usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

ElectionGuard: a Cryptographic Toolkit to Enable Verifiable Elections

Josh Benaloh and Michael Naehrig, *Microsoft Research;* Olivier Pereira, *Microsoft Research and UCLouvain;* Dan S. Wallach, *Rice University*

https://www.usenix.org/conference/usenixsecurity24/presentation/benaloh

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

ElectionGuard: a Cryptographic Toolkit to Enable Verifiable Elections

Josh Benaloh
Microsoft Research

Michael Naehrig
Microsoft Research

Olivier Pereira
Microsoft Research and UCLouvain

Dan S. Wallach[*]
Rice University

Abstract

ElectionGuard is a flexible set of open-source tools that—when used with traditional election systems—can produce end-to-end verifiable elections whose integrity can be verified by observers, candidates, media, and even voters themselves. ElectionGuard has been integrated into a variety of systems and used in actual public U.S. elections in Wisconsin, California, Idaho, Utah, and Maryland as well as in caucus elections in the U.S. Congress. It has also been used for civic voting in the Paris suburb of Neuilly-sur-Seine and for an online election by a Switzerland/Denmark-based organization.

The principal innovation of ElectionGuard is the separation of the cryptographic tools from the core mechanics and user interfaces of voting systems. This separation allows the cryptography to be designed and built by security experts without having to re-invent and replace the existing infrastructure. Indeed, in its preferred deployment, ElectionGuard does not replace the existing vote counting infrastructure but instead runs alongside and produces its own independently-verifiable tallies. Although much of the cryptography in ElectionGuard is, by design, not novel, some significant innovations are introduced which greatly simplify the process of verification.

This paper describes the design of ElectionGuard, its innovations, and many of the learnings from its implementation and growing number of real-world deployments.

1 Introduction

The ElectionGuard project was initiated by Microsoft in 2018 and announced publicly in 2019. The project intends to help support democratic institutions by building and deploying free, open-source tools that enable so-called, *end-to-end (E2E) verifiable* elections [8]—that is, elections in which voters and observers can confirm the accuracy of the results without having to trust the election software, hardware, or personnel outside of their control. ElectionGuard deters disinformation about election results by providing publicly-verifiable evidence that reported election outcomes are accurate.

One of the principal novelties of ElectionGuard is that it is *not* yet another static verifiable election system. Instead, it is a flexible set of tools that can be used by election equipment vendors within their own systems and processes. ElectionGuard encapsulates the cryptographic functionality and provides simple interfaces that can be used without cryptographic expertise. Broadly, the ElectionGuard tools support three phases of an election.

Key Generation and Set Up. In this phase, designated *guardians* engage, with the assistance of an *administrator*, in a protocol to jointly establish a public encryption key which will be used to encrypt all votes in one or possibly several elections. Additionally, a detailed manifest is provided to describe specifics of an election such as the contests, candidates, and voting rules. The resulting key and manifest will be incorporated into an ElectionGuard application that will run on devices used by voters to cast their votes during the election.

Ballot Encryption. In the course of an election, each time an enabled device used by a voter receives a voter's selections, it calls its embedded copy of the ElectionGuard application. ElectionGuard encrypts the ballot contents and returns back to the calling application a *confirmation code* in the form of a cryptographic hash of the voter's encrypted selections and other data. In most scenarios, the intent is that this confirmation code be given to the voter, and enables voters to confirm that their votes have been recorded without any change for inclusion in the tally. The systems that capture voter selections and call the ElectionGuard application can vary widely—including fully electronic voting interfaces, scanners which can read hand-marked and/or machine-marked ballots, ballots transmitted by physical mail, and even ballots cast online. Indeed, ElectionGuard is compatible with most existing modes of casting votes.

[*] Dr. Dan Wallach's contributions to this work were made while he was at Rice University and partially funded by DARPA prior to his becoming a DARPA program manager. The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Tally Decryption. In this final phase, the guardians reassemble together with the administrator to decrypt the election tallies and produce a verifiable record for publication. Voters can inspect this published *election record* to ensure that their encrypted votes are present, and the integrity of this election record can be verified by independent applications that can be written by any interested parties.

In addition to the encrypted votes, the election record produced by ElectionGuard includes numerous artifacts that together constitute a proof of the election's integrity—including the correctness of the announced election tallies. The integrity of an election record can be verified by applications written by voters, observers, or any other interested parties.

One of the guiding principles of ElectionGuard is to make independent verification as easy as possible. The desire is to promote the growth of an ecosystem of independent election verifiers by making the writing of a verifier a reasonable project for a first class in programming. Voters and observers can run verifiers from a variety of sources on the record of any election.

The first deployments of ElectionGuard were in 2020 in public elections in Wisconsin and California and then in the U.S. House Democratic caucus to elect its leadership. In 2021, civic votes using ElectionGuard began being conducted in Neuilly-sur-Seine, France. Using the learnings from these early deployments, work commenced on a version 2 of ElectionGuard which offered many advantages and simplifications. Some of these improvements were included in subsequent deployments in Idaho, Utah, and Maryland (detailed in Section 6).

While there have been a signficant number of implementations of various E2E-verifiable election designs in the past that have been used to cast millions of votes, none have followed this approach of building tools which can be incorporated into existing election equipment, and none have achieved anywhere close to this breadth of usage in different scenarios, including in-person voting on ballot marking devices, in-person voting with scanned hand-marked paper ballots, risk limiting audits, and Internet voting – see discussion in Section 6.

The remainder of this paper will describe the cryptographic design of the ElectionGuard protocol and the introduced innovations, some remarks and learnings about implementations of ElectionGuard, and details about various deployments within the U.S.

2 Cryptographic Design

The cryptographic design of ElectionGuard is largely inspired by the 1985 work of Cohen (now Benaloh) and Fischer [17] and the 1997 voting protocol by Cramer, Gennaro and Schoenmakers [20] that have also been adopted in systems like VoteBox [40], Helios 2.0 [1], STAR-Vote [4], and Belenios [18].

The two protocols [17, 20] support elections that compute the tally by adding the number of votes that each candidate received, including elections in which voters either pick a single candidate, a limited number, or as many candidates on the ballot as desired. The winner is determined by the number of votes each candidate received. This so-called homomorphic tallying approach is usually not suitable for voting methods like Instant-Runoff Voting (IRV) or Single Transferable Vote (STV) in which candidates are ranked and eliminated in multiple rounds: the simple addition of votes may not be sufficient to tally such elections. Techniques based on verifiable mixnets [39] are often more appropriate for such voting methods and have also been adopted in many systems, but they are more cumbersome to deploy.

A future variant of ElectionGuard may support mixnet methods, but the philosophy of ElectionGuard has been to cover the majority of scenarios with an approach that is as simple as possible to understand and verify. We have also found that the ability to assert to election administrators that cast ballots encrypted by ElectionGuard will *never* be (individually) decrypted has been a powerful tool to assuage concerns. For these reasons, we have started with homomorphic tallying, but a mixnet alternative (probably using elliptic curve groups) is definitely on the ElectionGuard road map.

2.1 Roles

ElectionGuard describes the various steps that a set of actors can take to obtain a verifiable election. These actors assume the following roles:

- *Guardians* $G_1, \ldots, G_n$ are jointly responsible for maintaining the privacy of the votes.
- *Voters* cast their ballots and may verify that they have been properly recorded.
- An election *administrator* facilitates protocols and procedures and also populates and publishes the election records.
- *Verifiers* verify the election records to confirm an election's integrity and, to some extent, ballot privacy.

It is important to note that the verifiers do not have a direct role in elections, and no assumptions are placed upon them. Instead, anyone—voter, candidate, observer, or other party—can choose at any time after the completion of an election to use their own means to verify the results of an election. There is no need for a verifier to register in any way or to inform anyone about performing this function.

2.2 Election Records

ElectionGuard makes the assumption that the election records are publicly accessible and that everyone has an identical view of them. This is a standard assumption for any election as there is always a central election authority that provides an authoritative list of voters, fixes the ballot contents, sets the voting times and locations, and announces the election tally.

As such, ElectionGuard assumes that the election administrator can set up a broadcast channel with similar properties and uses it to disseminate the ElectionGuard election records or, and this may be preferred, that the administrator uses an existing channel for this purpose.

In many verifiable election systems, it is assumed that a public bulletin board exists for publishing these records. In practice, it has most often taken the form of a simple web page,[1] as it is also common to publish the records of risk limiting audits [35] that may take place. Sometimes, this bulletin board can be implemented in the form of a more sophisticated distributed system [13, 30, 41]. A permissionless blockchain may be more problematic than helpful in this context: we only seek to authenticate a single authority that publishes a small number of records, not to encourage unidentified actors to replicate and compete on the views of these records.[2]

Digitally signing election records may or may not help depending on the context: signatures may offer means to authenticate records and confront a rogue administrator if inconsistent records are signed. However, a signature may not be beneficial if authenticating the legitimacy of a signature verification key is just as complicated as authenticating the legitimacy of the election records.[3] External mechanisms may be useful for that purpose.

As a complement, candidates may be invited to acknowledge, and possibly replicate, the election records through their own communication channels, just as it is customary that candidates publicly acknowledge the election outcome.

2.3 Election Parameters

This section describes the parameters that define an election and the logical structure for the ElectionGuard software as well as the cryptographic parameters used to implement the protocol components.

2.3.1 Election Manifest

Each election that uses ElectionGuard must have an *election manifest* that defines the election. It must at least contain a unique election label and a list of contests and selectable options for each contest, together with limits on the number of options that a voter can select within a contest (referred to as *contest selection limits*). It must also contain a list of ballot styles that indicate all subsets of contests that may be displayed to voters.[4] The election manifest may also contain various optional data fields, including details about voting devices, software versions, time information, etc.

The election manifest serves to guarantee that all actors have a common view of an election and makes it possible for the ElectionGuard software to record ballots properly.

2.3.2 Cryptographic Parameters

ElectionGuard uses ElGamal encryption in the multiplicative group $\mathbb{Z}_p^*$ of the ring of integers modulo a large prime p. In order to tally contests by relying on an additive homomorphism between the plaintexts and ciphertexts, choices are encoded exponentially as follows. A vote v (usually $v \in \{0,1\}$, but in some cases a larger v is possible) is mapped to the ElGamal group as g^v for some fixed group element $g \in \mathbb{Z}_p^*$. Multiplying two such powers of g together adds the exponents, as desired, and the discrete logarithm extraction needed to perform the inverse mapping remains efficient in practice since the exponent is bounded by the number of votes that a candidate can receive. An encryption of the vote v then is an ElGamal encryption of g^v and the multiplicative homomorphism inherent in the ElGamal encryption scheme through componentwise multiplication translates to an additive homomorphism.

The use of exponential ElGamal prompts for performing ElGamal encryption in a “small” subgroup of $\mathbb{Z}_p^*$. ElectionGuard uses a 4096-bit prime p and works in a cyclic subgroup of $\mathbb{Z}_p^*$ of order q, which is of size 256 bits.[5] This choice leads to more efficient group exponentiations compared to the standard group construction using safe primes without substantially reducing the security against best-known attacks.[6] For the latter, exponentiations are computed in the group of quadratic residues modulo p and are much more costly.

While ElectionGuard could be instantiated with an elliptic curve group, the decision to use integer groups instead was guided by making verifying the integrity of an election as easy as possible. Ideally, writing a verifier should be simple enough to be repeated many times and should not require intricate knowledge of elliptic curve implementations. Integer groups are a conceptually simpler choice with a hopefully lower barrier for implementing them.

On various occasions, ElectionGuard needs to hash values to $\mathbb{Z}_q$. Hashing thus requires some caution because outputs of hash functions are bit strings, so their interpretation as integers modulo q may not be uniformly distributed values in $\mathbb{Z}_q$. In some protocols, this discrepancy can create security issues [15]. There are standard solutions to this problem: for

[1] See, for instance: https://app.enhancedvoting.com/results/public/cc/CollegePark/nov23.

[2] For a discussion of the use of blockchains in voting systems, see pp. 103–105 of the U.S. National Academies report at https://nap.nationalacademies.org/catalog/25120

[3] In either case this could come down to checking a hash value.

[4] For instance, some contests may be available to all voters in an election while others may depend on the district in which a voter resides.

[5] The ElectionGuard specification [6] more generally describes how this p is to be computed for any desired size. But in practice, having verifiers check that a particular prime p is being used relieves them of the complex computational burden of confirming that p has been correctly computed.

[6] This choice also appears in the FIPS PUB 186-4 standard for instance https://csrc.nist.gov/pubs/fips/186-4/final.

instance, one can hash to bit strings substantially longer than q, so that their reductions modulo q are distributed statistically close to uniform. ElectionGuard takes a different approach by selecting $q = 2^{256} - 189$ which is the largest prime below 2^{256}. Hashing to 256-bit strings, converting them to integers, and reducing modulo q then provides values that are almost uniform modulo q when the hash function is modeled as a random oracle. In practice, no 256-bit hash value larger than q will ever occur, and the hash results taken modulo q will be indistinguishable from uniform.

As for the choice of p, ElectionGuard selects a 4096-bit prime such that its binary representation starts and ends with 256 bits set to one, is such that q divides $p-1$ and that $(p-1)/2q$ is also prime. The 3584 middle bits of p are determined as the bits corresponding to the smallest integer larger than the integer given by the first 3584 bits of $\ln(2)$ (the natural logarithm of 2) that make the resulting p compatible with the above constraints. The bits set to one support faster modular reduction, while the middle bits are chosen to prevent an effective use of SNFS [29]. Following this process that is determined by efficiency and security considerations leads to a unique prime p. The standard group generator g is simply chosen as $g = 2^{(p-1)/q} \bmod p$, which is different from 1 and therefore generates the group of order q.

Further cryptographic parameters are the number of guardians n and the quorum value k. The small integer n denotes the number of guardians that collaborate to generate the election public key used to encrypt all votes. To decrypt the election tallies that have been computed by homomorphic addition of encrypted votes, at least $k \leq n$ guardians are required to participate. The quorum can be chosen to be smaller then n to enable decryption even if some of the original guardians become unavailable or uncooperative.

2.4 Keeping track of the context

All cryptographic data produced by ElectionGuard are kept within a clear context. The fixed cryptographic public parameters are tied to all cryptographic operations, starting with key generation. As soon as the election context is known through the election manifest, it is additionally tied to all the election related operations, in order to prevent data reuse across elections. And, within an election, every cryptographic element is unambiguously labeled in order to prevent any internal reuse.

In order to offer public verifiability, ElectionGuard makes an abundant use of zero-knowledge (ZK) proofs that are traditional sigma protocols made non-interactive through the Fiat-Shamir heuristic [26]. The random oracle instance used to implement the Fiat-Shamir heuristic is used to tie all cryptographic data in the election records to their context.

In ElectionGuard, the random oracle is instantiated with HMAC-SHA-256, which is believed to offer a good option when used with keys of a fixed length smaller than the underlying hash function input block size [23]. Notation for the random oracle is $H(\cdot;\cdot)$, where the first input is passed to HMAC-SHA-256 as the key and the second as its input. In ElectionGuard, all HMAC-SHA-256 keys have a fixed length of 32 bytes. For hashing into $\mathbb{Z}_q$, the ouput of H is interpreted as an integer and reduced modulo q. This hash function is denoted $H_q(\cdot;\cdot) = H(\cdot;\cdot) \bmod q$.

As an illustration of this process, the first hash that is computed, the parameter hash $\mathrm{H}_P = H(\mathtt{ver};\mathtt{0x00},p,q,g,n,k)$ sets the cryptographic public parameters, where `ver` is the ElectionGuard specification version number that is used, padded to 32 bytes. From H_P and the election manifest, an election base hash is computed as $\mathrm{H}_B = H(\mathrm{H}_P;\mathtt{0x01},\mathtt{manifest})$, where `manifest` is a file containing the election manifest. The parameter hash H_P is used in the key generation process, which ElectionGuard does not link to a specific election.

After election public keys K and $\hat{K}$ have been generated, an extended base hash is computed as $\mathrm{H}_E = H(\mathrm{H}_B;\mathtt{0x14},K,\hat{K})$.

When a new ballot is encrypted, a random 32-byte selection encryption identifier id_B is chosen and an identifier hash is computed as $\mathrm{H}_I = H(\mathrm{H}_E;\mathtt{0x20},\mathrm{id}_B)$, which is then used in all proofs of validity of that ballot. The extended base hash H_E is also used in every decryption proof produced during the tallying operations.

All these hashes guarantee the soundness of the ZK proofs [10], and prevent carrying election data from one election to another, mixing data from different ballots, and, more generally, reusing cryptographic data in an unintended context.

2.5 Key Generation

ElectionGuard's key generation process uses a variant of Pedersen's distributed key generation (DKG) protocol [38] supporting a dishonest majority at the price of reduced robustness. The key generation ceremony runs between publicly identified parties, and there is little benefit for a malicious participant in introducing errors in the protocol execution if these errors are detected before the keys are actually used to encrypt sensitive material. Therefore, we assume that guardians abort key generation as soon as one of them detects an error, that they investigate the error out-of-band, and restart the protocol from scratch, possibly replacing guardians that are identified as acting maliciously.

Although distributed key generation has become relatively common, it can be quite challenging to get all of the details right, and many other efforts have missing or incorrect proofs.[7]

The election administrator publishes a quorum $k \leq n$ in the election record, which means that at least k guardians are needed to compute the election tally. It also means that ballot privacy will depend on the assumption that fewer than

[7] For example, the widely-used Belenios system references an incorrect proof [7].

k guardians are compromised—verifiability does not depend on the number of honest guardians. Pedersen's original protocol [38] was only designed to support $k < n/2$, which guarantees a majority of players are honest such that the protocol terminates and ciphertexts can always be decrypted. Our concern here is that corrupted guardians might want to decrypt individual votes that they should not decrypt, rather than that they refuse to decrypt ballots they are supposed to decrypt. Allowing a quorum less than n accommodates emergency situations on behalf of one or a few guardians, but it needs to be kept high enough to protect from curious guardians.

The protocol starts with each guardian G_i selecting a random polynomial $P_i(x) = \sum_{j=0}^{k-1} a_{i,j}x^j$ and publishing commitments $K_{i,j} = g^{a_{i,j}}$ for $0 \leq j < k$ in the election records, following Feldman's verifiable secret sharing (VSS) protocol [25]. Each guardian G_i also selects a random secret share encryption key ζ_i and publishes the corresponding public key $\kappa_i = g^{\zeta_i}$. Additionally, each guardian publishes Schnorr proofs [42] that it knows how to open the commitments it posts. Afterwards, each guardian G_i sends the value $P_i(j)$ to guardian G_j, encrypted with G_j's public share encryption key κ_j, for every $0 < j \leq n$. Then, each guardian G_j checks the validity of all posted Schnorr proofs, and verifies that $g^{P_i(j)} = \prod_{\ell=0}^{k-1}(K_{i,\ell})^{j^\ell}$ for $0 < i \leq n$. Finally, the public vote encryption key is set to $K = \prod_{i=1}^{n} K_{i,0}$ and each guardian G_i sets its private key share to $P(i) = \sum_{j=1}^{n} P_j(i)$.

The above key generation protocol is run a second time, either in parallel or sequentially, to generate a second public key $\hat{K}$ (together with commitments $\hat{K}_{i,j}$ and corresponding Schnorr proofs). This key is used for encrypting ballot data other than numerical votes, which is done with a different encryption mode.

At the end of this protocol execution, each guardian's software[8] displays a hash of its view of the public key generation protocol transcript records, including all the election parameters, $K, \hat{K}$ and all the $K_{i,j}$, $\hat{K}_{i,j}$, and κ_i. Each guardian compares that view with the one that appears in the election record. If this or any of the previous verification steps fail, the guardian complains and an out-of-band investigation is started before the protocol restarts from scratch, possibly with one or more new guardians. If no guardian complains, the content of the election record up to that point is validated and called the *guardian record*. The goal of these verification steps is to prevent a malicious election administrator from impersonating guardians or even to simply throw away the key generation transcript and to replace it by another fully simulated one—there is no public-key infrastructure (PKI) in ElectionGuard, so guardian messages are not signed.

As noted above, this protocol follows the general structure of Pedersen's [38] by producing a key computed as a sum of individual keys generated by guardians that are shared using a verifiable secret sharing (VSS) protocol. The differences with Pedersen's protocol are that (i) ElectionGuard does not start with a round during which every guardian commits to $K_{i,0}$ ($\hat{K}_{i,0}$) and then opens it, (ii) ElectionGuard adds Schnorr proofs for the $K_{i,j}$ ($\hat{K}_{i,j}$), and (iii) ElectionGuard relies on guardians checking authentic election records instead of signing their key shares.

The vast majority of proposed DKG protocols, including [12, 14, 28, 31, 33, 38] for instance, focus on the honest majority case, which offers robustness. The dishonest majority case has been considered in the context of threshold signatures [24, 32, 34], and ElectionGuard's DKG may be closest to the one proposed by Komlo and Goldberg for FROST [32]. In the following theorem, we claim the IND-CPA security of ElGamal encryption based on keys produced with the ElectionGuard DKG—we refer to the combined scheme as ElectionGuard ElGamal.

Theorem 1. ElectionGuard *ElGamal encryption is IND-CPA secure under static corruption of up to $k-1$ guardians ($1 \leq k \leq n$) if standard ElGamal encryption (performed with the standard single-party key generation) is IND-CPA secure with the same public parameters.*

The proof of this theorem follows the strategy from Braun et al. [12], adapted to the dishonest majority case.

Proof. Let k and n be fixed and define $C = \{1, \ldots, k-1\}$ and $H = \{k, \ldots, n\}$. Assume, w.l.o.g., that the guardians in the set $\{G_i\}_{i \in C}$ are corrupted, while the others are honest. We design an adversary $\mathcal{B}$ against standard ElGamal encryption that wins the IND-CPA game with a probability equal, up to a negligible difference, to the probability that an adversary $\mathcal{A}$ controlling the corrupted guardians breaks the IND-CPA security of the ElectionGuard ElGamal encryption.

Given $\mathcal{A}$, we design $\mathcal{B}$ as follows: when $\mathcal{B}$ receives the ElGamal public parameters $(\mathbb{Z}_p^*, q, g)$ and the public key K^* from the standard ElGamal challenger, it forwards the public parameters to $\mathcal{A}$. $\mathcal{B}$ honestly plays the role of all honest guardians, except for G_n. For emulating G_n, it simulates a share of the discrete logarithm of $K_{n,0} = K^*$ by picking $P_n(i)$ as a random element of $\mathbb{Z}_q$ for every $i \in C$. It then derives $\{K_{n,i}\}_{i \in C}$ so that $g^{P_n(i)} = \prod_{j=0}^{k-1}(K_{n,j})^{i^j}$ for every $i \in C$, by Lagrange interpolation "in the exponent". $\mathcal{B}$ also submits simulated Schnorr proofs for every $K_{n,i}$ with $i \in \{0\} \cup C$. The view of $\mathcal{A}$ is distributed in a way that is identical to what it would be in a normal execution of the ElectionGuard ElGamal key generation.

$\mathcal{B}$ then verifies the guardian records: it checks that all the expected $K_{i,j}$ values appear in the records, that the share verification succeeds for all the values submitted by $\mathcal{A}$, and that the Schnorr proofs are valid. If any of this fails, $\mathcal{B}$ aborts and $\mathcal{A}$ loses. If the verification succeeds, $\mathcal{B}$ extracts the discrete logarithms of the $K_{i,0}$ keys for $i \in C$ by rewinding $\mathcal{A}$ appropriately—this can be performed efficiently since it is a single round of extraction.

[8] Ideally, each guardian will use its own software obtained from a source of its own choosing.

Eventually, when $\mathcal{A}$ asks for the encryption of a pair of messages (m_0, m_1), $\mathcal{B}$ forwards it to the ElGamal challenger, who returns a ciphertext $(c_0, c_1) = (g^r, m_b(K^*)^r)$. $\mathcal{B}$ then submits to $\mathcal{A}$ the ciphertext $(c_0, c_1(c_0)^{\sum_{i=1}^{n-1} s_i}) = (g^r, m_b(\prod_{i=1}^{n} K_{i,0})^r)$ where each s_i is the discrete logarithm of $K_{i,0}$, which has either been extracted from the Schnorr proofs or has been selected by $\mathcal{B}$. When $\mathcal{A}$ outputs a guess b' on b, $\mathcal{B}$ forwards that guess to the ElGamal challenger. The probability that $b = b'$ is exactly the one that $\mathcal{A}$ makes a correct guess in the ElectionGuard ElGamal IND-CPA security game. The only possible discrepancy comes from the potential failure to extract a Schnorr proof, which can be made negligible. □

This protocol works for any quorum k. If a quorum $k = n$ is chosen, the VSS phase of the protocol could be avoided and the DKG could be as simple as asking each guardian to publish its public key together with a Schnorr proof that it knows the corresponding private key, as is done in the Helios voting system for instance [1]. In such a case, each guardian only needs to check that the election public key is indeed the product of a list of public keys that includes its own. In particular, no communication between guardians is needed, which may simplify the key generation ceremony by removing guardian coordination requirements. ElectionGuard chooses to offer a single key generation protocol to keep its specification and election verification as simple and uniform as possible.

2.6 Ballot Preparation

A typical ElectionGuard ballot consists of a sequence of ElGamal encryptions—usually of either 0 or 1, encoded in exponential form to support an additive homomorphism, together with ZK proofs that these encryptions are indeed encryptions of bits. There is one ciphertext per choice on the ballot, and, for most election types, it encrypts 1 if and only if the voter supports the corresponding choice.[9] The encrypted tally is computed by multiplying together all ciphertexts corresponding to the same choice across all cast ballots.

ElectionGuard also supports more general option selection limits and contest selection limits: the former allow a voter to assign a value in a given range to a specific option on the ballot (enabling systems like cumulative voting, range voting, and score voting), and the latter allow the number of options selected by a voter within a specific contest to lie in a specific range (enabling voters to select more than one option in a contest).

Again, ElectionGuard largely follows Cramer et al. [20], but makes some important tweaks and additions.

2.6.1 The ballot nonce

For each ballot B, a secret random nonce ξ_B is chosen in addition to the public random selection encryption identifier id_B. As discussed above, id_B is used to derive the ballot identifier hash H_I, which also incorporates the information related to the election public key and to the election manifest.

This identifier hash is used, together with the secret ballot nonce ξ_B, to derive the randomness $\xi_{i,j}$ used to encrypt the voter's choice for option j of contest i on the ballot:

$$\xi_{i,j} = H_q(\mathrm{H}_I; \texttt{0x21}, i, j, \xi_B).$$

This approach is useful in several use cases.

- Some ElectionGuard-enabled systems can take advantage of this nonce to efficiently challenge an encryption operation performed by an untrusted device. In such systems, the device is required to publish a ballot confirmation code, essentially a hash of all the ciphertexts on that ballot (we will discuss this further below), before the voter decides to cast the ballot. The voter may then choose to challenge the device, in which case the device only needs to provide the 32-byte ballot nonce ξ_B, which can then be used to re-derive, on a personal device, all the $\xi_{i,j}$ values, recompute all the ciphertexts on the ballot, and verify that the ballot confirmation code is actually consistent with the voter's choices. If a ballot is cast instead of being challenged, the ballot nonce is permanently erased.

- Some systems may use an offline ballot marking device (BMD) to print a human readable ballot with the voter's choices, and to compute and print the ballot confirmation code that the voter takes home. As the BMD is offline, there may be no way to transmit the full ciphertexts to the server that publishes the encrypted ballots for verification and in support of the tallying operations. When the paper ballots are anonymous (because they are dropped in a ballot box, for instance), a solution might be to print an encrypted version of ξ_B on the ballot so that, after scanning and decryption, an election server can recompute all ciphertexts on the ballot with the same randomness as the BMD, supporting the election verification steps.

2.6.2 Encryption

The encryption of a selection σ made by a voter, using randomness ξ derived as explained above, is computed as $(\alpha, \beta) = (g^{\xi}, K^{\xi+\sigma})$. Each ElGamal ciphertext is accompanied by a proof that it really encrypts a selection $\sigma \in \{0, 1\}$ (or, when allowed, in a specified larger domain), which is computed as a disjunctive version of the Chaum-Pedersen protocol [16, 19], following a most common choice in voting

[9] A blank vote in a contest would be represented by an encryption of 0 for every option.

systems [1, 4, 18, 20, 40]. The proof is generated by first computing a commitment consisting of two ciphertexts $(a_0, b_0) = (g^{u_0}, K^{u_0+\sigma c_0})$ and $(a_1, b_1) = (g^{u_1}, K^{u_1+(\sigma-1)c_1})$ for random choices of u_0, u_1, c_0, c_1 in $\mathbb{Z}_q$. Then, a global challenge is computed[10] as $c = H_q(\mathrm{H}_I; \texttt{0x24}, \alpha, \beta, a_0, b_0, a_1, b_1)$ and the selection challenge is updated to $c_\sigma = c - c_{1-\sigma}$. Observe that c_σ did not play any role in the computation of the commitment. Eventually, responses are computed as $v_0 = u_0 - c_0\xi$ and $v_1 = u_1 - c_1\xi$. The proof consists of (c_0, c_1, v_0, v_1) and is verified by checking that (α, β) are elements of the expected subgroup of $\mathbb{Z}_q$, recomputing the commitment, recomputing c using H, and verifying that c is the sum of c_0 and c_1.

The ElectionGuard encryption process differs from the traditional encryption method where ciphertexts are computed as $(g^\xi, g^\sigma K^\xi)$ [1, 4, 18, 20, 40]: this modification has no impact on security but reduces the cost of computing the ZK proof from 5 to 4 exponentiations, following an observation by Devillez et al. [22]. The recent literature offers numerous other options for computing these 0-1 encryption proofs, and these often lead to more compact and faster ways to compute proofs—several such methods are also described and compared by Devillez et al. [22]. ElectionGuard aims at keeping the programming of an election verifier as simple as possible, and therefore keeps these disjunctive Chaum-Pedersen proofs. In terms of proof size, a proof only takes 1024 bits compared to ciphertexts of 8192 bits, so the relative benefits of smaller proofs appear to be marginal. Furthermore, as will be described in the implementation section, the computation process can also be made fast enough for practical deployments by relying on other simple techniques, including fixed-base exponentiation methods.

2.6.3 Encryption of more sophisticated ballots

The encryption process just described can be used for traditional election methods and approval voting, i.e., contests in which voters can select as many options as they like. ElectionGuard supports several other types of contests:

- ElectionGuard includes range proofs, which are immediate extensions of the disjunctive proof technique used to demonstrate that a ciphertext encrypts a 0 or a 1. These can be used to prove that a single selection ciphertext is within an arbitrary range $[a, b]$, hence offering support to other types of voting methods, including cumulative voting and Borda count. These can also be used to prove that the product of the selection encryptions of a single contest is within a range, hence proving that a voter selected exactly or at most x options in that contest for instance.

- ElectionGuard supports contest write-in data, where a voter can enter text in a write-in field instead of selecting a listed option. Here, an extra counter is encrypted for the write-in, and the text itself is encrypted using the secondary ElGamal public key $\hat{K}$ in the threshold version of Signed ElGamal mode [43], hence offering NM-CPA security just as the security mode used to encrypt selections [10]. The counter is used to count the number of voters who used the write-in option and, when that number is low enough to guarantee that no write-in candidate could win (which is expected in most cases), the text of the write-ins is simply ignored and never decrypted. Otherwise, these write-in ciphertexts may be decrypted, providing that they offer sufficient anonymity guarantees.

2.6.4 Confirmation Codes

After all selections on a ballot have been encrypted, a *confirmation code* is prepared and, in most common use cases, given to the voter. The code is a hash value computed from the sequence of ciphertexts that constitute the encrypted ballot as follows. First, for each contest on the ballot, a *contest hash* is computed. Specifically, if (α_i, β_i) is the ciphertext encrypting the i-th selection of the l-th contest (with m options) on the ballot, the contest hash is

$$\chi_l = H(\mathrm{H}_I; \texttt{0x28}, \texttt{ind}_\texttt{c}(\Lambda_l), \alpha_1, \beta_1, \alpha_2, \beta_2, \ldots, \alpha_m, \beta_m),$$

where $\texttt{ind}_\texttt{c}(\Lambda_l)$ is the unique contest index for the index with label Λ_l specified by the election manifest. Ciphertexts are hashed in the order specified by the election manifest.

All contest hashes are then hashed in the order specified by the election manifest to obtain the confirmation code (assuming there are m_B contests on the ballot) as

$$\mathrm{H}_C = H(\mathrm{H}_I; \texttt{0x29}, \chi_1, \chi_2, \ldots, \chi_{m_B}, \mathrm{B}_C).$$

The last input B_C is the chaining field byte array that determines the ballot chaining mode and the additional input for ballot chaining.

ElectionGuard offers the option to chain ballots together via dependencies of their confirmation codes. A simple chaining mode would include the confirmation code of the previous ballot that was encrypted on the same device into the computation of the current confirmation code. The chaining field B_C is a fixed-length byte array that consists of a chaining mode identifier and the previous confirmation code. This means, the j-th confirmation code computed on a device is

$$\mathrm{H}_j = H(\mathrm{H}_I; \texttt{0x29}, \chi_1, \chi_2, \ldots, \chi_{m_B}, \mathrm{B}_{C,j}),$$

where $\mathrm{B}_{C,j} = i_C \parallel \mathrm{H}_{j-1}$ is the concatenation of the chaining mode identifier i_C and the confirmation code H_{j-1} of the previous ballot computed on the same device. The first ballot encludes an initial hash value that is derived from device information.

[10] ElectionGuard often hashes more context inputs like contest and selection indices that are omitted here for simplicity.

The goal of this chaining process is similar to that of the one proposed by Sandler and Wallach [41]: to make it easier to detect a modification to any ballot. When ballots are chained, any change in a ciphertext that appears on a ballot would require adapting all subsequent elements of the chain in order to keep the chain consistent and valid. As a result, any voter who verifies a confirmation code linked to a ballot appearing in the original chain after the modified ciphertext was included, would detect the modification. This makes detection more likely than in the absence of chaining: without chaining, only the voter holding the confirmation code of the modified ballot is able to detect the modification.

The ElectionGuard hashing process to compute confirmation codes does not include ZK proofs as inputs, which differs from other systems [1, 18]. Including only the ciphertexts is sufficient to commit to the content of the votes and, in some applications, it is convenient to be able to delay the computation of ballot validity proofs to a subsequent step in the election process.

Regardless of the chaining mode, the most a typical voter would do is check an election record to ensure that the expected confirmation code is present. The computations to confirm that confirmation codes are correct are more commonly delegated to election verifiers who, as part of the process of verifying the integrity of an election record, will confirm that each confirmation code in the record is correctly computed. Of course, anyone – including voters – can verify any or all of the contents of an election record.

A component of an election verifier can be a single ballot verifier which takes as its input all data used to generate a confirmation code (including, when appropriate, the chaining variable) and confirms the correct computation of a confirmation code. A single ballot verifier can be particularly useful when instant verification of challenges is offered to voters (for instance, at a poll site). In this scenario, a voter who challenges a ballot would receive information that would allow an app (perhaps installed on a mobile device) to check that the confirmation code is consistent with the selection made by the voter.

It is important to emphasize that since confirmation codes are derived entirely from encryptions of votes, their being given to voters and even their publication in properly deployed in-person applications does not in any way compromise voter privacy or enable coercion or vote-selling.

2.7 Tally

2.7.1 Update of the Election Record

Once voting has concluded, all encrypted ballots consisting of encrypted selections and corresponding ZK proofs of ballot well-formedness are added to the election record, with a clear mark indicating whether each ballot is intended to be included in the tally or is challenged, in which case it is not included in the tally and needs to be individually decrypted. The election record is committed to by the election administrator, and made available for everyone to check, in particular to the guardians who can compare their views of these election records. Again, requiring the administrator to sign the records may help to identify a rogue administrator who would distribute different views of the records to different parties.

2.7.2 Homomorphic tally computation

All ZK proofs of well-formedness are verified, the uniqueness of the selection encryption identifiers is verified, and the encrypted ballots that are intended to be tallied are aggregated to obtain the encrypted tallies. These are computed via the additive homomorphism of the exponentially encoded ElGamal encryption, that is, they are the componentwise products of all selection encryptions that correspond to the same option across all encrypted ballots containing that contest.

2.7.3 Decryption

When all encrypted tallies are available, at least k guardians reconvene together with the election administrator and jointly decrypt every tally for every selection in every contest of the election. Each available guardian G_i uses its private key share $P(i)$ to compute a partial decryption $M_i = A^{P(i)}$ of each given tally ciphertext (A,B). The available partial decryptions are combined to obtain the value $M = \prod_{i \in U} (M_i)^{w_i}$, where $U \subseteq \{1,2,\ldots,n\}$ is the set of indices of the available guardians, and the w_i for $i \in U$ are the corresponding Lagrange coefficients used to reconstruct the shared secret. The value M is therefore the value that would have been obtained by decrypting with the secret key corresponding to the election public key K. This secret key remains implicit in all ElectionGuard computations and is never explicitly reconstructed.

The value M allows decryption of the tally t via $K^t = BM^{-1}$ through the computation of a small discrete logarithm that can be computed efficiently (for example using a precomputed table) because tally values are typically bounded by the number of votes cast, a relatively small number.[11]

2.7.4 Proof of correct decryption

To enable verifiable decryption, the available guardians collaborate to produce a Chaum-Pedersen proof of correct decryption with the secret key corresponding to K. This proof can be computed in a way that is reminiscent of threshold signatures.

We suppose that a set $U \subseteq \{1,\ldots,n\}$ of guardians is available for performing the decryption of a ciphertext (A,B) and that $|U| \geq k$. Each guardian G_i, $i \in U$ starts by committing to

[11] In voting methods that allow a voter to place more than one vote on a candidate, the discrete logarithms may be slightly larger, but they are still easily computed.

the commitment of an individual Chaum-Pedersen proof, by selecting a random $u_i \leftarrow \mathbb{Z}_q$, computing $(a_i, b_i) = (g^{u_i}, A^{u_i})$, and sending $d_i = H(\mathrm{H}_E; \texttt{0x30}, i, A, B, a_i, b_i, M, U)$.

When G_i has received a d_j value from every other guardian G_j, $j \in U$, it sends the pair (a_i, b_i). After receiving the (a_j, b_j) from the other guardians, it then checks that d_j was computed correctly for all $j \in U$ and halts if any of these verification steps fail. This round of commit-reveal guarantees that corrupted guardians are unable to select their (a_i, b_i) pairs as a function of the pairs of the honest guardians, and that the product $(a, b) = (\prod_{i \in U} a_i, \prod_{i \in U} b_i)$ is uniformly distributed as long as one of the guardians G_i, $i \in U$ is honest.

After these verification steps, the proof challenge is computed as $c = H_q(\mathrm{H}_E; \texttt{0x31}, A, B, a, b, M)$. Each guardian G_i $(i \in U)$ can then use the i-th Lagrange coefficient w_i to compute an adjusted challenge $c_i = c \cdot w_i \bmod q$ and an individual response $v_i = u_i - c_i P(i) \bmod q$. The final response is computed as $v = \sum_{i \in U} v_i \bmod q$, and the decryption proof is the pair (c, v). Note that $v = \sum_{i \in U} u_i - c \sum_{i \in U} w_i P(i) = \sum_{i \in U} u_i - c \cdot s \bmod q$, but again, the secret election key s has not been explicitly reconstructed to compute this proof.

If the proof is valid, that is, if $v \in \mathbb{Z}_q$ and if c can be correctly recomputed as above, using $a = g^v K^c$ and $b = A^v M^c$, then it is published in the election record.

It would have been more direct for each guardian to publish an independent Chaum-Pedersen proof that it computed its partial decryption accurately. However, as ElectionGuard aims at making election verification as simple as possible, the present protocol, which is more complex for the guardians, offers the central benefit of making the role and identity of the participating guardians completely invisible to an election verifier. From the verifier perspective, the decryption proof is one single Chaum-Pedersen proof.

It can be observed that this protocol is very similar to the Sparkle threshold signature protocol [21]. Its security analysis partly follows similar arguments. First, we can observe that the protocol is sound: since it produces valid Chaum-Pedersen proofs, it follows that it has the same special soundness property and that obtaining two executions with the same commitments but two different challenges and valid responses makes it possible to extract the unique value s such that $K = g^s$ and $M = A^s$. We can also show that honest guardians do not reveal any information by computing the Chaum-Pedersen proof as described above.

We believe this approach of combining zero-knowledge proofs of partial decryptions into a single zero-knowledge proof of complete decryption to be novel. It is especially valuable under threshold conditions where some partial decryptions may not be available, and it completely removes one of the greatest challenges for verifiers, which is the handling of missing guardians and the associated Lagrange coefficients. This approach may have application outside of the domain of elections.

Theorem 2. *Let $U \subseteq \{1, 2, \ldots, n\}$ be the set of indices of the guardians participating when computing the decryption proof as described above. Let V be a non-empty subset of U containing the indices of the honest guardians. There exists a simulator S that, on inputs (A, B), M_i $(i \in V)$ and $K_{i,j}$ for $1 \leq i \leq n$ and $0 \leq j < k$, produces an interaction with the corrupted guardians with index in $U \setminus V$ that is statistically indistinguishable from an interaction in a real execution of the protocol. We let S control the function H, modeled as a random oracle.*

Proof. S selects a random $c \leftarrow \mathbb{Z}_q$, computes $c_i = cw_i \bmod q$ accordingly, and selects a random response $v_i \leftarrow \mathbb{Z}_q$ for each guardian with index in V. It then sets $a_i = g^{v_i} \left(\prod_{j=1}^{n} \prod_{m=0}^{k-1} K_{j,m}^{i^m} \right)^{c_i}$ and $b_i = A^{v_i} M_i^{c_i}$ for each guardian with index i in V. Note that the double product between the parentheses in the computation of a_i equals $g^{P(i)}$.

S then starts playing the protocol with the corrupted guardians, using the values above. The random choice of the v_i values and their independence from the (a_j, b_j) values of the corrupted guardians ensures that, with overwhelming probability $\geq (1 - t/q^2)$, where t is the number of queries that the corrupted parties make to the random oracle, the query that is made in order to compute the proof challenge will be fresh. S can then program the random oracle to offer c as the answer to that query. The resulting conversation will then be distributed exactly as a real conversation. □

We observe that the commit-reveal round of the protocol is essential to guarantee that the random oracle query made to compute the proof challenge is a fresh query with overwhelming probability.

2.7.5 Decryption of challenged ballots

The election record may also contain a set of challenged ballots, in order to support cast-as-intended verifiability, which will be discussed in Section 2.8.1. Valid challenged ballots are decrypted exactly in the same way as the aggregated ballots that are included in the tally. The verifiable decryption data provided for the challenged ballots is included within the election record.

2.8 Election Verification

ElectionGuard's main objective is to make elections verifiable. The value of a verifiable election is only fully realized, when the election is actually verified, for example by voters, election observers, or news organizations. ElectionGuard supports two central aspects of E2E-verifiability [8]:

1. *Cast-as-intended verifiability* makes it possible for a voter to verify that the selections made by the voter have been properly recorded.

2. *Tallied-as-cast verifiability* makes it possible for a voter to verify that every recorded vote is correctly included in the tally.

Eligibility verifiability, that is, the possibility to verify that ballots have been submitted by eligible voters, is also a desirable feature of an election. However, it typically relies on external voter identification mechanisms that are election dependent. As such, ElectionGuard enables verifying the number of ballots that are included in the tally, but not their origin. When paired with an external list of people who voted, this enables full verifiability of the results by participants and observers.

2.8.1 Cast-as-intended verifiability

The confirmation code of a ballot is the key element supporting cast-as-intended verifiability.

First, it makes it possible for a voter to challenge a ballot. This challenging operation enables a voter to verify that all selections were correctly encrypted and can be implemented in different ways.

- Most often, the encryption process happens after a voter expresses selections. In this case, the encryption device must commit to the ballot confirmation code and offer the choice to either cast or challenge the prepared ballot. The two choices are mutually exclusive, since the challenging operation reveals the selections on the ballot. A challenged ballot can never be cast and a cast ballot cannot be challenged anymore. The challenging process proceeds either by revealing the ballot nonces of the challenged ballots if they are available, or by decrypting the challenged ballots using the normal verifiable decryption process. It is again of crucial importance to verify, before decryption, that the decrypted ballots are valid and have a unique encryption identifier within the election. A ballot nonce can typically be revealed as soon as the ballot is challenged, while a ballot decryption operation will usually happen at tallying time. The voter can then verify that the selections made were correctly reflected in the decrypted ballot.

- ElectionGuard can also be used with pre-encrypted ballots: voters are then provided with vectors of ciphertexts encrypting every possible selection on a ballot, and select the ciphertexts that are claimed to match their selection. This can be useful in some vote-by-mail scenarios or for pre-printed ballots for in-person voting. In such a setting, it is possible to compute a confirmation code that includes hashes of every available encrypted selection, and the challenge process can then be used to verify that these ciphertexts actually encrypt the expected selections. This approach has two advantages: first, the challenging process is independent of any actual selection made by a voter, which is good for privacy and, second, the verification can be performed by anyone and does not depend on a voter accurately reporting selections, which offers accountability.

Whether a ballot is challenged or cast, it is included in the election record. Voters can then check that one or more ballots (a cast ballot and any challenged ballots) with the expected confirmation codes appear in the election records and for challenged ballots, that it shows the correct selections.

2.8.2 Tallied-as-cast verifiability

At the end of the tallying operations, the election records contain all the ballots and decryption proofs. Any voter can then verify that the ballot the voter cast is accurately incorporated in the homomorphic aggregation of the encrypted selections, and that the decryption proof demonstrates the validity of the announced tally with respect to the aggregated encrypted selections. That second verification can also be carried out by any observer, voter or not.

2.8.3 Privacy checks

All the verification steps that we described confirm the correctness of the election results and are independent of the honesty of the guardians: even a coalition of all guardians together with the administrator would not be able to fake the evidence offered on an election tally.

ElectionGuard however relies on the guardians to guarantee that the votes remain secret, and the guardians are expected to perform several verification steps as part of their role. These steps are not sufficient to guarantee that the votes remain secret: a ballot marking device may be corrupted and leak encryption nonces, or just use a broken pseudo-random number generator (PRG), and there is nothing that the guardians can do to prevent that.[12] Nevertheless, guardians should perform certain verifications to prevent ElectionGuard from being used to compromise voter privacy.

We summarize these steps here: their technical description has been provided above.

1. The guardian software must verify that the cryptographic parameters that are used offer the expected security level. This avoids accepting primes that would be too small for instance.

2. Guardians must check the guardian record at the end of the key generation phase: they must be certain that the published view of the key generation process is consistent with their own. This prevents a rogue administrator from replacing a guardian's key, or from subverting a guardian's key by sending malicious messages during the key generation process.

[12] Cryptographic means cannot ensure that there are no cameras hidden behind voters recording their actions.

3. Guardians must check that the ciphertexts they decrypt are those listed for decryption in the election record. This prevents a rogue administrator from asking for the decryption of individual cast ballots for instance. Guardians should also verify that ballots have been aggregated correctly and that all ballots have a unique selection encryption identifier and correct validity proofs. Though the latter task may be delegated to others, it must be verified before guardians engage in any decryption.

2.8.4 Verification software

In order to perform all these verification steps, voters, guardians and observers should use software that offers as much independence as possible. For instance, if the goal is to detect a malicious election administrator, it makes no sense to blindly trust software provided by the election administrator to perform these verification steps. Public code and availability of software from multiple sources are two important features for a meaningful use of ElectionGuard, and the ElectionGuard website[13] offers links to six independent verifiers (as well as one independent port of the full ElectionGuard implementation).

2.9 Trust

With the design described herein, privacy of ballots is dependent upon the cryptographic assumption that standard ElGamal encryption is IND-CPA secure as shown in Theorem 1 together with an assumption that the device on which the ElectionGuard application performing ballot encryption (such as an in-precinct ballot scanner) maintains ballot privacy. These assumptions need to be paired with necessary physical assumptions about any particular use case. Cryptography cannot prevent strategically-placed cameras or key-loggers from learning the inputs of voters.[14] A principal goal of ElectionGuard—as a technology that is intended to augment but not replace existing voting methods—is to add verifiability without materially weakening privacy.

What ElectionGuard achieves is the ability for voters and observers to verify the accuracy of election results with minimal trust—and critically with no trust at all in election administrators or any other specific entities (such as device manufacturers or software implementers). There is a very weak assumption on the collision resistance of the SHA-256 hash function (essentially that an adversarial agent should not be able to produce two distinct votes or ballots with the same hash). Beyond this, voters and observers may proxy their trust as they wish to one or more verifiers of their choosing or they may instead choose to write their own verifiers and confirm the integrity of the results entirely on their own.

Confidence in an election result requires confidence that the ballots being counted are genuine. A single, digitally-signed election record allows all voters to confirm that their votes are properly included. The discovery of two distinct signed election records immediately implicates the election administrator who published those records as maleficent.

An interesting challenge in some applications is so-called eligibility verification—ensuring that all votes are cast by voters who are eligible to vote and do not vote more often than allowed. In the U.S. and many other countries, the list of voters who cast ballots in any jurisdiction is a matter of public record. Eligibility is thereby achieved entirely through publicly-verifiable processes that are entirely outside the scope of ElectionGuard, and the only intersection is for interested parties to confirm that the number of ballots cast does not exceed the number of voters listed. There is an expectation that interested parties will scrutinize this list for fictitious voters and for voters who are listed as voting but who did not do so. When voting is in-person, observers can simply count the number of voters present. But these measures are beyond the scope of ElectionGuard. When the list of voters is not public, it is very difficult for observers to verify that insiders did not add additional ballots to the count and inflate counts of the number of voters.

3 Preferred Uses

The primary intended use of ElectionGuard is for in-person poll-site voting. But there are many possible designs that are compatible with a wide variety of equipment. Additionally, many elections offer different modes of voting. For example, hand-marked paper ballots for in-person voters together with machine-marked ballots for voters with disabilities that prevent them from marking physical ballots, together with mailed ballots for voters who are not able to vote in person. To be effective, ElectionGuard must enable all of these and other modes of voting in a single election. There are even applications in which voters are not present and ballot contents are encrypted administratively.

In all applications, an election using ElectionGuard begins with a key-generation ceremony in which an election administrator works with guardians to form election keys. Later, usually at the conclusion, the administrator will again work with guardians to produce verifiable tallies. What happens in between, however, can vary widely.

3.1 Precinct Ballot Scanners

An ideal use of ElectionGuard is on precinct-based ballot scanners. An in-person voter can mark a ballot—either by hand or by using a ballot-marking device—and then feed the ballot into a scanner which may read the contents of the ballot, display the contents as it interpreted them, and print a confirmation code for the voter.

[13] https://www.electionguard.vote

[14] It may be possible to use *code voting* to add a layer of indirection so that a voter's inputs do not reveal the voter's selections.

Once in possession of the paper confirmation code, the voter can choose to either accept the scanner's interpretation of the ballot contents and cast the ballot or instead challenge the ballot. A challenged ballot is marked as cancelled and the confirmation code is *opened*[15] by the scanner.[16]

A voter who challenges a ballot can then restart the voting process by marking a fresh ballot. Both cast and challenged ballots will be part of the election record.

3.2 Electronic Ballot Markers

ElectionGuard could instead be hosted by ballot marking devices. In one flow, a voter could make selections directly on an electronic (usually touch-screen) device which would then print a confirmation code and allow a voter to either cast the ballot or challenge the ballot and restart.

In another flow, a ballot marking device which can neither store ballot encryptions nor transmit them within a LAN could use the ballots themselves to convey encryptions by printing an encoding of the encryption directly onto a paper ballot.[17]

3.3 Vote by Mail

Pre-encrypted ElectionGuard ballots could be mailed to voters using the STROBE paradigm [3]. These paper ballots would have short codes printed beside each possible selection and voters could record short codes associated with their selections and check later to ensure that they appear correctly in the election record.

Unreturned ballots could serve as challenge ballots and would be opened to show that their short codes and associated confirmation codes are correct.

3.4 Internet Voting

Although not recommended for public elections[18], ElectionGuard could be used for Internet voting in a manner very similar to Helios [1]. Voters would make their selections online, receive confirmation codes, and then choose to either cast or open their ballots. The Helios model allows voters to prepare as many ballots as they wish—with only the last cast ballot counting.

3.5 Risk-Limiting Audits

ElectionGuard can also be used to protect voter privacy in risk-limiting audits (RLAs).

The most efficient post-election RLAs are *ballot-comparison audits* which operate as follows. The contents of each ballot are published together with ballot identifiers that match identifiers on corresponding paper ballots. Ballots are randomly selected from the published record and matched against the stored paper ballots to show that the stored ballots are consistent with announced tallies.

A major problem with ballot-comparison audits is that publication of ballot contents can compromise voter privacy. Voters can be coerced or sell their votes by completing a less important part of their ballots according to unusual pre-determined patterns that are likely to be unique within a voting precinct. This allows each ballot—including the remaining selections—to be associated with specific voters.

Following the VAULT paradigm of [9], ElectionGuard can be used to post encrypted versions of all ballots cast and to prove that these encrypted ballots are consistent with announced tallies. Randomly-selected ballots are decrypted and matched with corresponding paper ballots. These decryptions do not compromise voter privacy since coerced voters can simply claim that their assigned patterns did not appear because they were not among those that happened to be selected for audit.

In this application, ballots are encrypted administratively, and confirmation codes are not needed.

3.6 Other Uses

The above examples are just a sample of the wide variety of ways in which ElectionGuard can be used. The flexibility of ElectionGuard is novel and is one of its primary benefits.

It is important to note, however, that merely using the ElectionGuard tools does *not* ensure that an election's results will be publicly verifiable. There have, for example, been instances where users have sought to require voters to announce in advance when they intended to challenge a ballot or to even eliminate the challenge process entirely.

For at least the near future, it is important to include experts in the process of deciding how ElectionGuard will be used in any new deployments.

4 Implementations

Since its original specification in 2019 [6], there have been several implementations of ElectionGuard that have been used in various applications and pilot elections. Early implementations by various contributors included a Python reference implementation of the full ElectionGuard 1.0 specification, an encryption engine in C++ with C# bindings, implementations in Haskell, Java, and TypeScript. More recently, we have implemented ElectionGuard 2.0 in Kotlin and Rust.

This section discusses some of the lessons learned from building these implementations.

[15] This is accomplished by providing a verifiable decryption to demonstrate that the confirmation code is consistent with the voter's selections.

[16] It is also possible for challenged confirmation codes to be opened as part of the subsequent tally decryption ceremony.

[17] To save space, a symmetric encryption of a nonce could be recorded onto a paper ballot to allow a full encryption to be regenerated later.

[18] For a discussion of Internet voting, see the U.S. National Academies report at https://nap.nationalacademies.org/catalog/25120

4.1 Performance

Unsurprisingly, the modular exponentiation at the heart of most ElectionGuard operations imposes the highest computational cost among all computations and is the limiting factor in any performance analysis. Using fast libraries for modular arithmetic is crucial to achieve good performance so that latency due to ballot encryption and ZK proof generation does not impact usability in the voting place.

For code running on the Java virtual machine (including Kotlin), or on Android, Java's `BigInteger` class is the obvious choice. For Python, it is straightforward to use GnuMP, which has hand-tuned assembly routines. Inside a browser, modern JavaScript engines have a native `bigint` type which is also very performant. The Kotlin code when targeting "native" code rather than the JVM, uses Microsoft's HACL* [11]—a performant C implementation of a wide variety of cryptographic primitives which have been formally verified for correctness.

Fixed-base modular exponentiation. Most exponentiations in ElectionGuard have a fixed base, either the generator g or the election public key K. It is well-known [36, §14.6.3] that using pre-computed tables of certain powers of these bases makes encryption and proving operations substantially faster, as shown in the election context by Devillez et al. [22]. To compute g^x, one parses x in groups of k bits, e.g., for 8-bit groups, $g^x = g^{x_{0-7}} g^{(x_{8-15}) \ll 8} g^{(x_{16-23}) \ll 16} \cdots$. If x has ℓ bits, a precomputed table with all values $g^{2^{ki} \cdot j}$ for $0 \le i < \lceil \ell/k \rceil$, $0 \le j < 2^k$, turns the computation of g^x into a series of table lookups and multiplications. Using larger k leads to larger tables ($O(2^k)$ memory usage), but reduces the number of multiplications.

In the original Python implementation, 8-bit tables yielded a 5.1x performance improvement for encryption. 13-bit tables yielded a 7.4x improvement. 16-bit tables yielded an 8.6x improvement. (Measurements taken on a 2013 MacPro with a 3.5GHz Intel Xeon.) Verification operations do make some use of the generator g, but otherwise have variable bases, so we only see improvements of 1.6x-1.7x.

It is standard practice to use Montgomery multiplication [37] for implementing modular multiplication of large integers. In particular, entries in pre-computed tables should be directly provided in Montgomery form to avoid the costly conversion. HACL* provides the necessary primitives, giving a 2.5x performance improvement.

Using base-K encoded ElGamal encryption. As mentioned above in §2.6.2, using the public election key K as the base to encode voter selections for ElGamal encryption saves one exponentiation when encrypting the selection and generating the ZK proof of well-formedness. In our implementations, we observed a 13% speedup as expected.

Compact Chaum-Pedersen proofs. To keep memory requirements for the ZK proofs low, it is important to use the compact representation for proofs consisting of challenge and response values instead of including the commitments as well. Commitments have a large size of 512 bytes (4096-bit values), but they do not need to be stored as they can be recomputed from the challenge and response values, which only are 32 bytes in size each.

Side-channel attacks. Side-channel attacks on cryptographic operations are generally less of a threat for voting devices than with other applications. A poll site presents many more direct opportunities for an attacker to obtain a voter's selections. Nevertheless, implementations should be made side-channel resistant whenever practical.

4.2 Latency

Voter-perceived latency certainly needs to be considered, particularly if the voting device has a slow CPU. Luckily, beyond the optimizations discussed above, there are a variety of other options to hide this latency. For example, we also enable a precomputation approach since most of the computation for encrypting selections and generating the ZK proofs is independent of the voter's selections.

4.3 Compatibility

Every implementation of the ElectionGuard specification should be compatible with other implementations. The design specification only specifies the cryptographic operations in detail. In particular early on, the lack of a concrete implementation specification meant that implementation specifics were developed dynamically while several implementation efforts were underway. This naturally lead to challenges for agreeing on the specifics of data serialization into the JSON format. For example, the Python implementation initially used a base-64 encoding of cryptographic values into JSON strings, which was problematic for the C++ implementation. This lead to a less efficient hexadecimal encoding. Later on, the Kotlin implementation supported Google's Protocol Buffers, for an efficient binary representation, while the Rust code supported MongoDB's BSON (a binary variant of JSON).

The initial under-specification of how inputs to the cryptographic hash function should be serialized in the original version 1.0 specification has created unnecessary complications for achieving compatible implementations.

4.4 Scalability

The original definition of ElectionGuard specified a singular JSON election record. This was a problem for the VotingWorks implementation of VAULT [5], which was meant to read the tabular output of a plain paper ballot scanner and produce a public bulletin board with as many as a million encrypted ballots. Getting the necessary throughput required running on about 1000 CPU cores.

Fundamentally, ElectionGuard encryption is a CPU-bound operation. It is easy to express the encryption of a million ballots in any sort of data-parallel (e.g., MapReduce) paradigm, and as such, it is straightforward to scale the encryption throughput across multiple cores or multiple machines. The challenges come afterward. Before we had all the optimizations described above, each encrypted ballot was roughly 1MB of JSON data, so a million ballots required a terabyte of storage. On a cloud storage system like AWS S3, scalable read or write performance can only be achieved if your data is spread across multiple separate subdirectories, due to the way these systems do sharding. We ultimately settled on a design using one file per ballot, and building a Merkle tree into the directory structure, so any individual ballot can be verified without needing to read the full terabyte of data. This is what was used in the Inyo County pilot (see Section 6.2).

It will be necessary to allow for bulk storage and for efficient verification operations (e.g., voters, given receipts with the hashes of their ciphertext and knowledge of the root hash of the entire election, should be able to verify their ballots' correct representation in the election record without needing to read and process a terabyte of data).

5 In-Person Voting

While ElectionGuard has proven to be very flexible and usable in a wide variety of voting scenarios (e.g., in-person, mail-in, and Internet), the primary intent is for it to be used for in-person poll-site voting.

Even within the in-person setting, ElectionGuard can be used in many different ways: with hand-marked and/or machine-marked ballots, on ballot scanners, on ballot printers, or on ballot-marking devices. We describe in detail here a preferred scenario which matches the use of ElectionGuard in some of the deployments listed in section 6.

In an ideal setting, the ElectionGuard ballot encryption software can be housed on a ballot scanner located in a voting precinct. The use of precinct-based scanners has become common in the U.S. because of their reliability and flexibility (they can generally read either hand-marked or machine-marked ballots or a combination). Both the Idaho and Maryland deployments described in section 6 used the Hart Intercivic Verity ballot scanner which included both a display and a thermal printer.

Upon completing a paper ballot—either by hand or using a ballot-marking device—a voter feeds the paper ballot into the ballot scanner. After reading the contents of the ballot, the scanner displays its interpretation of the ballot contents and also calls the embedded ElectionGuard application with these same ballot contents. The ElectionGuard application encrypts these contents and returns a confirmation code which is printed for the voter. (N.b., the confirmation code is derived from the encryption of the voter selections and in no way reveals the selections themselves.) The voter collects the paper confirmation code and reviews the selections displayed on the screen. The voter may either indicate approval of the ballot—in which case the paper ballot is mechanically dropped into a bin beneath the scanner—or cancel the ballot—which causes the paper ballot to be returned to the voter. If the ballot is cancelled, the voter may make changes directly on the paper ballot or visit a poll-worker to obtain a fresh blank ballot or authorization to start again on a ballot-marking device.

Confirmation codes from cancelled ballots will be opened and serve to ensure correctness of the system as in [2]. The contents of cancelled ballots will be published in the election record together with additional data to enable verifiers to confirm that they match the given confirmation codes. It is also possible for voters to use personal apps from sources of their choosing together with supplemental data provided when a ballot is cancelled to instantly confirm that the confirmation code matches the voter's selections.[19]

6 Deployments and Other Use Cases

One of the principal benefits of the ElectionGuard toolkit model is its flexibility. This is shown by the variety of use cases seen in the deployments described in this section.

6.1 Wisconsin

The first use of ElectionGuard in a public election occurred in February of 2020 in the town of Fulton, Wisconsin. In Fulton, Microsoft partnered with VotingWorks[20] to deploy a system with five ballot-marking devices, a single ballot printer that was controlled by a laptop which also hosted the ElectionGuard vote encryption tools, and a ballot scanner.

Voters made their selections using one of the touchscreen ballot-marking devices and each voter's selections were written onto a smart card. Upon completing the selection process, a voter would take the smartcard to the printing station which would process the selections, call the ElectionGuard software, and print an ordinary ballot with the voter's selections and, on a separate yellow sheet, an ElectionGuard confirmation code. The voter would then have the option of either scanning the printed ballot or taking it to a poll worker to challenge the ballot and restart the voting process. In either case, the yellow sheets containing confirmation codes could be retained by the voters.

In Fulton, 398 voters cast ballots, and 4 ballots were challenged. After the ElectionGuard tallies were computed and released, a hand count of the paper ballots was initiated to confirm the results. The hand counted tally differed by 1 vote

[19] This instant verification feature was not available in recent deployments in Idaho and Maryland.

[20] https://www.voting.works/

from the ElectionGuard tally. But after a hand recount, it was discovered that one of the paper ballots had been put in the wrong pile, and once this was corrected, the results matched perfectly.

6.2 California

The next use of ElectionGuard in a public election was in November of 2020 in Inyo County, California [5]. But, unlike in Wisconsin, it was not part of the voting but instead part of the auditing process. California requires post-election audits of paper ballots to confirm that they are consistent with any machine counts that may have been done. The best modern audits, ballot-comparison *Risk-Limiting Audits (RLAs)* [35], typically begin by publishing the contents of every ballot and then perform random sampling to show that the published contents match the paper ballots.

The concern about ballot-comparison audits is that publication of ballot contents—even without any indication of the voters who cast each ballot—can compromise voter privacy.[21] The work in [9] shows how the tools used to achieve E2E-verifiability can also be used to conduct a ballot-comparison RLA without revealing raw ballot contents.

In Inyo County, California, ballot contents were encrypted by VotingWorks using ElectionGuard, and these encryptions were published instead of raw ballot contents. An election record was published to show that these encrypted ballots matched the announced tallies, and any ballot selected for auditing was decrypted and matched against the corresponding paper ballot.[22]

6.3 U.S. Congress

Shortly after the November 2020 U.S. general election, the respective political parties in the U.S. Senate and House of Representatives met to elect their leadership. At that time, the COVID-19 pandemic was a great concern, and discussions began between the U.S. House Democratic caucus and Microsoft about enabling leadership elections to be held remotely. While Congressional votes are public, the internal caucus elections are held by secret ballot, and ElectionGuard therefore provided a possible solution.

Internet voting poses many challenges, and although E2E-verifiability can mitigate many of these challenges, it does not offer a solution that is suitable for public elections [27] – it does nothing to protect the voting servers from large scale attacks for instance. Microsoft did a careful analysis of this scenario and concluded that for various reasons—including the small number of voters, the opportunity for direct voter education, the secondary channel available to each voter to confirm vote receipt, and, most importantly, the centrally managed mobile phones held by each voter—that this was an appropriate use of the ElectionGuard technology and partnered with Markup[23] to build a customized solution for the U.S. House Democratic caucus.[24]

House members downloaded a custom application onto their managed mobile devices which they used to make their selections of candidates for offices such as Speaker, Majority Leader, and Whip. Upon making their selections, each House member received a confirmation code and could choose to either cast the ballot or challenge it to verify that it reflected the selections that were made. After each round of voting, the lists of candidates were shortened and a new vote was conducted until a single winner was determined for each leadership position. The record of each round of voting was made available internally to House members and their staffs to enable verification.

6.4 Idaho

Starting in 2021, Microsoft partnered with election vendor Hart InterCivic[25] to integrate ElectionGuard into their voting equipment. The first public use of the Hart equipment with ElectionGuard took place in Preston, Idaho in the general election of November of 2022. Although the design was unchanged, numerous efficiency improvements were made to accommodate the Hart requirements, and new capabilities were added to meet their particular needs. MITRE Corporation[26] built its own independent verifier of the election record and took it to Preston to perform an immediate verification of the election record.

Voters had the choice to use ballot marking devices (which print paper ballots marked according to voter selections) or to fill out paper ballots by hand. Voters then had the option to insert their paper ballots into a traditional ballot scanner or a ballot scanner enhanced with ElectionGuard which provided a confirmation code for each vote on thermal paper. In either case, voters had an opportunity to review the scanners' interpretations of the selections on their ballots and could choose at that time to challenge their ballots and have the paper ballot returned. Voters could take challenged ballots to a poll worker and restart the voting process. In all, 111 voters in Preston chose to cast their votes on the ballot scanner enhanced with ElectionGuard.

[21] On a ballot with many contests and questions (as is typical in California), a voter can be instructed to make a very specific set of selections which is likely to be unique amongst all ballots cast within a precinct or jurisdiction. If the contents of each ballot are published, a coercer or vote-buyer can inspect the list of published ballot contents to ensure that a ballot matching the specific instructions is present.

[22] It is important to note that the small number of ballots that were decrypted and revealed do not enable coercion since a coerced voter can simply assert that the instructed ballot contents did not appear amongst the set of published raw votes because that ballot was, evidently, not amongst the audited ballots.

[23] https://www.markuplabs.com/

[24] This tool was also offered to the House Republican caucus and both Senate caucuses.

[25] https://www.hartintercivic.com/

[26] https://www.mitre.org/

6.5 Utah

Utah statute has allowed some voters to return ballots electronically since 2006. In the November 2023 general election, Enhanced Voting[27] used ElectionGuard to add integrity to this process in a Congressional special election and several local elections. Ballots that were electronically transmitted were also encrypted and tallied with ElectionGuard to verify that the tally had not been altered. 514 ballots across the state were processed with ElectionGuard.

6.6 Maryland

In November of 2023, ElectionGuard was used for municipal elections in College Park Maryland. Hart conducted this election with similar equipment to that used in Preston, Idaho. Although by this time, most, but not all, of the ElectionGuard version 2 features had been implemented. MITRE corporation revised their independent verifier to match the new features of this version and again ran their verifier immediately after the election. All 1468 voters in this election had their votes encrypted with ElectionGuard.

6.7 Neuilly-sur-Seine

Since late 2021, ElectionGuard has been used by Electis[28] for civic voting in the Paris suburb of Neuilly-sur-Seine.[29] This is a blockchain-based voting system of a kind explicitly recommended against by sources like the U.S. National Academies report.[30] However, ElectionGuard is open-source and may be used by anyone without payment or permission, and its use here demonstrates its broad applicability.

6.8 Concordium

Concordium[31] is a blockchain company based in Switzerland and Denmark. Its June 2024 internal elections[32] used its own Rust implementation of ElectionGuard. Here, the fact that an independent company did its own implementation demonstrates both the applicability and appeal of the ElectionGuard design.

6.9 Lessons

User surveys were conduced at the end of several ElectionGuard deployments.[33] These deployments happened in very different contexts, and were often accompanied by other important changes in the elections, like the introduction of electronic voting. As a result, it is hard to draw strong lessons, and more surveys in similar settings are needed.

At this stage, surveys showed that voters and poll workers expressed a strong increase in confidence in the accuracy of the election results when using ElectionGuard. In the eyes of technical experts observing the deployments, the key management by the guardians appears to be a central aspect that should motivate further research: the current deployments had guardians essentially blindly follow instructions provided by the election organizers, using software and hardware provided by these same organizers. Better ways of achieving independence in practice are needed.

7 Conclusions

As can be seen from the wide variety of scenarios described above, ElectionGuard has proven to be a very versatile design. When used in its primary application, ElectionGuard gives voters the ability to confirm on their own that their votes have been accurately counted—whether cast in-person, by mail, or online—without having to trust anyone or anything out of their control. ElectionGuard can also be used to protect voter privacy in post-election risk-limiting audits.

ElectionGuard also introduces new cryptographic approaches of independent interest that make ElectionGuard a flexible, efficient, and easy to use toolkit with numerous applications.

Acknowledgements

A great many people and organizations have contributed to the design, development, and deployment of ElectionGuard since its inception by Microsoft six years ago. There is not space here to properly thank everyone, but many are noted on the ElectionGuard website at https://www.electionguard.vote and in the ElectionGuard specification at https://www.electionguard.vote/spec/.

The authors do want to call out a few individuals whose help has been invaluable to this paper including John Caron, Henri Devillez, Moses Liskov, Arash Mirzaei, Thomas Peters, John Ramsdell, Marsh Ray, and Vanessa Teague.

The authors would also like to express deep thanks to the anonymous reviewers who made many valuable suggestions which greatly improved the quality of this work.

[27] https://www.enhancedvoting.com/

[28] https://electis.com

[29] https://xtz.news/adoption/the-neuilly-sur-seine-municipal-youth-council-elections-have-been-completed-using-the-tezos-based-voting-application-electis/

[30] https://nap.nationalacademies.org/catalog/25120/securing-the-vote-protecting-american-democracy

[31] https://www.concordium.com/

[32] https://medium.com/@concordium/concordiums-on-chain-voting-protocol-40d4aaafa880

[33] See, e.g., https://www.electionguard.vote/images/EAC%20Report%20Final.pdf

References

[1] Ben Adida, Olivier de Marneffe, Olivier Pereira, and Jean-Jacques Quisquater. Electing a university president using open-audit voting: Analysis of real-world use of Helios. In *2009 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '09*. USENIX Association, 2009.

[2] Josh Benaloh. Simple verifiable elections. In *2006 Electronic Voting Technology Workshop, EVT '06*. USENIX Association, 2006.

[3] Josh Benaloh. STROBE-Voting: Send Two, Receive One Ballot Encoding. In *Electronic Voting - 6th International Joint Conference, E-Vote-ID 2021*, volume 12900 of *LNCS*, pages 33–46. Springer, 2021.

[4] Josh Benaloh, Michael D. Byrne, Bryce Eakin, Philip T. Kortum, Neal McBurnett, Olivier Pereira, Philip B. Stark, Dan S. Wallach, Gail Fisher, Julian Montoya, Michelle Parker, and Michael Winn. STAR-Vote: A secure, transparent, auditable, and reliable voting system. In *2013 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '13*. USENIX Association, 2013.

[5] Josh Benaloh, Kammi Foote, Philip B. Stark, Vanessa Teague, and Dan S. Wallach. VAULT-style risk-limiting audits and the Inyo County pilot. *IEEE Security & Privacy*, 19(04):8–18, July 2021.

[6] Josh Benaloh, Michael Naehrig, and Olivier Pereira. ElectionGuard design specification version 2.1.0. https://www.electionguard.vote/spec/, June 2024.

[7] Josh Benaloh, Michael Naehrig, and Olivier Pereira. REACTIVE: Rethinking Effective Approaches Concerning Trustees in Verifiable Elections. https://eprint.iacr.org/2024/915, June 2024.

[8] Josh Benaloh, Ronald Rivest, Peter Y.A. Ryan, Philip Stark, Vanessa Teague, and Poorvi Vora. End-to-end verifiability. https://arxiv.org/abs/1504.03778, April 2015.

[9] Josh Benaloh, Philip Stark, and Vanessa Teague. VAULT: Verifiable Audits Using Limited Transparency. In *Electronic Voting - 4th International Joint Conference, E-Vote-ID 2019*, 2019.

[10] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. In *Advances in Cryptology - ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 626–643. Springer, 2012.

[11] Karthikeyan Bhargavan, Benjamin Beurdouche, Jean-Karim Zinzindohoué, and Jonathan Protzenko. HACL*: A verified modern cryptographic library. *ACM CCS*, September 2017.

[12] Lennart Braun, Ivan Damgård, and Claudio Orlandi. Secure multiparty computation from threshold encryption based on class groups. In *Advances in Cryptology - CRYPTO 2023*, volume 14081 of *LNCS*, pages 613–645. Springer, 2023.

[13] Craig Burton, Chris Culnane, and Steve A. Schneider. vVote: Verifiable electronic voting in practice. *IEEE Secur. Priv.*, 14(4):64–73, 2016.

[14] Ignacio Cascudo and Bernardo David. SCRAPE: scalable randomness attested by public entities. In *Applied Cryptography and Network Security - 15th International Conference, ACNS 2017*, volume 10355 of *LNCS*, pages 537–556. Springer, 2017.

[15] Nicholas Chang-Fong and Aleksander Essex. The cloudier side of cryptographic end-to-end verifiable voting: a security analysis of Helios. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016*, pages 324–335. ACM, 2016.

[16] David Chaum and Torben P. Pedersen. Wallet databases with observers. In *Advances in Cryptology - CRYPTO 1992*, volume 740 of *LNCS*, pages 89–105. Springer, 1992.

[17] Josh D. (Benaloh) Cohen and Michael J. Fischer. A robust and verifiable cryptographically secure election scheme. In *26th Annual Symposium on Foundations of Computer Science*. IEEE, 1985.

[18] Véronique Cortier, Pierrick Gaudry, and Stéphane Glondu. Belenios: A simple private and verifiable electronic voting system. In *Foundations of Security, Protocols, and Equational Reasoning - Essays Dedicated to Catherine A. Meadows*, volume 11565 of *LNCS*, pages 214–238. Springer, 2019.

[19] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Advances in Cryptology - CRYPTO 1994*, volume 839 of *LNCS*, pages 174–187. Springer, 1994.

[20] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Advances in Cryptology - EUROCRYPT 1997*, volume 1233 of *LNCS*, pages 103–118. Springer, 1997.

[21] Elizabeth C. Crites, Chelsea Komlo, and Mary Maller. Fully adaptive Schnorr threshold signatures. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023*, volume 14081 of *LNCS*, pages 678–709. Springer, 2023.

[22] Henri Devillez, Olivier Pereira, and Thomas Peters. How to verifiably encrypt many bits for an election? In *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security*, volume 13555 of *LNCS*, pages 653–671. Springer, 2022.

[23] Yevgeniy Dodis, Thomas Ristenpart, John P. Steinberger, and Stefano Tessaro. To hash or not to hash again? (In)differentiability results for H^2 and HMAC. In *Advances in Cryptology - CRYPTO 2012*, volume 7417 of *LNCS*, pages 348–366. Springer, 2012.

[24] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *2019 IEEE Symposium on Security and Privacy, SP 2019*, pages 1051–1066. IEEE, 2019.

[25] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science*, pages 427–437. IEEE Computer Society, 1987.

[26] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology - CRYPTO '86*, volume 263 of *LNCS*, pages 186–194. Springer, 1986.

[27] U.S. Vote Foundation. The future of voting: End-to-end verifiable internet voting - specification and feasibility study. https://www.usvotefoundation.org/E2E-VIV, July 2015.

[28] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptol.*, 20(1):51–83, 2007.

[29] Daniel M. Gordon. Discrete logarithms in *GF(P)* using the number field sieve. *SIAM J. Discret. Math.*, 6(1):124–138, 1993.

[30] Lucca Hirschi, Lara Schmid, and David A. Basin. Fixing the achilles heel of e-voting: The bulletin board. In *34th IEEE Computer Security Foundations Symposium, CSF 2021*, pages 1–17. IEEE, 2021.

[31] Stanislaw Jarecki and Anna Lysyanskaya. Adaptively secure threshold cryptography: Introducing concurrency, removing erasures. In *Advances in Cryptology - EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 221–242. Springer, 2000.

[32] Chelsea Komlo and Ian Goldberg. FROST: flexible round-optimized Schnorr threshold signatures. In *Selected Areas in Cryptography - SAC 2020*, volume 12804 of *LNCS*, pages 34–65. Springer, 2020.

[33] Benoît Libert, Marc Joye, and Moti Yung. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Theor. Comput. Sci.*, 645:1–24, 2016.

[34] Yehuda Lindell. Simple three-round multiparty Schnorr signing with full simulatability. *IACR Cryptol. ePrint Arch.*, page 374, 2022.

[35] Mark Lindeman and Philip B. Stark. A gentle introduction to risk-limiting audits. *IEEE Security and Privacy*, 10(5):42–49, 2012.

[36] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[37] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, (170):519–521, 1985.

[38] Torben P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract). In *Advances in Cryptology - EUROCRYPT 1991*, volume 547 of *LNCS*, pages 522–526. Springer, 1991.

[39] Kazue Sako and Joe Kilian. Receipt-free mix-type voting scheme - A practical solution to the implementation of a voting booth. In *Advances in Cryptology - EUROCRYPT '95*, volume 921 of *LNCS*, pages 393–403. Springer, 1995.

[40] Daniel Sandler, Kyle Derr, and Dan S. Wallach. VoteBox: A tamper-evident, verifiable electronic voting system. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium*, pages 349–364. USENIX Association, 2008.

[41] Daniel Sandler and Dan S. Wallach. Casting votes in the auditorium. In Ray Martinez and David A. Wagner, editors, *2007 USENIX/ACCURATE Electronic Voting Technology Workshop, EVT'07, Boston, MA, USA, August 6, 2007*. USENIX Association, 2007.

[42] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Advances in Cryptology - CRYPTO '89*, volume 435 of *LNCS*, pages 239–252. Springer, 1989.

[43] Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *J. Cryptol.*, 15(2):75–96, 2002.