Vote ID
- $t_{\texttt{submitted}}$: `Time`. Submission time
- $t_{\texttt{reg}}$: `Time`. Registration/finalisation time
- *v* : `BDOC`. The ballot container
- $id_{\texttt{voter}}$: `String`. Identifies the ballot signer
- *cnt* : `Int`. Time a vote has been verified
- $\texttt{resp}_{\texttt{tsp}}$: `TSPResponse`. TSP qualification response
- $\texttt{resp}_{\texttt{ocsp}}$: `OCSPResponse`. OCSP qualification response
- `isTestVote` : `Bool`. Whether the registration time falls before election start