

Extending ElectionGuard with Eligibility Verifiability to Prove the Absence of Ballot Stuffing

Master's thesis of

Kaiser, Jonas

At the KIT Department of Informatics

KASTEL – Institute of Information Security and Dependability

First examiner: Prof. Dr. Jörn Müller-Quade

Second examiner: Prof. Dr. Thorsten Strufe

Advisor: Dr. Felix Dörre

27. April 2026 – 28. September 2026

Abstract

ElectionGuard is a cryptographic toolkit that makes elections end-to-end verifiable but leaves eligibility verifiability open. Without this property, ballot stuffing, i.e., the addition of ballots beyond one per eligible voter, cannot be detected from the election record.

This thesis extends ElectionGuard with eligibility verifiability while preserving its applicability to different voting systems and the simplicity of its verification. Every ballot is signed under a credential whose key shares are generated independently by several registrars using a multi-signature scheme, so that the registrars can only collectively produce a usable credential. In addition, the electoral roll is fixed by a perfectly hiding commitment, which allows designated auditors to inspect the same roll the credentials were issued for without publishing it. It is proven that, as long as one registrar is honest, the extension is verifiable under the unforgeability of the multi-signature scheme and the soundness of the tally proofs. The correctness of the roll itself is established by a statistical audit whose error probability is bounded by the number of sampled entries. Since the published credentials carry no reference to a voter, everlasting ballot privacy is preserved as long as the registrars keep the link between voters and credentials secret.

A prototype of ElectionGuard instantiates the construction with MuSig and a Pedersen commitment. Election records produced by it remain valid for unmodified verifiers, and the benchmarks show a constant overhead of about one kilobyte and less than one millisecond per ballot, which amounts to about 4% for the smallest ballot style evaluated and decreases for larger ones.

Zusammenfassung

ElectionGuard ist ein kryptographisches Toolkit, das Wahlen Ende-zu-Ende-verifizierbar macht, die Überprüfbarkeit der Wahlberechtigung (Eligibility Verifiability) jedoch offenlässt. Ohne diese Eigenschaft lässt sich Ballot Stuffing, also das Hinzufügen von Stimmzetteln über einen pro wahlberechtigter Person hinaus, anhand des Wahlprotokolls nicht erkennen.

In dieser Arbeit wird ElectionGuard um Eligibility Verifiability erweitert, wobei die Anwendbarkeit auf verschiedene Wahlsysteme und die Einfachheit der Verifikation erhalten bleiben. Jeder Stimmzettel wird unter einem Credential signiert, dessen Schlüsselanteile mithilfe eines Multi-Signatur-Verfahrens unabhängig voneinander von mehreren Registraren erzeugt werden, sodass die Registrare nur gemeinsam ein verwendbares Credential erstellen kann. Zusätzlich wird das Wählerverzeichnis durch ein perfekt verbergendes Commitment festgelegt. Dadurch können ausgewählte Prüfer dasselbe Verzeichnis einsehen, für das die Credentials ausgestellt wurden, ohne dass es veröffentlicht werden muss. Es wird bewiesen, dass die Erweiterung verifizierbar ist, solange ein Registrar ehrlich ist, unter der Voraussetzung der Unfälschbarkeit des Multi-Signatur-Verfahrens und der Korrektheit der Auszählungsbeweise. Die Korrektheit des Verzeichnisses selbst wird durch ein statistisches Audit sichergestellt, dessen Fehlerwahrscheinlichkeit durch die Anzahl der geprüften Einträge beschränkt ist. Da die veröffentlichten Credentials keinen Bezug zu einer Person enthalten, bleibt Everlasting Ballot Privacy erhalten, solange die Registrare die Zuordnung zwischen Wählenden und Credentials geheim halten.

Ein Prototyp von ElectionGuard instanziiert die Konstruktion mit MuSig und einem Pedersen-Commitment. Die damit erzeugten Wahlprotokolle bleiben für unveränderte Verifier gültig, und die Benchmarks zeigen einen konstanten Mehraufwand von etwa einem Kilobyte und weniger als einer Millisekunde pro Stimmzettel. Das entspricht etwa 4 % beim kleinsten untersuchten Stimmzettel und sinkt bei größeren Stimmzetteln weiter.

Contents

Abstract	i
Zusammenfassung	iii
1 Introduction	1
2 Preliminaries	3
2.1 Privacy and Verifiability of Electronic Voting Systems	3
2.2 Discrete Log Assumption	4
2.3 Multi-Signatures	5
2.3.1 MuSig Signature	6
2.4 Commitments	8
2.4.1 Pedersen Commitment	8
2.5 ElectionGuard	9
2.5.1 Assumptions and Security Guarantees	11
3 Existing Approaches to Eligibility Verifiability	15
3.1 Approaches with System-Issued Credentials	15
3.1.1 Distributed Credential Shares	15
3.1.2 Token-Controlled Encryption	16
3.1.3 Credential-Based Ballot Signatures	16
3.2 Approaches with Preexisting Credential Infrastructure	17
3.2.1 Null Ballots over a Public Key Infrastructure	17
3.2.2 OpenID Authentication	18
3.2.3 Anonymous Signatures via Public Key Shuffling	18
3.3 Applicability to ElectionGuard	18
4 Multi-Signature-Based Approach to Eligibility Verifiability	21
4.1 Syntax of a Voting Scheme and Verifiability with One Honest Registrar	21
4.2 ElectionGuard with Multi-Signature-Based Credentials	23
4.2.1 Assumptions and Security Guarantees	29
4.3 Proof of Eligibility Verifiability	32

5 Prototypical Implementation	39
5.1 Design and Cryptographic Instantiation	39
5.2 Evaluation	43
5.2.1 Correctness	43
5.2.2 Benchmark Methodology	43
5.2.3 Performance of the Cryptographic Primitives	44
5.2.4 Performance Overhead during an Election	45
6 Conclusion	49
Bibliography	53

1 Introduction

Elections are a fundamental institution of democratic societies. However, public confidence in their correctness cannot be taken for granted. In a 2024 survey, only 57% of U.S. adults were at least somewhat confident that the votes in that year’s presidential election would be accurately cast and counted, and 44% expected votes cast by persons not eligible to vote to be a major problem [1]. Such votes are a form of election fraud called ballot stuffing, where the cast ballots exceed one ballot per eligible voter. Irrespective of how often ballot stuffing actually occurs, conventional elections offer voters no means to convince themselves that this fraud has not occurred.

Electronic voting can address this by making the individual steps of an election publicly verifiable through cryptographic means, so that the correctness of an outcome no longer has to be taken on trust. At the same time, there is a demand for the tally to remain verifiable by laypersons. In Germany, the Federal Constitutional Court ruled that the essential steps of an election must be verifiable by citizens without specialist knowledge [2], and federal elections have since been conducted exclusively with paper ballots. This suggests that cryptographic verifiability is more likely to find acceptance if it complements paper ballots rather than replacing them.

ElectionGuard is a cryptographic toolkit that follows this approach [3]. It is intended to be integrated into existing voting systems to make elections end-to-end verifiable, i.e., to allow voters to verify that their ballots were recorded as intended and anyone to verify that the tally was computed correctly from the recorded ballots. Possible deployments include, among others, in-person voting with paper ballots or voting machines as well as online voting. Another design goal is to keep the verification simple enough for first-year programming students to implement it. However, ElectionGuard does not specify a method for verifying whether every ballot counted was cast by an eligible voter or whether more than one ballot per eligible voter was counted. This property is known as eligibility verifiability.

This thesis investigates how ElectionGuard can be extended to provide eligibility verifiability. The extension is intended to preserve the design goals of ElectionGuard, so that it remains applicable to a variety of voting systems and its verification remains simple. Furthermore, it should neither require additional trust assumptions for the existing verifiability properties nor significantly increase the trust required for ballot privacy. The proposed approach is assessed with respect to two criteria, namely whether the claimed

security properties can be proven formally and whether the approach is practical, which is examined through benchmarks of a prototype implementation.

The remainder of this thesis is structured as follows. First, the preliminaries are introduced, comprising definitions of eligibility verifiability and related security properties, an overview of ElectionGuard, and the cryptographic primitives on which the extension builds. Subsequently, the approaches of existing voting systems to eligibility verifiability are analyzed with regard to their applicability to ElectionGuard. This is followed by the description of the proposed extension, the trust assumptions under which it guarantees eligibility verifiability, and a formal proof of this guarantee. Afterward, the prototype implementation is presented, and the practicality of the presented extension is evaluated based on it. Finally, the thesis is concluded by discussing the approach, including its limitations and possible future research.

2 Preliminaries

This chapter outlines the conceptual and cryptographic foundations underlying the extension described in Chapter 4 and the implementation described in Chapter 5. First, desirable privacy and verifiability properties of electronic voting systems are listed and explained. Second, the cryptographic building blocks used in the remainder of this work are introduced, namely the discrete logarithm assumption, multi-signatures together with the MuSig scheme, and commitment schemes together with the Pedersen commitment. Finally, the cryptographic toolkit ElectionGuard is presented, including the assumptions it relies on and the security guarantees it provides, as these form the starting point of this work.

2.1 Privacy and Verifiability of Electronic Voting Systems

Electronic voting systems specifically aim at two groups of security properties. The first concerns vote privacy, and the second concerns the correctness of the election process. The two groups are not independent of each other, and strengthening a property from one group can require weakening a property from the other one or strengthening the trust assumptions it relies on. It is therefore important to treat the properties together. Below follows an informal and non-exhaustive list of desirable properties for each group.

Privacy. No one can infer more about the votes and voters than is revealed by the final election result.

Ballot Privacy. No one with access to all public output can find out any information about how someone has voted other than what can be deduced from the election result.

Participation Privacy. No one can find out who actually voted from the public output.

Voter-Ballot Unlinkability. No one can link a ballot to a voter from the public output.

Receipt-Freeness. No voter can prove their vote even if they actively collude with a coercer and deviate from the protocol [4].

Everlasting Ballot Privacy. Ballot privacy holds even for a computationally unbounded party [5].

End-to-End Verifiability / Correctness. The correctness of an election process is most often guaranteed by an end-to-end verifiable voting system. This means that it provides

voters and external auditors with all of the following means of verification, which are drawn from S. Kremer *et al.* [6] and J. Benaloh *et al.* [7].

Individual Verifiability. A voter can verify that their vote has been cast as intended and recorded as cast.

Universal Verifiability. Everyone can verify that all recorded votes have been tallied correctly.

Eligibility Verifiability. Everyone can verify that all recorded votes are cast by eligible voters and that there is only one recorded vote per voter.

Eligibility verifiability is directly connected to voter-ballot unlinkability and participation privacy, because ideally ballots include a proof that they have been cast by eligible voters, without revealing which voter cast them. Therefore, it is difficult to meet this goal without weakening the other two or increasing the trust assumptions for them. In addition, having a link between voter and ballot precludes everlasting ballot privacy for systems with computationally hiding encryptions, since the encryption will eventually be broken, allowing anyone to view any voter's ballot. This connection between the properties needs to be considered when designing a voting system that lets people verify the eligibility of the participating voters.

However, eligibility verifiability is important to counter *ballot stuffing*, which refers to an attack on an election in which one or more ballots are added without authorization. The attacker can be any party, including parties that are part of the election process themselves, such as the registration authority or the operator of the ballot box. Consequently, in a successful attack, the number of recorded votes exceeds the number of votes actually cast by eligible voters, which results in an incorrect election result. Strictly speaking, eligibility verifiability does not prevent such an attack, but it allows anyone to detect the unauthorized ballots from the public output and thus to reject the implicitly incorrect result. In the remainder of this work, therefore, the security against ballot stuffing and the verifiability of eligibility are treated as equivalent.

2.2 Discrete Log Assumption

The discrete logarithm assumption is the hardness assumption underlying the encryption scheme used by ElectionGuard, the multi-signature scheme MuSig described in Section 2.3.1, and the binding property of the Pedersen commitment introduced in Section 2.4.1.

The discrete logarithm (DLog) problem asks, for a cyclic group $\mathbb{G}$ of order q with generator g , to find for a given value $X = g^x \in \mathbb{G}$ its discrete logarithm, i.e., the value x . Here, $\mathbb{G}$, g , and q implicitly depend on the security parameter κ . The DLog assumption states that, for a sufficiently large q , no probabilistic polynomial-time algorithm can compute the discrete

logarithm with non-negligible probability. The advantage of an algorithm $\mathcal{A}$ in solving the discrete logarithm problem is defined as

$$\text{Adv}_{\mathcal{A}}^{\text{DLog}}(\kappa) = \Pr \left[g^{x'} = X \mid X \xleftarrow{\$} \mathbb{G}; x' \leftarrow \mathcal{A}(\mathbb{G}, g, q, X) \right].$$

2.3 Multi-Signatures

Multi-signature schemes are signature schemes in which multiple parties can sign a message only jointly [8]. They are the main building block of the voting scheme in Chapter 4.

Each party possesses its own public-secret key pair. The parties then generate the signature interactively by sending non-secret messages to one another. From the first round to the second-to-last round, each party sends a message to every other party. Starting with the second round, the received messages serve as input for that round. If the protocol succeeds, the output of the last round is the signature itself and is not sent again.

Like classical signature schemes, multi-signature schemes consist of the four algorithms $\text{MS} = (\text{Setup}, \text{KeyGen}, \text{Sign}, \text{Verify})$. The Setup algorithm is performed once by a central authority on the security parameter κ to generate the system parameters $\text{pars} \leftarrow \text{Setup}(1^\kappa)$. Afterward, each party can independently generate its key pair consisting of a secret and a public key, $(\text{sk}_s, \text{pk}_s) \leftarrow \text{KeyGen}()$, using the KeyGen algorithm. The Sign algorithm is performed interactively by each party to produce a signature σ . The signature for a message m can then be verified by anyone with the list of public keys $L = \{\text{pk}_{s_1}, \dots, \text{pk}_{s_n}\}$ for each of the n parties, using the $\text{Verify}(L, m, \sigma)$ algorithm, returning 1 if successful and 0 otherwise. The system parameters pars returned by Setup are implicit inputs to all other algorithms. Such a multi-signature scheme is then called EUF-CMA secure if the advantage $\text{Adv}_{\mathcal{F}, \text{MS}}^{\text{EUF-CMA}}(\kappa) = \Pr[\text{Exp}_{\mathcal{F}, \text{MS}}^{\text{EUF-CMA}}(1^\kappa) = 1]$ is a negligible function of κ for every probabilistic polynomial-time adversary $\mathcal{F}$.

The experiment $\text{Exp}_{\mathcal{F}, \text{MS}}^{\text{EUF-CMA}}(1^\kappa)$ is shown in Figure 1. The challenger first runs the algorithms Setup and KeyGen to generate the system parameters pars and the honest party's key pair $(\text{sk}_s^*, \text{pk}_s^*)$. The forger $\mathcal{F}$ is given the public key pk_s^* , together with oracle access to $\mathcal{OSign}$, an interactive oracle in which the challenger simulates the honest party, potentially across several concurrent signing sessions. Since $\mathcal{F}$ may choose the remaining public keys placed in the key list L itself, and therefore knows the corresponding secret keys, no explicit corruption oracle is required. This directly realizes the assumption that exactly one party is honest, while all others may be corrupted by $\mathcal{F}$. The set Q records the key-list/message pairs (L, m) that were queried to $\mathcal{OSign}$, so that $\mathcal{F}$ wins if it outputs a key list L containing the honest key pk_s^* , a message m for which (L, m) was never queried, and a signature σ that verifies under L and m .

$$\text{Exp}_{\mathcal{F}, \text{MS}}^{\text{EUF-CMA}}(1^\kappa)$$

```

1  $pars \leftarrow \text{Setup}(1^\kappa)$ 
2  $(sk^*, pk^*) \leftarrow \text{KeyGen}()$ 
3  $Q := \emptyset$ 
4  $(L, m, \sigma) \leftarrow \mathcal{F}^{\mathcal{OSign}}(pars, pk^*)$ 
5 return  $pk^* \in L \wedge (L, m) \notin Q \wedge \text{Verify}(L, m, \sigma) = 1$ 

```

Figure 1: EUF-CMA security experiment for multi-signature schemes by M. Bellare and G. Neven [8]. The sign oracle $\mathcal{OSign}$ is an interactive oracle where the challenger simulates one honest signer, and the forger attempts to produce a valid signature afterward. The forger can engage in multiple concurrent signing sessions. Whenever a signing session for a key list L and a message m is started, the oracle adds the pair (L, m) to Q .

2.3.1 MuSig Signature

MuSig is a multi-signature scheme by G. Maxwell *et al.* [9] and is used as the instantiation of the multi-signature building block in the prototypical implementation in Chapter 5. It is provably secure in the random oracle model under the DLog assumption defined in Section 2.2, for the EUF-CMA definition described in Section 2.3. The scheme consists of a three-round interactive signing protocol between the signing parties, and its algorithms are shown in Figure 2.

First, the Setup algorithm generates the group parameters, selects three hash functions $H_{\text{agg}}, H_{\text{com}}, H_{\text{sig}} : \{0, 1\}^* \rightarrow \mathbb{Z}_q$, and returns all of them as system parameters. The KeyGen algorithm draws a random secret key $x \xleftarrow{\$} \mathbb{Z}_q$, computes the public key $X := g^x$, and returns the key pair. All public keys need to be published in some way, so that every signer has access to the uniquely encoded multiset $L := \{X_1, \dots, X_n\}$.

To sign a message m , each signer i first draws a random value $r_i \xleftarrow{\$} \mathbb{Z}_q$, computes $R'_i := g^{r_i}$, and sends the commitment $t_i := H_{\text{com}}(R'_i)$ to every other signer. After receiving all commitments, each signer sends R'_i and checks for every received group element R_i whether the recomputed commitment $H_{\text{com}}(R_i)$ matches the value t_i received in the first round. Note that the random value r_i itself is never revealed. If all commitments match, every signer aggregates the public keys to get $\tilde{X}$, using the product of the independent keys, where every key X_i is raised to the power of $a_i := H_{\text{agg}}(L, X_i)$. Each signer then sends $s'_i = r_i + c \cdot a_i \cdot x_i \bmod q$ to every other signer, where $c := H_{\text{sig}}(\tilde{X}, \tilde{R}, m)$ and $\tilde{R} := \prod_{i=1}^n R_i$. Finally, each signer can produce a valid signature $\sigma := (\tilde{R}, s)$ with $s := \sum_{i=1}^n s'_i \bmod q$.

A signed message can be verified by anyone with access to the public keys L or the aggregated public key $\tilde{X}$ by computing c and comparing $g^s \stackrel{?}{=} \tilde{R} \cdot \tilde{X}^c$.

2.3 Multi-Signatures

Setup(1^κ) <pre> 1 $(\mathbb{G}, q, g) \xleftarrow{\\$} \text{GrGen}(1^\kappa)$ 2 Select three hash functions: 3 $H_{\text{agg}}, H_{\text{com}}, H_{\text{sig}} : \{0, 1\}^* \rightarrow \mathbb{Z}_q$ 4 $\text{pars} := ((\mathbb{G}, q, g), H_{\text{agg}}, H_{\text{com}}, H_{\text{sig}})$ 5 return pars </pre> KeyGen() <pre> 1 $x \xleftarrow{\\$} \mathbb{Z}_q$ 2 $X := g^x$ 3 return (x, X) </pre> KeyAgg(L) <pre> 1 $\{X_1, \dots, X_n\} := L$ 2 for $i := 1, \dots, n$: 3 $a_i := H_{\text{agg}}(L, X_i)$ 4 $\tilde{X} := \prod_{i=1}^n X_i^{a_i}$ 5 return $\tilde{X}$ </pre> Sign() <pre> 1 $r \xleftarrow{\\$} \mathbb{Z}_q$ 2 $R' := g^r$ 3 $t := H_{\text{com}}(R')$ 4 $\text{out} := t$ 5 $\text{state} := (r, R')$ 6 return $(\text{out}, \text{state})$ </pre> Sign'($(\text{out}_1, \dots, \text{out}_n), \text{state}$) <pre> 1 $(r, R') := \text{state}$ 2 $(t_1, \dots, t_n) := (\text{out}_1, \dots, \text{out}_n)$ 3 $\text{out}' := R'$ 4 $\text{state}' := (r, (t_1, \dots, t_n))$ 5 return $(\text{out}', \text{state}')$ </pre>	Sign''($L, (\text{out}'_1, \dots, \text{out}'_n), \text{state}', x, X, m$) <pre> 1 $(r, (t_1, \dots, t_n)) := \text{state}'$ 2 for $i \in \{1, \dots, n\}$: 3 $R_i := \text{out}'_i$ 4 if $H_{\text{com}}(R_i) \neq t_i$: abort 5 $\tilde{X} := \text{KeyAgg}(L)$ 6 $\tilde{R} := \prod_{i=1}^n R_i$ 7 $c := H_{\text{sig}}(\tilde{X}, \tilde{R}, m)$ 8 $a := H_{\text{agg}}(L, X)$ 9 $s' := r + c \cdot a \cdot x \bmod q$ 10 $\text{state}'' := \tilde{R}$ 11 $\text{out}'' := s'$ 12 return $(\text{out}'', \text{state}'')$ </pre> Sign'''($(\text{out}''_1, \dots, \text{out}''_n), \text{state}''$) <pre> 1 $(s'_1, \dots, s'_n) := (\text{out}''_1, \dots, \text{out}''_n)$ 2 $\tilde{R} := \text{state}''$ 3 $s := \sum_{i=1}^n s'_i \bmod q$ 4 $\sigma := (\tilde{R}, s)$ 5 return σ </pre> Verify(L, m, σ) <pre> 1 $(\tilde{R}, s) := \sigma$ 2 $\tilde{X} := \text{KeyAgg}(L)$ 3 $c := H_{\text{sig}}(\tilde{X}, \tilde{R}, m)$ 4 return $g^s \stackrel{?}{=} \tilde{R} \cdot \tilde{X}^c$ </pre>
--	---

Figure 2: The MuSig scheme by G. Maxwell *et al.* [9]. The public parameters pars returned by Setup are implicitly given as input to all other algorithms. The multiset of public keys L can always be disassembled into $\{X_1, \dots, X_n\}$ and is assumed to be encoded uniquely, e.g., by lexicographical order. The values in out are sent to the other signers during one signing session.

2.4 Commitments

A commitment scheme makes it possible to publicly fix a value without revealing it. The committing party can then reveal the value later on, allowing everyone to verify that the revealed value is the same one committed to before. A commitment scheme is therefore *hiding*, as the commitment conceals the value, and *binding*, as the committed value cannot be changed afterward.

Such a scheme can be formalized as a tuple of three algorithms $\mathbf{Com} = (\text{Setup}, \text{Commit}, \text{Open})$. The Setup algorithm is performed once, by a central authority, on the security parameter κ , to generate the system parameters $\text{pars} \leftarrow \text{Setup}(1^\kappa)$. The other two algorithms receive the system parameters implicitly. Afterward, any party can commit to a value x by running the $(\text{com}, \text{dec}) \leftarrow \text{Commit}(x)$ algorithm, which takes the message x and outputs a commitment com together with the opening information dec . The commitment can then be published, while the opening information is kept secret. Later on, the party can reveal the committed value by running the $\text{Open}(\text{com}, \text{dec}, x)$ algorithm on the commitment, the opening information, and the message, which outputs 1 if the commitment matches the recomputed commitment, or 0 otherwise.

Commitments can differ in the strength of their hiding and binding properties. A commitment is called *computationally binding* or *hiding* if a probabilistic polynomial-time adversary cannot break the binding or hiding property. For the binding property, this is formalized by the advantage

$$\text{Adv}_{\mathcal{B}, \mathbf{Com}}^{\text{binding}}(\kappa) = \Pr[x \neq x' \wedge \text{Open}(\text{com}, \text{dec}, x) = 1 \\ \wedge \text{Open}(\text{com}, \text{dec}', x') = 1 \mid (\text{com}, x, \text{dec}, x', \text{dec}') \leftarrow \mathcal{B}(\text{pars})]$$

of an adversary $\mathcal{B}$ in opening one commitment to two different messages, which is required to be negligible in κ for every probabilistic polynomial-time $\mathcal{B}$. If even a computationally unbounded adversary cannot break the binding or hiding property, the commitment is called *perfectly binding* or *hiding*.

2.4.1 Pedersen Commitment

A Pedersen commitment is a *computationally binding* and *perfectly hiding* commitment scheme by T. P. Pedersen [10]. The scheme is defined over a cyclic group $\mathbb{G}$ of prime order q with generator g . The Setup algorithm selects a second generator $h \in \mathbb{G}$, with an unknown discrete logarithm $\log_g h$, and returns the system parameters $\text{pars} := (\mathbb{G}, q, g, h)$. The Commit algorithm takes a message $x \in \mathbb{Z}_q$, draws the opening information $\text{dec} \xleftarrow{\$} \mathbb{Z}_q$, and computes the commitment $\text{com} := g^x \cdot h^{\text{dec}}$. The Open algorithm takes a commitment, the

opening information, and a message, checks whether $com \stackrel{?}{=} g^x \cdot h^{dec}$, and returns 1 if the equality holds and 0 otherwise.

The binding property holds under the DLog assumption defined in Section 2.2. Suppose an adversary finds a collision, i.e., a commitment $com = g^x \cdot h^{dec} = g^{x'} \cdot h^{dec'}$ that opens to the pairs of message and opening information (x, dec) and (x', dec') , where $x' \neq x$, which implies $dec' \neq dec$. Then the adversary can compute the discrete logarithm $a = \log_g h$ by rearranging $com = g^{x+a \cdot dec} = g^{x'+a \cdot dec'}$, which yields $a = \frac{x-x'}{dec'-dec} \bmod q$.

The hiding property, in contrast, is information-theoretically secure, since h is a generator and therefore, for every message $x' \in \mathbb{Z}_q$, exactly one opening information $dec' \in \mathbb{Z}_q$ with $com = g^{x'} \cdot h^{dec'}$ exists, meaning com itself reveals nothing about which message was actually committed to.

2.5 ElectionGuard

ElectionGuard, described by J. Benaloh *et al.* [3], is an open-source toolkit initiated by Microsoft that includes cryptographic primitives to enable end-to-end verifiable elections, with the exception of eligibility verifiability. The authors deliberately leave this responsibility to the election organizers, who may address it through external means or not at all.

What makes ElectionGuard unique is that it is not a standalone voting system but is intended as an extension for new and existing voting systems. This allows the cryptographic primitives to be developed separately from other components, such as the user interface, and makes it easier for security experts to develop and refine them. Additionally, it offers existing voting systems, including paper-based ones, a simple way to add verifiability without discarding the existing infrastructure. In its preferred deployment, ElectionGuard does not replace the existing vote-counting infrastructure but runs alongside it and produces its own independently verifiable tallies.

ElectionGuard involves participants with four clearly defined roles during and after the election process.

Guardians. During the key-generation ceremony, n_G guardians meet with the administrator. Each guardian verifies that the correct cryptographic parameters are being used, for example, that the primes are of the prescribed size. Subsequently, each guardian generates a secret share of the decryption key and publishes the corresponding public share along with a zero-knowledge proof demonstrating knowledge of the associated secret share. Upon completion of the key-generation ceremony, the guardians reconcile their respective views of the process with the public records. After the guardians' validation, the election record is also referred to as the guardian record. At the end of the election process, at least $k \leq n_G$ guardians gather to decrypt the tallies. In doing so, they first verify that the ciphertexts to be decrypted are valid and correctly recorded in the election record, in particular that

they are aggregated tallies and not individual cast ballots. After the final election record is published, the guardians reconcile their view of the records with the published record.

Administrator. At the beginning of the election process, the administrator sets a quorum $k \leq n_G$, which defines the minimum number of guardians required for the decryption of the tallies. The key-generation ceremony then takes place together with all n_G guardians. During this ceremony, the administrator defines the election manifest, which contains the candidates, election rules, and other election-specific parameters. Both the election keys and the manifest are embedded in all applications through which voters can cast their votes. After voting has concluded, the administrator collects the decrypted tallies and publishes them in the final, publicly verifiable election record.

Voters. Voters can cast ballots and verify them. Depending on the election manifest and, for example, the electoral district, voters participate in different contests and have correspondingly different choices. The selection is encrypted either before voting, based on prepared ElGamal ciphertexts [11], or after selection by the device, depending on the specific deployment. Each encrypted selection is accompanied by a zero-knowledge proof based on a disjunctive variant of the Chaum-Pedersen protocol [12], which proves that a valid selection was indeed encrypted. After encryption, the voter receives a confirmation code.

Before the final vote is cast, the voter can challenge their ballot to verify that their selection was encrypted as intended (cast-as-intended verifiability). This mechanism was introduced in [13] and is commonly referred to as the Benaloh challenge. In ElectionGuard, it can take two forms depending on the actual deployment. In the first form, the device discloses the encryption nonce ξ_B , allowing the voter to reproduce the ciphertexts independently on their own device. In the second form, verification occurs as part of the tallying process, where the guardians decrypt the ballot and the result is published in the election record. In both cases, however, the challenged ballot is not included in the final tally, since otherwise the ballot would prove to anyone what it contains, including a coercer.

After the election, each voter can use their confirmation code to verify that their encrypted ballot is included in the published election record (recorded-as-cast verifiability).

Verifiers. After the election process is over and the election record is published, anyone can be a verifier, regardless of whether that party was already involved in the election process or is an outsider. Both existing and self-developed software can be used for verification. As part of the verification process, several properties can be checked using the election record. First, zero-knowledge proofs can be used to verify the well-formedness of the encrypted ballots. Second, it can be verified that the ballots were correctly aggregated to obtain the published tallies and that their decryptions were performed correctly using Chaum-Pedersen proofs. This includes verifying that challenged ballots were correctly decrypted and were not included in the aggregated tallies (tallied-as-recorded verifiability).

Verification does not remove trust from the process but relocates it. The cryptographic proofs contained in the election record establish the correctness of the recorded ballots, their aggregation, and their decryption, and these guarantees hold independently of who performs the verification. The confidence a voter draws from them, however, does not, since verification software can be faulty or biased. A single verifier therefore remains a point of trust in an otherwise cryptographically protected process. This residual trust is reduced when several parties audit the same election record with independently developed implementations, since an undetected error or manipulation would then have to occur in all of them simultaneously.

2.5.1 Assumptions and Security Guarantees

The rest of this section discusses the security properties guaranteed by ElectionGuard and the assumptions needed for them to be ensured. The assumptions are listed first, so that each of them can be referenced individually afterward.

The following cryptographic assumptions concern the primitives ElectionGuard is built from.

DLog Assumption. It is assumed that the DLog assumption as explained in Section 2.2 holds.

Collision-Resistant Hash Assumption. It is assumed that ElectionGuard uses a collision-resistant hash function. A hash function $h : \mathcal{X} \rightarrow \mathcal{Y}$ is called collision resistant if, for every probabilistic polynomial-time algorithm $\mathcal{A}$, the probability that $\mathcal{A}$ finds two $m, m' \in \mathcal{X}$ with $m \neq m'$ and $h(m) = h(m')$ is negligible.

Tally Soundness Assumption. It is assumed that the proofs published alongside the tally cannot be forged, that is, for every probabilistic polynomial-time adversary $\mathcal{A}$, the probability that $\mathcal{A}$ produces an accepting proof for a tally result that differs from the correct tabulation of the recorded ballots is negligible. Unforgeability is what the verification of the tally ultimately rests on, since a verifier can never recompute the plaintext tally but only checks the accompanying proofs. In ElectionGuard, these are Chaum-Pedersen proofs made non-interactive via the Fiat-Shamir transformation [14], so their soundness in turn rests on further assumptions, in particular on the hash function being modeled as a random oracle. Reducing tally soundness to those assumptions is out of scope for this work, which is why it is stated as an assumption of its own.

The following trust assumptions concern the parties involved in the election and the environment in which it is held.

Honest Guardian Threshold Assumption. It is assumed that at least $n_G - k + 1$ guardians are honest and only decrypt the agreed-upon result and challenged ballots.

Voter Ballot Unlinkability Assumption. It is assumed that no public link between ballot and voter exists.

Trustworthy Voting Environment Assumption. It is assumed that the voting environment is trustworthy. This includes the voting device and the physical environment of the voter.

In the following, the security guarantees of ElectionGuard are listed together with an explanation of why and under which cryptographic and trust assumptions they are ensured. Not every property of Section 2.1 appears in that list, since some of them depend entirely on the deployed system rather than on the toolkit. ElectionGuard, for example, does not model voter identities and therefore also does not store or publish which voters participated and which ballot is matched to them. Participation Privacy and Voter-Ballot Unlinkability can therefore be assumed if the organization operating the deployed system handles this data in a trustworthy manner. Its privacy guarantees are the following.

Ballot Privacy. Ballot privacy is ensured along two alternative routes, either of which suffices on its own. The first one relies on the

- *Voter Ballot Unlinkability Assumption*, since a ballot that cannot be attributed to a voter reveals nothing about that voter's choice, even if it is maliciously decrypted.

The second one also holds when a ballot can be matched to a particular voter, but then rests on three further assumptions, all of which are required together.

- *Trustworthy Voting Environment Assumption*, since a malicious device could retain the randomness used during encryption, which could then break ballot privacy.
- *Honest Guardian Threshold Assumption* prevents guardians from decrypting individual votes.
- *DLog Assumption* prevents anyone from computing the discrete logarithm of the public encryption key and decrypting votes themselves.

Receipt-Freeness. A voter who is able to retain the randomness used during encryption can present it to a coercer, who can then recompute the ciphertext and thereby learn the voter's selection. Receipt-freeness therefore rests on the

- *Trustworthy Voting Environment Assumption*, since it excludes an environment in which the voter can deviate from the protocol to the extent of obtaining the encryption randomness.

Everlasting Ballot Privacy. For a computationally unbounded adversary, the *DLog Assumption* does not hold, and the adversary can therefore compute the discrete logarithm of the public encryption key and thereby decrypt every recorded ballot. The decrypted selections remain uninformative, however, as long as no ballot can be matched to a particular voter, which is why the property relies on the

- *Voter Ballot Unlinkability Assumption*, since it is the only remaining barrier once the encryption itself has been broken.

Beyond the privacy guarantees, the following correctness guarantees are ensured by ElectionGuard.

Individual Verifiability. For every encrypted ballot, a confirmation code exists, and the voter can decide to either cast or challenge the ballot. If the code were not generated by a collision-resistant hash function, the device could open a different ballot that matches the displayed confirmation code, and when the voter casts, the device could cast a different ballot from the one the voter intended. The mapping between the ballot during casting and the published ballot is also based on the confirmation code alone, so a second ballot matching the same code would prevent the voter from recognizing the recorded ballot as their own. Both cast-as-intended and recorded-as-cast verifiability therefore depend on the

- *Collision-Resistant Hash Assumption*, since both rely on the confirmation code identifying exactly one ballot.

Universal Verifiability. Verifiers never recompute the plaintext tally themselves but check the proofs published alongside it, which in ElectionGuard are the Chaum-Pedersen proofs the guardians provide for each decryption. That the result matches the recorded ballots therefore rests on the

- *Tally Soundness Assumption*, since an accepted proof for a deviating result would otherwise remain undetected.

Taken together, ElectionGuard covers individual and universal verifiability, while eligibility verifiability is the one end-to-end property that it deliberately leaves open, since no notion of voter identity is part of the toolkit at all. Closing this gap without sacrificing its other security guarantees discussed here is the objective of the extension presented in Chapter 4.

3 Existing Approaches to Eligibility Verifiability

This chapter surveys existing voting systems that provide eligibility verifiability. For each of them, it outlines how eligibility is established, which parties have to be trusted for it, and whether participation privacy and everlasting ballot privacy are preserved. The last section derives which of these ideas carry over to ElectionGuard and thereby motivates the construction in Chapter 4.

3.1 Approaches with System-Issued Credentials

The following approaches issue credentials to eligible voters that are specific to an election. Before the voting phase, one or more registrars generate a credential for every voter on the electoral roll, hand its secret part to the voter, and publish information that allows anyone to check it later. A ballot is then accepted only if it proves possession of such a credential, which the approaches realize in different ways, ranging from encrypted credentials with zero-knowledge proofs to token-controlled encryption and signatures. Since the registrars issue the credentials, they are central to the trust assumptions of these approaches, both for eligibility and for keeping the link between a voter and their credential secret.

3.1.1 Distributed Credential Shares

Civitas [15] distributes the generation of credentials across several registration tellers. The electoral roll listing all eligible voters is public. For every entry on it, each registration teller generates one share of a credential and publishes an encryption of this share next to the entry. The tellers then hand the secret shares to the respective voter, who combines them into a credential that no single teller knows. When casting a ballot, the voter includes an encryption of this credential and a zero-knowledge proof of knowledge of it.

The main focus of Civitas is coercion resistance, not eligibility verifiability. Following A. Juels *et al.* [16], Civitas therefore enables voters to derive a fake credential that is indistinguishable from the genuine one, hand it to a coercer, and cast the actual vote under the genuine one. Accordingly, the credentials must remain encrypted, so that it is never revealed which ballots are tallied and which are discarded. Instead, the ballots and the

published credentials are mixed separately, and the encrypted credentials are compared by plaintext equivalence tests (PETs) to eliminate every ballot whose credential encrypts the same plaintext as the credential of another ballot, or as none of the published credentials. A PET requires the decryption key, which is shared among the tabulation tellers, so only they can perform the elimination. They therefore publish a proof for each test that anyone can verify. Since the mixing hides which ballots are discarded, these proofs show that only ballots under valid credentials are counted without revealing which ones.

Eligibility verifiability holds as long as one registration teller is honest, since only all of them together can forge a credential. The mix-net provides participation privacy, but not everlasting ballot privacy, since the published credential of a voter and the one in their ballot encrypt the same value and can be matched once the encryption is broken. The additionally achieved coercion resistance comes at the cost of scalability, as the PETs compare pairs of credentials, which requires $O(N^2)$ operations for N submitted ballots.

3.1.2 Token-Controlled Encryption

S. Srinivasan *et al.* [17] extend Helios [18] with token-controlled encryption (TCE). Before the election, a registrar sends each eligible voter a secret token and commits to the assignment of tokens to voters by publishing its hash. When voting, the ballot is encrypted twice. The first encryption hides the selected options, as in Helios. The second one encrypts the result under an election-specific TCE key and the voter's token. To cast the ballot, the voter additionally authenticates to the voting server with a password. After the election, the TCE secret key and the assignment of tokens to voters are published, and only ballots encrypted under an issued token decrypt successfully.

Eligibility verifiability holds as long as either the registrar issuing the tokens or the voting server authenticating the voters during casting is honest. Publishing the assignment of tokens to voters additionally allows anyone to check that tokens were only issued to eligible voters. This comes at the cost of privacy, since every ballot is thereby linked to its voter, so neither participation privacy nor everlasting ballot privacy is provided.

3.1.3 Credential-Based Ballot Signatures

Belenios [19] instantiates the generic construction of V. Cortier *et al.* [20] for Helios. A registrar generates a signing key pair for each eligible voter, sends the secret key to the voter, and passes the public keys in random order to the voting server, which publishes them. When voting, a voter signs their ballot, so anyone can verify afterward that every ballot carries a valid signature under a distinct published key. Since the voting server additionally authenticates voters by password, eligibility verifiability holds as long as either the registrar or the voting server is honest. Both, however, receive the electoral roll from

the election administrator, whom the generic construction assumes to be honest. Even if the roll were published to be audited, nothing would guarantee that the registrar and the voting server received the same one.

The published keys carry no reference to a voter, and ballots appear only next to these keys. Participation privacy and everlasting ballot privacy are therefore preserved, as long as the registrar and the voting server, which learns the link while accepting a ballot, do not disclose it.

3.2 Approaches with Preexisting Credential Infrastructure

Rather than issuing credentials specifically for an election, the following approaches rely on an existing infrastructure, such as a public key infrastructure (PKI) or an OpenID provider, to prove the eligibility of a voter. No credentials have to be distributed before the election, since voters already hold keys or accounts that identify them. In turn, trust in eligibility shifts to the operator of the infrastructure, and because the infrastructure exists precisely to tie identities to keys or accounts, each approach has to hide this link when ballots are cast under these identities to preserve participation privacy.

3.2.1 Null Ballots over a Public Key Infrastructure

O. Kulyk *et al.* [21] organize the bulletin board in one row per public key of an eligible voter, taken from an existing PKI. Since the PKI links each key publicly to a voter, a ballot in a row would reveal that this voter participated. To preserve participation privacy, ballots can therefore be added to any row, not only to one's own, which is mainly done by dedicated posting proxies. A ballot that affects the result, however, can only be added by the holder of the secret key of the row, while anyone else can only add null ballots, i.e., encryptions of the neutral element. Each ballot therefore carries a zero-knowledge proof that it either encrypts the neutral element or was cast by the holder of the secret key of its row. Since null ballots are indistinguishable from genuine ones, they hide whether a voter actually participated, while the proofs let anyone verify that only the eligible voter can influence their row. For tallying, the ballots of each row are combined homomorphically, and the resulting ciphertexts are mixed before decryption.

No registrar or voting server has to be trusted for eligibility verifiability. Instead, the PKI has to be trusted to certify only eligible voters. Participation privacy holds against a passive adversary as long as enough null ballots are posted. Everlasting ballot privacy does not hold, since every row, and thereby its combined ciphertext, is publicly linked to a voter.

3.2.2 OpenID Authentication

V. Cortier *et al.* [22] use the standardized OpenID protocol [23] to authorize ballots. The voting client hashes the ballot together with a nonce and sends the hash to the voting server, which derives a second nonce from it and includes it in the authentication request to the identity provider. After successful authentication, the identity provider issues a signed token containing the voter's identity and this nonce.

Instead of the token, the voting server publishes its hash together with a zero-knowledge proof that it knows a voter identity and a nonce from which this hash and the ballot commitment can be reconstructed. Together with the signature of the identity provider, this proof is publicly verifiable, while hiding which voter cast which ballot. Therefore, participation privacy and everlasting ballot privacy are preserved. Eligibility verifiability, however, requires a trustworthy identity provider, which is a strong assumption, since a single authority is responsible for authorizing every ballot.

3.2.3 Anonymous Signatures via Public Key Shuffling

R. Haenni and O. Spycher [24] build on an existing DSA public key infrastructure [25], in which voters sign their ballots with their existing key pairs. To unlink voters and ballots, a set of trustees, called anonymizers, successively shuffle the public keys and raise them, together with the group generator, to a secret exponent. Each anonymizer publishes a proof of correct shuffling. Voters then sign their ballot with their unchanged secret key, but relative to the new generator. Thus, anyone can check that each ballot verifies under one of the anonymized keys without learning which voter it belongs to.

As before, eligibility verifiability rests on a trustworthy PKI. Participation privacy holds as long as one anonymizer keeps its permutation and exponent secret. Everlasting ballot privacy, however, does not hold, since an anonymized key is derived from the same secret key as the original one. A computationally unbounded adversary can compute the discrete logarithms of both and match them, which links every ballot to its author.

3.3 Applicability to ElectionGuard

The surveyed approaches differ mainly in where the credentials originate, which party besides their issuer has to be trusted, and how the credentials are tied to the electoral roll. The following section examines, for each of these aspects, which approach suits ElectionGuard best and can thus serve as a starting point for the extension.

Regarding their origin, credentials issued specifically for the election are preferable to using PKI keys as credentials. A PKI links each voter publicly to their key, which precludes

everlasting ballot privacy and makes additional mechanisms necessary to restore participation privacy. However, a PKI can be used to deliver the system-issued credentials to the voters, as long as the ballots are not cast under the voters' own keys. Among the approaches with issued credentials, Civitas pays for coercion resistance with considerable complexity, which is not justified for ElectionGuard, since ElectionGuard itself does not aim at coercion resistance.

As for the trusted parties, TCE, Belenios, and the OpenID-based approach rely on a second party that authenticates every voter during casting, namely the voting server or the identity provider. ElectionGuard, however, is also deployed alongside in-person elections, in which such a party is not necessarily available. Civitas works around this by having several registration tellers generate every credential in shares beforehand, so that a single honest teller suffices.

The same idea is mentioned as an option for the TCE approach and for the generic construction behind Belenios [17], [20], but is not worked out there. Of these two mechanisms, however, TCE offers no real advantage over signatures. While a signature is verified under a public key, checking a ballot under TCE requires the secret token itself. As anyone who knows a token can cast a ballot under it, the tokens can only be revealed after the voting phase. The registrars would then have to remain involved after the election, the bulletin board would have to be fixed before the tokens are revealed, and invalid ballots could not be rejected when they are cast.

Finally, concerning the electoral roll, binding it publicly to the credentials, as in Civitas and TCE, requires the roll itself to be published. In many elections, however, this is not permitted, and the roll is only disclosed to designated parties [26]. Yet, without a binding, as in Belenios, the roll itself has to be trusted. Therefore, combining signatures with distributed credential issuance and with a binding to the electoral roll that keeps the roll confidential is the starting point of the construction in Chapter 4.

4 Multi-Signature-Based Approach to Eligibility Verifiability

This chapter works out the construction motivated at the end of Chapter 3 and extends the ElectionGuard toolkit so that anyone can verify that only votes cast by eligible voters were counted. To this end, it begins by formally defining a general notion of a voting scheme and the verifiability of a scheme in which the adversary has access to all trustees other than one registrar. This is followed by a textual and a formal description of the extension, its new trust assumptions, and its security guarantees. Finally, eligibility verifiability is proven for the extended scheme.

4.1 Syntax of a Voting Scheme and Verifiability with One Honest Registrar

Throughout this chapter, the following general notion of a voting scheme is used. It consists of the algorithms $V = (\text{Setup}, \text{CredentialShare}, \text{ValidateRegistration}, \text{Vote}, \text{ValidateBallot}, \text{VerifyVote}, \text{Box}, \text{ValidateBoard}, \text{Tally}, \text{Verify})$. The security parameter κ is given to Setup in unary as 1^κ . Every other algorithm receives the public part pub of the output of Setup as an implicit input.

The voting scheme is defined relative to a result function $\rho : \mathbb{V}^* \rightarrow Res$, which maps a finite sequence of votes $v \in \mathbb{V}$ to a result $r \in Res$. It is applied by Tally to the recorded votes and is permutation invariant, since the order in which ballots are recorded must not affect the result.

The individual algorithms are defined as follows.

Setup(1^κ). Is executed once by the election administrator. It returns a pair consisting of public and secret data (pub, sec) . The secret data includes the electoral roll $\mathcal{L}$ and the key sk to decrypt the votes or the tally. Public data includes the set of registrars $\mathcal{R}$ of size $m := |\mathcal{R}|$, the size of the electoral roll $n := |\mathcal{L}|$, a registration list $\mathcal{L}_{\text{Reg}}$, given as a set of credentials, and the encryption key pk used during voting. The bulletin board BB , in contrast, is a sequence of ballots, since the order in which ballots are recorded is observable. The public part is published and implicitly passed to every other algorithm, whereas the secret part is passed only to those algorithms that require it.

CredentialShare($id, \mathcal{L}$). Is executed by a single registrar $\mathcal{R}_i \in \mathcal{R}$ in order to contribute one share of the credential of the voter id . If $id \notin \mathcal{L}$, no share is produced. It returns a public share pcs_i , which is published, and a secret share scs_i , which is sent to the voter over a secret channel. Once every registrar has contributed, the public credential of the voter is the m -tuple $pc_{id} := (pcs_1, \dots, pcs_m)$, ordered by registrar index, which is added to $\mathcal{L}_{\text{Reg}}$, and the secret credential known to the voter is $sc_{id} := (scs_1, \dots, scs_m)$.

ValidateRegistration($\cdot$). Is executed by any verifier and returns 1 if the registration list $\mathcal{L}_{\text{Reg}}$ is well-formed and 0 otherwise.

Vote(pc, sc, v). Is executed by a voter in order to cast a vote $v \in \mathbb{V}$. It returns a ballot b which is sent to the bulletin board.

ValidateBallot(b). Is executed by any verifier and returns 1 if the ballot b is well-formed and 0 otherwise.

VerifyVote(BB, id, b). Can be executed by a voter in order to check that the ballot b cast under the identity id is contained in the bulletin board BB and will therefore be included in the tally. It returns 1 if the check succeeds and 0 otherwise.

Box(BB, b). Is executed by the bulletin board manager upon receiving a ballot b . It updates the bulletin board if some checks including **ValidateBallot**(b) succeed. Otherwise, it leaves BB unchanged.

ValidateBoard(BB). Is executed by any verifier and returns 1 if the bulletin board is well-formed, e.g., every ballot on it passes **ValidateBallot**, and 0 otherwise.

Tally(BB, sk). Is executed by the trustees after the voting phase has ended. It outputs the result $r \in \text{Res}$ together with a proof of correct tabulation Π . If some checks, including **ValidateRegistration**($\cdot$) or **ValidateBoard**(BB), fail, it outputs $r := \perp$, meaning that the election has been declared invalid.

Verify(BB, r, Π). Can be executed by anyone. It returns 1 if **ValidateBoard**(BB) = 1, **ValidateRegistration**($\cdot$) = 1, and Π is a correct proof of the tally result r of the bulletin board BB , and 0 otherwise.

A voting scheme $\mathbf{V}$ with at least one honest registrar is said to be verifiable against all other trustees being dishonest if, for every probabilistic polynomial-time adversary $\mathcal{A}$, the advantage

$$\text{Adv}_{\mathcal{A}, \mathbf{V}}^{\text{ver}}(\kappa) := \Pr[\text{Exp}_{\mathcal{A}, \mathbf{V}}^{\text{ver}}(1^\kappa) = 1]$$

is negligible in κ . This single notion covers eligibility verifiability and universal verifiability at the same time, as the decomposition of the winning condition at the end of this section makes explicit.

The experiment $\text{Exp}_{\mathcal{A}, \mathcal{V}}^{\text{ver}}$ is defined in Figure 3 and is based on the definition of V. Cortier *et al.* [20] against a dishonest bulletin board and an honest registrar, adapted to m registrars of which only one needs to be honest. Without loss of generality, this registrar is $\mathcal{R}_1$.

The experiment first sets up the electoral roll and the election parameters and hands them to the adversary, together with access to the four oracles $\mathcal{O}\text{Reg}$ (split into two steps for interactivity), $\mathcal{O}\text{Vote}$, $\mathcal{O}\text{Check}$, and $\mathcal{O}\text{Corrupt}$. Registering a voter first requires a call to $\mathcal{O}\text{Reg}_1$, in which the adversary supplies the *id* of the voter to register. The secret and public shares of $\mathcal{R}_1$ are produced by the experiment, and the public share is returned to the adversary. It can then produce the shares of the other $m - 1$ registrars it controls, even as a function of the received honest share, and can call $\mathcal{O}\text{Reg}_2$ with them. The secret shares supplied by the adversary simulate the ones sent to the voter and do not need to be correct. If they are not, the adversary can still query $\mathcal{O}\text{Vote}$ for this voter, yet the resulting ballot does not pass ValidateBallot and therefore never reaches the bulletin board.

Legitimate ballots for registered voters are obtained through $\mathcal{O}\text{Vote}$, which answers at most one query per voter. The individual verification performed by a voter, who checks that the ballot has been recorded on the bulletin board, is modeled by $\mathcal{O}\text{Check}$. Finally, $\mathcal{O}\text{Corrupt}$ hands the secret credential of a registered voter to the adversary, who can then construct ballots for that voter without the voting oracle. The experiment maintains the table $\mathcal{U}$ of registered voters together with their credentials.

The adversary wins if it produces a bulletin board, a result, and a proof of correct tabulation such that verification succeeds, all checked ballots are contained in the bulletin board, and the result differs from any result that counts all checked votes, a subset of the unchecked honest votes, and at most one vote per corrupted voter.

The winning condition decomposes into two components. The constraint $0 \leq n_B \leq |\mathcal{U}|$ is the eligibility component, which means that no matter how the adversary manipulates the bulletin board, it cannot contribute more votes than it has corrupted voters. This is exactly the absence of ballot stuffing for a fixed electoral roll. The remaining part of the condition, namely that all checked votes and a subset of the unchecked honest votes are counted with their intended plaintexts, is the universal verifiability component. Recorded-as-cast verifiability is covered by $\mathcal{O}\text{Check}$, whereas cast-as-intended verifiability is not, since the honest votes are encrypted by the experiment rather than by a device the adversary controls. In the construction, this gap is closed by the Benaloh challenge inherited from ElectionGuard.

4.2 ElectionGuard with Multi-Signature-Based Credentials

This section describes how ElectionGuard is extended so that voting systems built on it can provide eligibility verifiability. Since ElectionGuard is not a voting system itself, but

$\text{Exp}_{\mathcal{A}, \mathcal{V}}^{\text{ver}}(1^\kappa)$	
$1 \ \mathcal{U} := \emptyset, \ \mathcal{CU} := \emptyset$ $2 \ \text{HVote} := \emptyset, \ \text{Checked} := \emptyset$ $3 \ (pub, sec) \leftarrow \text{Setup}(1^\kappa)$ $4 \ session := []$ $5 \ (BB, r, \Pi) \leftarrow \mathcal{A}^{\mathcal{OReg}, \mathcal{OVote}, \mathcal{OCheck}, \mathcal{OCorrupt}}(pub, sec)$ $6 \ \text{if } \text{Verify}(BB, r, \Pi) = 0 \ \text{return } 0$ $7 \ \text{if } \exists (id, *, b) \in \text{Checked} : \text{VerifyVote}(BB, id, b) \neq 1 \ \text{return } 0$ $8 \ \text{if } \exists (id_1^A, v_1^A, *), \dots, (id_{n_A}^A, v_{n_A}^A, *) \in \text{HVote} \setminus \text{Checked}$ $\quad \exists v_1^B, \dots, v_{n_B}^B \in \mathbb{V} \text{ s.t. } 0 \leq n_B \leq \mathcal{CU} $ $\quad \text{s.t. } r = \rho(v_1^A, \dots, v_{n_A}^A, v_1^B, \dots, v_{n_B}^B, v_1^C, \dots, v_{n_C}^C)$ $\quad \text{where } \text{Checked} = \{(id_1^C, v_1^C, b_1^C), \dots, (id_{n_C}^C, v_{n_C}^C, b_{n_C}^C)\}$ $9 \quad \ \text{return } 0$ $10 \ \text{else return } 1$	
$\mathcal{OReg}_1(id)$	$\mathcal{OCorrupt}(id)$
$1 \ \text{if } id \notin \mathcal{L} \vee (id, *, *) \in \mathcal{U} \vee$ $\quad session[id] \neq \perp$ $2 \quad \ \text{return } \perp$ $3 \ (pcs_1, scs_1) \leftarrow \text{CredentialShare}(id, \mathcal{L})$ $4 \ session[id] := (pcs_1, scs_1)$ $5 \ \text{return } pcs_1$	$1 \ \text{if } (id, *, *) \notin \mathcal{U} \ \text{return } \perp$ $2 \ \mathcal{CU} := \mathcal{CU} \cup \{id\}$ $3 \ \text{return } sc_{id} \text{ with } (id, *, sc_{id}) \in \mathcal{U}$
$\mathcal{OReg}_2(id, (pcs_i, scs_i)_{i=2}^m)$	$\mathcal{OVote}(id, v)$
$1 \ \text{if } session[id] = \perp \ \text{return } \perp$ $2 \ (pcs_1, scs_1) := session[id]$ $3 \ pc_{id} := (pcs_1, \dots, pcs_m)$ $4 \ sc_{id} := (scs_1, \dots, scs_m)$ $5 \ \mathcal{U} := \mathcal{U} \cup \{(id, pc_{id}, sc_{id})\}$ $6 \ \mathcal{L}_{\text{Reg}} := \mathcal{L}_{\text{Reg}} \cup \{pc_{id}\}$ $7 \ session[id] := \perp$ $8 \ \text{return } pc_{id}$	$1 \ \text{if } (id, *, *) \notin \mathcal{U} \vee v \notin \mathbb{V}$ $2 \quad \ \text{return } \perp$ $3 \ \text{if } \exists v' : (id, v', *) \in \text{HVote} \ \text{return } \perp$ $4 \ b \leftarrow \text{Vote}(pc_{id}, sc_{id}, v) \text{ with } (id, pc_{id}, sc_{id}) \in \mathcal{U}$ $5 \ \text{HVote} := \text{HVote} \cup \{(id, v, b)\}$ $6 \ \text{return } b$
	$\mathcal{OCheck}(id, b)$
	$1 \ \text{if } \nexists v : (id, v, b) \in \text{HVote}$ $2 \quad \ \text{return } \perp$ $3 \ \text{Checked} := \text{Checked} \cup \{(id, v, b)\}$ $4 \ \text{return } 1$

Figure 3: Security experiment for verifiability with one honest registrar. All registrars except $\mathcal{R}_1$ and all other trustees are controlled by the adversary.

a collection of cryptographic protocols from which a voting system can be assembled, the extension is still intended to work with different types of voting systems based on the toolkit.

The main addition and the component that provides eligibility verifiability, for a given electoral roll, is the inclusion of a multi-signature scheme. In the extension, every ballot carries a signature and a public verification key, also called a public credential, under which it verifies. A ballot is only recorded if its signature is valid under that credential, and the credential must have been published before the voting phase began. Every eligible voter therefore has to hold a secret credential in advance to sign their ballot. However, this credential does not consist of a single key but of a tuple of key shares, each generated independently by a separate trustee. A signature is only valid once every share has contributed to it. No single trustee can therefore produce a credential that can be used to cast a ballot.

Supporting this component requires two additions to the election record. Before the voting phase, the list of public credentials and a commitment to the list of eligible voters are published. The credential list is what makes it possible to record only ballots carrying a valid credential and signature, and what allows anyone to verify this afterward from the record alone. The commitment allows auditors to obtain the same electoral roll the credentials were generated for and to verify that these voters are indeed eligible.

Accordingly, the extension introduces two new roles, the registrars and the auditor, and extends three of the roles described in Section 2.5. Only the guardians remain unaffected.

Registrars. The registrars are a new group of trustees, of which there must be at least two, and they carry out the registration phase before the voting phase. Before generating any share, each registrar recomputes the published commitment from the electoral roll it received and aborts the registration if the two do not match. Since a credential is only usable once every registrar has contributed its share, a single honest registrar performing this check is enough to prevent credentials from being generated for a roll other than the committed one. Each registrar then independently generates one share of the credential of every eligible voter, consisting of a secret and a public part. The public shares are published, and their order within a credential is determined by the registrar they originate from. The secret shares are sent to the corresponding voter over a secure channel, which is specified by the contact information for each voter that the registrar receives beforehand. The registrars therefore know which credential belongs to which voter, but the published credentials themselves carry no reference to a voter. After the registration phase, each registrar reconciles the published credential list with its own view and complains if a credential appears that it did not contribute a share to, or if one of its shares occurs in more than one credential.

Administrator. In addition to its previous tasks, the administrator selects the set of registrars and defines the electoral roll of the election, which can consist of several contests with

different eligible voters. The roll therefore specifies in which contests a voter is authorized to vote. Before the registration phase, the administrator commits to the roll and publishes the commitment. Depending on the election, the roll must be handled confidentially, and its opening is only shared with specific parties, such as the registrars and auditors.

Guardians. The role is not affected by the extension. Key generation, decryption of the tallies, and publication of the election record proceed exactly as before.

Voters. A voter receives the secret credential shares during the registration phase and has to retain them until voting. Option selection, ballot encryption, and challenging the ballot work as in standard ElectionGuard. When the ballot is to be cast, the voter additionally supplies the secret shares, which are used to sign the encrypted ballot, and the ballot is then cast together with the public credential and the signature. A ballot whose signature does not verify, whose credential is not registered, or whose credential already occurs on a recorded ballot is rejected, so no voter can have more than one ballot recorded.

Verifiers. A verifier still checks the correctness of the tally aggregation, its decryption, and the well-formedness of every recorded ballot. In addition, the registration process and the link between ballots and credentials become verifiable. On the published registration list, a verifier checks that the credentials are pairwise distinct, that there are no more of them than there are eligible voters, and that each of them consists of one share per registrar. On the ballots, a verifier checks that the attached signature is valid under the public credential the ballot carries, that this credential occurs on the registration list, and that no credential occurs on more than one ballot included in the tally. Together, these checks decide eligibility from the election record alone, without any knowledge of who is on the electoral roll.

Auditor. An auditor examines the electoral roll by checking its entries against an independent record of eligibility such as a population register. This cannot be done universally by any verifier, since the roll has to be treated confidentially in some elections. Since this process differs considerably between elections, the extension does not specify who becomes an auditor, when the inspection takes place, or how it is done. Instead, it ensures that an auditor receives the same roll that the registrars received to generate the credentials. To this end, an auditor receives opening information for the published commitment. If opening the commitment fails, the auditor can contest the election, which is outside the scope of this specification.

To make the extension accessible to the security experiment of Section 4.1, a basic voting scheme built from the extended toolkit is described formally in the following. It consists of the algorithms $V^{EG+} = (\text{Setup}, \text{CredentialShare}, \text{ValidateRegistration}, \text{Vote}, \text{ValidateBallot}, \text{VerifyVote}, \text{Box}, \text{ValidateBoard}, \text{Tally}, \text{Verify})$ as defined in Section 4.1.

The result function $\rho : \mathbb{V}^* \rightarrow \text{Res}$ is the counting function of ElectionGuard, where a vote $v \in \mathbb{V}$ is a vector of selections and ρ returns their component-wise sum, i.e., the number of votes for each option. It is defined on finite sequences and is permutation invariant as required in Section 4.1.

The credentials of the eligible voters are issued jointly by a set of registrars $\mathcal{R} = \{\mathcal{R}_1, \dots, \mathcal{R}_m\}$, using a multi-signature scheme $\mathbf{MS} = (\text{Setup}, \text{KeyGen}, \text{Sign}, \text{Verify})$ as defined in Section 2.3, which is existentially unforgeable under chosen-message attacks in the sense of Figure 1. The public credential pc of a voter takes the role of the key list L of that definition. Since pc is an m -tuple ordered by registrar index, it is itself the unique encoding of the multiset $\{pcs_1, \dots, pcs_m\}$ required there. It is required that an honest execution of all m roles of $\mathbf{MS}.\text{Sign}$ on a message c , under a credential whose shares were produced by m independent executions of $\mathbf{MS}.\text{KeyGen}$, yields a signature σ with $\mathbf{MS}.\text{Verify}(pc, c, \sigma) = 1$ with overwhelming probability, and that $\mathbf{MS}.\text{Verify}$ is deterministic on all inputs, including public shares outside the range of $\mathbf{MS}.\text{KeyGen}$.

Votes are encrypted under a public key encryption scheme with algorithms Enc and Dec , and a homomorphic aggregation of ciphertexts, whose key pair is generated during Setup . The scheme is required to be correct, that is, $\text{Dec}_{sk}(\text{Enc}_{pk}(v)) = v$ holds with overwhelming probability for every $v \in \mathbb{V}$, and the aggregation is required to be compatible with ρ , that is, decrypting the aggregate of $\text{Enc}_{pk}(v_1), \dots, \text{Enc}_{pk}(v_\ell)$ yields $\rho(v_1, \dots, v_\ell)$ with overwhelming probability.

The electoral roll $\mathcal{L}$ is fixed publicly by a perfectly hiding commitment scheme $\mathbf{Com} = (\text{Setup}, \text{Commit}, \text{Open})$ as defined in Section 2.4, where $\mathbf{Com}.\text{Setup}$ generates the system parameters, $(com, dec) \leftarrow \mathbf{Com}.\text{Commit}(\mathcal{L})$ produces a commitment com together with the opening information dec , and $\mathbf{Com}.\text{Open}(com, dec, \mathcal{L}) \in \{0, 1\}$ checks an opening.

The individual algorithms are defined as follows.

Setup(1^κ). Is executed once by the election administrator. It generates the system parameters $pars \leftarrow \mathbf{MS}.\text{Setup}(1^\kappa)$ of the multi-signature scheme, the system parameters of the commitment scheme through $\mathbf{Com}.\text{Setup}(1^\kappa)$, and the standard ElectionGuard system parameters and key pair (pk, sk) used to encrypt and decrypt votes. It also selects the set of registrars $\mathcal{R}$ of size $m := |\mathcal{R}|$ and the electoral roll $\mathcal{L}$ of size $n := |\mathcal{L}|$, commits to the roll as $(com, dec) \leftarrow \mathbf{Com}.\text{Commit}(\mathcal{L})$, and initializes the registration list $\mathcal{L}_{\text{Reg}}$ as empty. It returns a pair (pub, sec) with $pub := (pars, \mathcal{R}, m, n, \mathcal{L}_{\text{Reg}}, pk, com)$ and $sec := (\mathcal{L}, dec, sk)$.

CredentialShare($id, \mathcal{L}, dec$). Is executed by a single registrar $\mathcal{R}_i \in \mathcal{R}$ in order to contribute one share of the credential of the voter id . Compared to the syntax of Section 4.1, it additionally receives the opening information dec from the secret data sec , which the experiment can supply, since it holds sec . If $id \notin \mathcal{L}$ or if $\mathbf{Com}.\text{Open}(com, dec, \mathcal{L}) = 0$, no share is produced. Otherwise, the registrar runs $\mathbf{MS}.\text{KeyGen}()$ and returns a share pair

(pcs_i, scs_i) . The public share pcs_i is published, and the secret share scs_i is sent to the voter over a secret channel. Once every registrar has contributed, the public credential of the voter is the m -tuple $pc_{id} := (pcs_1, \dots, pcs_m)$, ordered by registrar index, which is added to $\mathcal{L}_{\text{Reg}}$, and the secret credential known to the voter is $sc_{id} := (scs_1, \dots, scs_m)$.

ValidateRegistration(). Is executed by any verifier and returns 1 if the registration list $\mathcal{L}_{\text{Reg}}$ contains at most n credentials for an electoral roll of n voters, if these credentials are pairwise distinct, and if every credential in it consists of exactly m shares, one for each registrar, and 0 otherwise.

Vote(pc, sc, v). Is executed by a voter in order to cast a vote $v \in \mathbb{V}$. It first encrypts the vote into a ciphertext $c \leftarrow \text{Enc}_{pk}(v)$ and then signs c by executing all m roles of MS.Sign locally, using the shares contained in sc . Finally, it returns the ballot $b := (c, \sigma, pc)$, which is sent to the bulletin board.

ValidateBallot(b). Is executed by any verifier and returns 1 if the ballot $b = (c, \sigma, pc)$ is well-formed and 0 otherwise. A ballot is well-formed if c is a valid encryption of exactly one vote in $\mathbb{V}$, if $\text{MS.Verify}(pc, c, \sigma) = 1$, and if pc is one of the registered credentials in $\mathcal{L}_{\text{Reg}}$.

VerifyVote(BB, id, b). The generic definition from Section 4.1 is adopted unchanged. The check returns 1 if and only if the ballot b is contained in the bulletin board BB , and 0 otherwise.

Box(BB, b). Is executed by the bulletin board manager upon receiving a ballot $b = (c, \sigma, pc)$. It updates the bulletin board if $\text{ValidateBallot}(b) = 1$ and no ballot $b' \in BB$ carries the same public credential pc . Otherwise, it leaves BB unchanged.

ValidateBoard(BB). Is executed by any verifier and returns 1 if every ballot of the bulletin board $b = (c, \sigma, pc) \in BB$ satisfies $\text{ValidateBallot}(b) = 1$ and no other ballot $b' \in BB$ carries the same public credential pc , and 0 otherwise.

Tally(BB, sk). Is executed by the guardians after the voting phase has ended. It homomorphically aggregates the ciphertexts of the recorded ballots and decrypts only the resulting aggregate using Dec_{sk} , so that no individual ballot is decrypted. By the compatibility of the aggregation with ρ , the decrypted aggregate is the result $r \in \text{Res}$ of the recorded votes, which is returned together with a proof of correct tabulation Π . If $\text{ValidateRegistration}() = 0$ or $\text{ValidateBoard}(BB) = 0$, it returns $r := \perp$, meaning that the election has been declared invalid.

Verify(BB, r, Π). Can be executed by anyone. It returns 1 if $\text{ValidateBoard}(BB) = 1$, $\text{ValidateRegistration}() = 1$, and Π is a correct proof of the tally result r of the bulletin board BB , and 0 otherwise.

The voting scheme is also assumed to admit tally soundness, that is, for every probabilistic polynomial-time algorithm $\mathcal{T}$, the advantage

$$\text{Adv}_{\mathcal{T}, \mathbf{V}^{\text{EG}+}}^{\text{tally-snd}}(\kappa) := \Pr[\text{Verify}(BB, r, \Pi) = 1 \wedge r \neq \rho(\text{Dec}_{sk}(c_1), \dots, \text{Dec}_{sk}(c_\ell)) \mid (BB, r, \Pi) \leftarrow \mathcal{T}(\text{pub}, \text{sec})]$$

is negligible in κ , where $BB = (b_1, \dots, b_\ell)$ and c_j denotes the ciphertext of the ballot b_j . The algorithm $\mathcal{T}$ receives the secret data sec as well, since the guardians producing the tally are the party quantified over. This means that the resulting tally r and the proof Π output by the guardians are accepted by the public verification algorithm only if r is indeed the result of the encrypted votes contained in the ballots on the bulletin board. This is the *Tally Soundness Assumption* of Section 2.5.1, stated concretely for $\mathbf{V}^{\text{EG}+}$.

4.2.1 Assumptions and Security Guarantees

The extension inherits the assumptions of ElectionGuard listed in Section 2.5.1 and adds assumptions of its own. Below, the new assumptions and a refined *Voter Ballot Unlinkability Assumption* are described and grouped into cryptographic assumptions and trust assumptions on the parties involved.

The cryptographic assumptions are the following:

Multi-Signature Unforgeability Assumption. It is assumed that the multi-signature scheme MS used to issue the credentials is EUF-CMA secure in the sense of Figure 1.

Commitment Binding Assumption. It is assumed that the commitment scheme Com used to fix the electoral roll is computationally binding as defined in Section 2.4.

Commitment Perfectly Hiding Assumption. It is assumed that the commitment scheme Com used to fix the electoral roll is perfectly hiding as defined in Section 2.4.

The trust assumptions are the following:

One Honest Registrar Assumption. It is assumed that at least one of the registrars is honest. This includes that the registrar does not collude with the other ones by sharing its secret credential shares, and that the registrar reconciles the published credential list with its own view and complains about any deviation. Furthermore, a secret channel must be used to send the credential share to the voter.

Voter Ballot Unlinkability Assumption. As defined in Section 2.5.1, it is assumed that no public link between voter and ballot exists. With the introduction of the registrars and the credentials, it is refined by two further requirements. First, all registrars are honest and do not share the link between credential and voter with another party. Second, all channels used by the registrars to send the credential share to a voter are secure.

Audited Electoral Roll Assumption. It is assumed that an independent record of eligibility exists, that the electoral roll is audited against it, and that a failed audit leads to the election being contested.

Complaining Voters Assumption. It is assumed that a voter who receives no credential or an invalid one complains, and that a voter who finds a ballot cast under their own credential without their participation complains as well. It is further assumed that the election organization acts on such complaints.

The security guarantees of the extension are discussed below. First, the following two privacy guarantees are revisited, since they have refined assumptions.

Ballot Privacy. As in standard ElectionGuard, ballot privacy is ensured along two alternative routes. The extension adds a link between a voter and a recorded ballot through the credential. However, this link is known only to the registrars. Therefore, the property holds either if the following two assumptions hold:

- *Voter Ballot Unlinkability Assumption*, since the credential is the only link the extension adds between a voter and a recorded ballot.
- *Commitment Perfectly Hiding Assumption*, since the published commitment would otherwise disclose the electoral roll the credentials were issued for.

Or, as before, if they do not hold, it still rests on the following three assumptions:

- *Trustworthy Voting Environment Assumption*, since a malicious device could retain the randomness used during encryption, which could then break ballot privacy.
- *Honest Guardian Threshold Assumption* prevents guardians from decrypting individual votes.
- *DLog Assumption* prevents anyone from computing the discrete logarithm of the public encryption key and decrypting votes themselves.

Everlasting Ballot Privacy. As in standard ElectionGuard, this property follows from the unlinkability between voter and ballot and therefore also rests on the following two assumptions:

- *Voter Ballot Unlinkability Assumption*, since it is the only remaining barrier once the encryption itself has been broken.
- *Commitment Perfectly Hiding Assumption*, since the commitment stays published after the election and only a perfectly hiding one keeps the roll secret from an adversary that is computationally unbounded.

Lastly, the correctness guarantees, established through individual and universal verifiability, remain unchanged from standard ElectionGuard under the same assumptions and are therefore not revisited here. Eligibility verifiability is the property the extension is built to provide. The assumptions it rests on are described below.

Eligibility Verifiability. The guarantee is established in two layers. First, the signed ballots ensure that no adversary can contribute more ballots than there are credentials generated by the registrars, and that the registrars generate one credential per voter specified by an electoral roll. This layer relies on the following cryptographic assumptions:

- *Multi-Signature Unforgeability Assumption* prevents forging signatures under valid credentials.
- *Tally Soundness Assumption* prevents forging valid tabulation proofs for a result that does not match the tally of the eligible ballots.

It further relies on the following trust assumption:

- *One Honest Registrar Assumption*, since the registrars could otherwise collude and stuff ballots.

The second layer confirms that the roll itself corresponds to the actually eligible voters. This is established statistically by an audit and therefore depends on one more trust assumption:

- *Audited Electoral Roll Assumption*, since an adversary who controls the roll itself can add identities to it and obtain legitimate credentials for them.

The commitment to the electoral roll ties the two layers together, since it binds the roll the credentials were issued for to the roll the auditors inspect, which requires one more cryptographic assumption:

- *Commitment Binding Assumption*, since a roll that opens the commitment in two ways would let the registrars issue credentials for one roll while the audit examines another.

Eligibility verifiability is proven in the next section, Section 4.3, given the stated assumptions.

Below, a weaker form of eligibility verifiability is described, which holds even if all registrars collude and behave dishonestly. It is neither formalized nor proven in Section 4.3. It describes what remains of eligibility verifiability once trust in the registrars fails entirely, rather than an additional guarantee on top of the main result.

Individual Eligibility Verifiability. Every eligible voter can detect dishonest behavior of the registrars and complain to the election organization. The number of issued credentials and the number of eligible voters are both public and can be compared by any verifier, so dishonest registrars cannot issue more credentials than there are eligible voters. Stuffing a ballot therefore requires the credential of an eligible voter. That voter either receives no credential or an invalid one, or receives a valid one and finds a second ballot under it on the bulletin board. In both cases, the voter can complain. The guarantee rests on the following cryptographic assumptions:

- *Commitment Binding Assumption*, since the roll the credentials were issued for would otherwise differ from the audited one.

- *Multi-Signature Unforgeability Assumption*, which prevents forging a signature under a valid credential.
- *Tally Soundness Assumption*, which prevents forging a tabulation proof for a result that does not match the recorded ballots.

In addition, the following two trust assumptions are needed:

- *Audited Electoral Roll Assumption*, since an adversary who controls the roll itself can add identities to it and obtain legitimate credentials for them.
- *Complaining Voters Assumption*, since a detection that no one acts on changes nothing.

4.3 Proof of Eligibility Verifiability

The previous section defined the voting scheme $V^{\text{EG}+}$ representing the ElectionGuard extension and divided its eligibility verifiability guarantee into two layers with the specific trust and cryptographic assumptions needed for it to hold. In this section, the property is formalized in separate theorems. The first layer is captured by the security experiment defined in Section 4.1, instantiated for the voting scheme $V^{\text{EG}+}$, where the trust assumption is implemented by the oracles and the cryptographic assumptions are the ones the experiment is reduced to. It is expressed by the following theorem.

Theorem 1. *Let $V^{\text{EG}+}$ be a voting scheme whose credentials are issued by the multi-signature scheme MS and let $n := |\mathcal{L}|$ denote the number of eligible voters. For every probabilistic polynomial-time adversary $\mathcal{A}$ there exist probabilistic polynomial-time algorithms $\mathcal{F}$ and $\mathcal{T}$ such that*

$$\text{Adv}_{\mathcal{A}, V^{\text{EG}+}}^{\text{ver}}(\kappa) \leq n \cdot \text{Adv}_{\mathcal{F}, \text{MS}}^{\text{EUF-CMA}}(\kappa) + \text{Adv}_{\mathcal{T}, V^{\text{EG}+}}^{\text{tally-snd}}(\kappa).$$

Proof of Theorem 1. The proof proceeds in two steps. A first game hop removes the possibility that the announced result deviates from the content of the bulletin board. The remaining game is then reduced to the unforgeability of the multi-signature scheme.

Let G_0 denote the experiment $\text{Exp}_{\mathcal{A}, V^{\text{EG}+}}^{\text{ver}}(1^\kappa)$. As a first step, it is ruled out that $\mathcal{A}$ wins by making the verifier accept an announced result that does not match the bulletin board. Let $\text{bad}_{\text{tally}}$ denote the event that, in a run of G_0 , $\mathcal{A}$ outputs (BB, r, Π) with $BB = (b_1, \dots, b_\ell)$ such that $\text{Verify}(BB, r, \Pi) = 1$ although $r \neq \rho(\text{Dec}_{sk}(c_1), \dots, \text{Dec}_{sk}(c_\ell))$ for the ciphertexts c_j contained in the ballots of BB . Let G_1 be identical to G_0 , except that it outputs 0 whenever $\text{bad}_{\text{tally}}$ occurs. The probability that G_0 is won is therefore bounded by

$$\Pr[G_0 = 1] \leq \Pr[G_1 = 1] + \Pr[\text{bad}_{\text{tally}}].$$

To bound $\Pr[\text{bad}_{\text{tally}}]$, consider an algorithm $\mathcal{T}$ against the tally soundness of $V^{\text{EG}+}$. On input (pub, sec) , $\mathcal{T}$ invokes $\mathcal{A}$ on the same input and answers every oracle query of $\mathcal{A}$ exactly as G_0 would, maintaining the sets $\mathcal{U}$, $\mathcal{L}_{\text{Reg}}$, $\mathcal{CU}$, HVote , and Checked itself. This

is possible since sec contains the decryption key sk and the electoral roll $\mathcal{L}$. When $\mathcal{A}$ halts with output (BB, r, Π) , $\mathcal{T}$ outputs the same triple. The run of G_0 simulated by $\mathcal{T}$ is thus distributed exactly as G_0 itself, so the output of $\mathcal{T}$ meets the condition of $\text{Adv}_{\mathcal{T}, \mathbf{V}^{\text{EG}+}}^{\text{tally-snd}}$ precisely if $\text{bad}_{\text{tally}}$ occurs in that run, and therefore $\Pr[\text{bad}_{\text{tally}}] = \text{Adv}_{\mathcal{T}, \mathbf{V}^{\text{EG}+}}^{\text{tally-snd}}(\kappa)$. Combining this with the game hop gives

$$\Pr[G_0 = 1] \leq \Pr[G_1 = 1] + \text{Adv}_{\mathcal{T}, \mathbf{V}^{\text{EG}+}}^{\text{tally-snd}}(\kappa).$$

By definition of G_1 , whenever G_1 does not abort due to $\text{bad}_{\text{tally}}$, the announced result equals the result function applied to the plaintexts of the ballots on the bulletin board.

It remains to bound $\Pr[G_1 = 1]$. From an adversary $\mathcal{A}$ in G_1 a forger $\mathcal{F}$ against **MS** is constructed. $\mathcal{F}$ is given the parameters $pars$ of **MS** and the public key pks^* of the single honest signer, and it has access to the signing oracle $\mathcal{OSign}$ of the experiment $\text{Exp}_{\mathcal{F}, \text{MS}}^{\text{EUF-CMA}}$.

The forger completes the election parameters by generating the encryption key pair (pk, sk) itself and by selecting the set of registrars $\mathcal{R}$ and the electoral roll $\mathcal{L} := \{id_1, \dots, id_n\}$. It plays the role of the honest registrar $\mathcal{R}_1$, while the remaining registrars are controlled by $\mathcal{A}$. Instead of sampling its own multi-signature parameters, $\mathcal{F}$ passes on the parameters $pars$ of its own experiment. It commits to the roll as $(com, dec) \leftarrow \text{Com.Commit}(\mathcal{L})$ and assembles $pub := (pars, \mathcal{R}, m, n, \mathcal{L}_{\text{Reg}}, pk, com)$ with $\mathcal{L}_{\text{Reg}} := \emptyset$ and $sec := (\mathcal{L}, dec, sk)$. Since **MS.Setup** is invoked independently of all other components of **Setup**, the parameters seen by $\mathcal{A}$ are distributed exactly as in G_1 .

Additionally, $\mathcal{F}$ samples an index i^* uniformly at random from $\{1, \dots, n\}$ and sets $id^* := id_{i^*}$. Intuitively, $\mathcal{F}$ guesses the voter under whose credential $\mathcal{A}$ will later place a ballot on the bulletin board that it did not obtain from the voting oracle. The index i^* is never revealed to $\mathcal{A}$. Finally, $\mathcal{F}$ invokes $\mathcal{A}$ on (pub, sec) and maintains the sets $\mathcal{U}$, $\mathcal{L}_{\text{Reg}}$, $\mathcal{CU}$, **HVote**, and **Checked**, as well as the table *session* exactly as the experiment does. The queries of $\mathcal{A}$ are answered as follows.

OReg₁. If $id \notin \mathcal{L}$, or if id already occurs in $\mathcal{U}$, or if a registration for id is already in progress, $\mathcal{F}$ returns $\perp$. If $id \neq id^*$, it runs $(pcs_1, scs_1) \leftarrow \text{MS.KeyGen}()$ and stores the resulting share pair in $session[id]$. If $id = id^*$, it embeds its challenge key by setting $pcs_1 := pks^*$ and, without knowing the corresponding secret share, $scs_1 := \perp$. Since pks^* is generated by **MS.KeyGen** in $\text{Exp}_{\mathcal{F}, \text{MS}}^{\text{EUF-CMA}}$, the two cases are perfectly indistinguishable from the view of $\mathcal{A}$. In both cases, $\mathcal{F}$ returns pcs_1 .

OReg₂. If a registration for id is in progress, $\mathcal{F}$ takes the shares of the corrupted registrars from the query, assembles pc_{id} , adds the entry to $\mathcal{U}$ and pc_{id} to $\mathcal{L}_{\text{Reg}}$, clears $session[id]$, and returns pc_{id} .

OVote(id, v). If id is not registered, $v \notin \mathbb{V}$, or an entry $(id, *, *) \in \text{HVote}$ already exists, then $\mathcal{F}$ returns $\perp$. Otherwise, it encrypts the vote as $c \leftarrow \text{Enc}_{pk}(v)$. For $id \neq id^*$, the forger holds the complete secret credential sc_{id} and therefore executes all m roles of

MS.Sign on c itself. For $id = id^*$, it holds every share of sc_{id^*} except the one belonging to pks^* . It therefore runs the signing protocol on the message c under the credential pc_{id^*} , playing the $m - 1$ roles it knows and obtaining the contribution of the remaining role by running a signing session with $\mathcal{OSign}$ for the key list pc_{id^*} and the message c , forwarding the protocol messages of every round. Both cases are identically distributed, since a local execution of all m roles is by definition a protocol run with m honest participants, and a run in which $m - 1$ roles are played by $\mathcal{F}$ according to the protocol and one role is played by the signing oracle is the same run with the roles distributed differently. In both cases, $\mathcal{F}$ sets $b := (c, \sigma, pc_{id})$, updates $\text{HVote} := \text{HVote} \cup \{(id, v, b)\}$, and returns b .

OCheck(id, b). The query is answered exactly as in Figure 3, which $\mathcal{F}$ can do without any secret information.

OCorrupt(id). If $id = id^*$, then $\mathcal{F}$ aborts and outputs $\perp$. This event is denoted by $\text{bad}_{\text{corrupt}}$. Otherwise, $\mathcal{F}$ knows sc_{id} and returns it after adding id to $\mathcal{U}$.

Unless $\text{bad}_{\text{corrupt}}$ occurs, the view of $\mathcal{A}$ in the reduction is identical to its view in G_1 .

Eventually, $\mathcal{A}$ outputs a bulletin board BB , a result r , and a proof Π . If G_1 returns 0 on this output, $\mathcal{F}$ aborts. Otherwise, there exists a ballot $b^* := (c^*, \sigma^*, pc_{id'}) \in BB$ such that $pc_{id'} \in \mathcal{L}_{\text{Reg}}$ for a voter $id' \notin \mathcal{U}$ and c^* was never signed under $pc_{id'}$ in a query to $\mathcal{OVote}$. Should several such ballots exist, let b^* be the first of them in the order of BB , so that id' is uniquely determined.

To see that such a ballot exists, recall that G_1 proceeds beyond its verification steps only if $\text{Verify}(BB, r, \Pi) = 1$ and $r \neq \perp$. By the definition of Verify this implies $\text{ValidateRegistration}() = 1$ and $\text{ValidateBoard}(BB) = 1$. Consequently, every ballot on BB carries a signature that verifies under a public credential contained in $\mathcal{L}_{\text{Reg}}$, no public credential occurs on more than one ballot, and every ciphertext on BB decrypts to a vote in $\mathbb{V}$. Furthermore, the experiment did not abort in the individual verifications, so every entry $(id, v, b) \in \text{Checked}$ satisfies $\text{VerifyVote}(BB, id, b) = 1$, and the corresponding ballot is contained in BB . The votes $v_1^C, \dots, v_{n_C}^C$ of all entries in Checked are therefore among the plaintexts on the board.

Now assume that no ballot b^* of the stated form exists. Then the public credential of every ballot on BB either belongs to a corrupted voter, or the ciphertext of the ballot was signed under that credential in a query to $\mathcal{OVote}$. In the first case, distinct ballots belong to distinct corrupted voters, because $\text{ValidateBoard}(BB) = 1$ rules out that a public credential occurs on more than one ballot, and because the registration oracles admit at most one completed registration per voter and therefore rule out that a single voter holds more than one credential. Hence, at most $|\mathcal{U}|$ ballots fall into this case, and their plaintexts are admissible votes that can be written as $v_1^B, \dots, v_{n_B}^B$ with $0 \leq n_B \leq |\mathcal{U}|$. In the second case, $\mathcal{OVote}$ answers at most one query per voter, so the ciphertext of the ballot is the

ciphertext of the unique entry $(id, v, b) \in \text{HVote}$ for that voter, and it decrypts to v . These plaintexts are therefore the votes recorded in HVote , split into those belonging to entries in Checked , all of which occur by the argument above, and those belonging to entries in $\text{HVote} \setminus \text{Checked}$. Since the announced result in G_1 equals the result function applied to the plaintexts of the ballots on the bulletin board, and these plaintexts are, up to reordering, exactly $v_1^A, \dots, v_{n_A}^A, v_1^B, \dots, v_{n_B}^B, v_1^C, \dots, v_{n_C}^C$, permutation invariance of ρ yields

$$r = \rho(v_1^A, \dots, v_{n_A}^A, v_1^B, \dots, v_{n_B}^B, v_1^C, \dots, v_{n_C}^C).$$

This is precisely the situation in which G_1 returns 0, contradicting the assumption that $\mathcal{A}$ wins. Hence, a ballot b^* of the stated form exists.

If the voter id' identified above satisfies $id' \neq id^*$, then $\mathcal{F}$ aborts. This event is denoted by $\text{bad}_{\text{guess}}$. Otherwise, $\mathcal{F}$ outputs $(pc_{id^*}, c^*, \sigma^*)$ as its forgery.

It remains to argue that this output wins $\text{Exp}_{\mathcal{F}, \text{MS}}^{\text{EUF-CMA}}$. The credential pc_{id^*} contains the challenge key pks^* , because $\mathcal{F}$ embedded it during the registration of id^* . The signature verifies, since $\text{ValidateBoard}(BB) = 1$ implies $\text{MS.Verify}(pc_{id^*}, c^*, \sigma^*) = 1$, and this verdict does not depend on which verifier evaluates it. Finally, the message c^* was never signed under pc_{id^*} in a query to $\mathcal{O}\text{Vote}$, and since $\mathcal{F}$ queries $\mathcal{O}\text{Sign}$ exclusively during such queries, the pair (pc_{id^*}, c^*) was never submitted to the signing oracle. Therefore, this output wins $\text{Exp}_{\mathcal{F}, \text{MS}}^{\text{EUF-CMA}}$.

The success probability of $\mathcal{F}$ depends on two events, namely on $\text{bad}_{\text{corrupt}}$ and on $\text{bad}_{\text{guess}}$, and both of them depend on the guessed index i^* .

The two abort conditions do not interfere with each other. If the guess is correct, then $id' = id^*$ holds and $id' \notin \mathcal{U}$ by the choice of b^* , so $\mathcal{O}\text{Corrupt}(id^*)$ was never queried and $\text{bad}_{\text{corrupt}}$ did not occur. The guess is correct with probability $1 - \Pr[\text{bad}_{\text{guess}}] = \frac{1}{n}$. Whenever G_1 returns 1 and the guess is correct, $\mathcal{F}$ consequently outputs a valid forgery, which gives

$$\text{Adv}_{\mathcal{F}, \text{MS}}^{\text{EUF-CMA}}(\kappa) \geq \Pr[G_1 = 1 \wedge id' = id^*] = \frac{1}{n} \cdot \Pr[G_1 = 1].$$

Combining this with the bound of the game hop gives

$$\text{Adv}_{\mathcal{A}, \text{VEG}^+}^{\text{ver}}(\kappa) = \Pr[G_0 = 1] \leq n \cdot \text{Adv}_{\mathcal{F}, \text{MS}}^{\text{EUF-CMA}}(\kappa) + \text{Adv}_{\mathcal{T}, \text{VEG}^+}^{\text{tally-snd}}(\kappa),$$

which concludes the proof. $\square$

This establishes the first layer. The second layer, the correctness of the electoral roll itself, is not a cryptographic argument but a statistical one. The following theorem bounds the probability that a manipulated electoral roll goes undetected, for a given fraction of ineligible entries and a given number of entries the auditor samples.

Theorem 2. Let $\mathcal{E}$ be an independent record of eligibility. For every adversary $\mathcal{A}$ producing an electoral roll $\mathcal{L}$ with a fraction of illegitimate entries $\theta := \frac{|\mathcal{L} \setminus \mathcal{E}|}{|\mathcal{L}|}$, let $S \subseteq \mathcal{L}$ be a uniformly random subset of size $k := |S|$. Then

$$\text{Adv}_{\mathcal{A}}^{\text{audit}} := \Pr \left[S \cap (\mathcal{L} \setminus \mathcal{E}) = \emptyset \mid \mathcal{L} \leftarrow \mathcal{A}(\mathcal{E}), \quad S \xleftarrow{\$} \{S' \subseteq \mathcal{L} : |S'| = k\} \right] \leq (1 - \theta)^k.$$

Proof of Theorem 2. Let $n := |\mathcal{L}|$ and $t := |\mathcal{L} \setminus \mathcal{E}|$ denote the total number of roll entries and the number of illegitimate ones, so that $\theta = \frac{t}{n}$. Since S is drawn uniformly at random from all size- k subsets of $\mathcal{L}$, the event $S \cap (\mathcal{L} \setminus \mathcal{E}) = \emptyset$ occurs precisely when all k elements of S are drawn from the $n - t$ legitimate entries. There are $\binom{n-t}{k}$ such subsets out of $\binom{n}{k}$ subsets in total, so

$$\text{Adv}_{\mathcal{A}}^{\text{audit}} = \frac{\binom{n-t}{k}}{\binom{n}{k}} = \frac{(n-t)!}{(n-t-k)!} \frac{(n-k)!}{n!} = \prod_{i=0}^{k-1} \frac{n-t-i}{n-i}.$$

For each $i \in \{0, \dots, k-1\}$, it holds that $\frac{n-t-i}{n-i} = 1 - \frac{t}{n-i} \leq 1 - \frac{t}{n} = \frac{n-t}{n} = 1 - \theta$. Taking the product over each i yields

$$\text{Adv}_{\mathcal{A}}^{\text{audit}} \leq (1 - \theta)^k,$$

which concludes the proof. $\square$

The bound is stated in terms of θ , which the auditor does not know beforehand. For a single plurality contest, however, only a manipulation large enough to change the outcome matters, which fixes a smallest relevant θ and turns the bound into a required sample size.

Corollary 1. Let $\mathcal{L}$ be an electoral roll of size n in a single plurality contest, let $\Delta \geq 1$ be the vote difference between the first- and second-placed candidates in the announced result, let $t := |\mathcal{L} \setminus \mathcal{E}|$ be the number of ineligible entries in the roll with respect to an independent record of eligibility $\mathcal{E}$, and let $\alpha \in (0, 1)$ be an error bound. If the auditor checks

$$k \geq \frac{\ln \alpha}{\ln(1 - \frac{\Delta}{n})}$$

uniformly random entries of $\mathcal{L}$ and accepts only if none of them is ineligible, then a roll with enough ineligible entries to have changed the outcome is accepted with probability at most α .

Proof of Corollary 1. Since each credential admits at most one counted ballot, each ineligible entry contributes at most one vote to the announced winner. The winner therefore retains a strict lead whenever $t < \Delta$, and it suffices to bound the probability of a clean audit in the remaining case $t \geq \Delta$, i.e., $\theta \geq \theta_0 := \Delta/n$. As $(1 - \theta)^k$ decreases in θ , Theorem 2 gives

$$\text{Adv}_{\mathcal{A}}^{\text{audit}} \leq \left(1 - \frac{\Delta}{n}\right)^k.$$

Requiring this bound to be at most α and rearranging it yields the claimed bound on k . $\square$

Both layers are combined in the following theorem, which bounds the probability that an election whose announced winner was produced by ballot stuffing passes the audit. It considers the experiment $\text{Exp}_{\mathcal{A}, \mathbf{V}^{\text{EG}+}}^{\text{ver}}$ with the following modifications, denoted as $\text{Exp}_{\mathcal{A}, \mathbf{V}^{\text{EG}+}}^{\text{ver}+}$. Before Setup, the adversary fixes a roll $\mathcal{L}$ of size n and publishes the commitment com obtained from $(\text{com}, \text{dec}) \leftarrow \text{Com.Commit}(\mathcal{L})$. It then hands $(\mathcal{L}, \text{dec})$ to the honest registrar $\mathcal{R}_1$, who opens com and aborts unless the opening succeeds. After the result has been announced, let Δ denote its margin and set $\theta_0 := \Delta/n$. The adversary provides an opening $(\mathcal{L}', \text{dec}')$ of com to the auditors, who abort unless opening the commitment succeeds. They then draw a uniformly random sample $S \subseteq \mathcal{L}'$ of size $k \geq \ln \alpha / \ln(1 - \theta_0)$ and reject if any sampled entry is not contained in $\mathcal{E}$.

Theorem 3. *Let $\mathbf{V}^{\text{EG}+}$ be instantiated with the multi-signature scheme $\mathbf{MS}$ and a commitment scheme $\mathbf{Com}$, restricted to a single plurality contest whose winner is the option with the most votes in the result, and let $\mathcal{E}$ denote an independent record of eligibility. For every probabilistic polynomial-time adversary $\mathcal{A}$ in $\text{Exp}_{\mathcal{A}, \mathbf{V}^{\text{EG}+}}^{\text{ver}+}$ there exist probabilistic polynomial-time algorithms $\mathcal{F}$, $\mathcal{T}$, and $\mathcal{B}$ such that, whenever the announced winner differs from the winner determined by the votes of entries in $\mathcal{L} \cap \mathcal{E}$ alone, the audit accepts with probability at most*

$$n \cdot \text{Adv}_{\mathcal{F}, \mathbf{MS}}^{\text{EUF-CMA}}(\kappa) + \text{Adv}_{\mathcal{T}, \mathbf{V}^{\text{EG}+}}^{\text{tally-snd}}(\kappa) + \text{Adv}_{\mathcal{B}, \mathbf{Com}}^{\text{binding}}(\kappa) + \alpha.$$

Proof Sketch of Theorem 3. The claim follows from three successive game hops, each removing one way in which the audit can accept an outcome that ballot stuffing has changed.

The first hop rules out that the roll opened to the auditors differs from the one under which credentials were issued. Two valid openings of com break the binding property, so this costs $\text{Adv}_{\mathcal{B}, \mathbf{Com}}^{\text{binding}}(\kappa)$.

The second hop rules out that the announced result deviates from the votes cast under registered credentials of that roll. This is exactly Theorem 1 and costs $n \cdot \text{Adv}_{\mathcal{F}, \mathbf{MS}}^{\text{EUF-CMA}}(\kappa) + \text{Adv}_{\mathcal{T}, \mathbf{V}^{\text{EG}+}}^{\text{tally-snd}}(\kappa)$.

In the remaining game, every vote not cast by an eligible voter originates from a distinct entry of $\mathcal{L} \setminus \mathcal{E}$, so changing the outcome requires at least Δ such entries. Since $\mathcal{L}$ is fixed by the commitment before the sample is drawn, Corollary 1 applies and bounds the probability of a clean audit in this case by α . Summing the three bounds yields the claim. $\square$

4.3 Proof of Eligibility Verifiability

The first three terms are negligible in κ , so the probability that a stuffed election passes the audit is essentially α . Unlike the other terms, α is not a property of the primitives but a parameter of the audit, set through the number of entries the auditor inspects. Eligibility verifiability of the extension therefore rests on cryptography for everything contained in the election record, and on the sample size for the audit of the electoral roll behind it. Consequently, the resulting probability can be pushed arbitrarily low by inspecting more entries, but can only be reduced to zero by inspecting the roll in full.

5 Prototypical Implementation

This chapter presents a prototype implementation¹ of the extension described in Chapter 4. It serves as an exemplary instantiation of the abstract building blocks, i.e., the multi-signature scheme and the commitment scheme, and shows how the extension integrates into an existing ElectionGuard implementation. Finally, the performance of the primitives used is evaluated both in isolation and in terms of the overhead introduced during an election.

The ElectionGuard toolkit is implemented in several languages. One of them is the Python reference implementation², which implements version 1.0 of the specification [3], even though its documentation still refers to version 0.95. There are also implementations in Kotlin and Rust, both based on the ElectionGuard 2.0 specification. Nevertheless, this prototype extends the Python implementation, as it covers the complete election process, including a verifier, and is well documented.

Building on an earlier specification version, however, does not limit the intent of the prototype. Version 2.0 mostly introduced changes to reduce the size of the election record by roughly a factor of three and to compute proofs 20% faster [27], [28]. Those changes concern parts of the protocol that this extension does not touch, so the newly introduced features carry over to implementations of the newer specification. Their absolute cost carries over as well, since the added operations depend only on the group arithmetic, and the bit lengths of the group parameters are unchanged in version 2.0. The relative overhead reported in Section 5.2 would therefore come out larger on version 2.0, since the same absolute cost is measured against a smaller election record.

5.1 Design and Cryptographic Instantiation

The construction of Chapter 4 requires a multi-signature scheme and a commitment scheme without specifying either of them. This section specifies both schemes and then follows the algorithms of Section 4.1 to show where the extension integrates into the existing implementation.

¹<https://github.com/ionas-dev/electionguard-python>

²<https://github.com/Election-Tech-Initiative/electionguard-python>

Both the existing implementation and the extension perform the expensive modular arithmetic through gmpy2, a Python wrapper around the GMP library. The extension uses the same group primitives as the ElectionGuard core, so no separate arithmetic backend is introduced.

The multi-signature scheme is instantiated with MuSig as described in Section 2.3.1. Its central property for this implementation is that the shares of a credential can be aggregated into a single key. Since the voter receives all secret shares, the three-round signing protocol collapses into a local aggregation followed by an ordinary Schnorr signature [29]. The first two rounds, and with them the hash function H_{com} , are only needed to keep a malicious cosigner from choosing its nonce in response to the others, which cannot occur when a single party executes all roles. The aggregated secret key is $\tilde{sc} := \sum_{i=1}^m scs_i \cdot a_i$ with $a_i := H_{\text{agg}}(pc, pcs_i)$ and $pc = (pcs_1, \dots, pcs_m)$. The corresponding public key is the aggregate $\tilde{pc} := \prod_{i=1}^m pcs_i^{a_i}$ published in the registration list. Ballots are therefore signed and verified exactly as with an ordinary Schnorr signature, and the ballot only needs to carry the aggregate instead of all separate shares. This keeps the ballot size small and the verification simple to implement, which is one of ElectionGuard’s guiding principles [3].

The commitment scheme is instantiated with a Pedersen commitment, as described in Section 2.4.1, and the commitment is taken over the hash of the electoral roll. The scheme is easy to introduce in ElectionGuard, since it reuses the existing group parameters and only adds a second group generator to them. In addition, no new cryptographic assumptions are needed, since the binding property rests only on the DLog assumption and the collision resistance of the hash function, which are both already assumed.

MuSig requires the aggregation coefficients and the signature challenge to be derived from distinct hash functions. Domain separation is therefore added to the existing hash implementation. That implementation is defined over a sequence of hashable elements and builds on SHA-256. Its input begins with the character |, and every element is appended followed by the same character. For the existing uses of the hash function, this format remains unchanged. The two uses introduced by MuSig, denoted H_{agg} and H_{sig} above, prepend a domain tag to the input instead. The tags are of equal length and pairwise distinct, so inputs of different domains cannot coincide, while every hash computed by unmodified ElectionGuard keeps its value.

The algorithms of Section 4.2 are implemented as follows, where some of them are already covered by the reference implementation to a large extent and are therefore only mentioned briefly. VerifyVote is unaffected by the extension, since a voter still locates their ballot through the confirmation code alone. ValidateBoard is extended by the checks described below for ValidateBallot and Box. Tally includes the same checks and, in addition, those described for ValidateRegistration. The remaining algorithms carry the changes of the extension and are described in detail in the following.

Setup(1^κ). This algorithm is spread across several places in the existing implementation. The newly introduced multi-signature scheme reuses the group parameters already present in ElectionGuard, that is, the prime modulus p , the subgroup order q , the cofactor $r := \frac{p-1}{q}$, and the generator $g := 2^r \bmod p$ of the subgroup $\mathbb{G}$ of order q in $\mathbb{Z}_p^*$. The Pedersen commitment also reuses these parameters, but needs an additional group generator h with an unknown discrete logarithm with respect to the existing generator g . To guarantee this, the new generator $h := 3^r \bmod p$ is specified similarly to g and is set as a project constant like the other parameters. The electoral roll is implemented as an object of `class ElectoralRoll` that stores a list of voters, each of which contains contact information and the id of the ballot style the voter is eligible for. A specific ballot style contains the set of contests and options presented to that voter.

CredentialShare($id, \mathcal{L}, dec$). The generation of a credential share for a voter by a registrar is implemented as part of a batch generation for all voters in the electoral roll $\mathcal{L}$.

```
def generate_credentials(self):
    for (i, voter) in enumerate(self.electoral_roll.voters):
        nonce = self.nonces.get_with_headers(i, voter.object_id)
        credential = self.generate_credential(nonce)
        self.credentials[voter.object_id] = credential
```

The randomness of each share is derived from a sequence of nonces that is deterministically specified by a seed and a registrar-specific header. The generated credentials are stored and are exposed through the functions `publish_public_credentials(self)` and `send_credential_to_voter(self, id)`. Before generating any share, a registrar needs to receive the commitment opening and compare the received roll with the published commitment through `verify_electoral_roll_commitment(self, commitment, opening)`.

ValidateRegistration(). The registration list $\mathcal{L}_{\text{Reg}}$ is implemented as an object of `class CredentialRegistry`. An ElectionGuard election can contain multiple contests and ballot styles, where each ballot style covers a subset of the contests. The credential registry therefore refines the notion of eligibility used in the construction and registers a credential not for the election as a whole but for one ballot style. Validation is then performed per ballot style and checks that no more credentials are registered than there are eligible voters for that style, that the aggregated keys are pairwise distinct, and that every credential consists of one share per registrar. In addition, the implementation compares the aggregate recomputed from the shares with the one stored in the registry, which ties the aggregate used for verification to the shares the registrars published.

Vote(pc, sc, v). Encrypting a selection v into a well-formed ballot c is already implemented in ElectionGuard and results in an object of `class CiphertextBallot`. The extension adds objects of `class SignedBallot`, which are created by signing the hash of the ciphertext ballot with the aggregated credential key pair.

```
def sign(ballot, key_pair, nonce):
    message = ballot.crypto_hash
    nonce = nonce if nonce is not None else rand_q()
    signature = schnorr_sign(nonce, message, key_pair)
    return make_ciphertext_signed_ballot(ballot, signature,
                                         key_pair.public_key)
```

Signing the ballot reuses the hash that ElectionGuard already computes for every ballot. The signed ballot contains all fields of the ciphertext ballot and adds the signature and the public credential, so that encryption remains untouched and the extension is purely additive.

ValidateBallot(b). Validating the well-formedness of a ciphertext ballot is already implemented. The validation of the signature and of the credential is new. The signature is verified directly on the ballot objects through `verify_signature(self)`, since the ballot carries both the public credential and the signature. Because credentials are registered per ballot style, as described for `ValidateRegistration()`, validation additionally requires that the presented credential is registered for the style of the submitted ballot, which is checked through `is_registered(self, ballot_style_id, public_key)` on an object of the credential registry.

Box(BB, b). Submitting a ballot to the box, including the validation of its well-formedness, is also already implemented by ElectionGuard. This includes preventing the submission of a ballot whose id was already used by another submitted ballot. The extended implementation now includes the additional checks described above for `ValidateBallot( $b$ )`, as well as preventing the submission of a ballot whose credential was already used by another submitted ballot. To this end, the object of `class BallotBox` additionally stores the ballots mapped by their credential in a dictionary.

Verify(BB, r, Π). ElectionGuard does not provide a single verification function but separate ones for the well-formedness of the ballots, the aggregation of the tally, and the decryption proofs. The extended implementation follows this structure and adds three further functions alongside them rather than modifying any existing one. The function `verify_ballot_eligibility(ballot, registry)` checks the signature and the credential of one ballot, and `verify_credential_registry(registry)` checks the registration list as described for `ValidateRegistration()`. The function `verify_aggregation_with_credentials(submitted_ballots, tally, credential_registry, manifest, context)` checks that the credentials of the submitted ballots are pairwise distinct, aggregates the ballots into a tally, and compares the result with the given tally. Existing verifier code therefore remains untouched, and a standard verifier can be extended by adding the two eligibility checks and replacing its aggregation check with the extended one.

5.2 Evaluation

The prototype is evaluated in two ways. The first is correctness, which is addressed through a test suite that is described in the following. The second is performance, which is measured through microbenchmarks for the newly introduced primitives and through benchmarks that isolate the overhead the extension causes for each party involved in an election.

5.2.1 Correctness

Based on the lines that were added or modified compared to the pre-extension state, the test suite achieves ~97% line coverage, meaning that the lines are executed at least once during the test run. This is a helpful but far from sufficient condition. It rules out untested code paths, yet it establishes neither that the primitives instantiate the schemes of Section 4.2 faithfully nor that a check which is executed also rejects what it is supposed to reject.

The tests are therefore organized around the properties the extension is meant to provide. For the primitives, this covers correctness in the sense that honestly produced signatures and commitments verify and open, respectively, and modified ones do not. For the extension as a whole, this means testing against the situations the algorithms in Section 4.2 are meant to prevent. In addition, an end-to-end test runs a complete election from the generation of credentials through the casting and signing of ballots to the tally, to confirm that a complete election works as intended.

Since the extension is purely additive with the single exception of the acceptance of a ballot, an election record produced by the extended implementation remains a valid record of the unextended one. This is tested by verifying such a record with an unmodified verifier. The additional guarantee therefore does not come at the price of breaking compatibility with existing verifier implementations.

5.2.2 Benchmark Methodology

All benchmarks were computed on a MacBook Air with an M1 processor and 8 cores. The Python version was 3.13.5, and the gmpy2 version was 2.3.0. The latest commit of the Python reference implementation was from July 19, 2026 with commit id a975384. All measurements were taken on a single core, so the reported times do not reflect the parallelism that verification in particular would allow in practice.

Every benchmark reports the sample mean and the sample standard deviation over repeated executions of a single operation. The number of repetitions was chosen such that the 95% confidence interval of the mean has a margin of error below 1% of the mean. Since neither the mean nor the standard deviation is known beforehand, both were estimated in

a pilot run with an arbitrarily chosen number of repetitions n_0 . Rearranging the margin of error $e = z \cdot \frac{s}{\sqrt{n}}$ for n yields the number of repetitions required to reach the target precision as

$$n \geq \left(\frac{z \cdot s}{e} \right)^2, \quad z := 1.96, \quad e := 0.01 \cdot \bar{x},$$

where $\bar{x}$ and s denote the sample mean and the sample standard deviation of the pilot run. If $n \leq n_0$ held, the pilot run already satisfied the target precision and its measurement is the reported one. Otherwise, the benchmark was repeated with $n_1 \geq n$ repetitions.

The benchmarks that compare the extended processes against the unextended ones are taken from the same code base. The difference between the two therefore reflects the extension itself and not a difference in the underlying arithmetic backend.

5.2.3 Performance of the Cryptographic Primitives

The signature operations were benchmarked using 100 repetitions each. Signing and verification were performed on the same ballot, while each signature was made with a freshly generated key pair and each verification was performed on a fresh signature. The results are shown in Table 1.

Key generation consists of a single exponentiation in $\mathbb{G}$ and can therefore serve as a reference value for the cost of one exponentiation on the benchmark machine. Signing adds the hashing of the ballot to one such exponentiation and stays close to that value. Verification takes roughly twice as long, which corresponds to the two exponentiations it requires. The runtime of all three operations is independent of the number of voters and of the size of the ballot, since only the ballot hash enters the signature.

The commitment operations were measured using 250 repetitions each. As messages, lists of 10, 100, and 1,000 voters were used. The results are shown in Table 2.

Committing and opening cost essentially the same, which is expected since opening is practically the same operation with an additional comparison at the end. Both consist of

Operation	Time (ms)
Sign	(0.891 ± 0.010) ms
Verify	(1.801 ± 0.037) ms
Key Generation	(0.883 ± 0.015) ms

Table 1: Runtime of the signature operations, given as mean and standard deviation over 100 repetitions on the same ballot.

Operation	Time (ms)		
	10 Voters	100 Voters	1,000 Voters
Commit	(2.022 ± 0.034) ms	(4.522 ± 0.039) ms	(29.9 ± 1.1) ms
Open	(2.016 ± 0.024) ms	(4.533 ± 0.045) ms	(29.82 ± 0.42) ms

Table 2: Runtime of committing to and opening an electoral roll of 10, 100, and 1,000 voters, given as mean and standard deviation.

a constant part of two exponentiations and a part that is linear in the number of voters, caused by hashing the electoral roll. For 10 voters, the constant part dominates, whereas for 1,000 voters the linear part does. The commitment itself remains a single group element regardless of the size of the roll.

5.2.4 Performance Overhead during an Election

The performance overhead is reported separately for each party, since no single device runs the protocol from beginning to end. A registrar issues credentials once before the election, a voter casts a single ballot, and a verifier processes the entire election record. In addition, the size overhead of a ballot is reported.

Registrars. The main operations of a registrar are generating the credentials, verifying that all credentials are published in the credential registry, and verifying the commitment of the electoral roll. All three were measured as the work of a single registrar for an electoral roll of 10, 100, 1,000, and 10,000 voters. The results are shown in Figure 4.

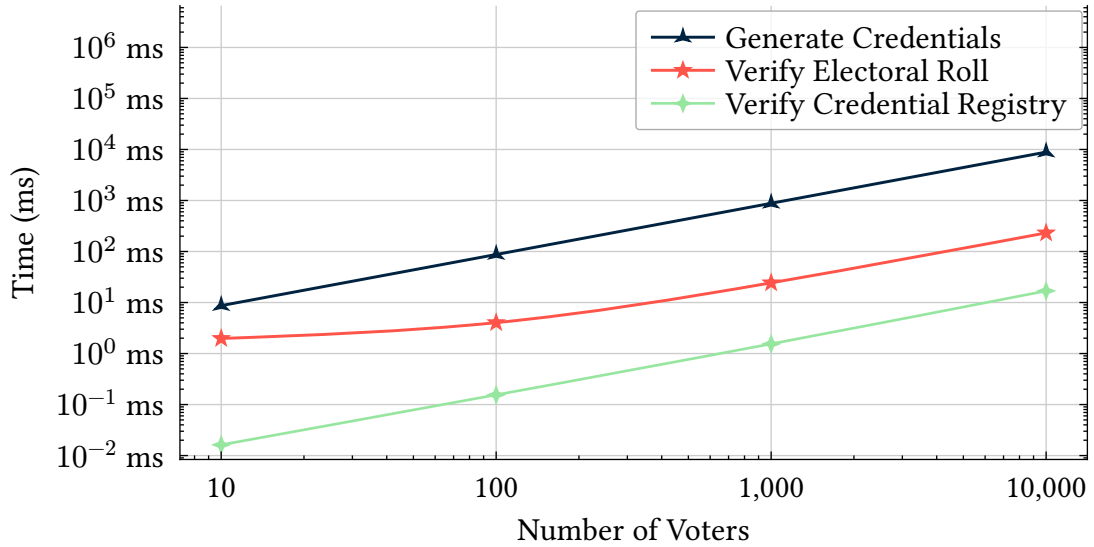


Figure 4: Runtime of a registrar's main operations over the number of voters, on logarithmic axes.

All three operations grow linearly in the number of voters. Generating the shares dominates and costs about one exponentiation per voter, since a share is a freshly generated key pair. Verifying the credential registry is cheaper by more than two orders of magnitude, because it compares published values instead of sampling fresh randomness. Verifying the commitment of the electoral roll lies between the two and follows the behavior already observed in Table 2, with the constant part dominating for small rolls and the hashing of the roll dominating from about 1,000 voters onward. Even for 10,000 voters, all three operations stay in the range of seconds. Since every registrar performs this work independently on its own shares, even longer executions for very large electorates are unproblematic.

The remaining benchmarks were taken for different ballot styles with a varying number of contests and candidates, which results in different ballot sizes. The ballot styles are described in Table 3.

Ballot Size. A signed ballot carries the signature and the public credential in addition to the fields of an ordinary ciphertext ballot. The size overhead is constant and independent of the rest of the ballot, so it shrinks relative to the total size as the ballot grows through additional contests or candidates. The relative overhead with respect to the total size of a ballot is shown in Figure 5.

Across the ballot styles considered here, the additional data amounts to about one kilobyte per ballot. This corresponds to slightly more than 4% for the smallest ballot style with a single contest and drops below 0.5% from the third style onward. Since the election record contains one entry per ballot, the same factor applies to the growth of the record as a whole.

Voters. The time a voter needs to encrypt, sign, and cast a ballot was measured using 50 repetitions for each of the ballot styles of Table 3. The results are shown in Figure 6.

Contests	Candidates	Size (kB)		Mark
		Unsigned	Signed	
1	2	29 kB	30 kB	●
2	5	100 kB	101 kB	▲
3	10	256 kB	257 kB	■
4	20	625 kB	626 kB	◆

Table 3: Ballot styles used for the size and runtime benchmarks, with their size before and after signing. The mark identifies the style in the following figures.

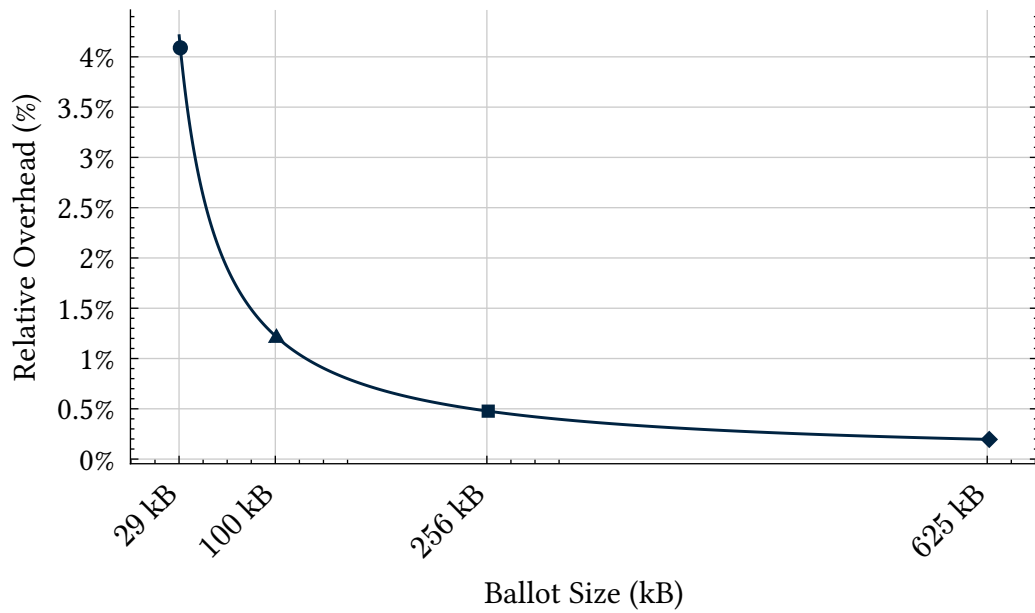


Figure 5: Relative size overhead of a signed ballot with respect to the total ballot size. The marks refer to the ballot styles of Table 3.

The absolute runtime is dominated by the encryption of the selections and by the proofs of their well-formedness, both of which grow with the number of contests and candidates. Signing, in contrast, is independent of the ballot size, as only the hash of the ballot is signed, and the hash is already computed by the unextended implementation. The absolute overhead is therefore constant at roughly the cost of one signature, and the relative overhead shrinks from about 3.7% for the smallest ballot style to about 0.6% for the largest.

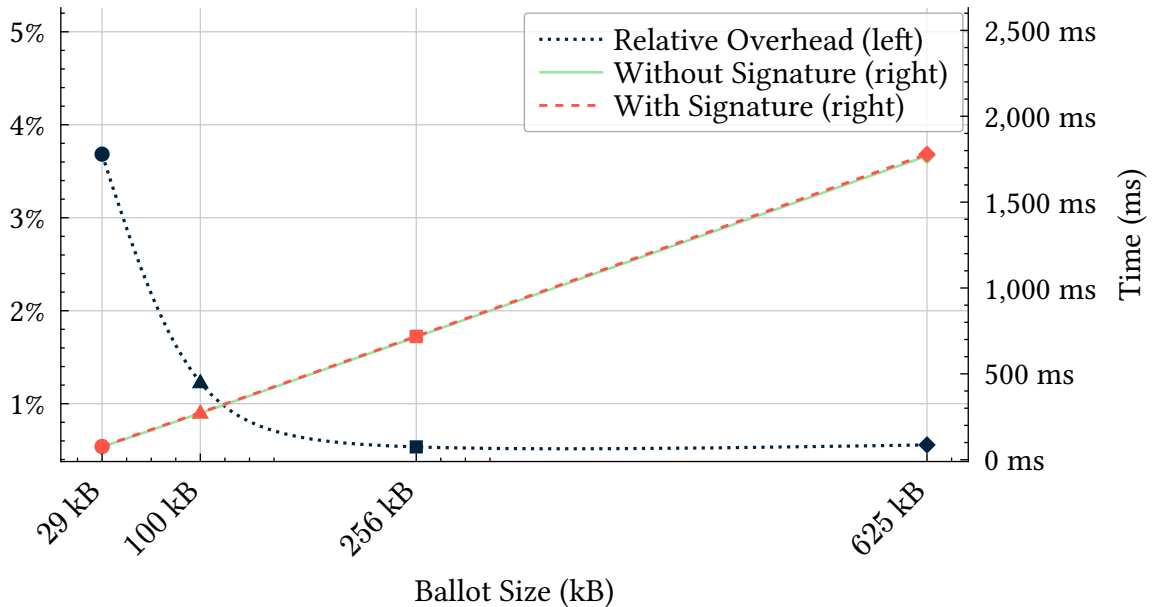


Figure 6: Time to vote a ballot, with and without the signature (right axis), and the resulting relative overhead (left axis). The marks refer to the ballot styles of Table 3.

In absolute terms, the additional time a voter spends stays below one millisecond, which is negligible against the seconds already spent on encryption.

Verifiers. The additional task of a verifier is to establish that all tallied ballots were cast by eligible voters. It was measured together with the verification of ballot well-formedness, which includes checking the proofs that the encrypted selections are valid. The results are shown in Figure 7.

The results match those observed for voting. Verifying a signature and looking up a credential in the registry are independent of the ballot size, whereas verifying well-formedness grows with it, so the relative overhead shrinks from about 3.7% for the smallest ballot style to below 0.2% for the largest. Both quantities are linear in the number of cast ballots, which leaves the relative overhead unchanged for an election of any size. On top of the per-ballot cost, a verifier checks the credential registry once, which Figure 4 shows to be negligible compared to the per-ballot work.

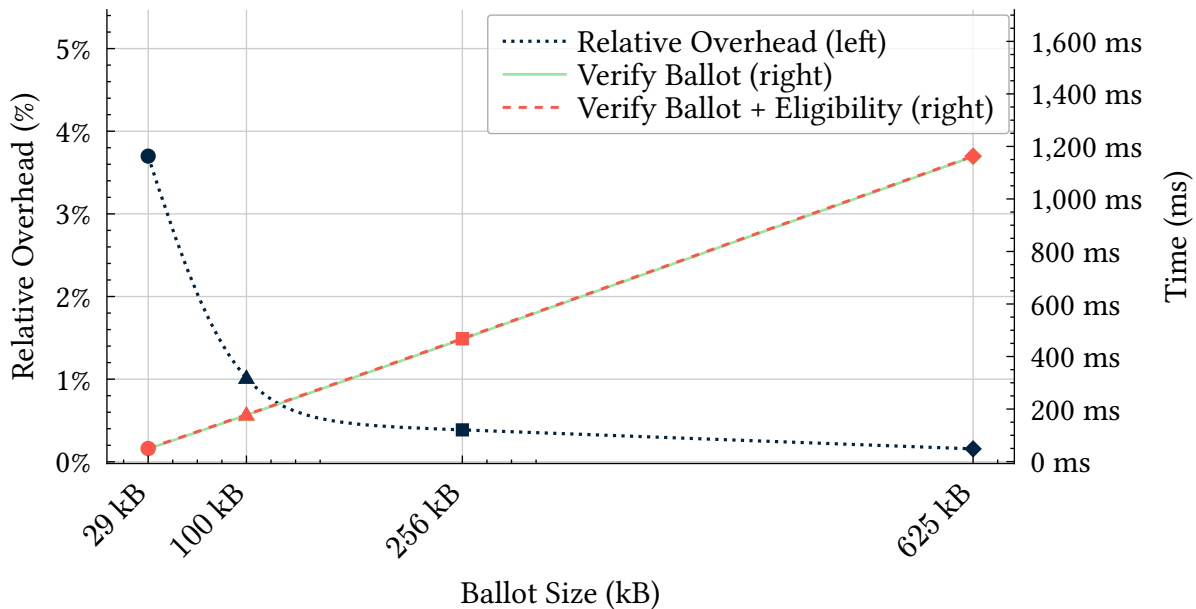


Figure 7: Time a verifier needs to check ballot well-formedness with and without the additional eligibility check (right axis), and the resulting relative overhead (left axis). The marks refer to the ballot styles of Table 3.

6 Conclusion

In this thesis, ElectionGuard is extended along two axes in order to prevent ballot stuffing. First, every ballot is signed under a credential consisting of shares generated independently by several trustees, referred to as registrars. Second, the set of issued credentials is publicly bound to the roll of eligible voters through a commitment, which makes the registration process auditable.

The extension is shown to be both sound and practical. Eligibility verifiability is proven in two parts. For a fixed electoral roll and one honest registrar, it reduces to tally soundness and to the EUF-CMA security of the underlying multi-signature scheme. That the roll itself is not manipulated is not assumed but statistically audited, and the probability that a manipulated roll passes the audit is bounded by the number of entries the auditor inspects. The security therefore also relies on the binding property of the commitment scheme to tie both parts together. The accompanying prototype implementation, together with the reported benchmarks, demonstrates that the additional cost remains moderate.

The new correctness guarantee comes with two new trust assumptions. The first is that at least one of the registrars behaves honestly and sends the secret shares of the credentials over a secret channel to the corresponding voters. This assumption can be realized in a straightforward manner in practice, as the registrars could be staffed by parties related to opposing election candidates and would therefore have no interest in colluding and manipulating the election together. Furthermore, the channel could be implemented by postal mail or by a preexisting PKI used to encrypt the shares of the voters. The second assumption is that the electoral roll is statistically audited to detect a manipulation before credential issuance.

Moreover, the new security guarantee does not weaken the preexisting individual and universal verifiability guarantees but complements them. This holds even if all registrars are dishonest and issue credentials to ineligible voters. In that case, the preexisting verification mechanisms stay the same, and eligibility can still be verified individually by the eligible voters, by inspecting the published set of tallied ballots and determining whether a ballot signed under their own credential was cast without their participation, or by noticing that they received no credential or an invalid one in the first place. This yields a weaker form of individual eligibility verifiability that survives the failure of the assumptions required for the main result.

The presented approach is similar to that of Belenios, which combines a credential issued by a registrar with a password to authenticate at the voting server. However, it shows that the same level of protection can be achieved without a voting server and without an additional password, by distributing the credential generation across multiple registrars and signing the ballot under all shares. The resulting trust assumption is potentially even weaker, since honesty is required of only one of the registrars rather than of a single designated registrar or voting server. Civitas likewise distributes the generation of each credential across independent registration tellers, but additionally allows voters to construct fake credentials and thereby achieves stronger forms of coercion resistance. This property is not provided by the presented approach, which in turn is simpler to verify and computationally considerably less expensive. The gap, however, is not specific to the extension, since unextended ElectionGuard does not provide coercion resistance in this form either. Compared to Civitas and the approaches based on a public key infrastructure, the presented approach preserves everlasting ballot privacy, since credentials are published without reference to a voter and the commitment to the roll of eligible voters is perfectly hiding.

The extension is also the only one of the compared approaches that allows auditing the electoral roll while keeping it confidential. The other approaches either publish the roll or assume it to be correct, whereas the presented extension lets designated auditors inspect the same roll the registrars used to generate the credentials.

Limitations. The extension still has a few limitations introduced by the registrars issuing the credential shares. Since all shares are required to sign a ballot, a single registrar withholding a share from a voter is able to prevent that voter from casting a ballot. Resolving this availability issue currently lies outside the scope of the extension. A comparable failure exists in any election in which a voter depends on receiving material beforehand, such as a polling notification or postal voting documents. Another limitation is that each registrar knows the link between a ballot and a voter and could, for instance, collude with a sufficient number of guardians to break the voters' ballot privacy.

Future Research. Addressing the limitations above requires limiting the registrars' abilities and knowledge. One promising direction is investigating threshold signature schemes as an alternative to the multi-signature scheme, which would address a registrar withholding a share. To limit their knowledge, further research on a verifiable shuffle could be conducted, so that registrars do not learn which voter receives their shares.

Beyond these limitations, two extensions follow naturally from the presented construction. The commitment could alternatively be instantiated with a perfectly hiding commitment scheme that allows opening only a part of the roll to an auditor. This would permit an audit of rolls that must not be disclosed in full, without giving up any of the security gained here. Finally, the credentials introduced by the extension offer a natural basis for a basic

form of revoting, since a new ballot can be linked to a voter's earlier ballot through the credential and thus replace it, allowing voters to correct mistakes. Deniable revoting, which is required for coercion resistance, is considerably harder to achieve, as the linkage must then remain hidden, and is left for future research.

Bibliography

- [1] Gallup, Inc., “Gallup Poll Social Series: Governance. Final Topline,” Sep. 2024. Accessed: Sep. 19, 2026. [Online]. Available: <https://news.gallup.com/poll/651185/partisan-split-election-integrity-gets-even-wider.aspx>
- [2] Bundesverfassungsgericht, “Urteil des zweiten Senats vom 3. März 2009, 2 BvC 3/07, 2 BvC 4/07.” Accessed: Sep. 19, 2026. [Online]. Available: https://www.bverfg.de/e/cs20090303_2bvc000307
- [3] J. Benaloh, M. Naehrig, O. Pereira, and D. S. Wallach, “ElectionGuard: A Cryptographic Toolkit to Enable Verifiable Elections,” in *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA: USENIX Association, Aug. 2024, pp. 5485–5502. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity24/presentation/benaloh>
- [4] M. Bernhard *et al.*, “Public Evidence from Secret Ballots,” in *Electronic Voting*, R. Krimmer, M. Volkamer, N. Braun Binder, N. Kersting, O. Pereira, and C. Schürmann, Eds., Cham: Springer International Publishing, 2017, pp. 84–109. doi: 10.1007/978-3-319-68687-5_6.
- [5] T. Moran and M. Naor, “Receipt-Free Universally-Verifiable Voting with Everlasting Privacy,” in *Advances in Cryptology - CRYPTO 2006*, C. Dwork, Ed., Berlin, Heidelberg: Springer, 2006, pp. 373–392. doi: 10.1007/11818175_22.
- [6] S. Kremer, M. Ryan, and B. Smyth, “Election Verifiability in Electronic Voting Protocols,” in *Computer Security – ESORICS 2010*, D. Gritzalis, B. Preneel, and M. Theoharidou, Eds., Berlin, Heidelberg: Springer, 2010, pp. 389–404. doi: 10.1007/978-3-642-15497-3_24.
- [7] J. Benaloh, R. Rivest, P. Y. A. Ryan, P. Stark, V. Teague, and P. Vora, “End-to-End Verifiability,” arXiv preprint, Apr. 2015. doi: 10.48550/arXiv.1504.03778.
- [8] M. Bellare and G. Neven, “Multi-Signatures in the Plain Public-Key Model and a General Forking Lemma,” in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, in CCS ’06. New York, NY, USA: Association for Computing Machinery, Oct. 2006, pp. 390–399. doi: 10.1145/1180405.1180453.
- [9] G. Maxwell, A. Poelstra, Y. Seurin, and P. Wuille, “Simple Schnorr Multi-Signatures with Applications to Bitcoin,” *Designs, Codes and Cryptography*, vol. 87, no. 9, pp. 2139–2164, Sep. 2019, doi: 10.1007/s10623-019-00608-x.

- [10] T. P. Pedersen, “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing,” in *Advances in Cryptology — CRYPTO ’91*, J. Feigenbaum, Ed., Berlin, Heidelberg: Springer, 1992, pp. 129–140. doi: 10.1007/3-540-46766-1_9.
- [11] T. Elgamal, “A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms,” *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, Jul. 1985, doi: 10.1109/TIT.1985.1057074.
- [12] D. Chaum and T. P. Pedersen, “Wallet Databases with Observers,” in *Advances in Cryptology — CRYPTO’ 92*, E. F. Brickell, Ed., Berlin, Heidelberg: Springer, 1993, pp. 89–105. doi: 10.1007/3-540-48071-4_7.
- [13] J. Benaloh, “Simple Verifiable Elections,” in *2006 USENIX/ACCURATE Electronic Voting Technology Workshop (EVT 06)*, Vancouver, B.C.: USENIX Association, Aug. 2006. [Online]. Available: <https://www.usenix.org/conference/evt-06/simple-verifiable-elections>
- [14] A. Fiat and A. Shamir, “How To Prove Yourself: Practical Solutions to Identification and Signature Problems,” in *Advances in Cryptology — CRYPTO’ 86*, A. M. Odlyzko, Ed., Berlin, Heidelberg: Springer, 1987, pp. 186–194. doi: 10.1007/3-540-47721-7_12.
- [15] M. R. Clarkson, S. Chong, and A. C. Myers, “Civitas: Toward a Secure Voting System,” in *2008 IEEE Symposium on Security and Privacy (SP 2008)*, May 2008, pp. 354–368. doi: 10.1109/SP.2008.32.
- [16] A. Juels, D. Catalano, and M. Jakobsson, “Coercion-Resistant Electronic Elections,” in *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, in WPES ’05. New York, NY, USA: Association for Computing Machinery, Nov. 2005, pp. 61–70. doi: 10.1145/1102199.1102213.
- [17] S. Srinivasan, C. Culnane, J. Heather, S. Schneider, and Z. Xia, “Countering Ballot Stuffing and Incorporating Eligibility Verifiability in Helios,” in *Network and System Security*, M. H. Au, B. Carminati, and C.-C. J. Kuo, Eds., Cham: Springer International Publishing, 2014, pp. 335–348. doi: 10.1007/978-3-319-11698-3_26.
- [18] B. Adida, “Helios: Web-based Open-Audit Voting,” in *17th USENIX Security Symposium (USENIX Security 08)*, San Jose, CA: USENIX Association, Jul. 2008, pp. 335–348. [Online]. Available: <https://www.usenix.org/conference/17th-usenix-security-symposium/helios-web-based-open-audit-voting>
- [19] V. Cortier, P. Gaudry, and S. Glondou, “Belenios: A Simple Private and Verifiable Electronic Voting System,” in *Foundations of Security, Protocols, and Equational Reasoning*, vol. 11565, J. D. Guttman, C. E. Landwehr, J. Meseguer, and D. Pavlovic, Eds., Cham: Springer International Publishing, 2019, pp. 214–238. doi: 10.1007/978-3-030-19052-1_14.

- [20] V. Cortier, D. Galindo, S. Glondou, and M. Izabachène, “Election Verifiability for Helios under Weaker Trust Assumptions,” in *Computer Security - ESORICS 2014*, M. Kutylowski and J. Vaidya, Eds., Cham: Springer International Publishing, 2014, pp. 327–344. doi: 10.1007/978-3-319-11212-1_19.
- [21] O. Kulyk, V. Teague, and M. Volkamer, “Extending Helios Towards Private Eligibility Verifiability,” in *E-Voting and Identity*, R. Haenni, R. E. Koenig, and D. Wikström, Eds., Cham: Springer International Publishing, 2015, pp. 57–73. doi: 10.1007/978-3-319-22270-7_4.
- [22] V. Cortier, A. Debant, A. Goetschmann, and L. Hirschi, “Election Eligibility with OpenID: Turning Authentication into Transferable Proof of Eligibility,” in *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA: USENIX Association, Aug. 2024, pp. 3783–3800. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity24/presentation/cortier>
- [23] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, “OpenID Connect Core 1.0 incorporating errata set 2,” OpenID Foundation, Dec. 2023. Accessed: Sep. 20, 2026. [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html
- [24] R. Haenni and O. Spycher, “Secure Internet Voting on Limited Devices with Anonymized DSA Public Keys,” in *2011 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE 11)*, San Francisco, CA: USENIX Association, Aug. 2011. [Online]. Available: <https://www.usenix.org/conference/evtwote-11/secure-internet-voting-limited-devices-anonymized-dsa-public-keys>
- [25] National Institute of Standards and Technology, “Digital Signature Standard (DSS),” Federal Information Processing Standard (FIPS) 186–4(Withdrawn), Jul. 2013. doi: 10.6028/NIST.FIPS.186-4.
- [26] Die Bundeswahlleiterin, “Voters’ Register.” Accessed: Sep. 21, 2026. [Online]. Available: <https://www.bundeswahlleiterin.de/en/service/glossar/w/waehlerverzeichniss.html>
- [27] Election-Tech-Initiative, “ElectionGuard Specifications.” Accessed: Aug. 28, 2026. [Online]. Available: <https://electionguard.vote/spec/>
- [28] Election-Tech-Initiative, “ElectionGuard GitHub Specifications.” Accessed: Sep. 16, 2026. [Online]. Available: <https://github.com/Election-Tech-Initiative/electionguard/blob/main/docs/spec/index.md>
- [29] C. P. Schnorr, “Efficient Signature Generation by Smart Cards,” *Journal of Cryptology*, vol. 4, no. 3, pp. 161–174, Jan. 1991, doi: 10.1007/BF00196725.