ElectionGuard

Design Specification

Version 2.1.0

August 12, 2024

Josh Benaloh, Michael Naehrig, and Olivier Pereira
Microsoft Research

Contents

1 Introduction

This document describes the cryptographic details of the design of ElectionGuard, which can be used in conjunction with many new and existing voting systems to enable both end-to-end (E2E) verifiability and privacy-enhanced risk-limiting audits (RLAs). ElectionGuard is not a complete election system. It instead provides components that are designed to be flexible and to promote innovation by election officials and system developers. When properly used, it can promote voter confidence by empowering voters to independently verify the accuracy of election results.

End-to-End (E2E) Verifiability

An E2E-verifiable election provides artifacts which allow voters to confirm that their votes have been accurately recorded and counted. Specifically, an election is End-to-end (E2E) verifiable if two properties are achieved.

1. Individual voters can verify that their votes have been accurately recorded.
2. Voters and observers can verify that all recorded votes have been accurately counted.

An E2E-verifiable election does not guarantee that the recorded votes have been cast by legitimate voters: this needs to be ensured through the traditional voter identification mechanisms that are already deployed in elections.

An E2E-verifiable tally can be used as the primary tally in an election or as a verifiable secondary tally alongside traditional methods. ElectionGuard is compatible with in-person voting—either using an electronic ballot-marking device or an optical scanner capable of reading hand-marked or machine-marked ballots, with voting by mail, and even with Internet voting.[1]

Risk-Limiting Audits (RLAs)

RLAs offer election administrators efficient methods to validate reported election tallies against physical ballot records. There are several varieties of RLAs, but the most efficient and practical are *ballot-comparison audits*, in which electronic *cast-vote records* (CVRs) are individually compared against physical ballots.

The challenge with ballot-comparison audits is that public release of the full set of CVRs can compromise voter privacy while an audit without public disclosure of CVRs offers no basis for public confidence in the outcome. ElectionGuard can bridge this gap by enabling public disclosure of encrypted ballots that can be matched directly to physical ballots selected for auditing and can also be proven to match the reported tallies.

Making an election E2E-verifiable, in addition to conducting an RLA, can offer guarantees that the physical ballot records used in the RLA are those produced by the voters.

[1]Note that there are many challenges to responsible Internet voting that are mitigated but not fully solved by E2E-verifiability. The 2015 U.S. Vote Foundation report at https://www.usvotefoundation.org/E2E-VIV details many of these issues, and the 2018 National Academies report at https://nap.nationalacademies.org/catalog/25120/securing-the-vote-protecting-american-democracy includes a section on Internet voting (pp. 101–105). These conclusions were reaffirmed by a 2022 study conducted by Berkeley's Goldman School of Public Policy available at https://gspp.berkeley.edu/index.php/research-and-impact/news/recent-news/csp-working-group-findings-on-mobile-voting.

About this Specification

This specification can be used by expert reviewers to evaluate the details of the ElectionGuard process and by independent parties to write ElectionGuard implementations. This document is not intended to be a fully detailed implementation specification. It does not specify serialization and data structures for the election record or mappings between the notation used here and the corresponding data types and components of a specific implementation. However, this document, together with a detailed implementation specification or a well-documented ElectionGuard implementation, can be used by independent parties to write ElectionGuard verifiers to confirm the consistency of election artifacts with announced election results.

2 Overview

This section gives a very brief and high-level overview of the ElectionGuard system's functionality and how it can be used during an election or in a post-election audit such as an RLA.

To use ElectionGuard in an election, a set of *guardians* is enlisted to serve as trustees who manage cryptographic keys. The members of a canvassing board can serve as guardians.[2] Prior to the commencement of voting or auditing, the guardians work together to form a public encryption key that will be used to encrypt individual ballots.

After the conclusion of voting or auditing, a *quorum* of guardians is necessary to produce the artifacts required to enable public verification of the tally.

Prior to, throughout, and after an election (or audit) using ElectionGuard, an *election administrator* facilitates the ElectionGuard protocols and procedures and is responsible for populating and publishing the artifacts of an election in the form of the *election record*.

Key Generation

Prior to the start of voting (for an E2E-verifiable election) or auditing (for an RLA), the election guardians participate in a process wherein they generate public keys to be used in the election. Each guardian generates its own public-secret key pairs. One set of guardian public keys will be combined to form a single public key with which votes are encrypted, another set will be combined to form a second public key to encrypt other ballot data. Each guardian also generates an additional communication key pair used to communicate privately with other guardians to exchange information about secret keys so that the election record can be produced after voting or auditing is complete—even if not all guardians are available at that time.

The key generation ceremony begins with each guardian publishing its public keys together with proofs of knowledge of the associated secret keys. Once all public keys are published, each guardian uses each other guardian's public communication key to encrypt shares of its own secret keys. Finally, each guardian decrypts the shares it receives from other guardians and checks them for

[2]It might also be possible to use trusted hardware to perform the duties of guardians. This is explored in https://eprint.iacr.org/2024/915. It is important to note that the role of guardians is to protect confidentiality of votes. Compromised guardians—whether instantiated as humans or hardware—cannot compromise the integrity of the election tallies.

consistency. If the received shares verify, the receiving guardian announces its completion. If any guardian notices any irregularities such as a failed share verification, a failed proof verification, or an inconsistent set of keys, the guardian complains and halts the protocol. If an investigation identifies a misbehaving guardian, it is removed and the protocol restarted from scratch. It is the responsibility of the guardians together with the election administrator to conclude the key generation.

Ballot Encryption

In most uses, the election system makes a single call to the ElectionGuard API after each voter completes the process of making selections or with each ballot to be encrypted for an RLA. ElectionGuard will encrypt the selections made by the voter and return a confirmation code which the system should give to the voter.[3]

This is the only point where an existing election system must interface with ElectionGuard. In most uses of ElectionGuard, voters will have an opportunity to challenge their encrypted ballots and view their decryptions to ensure that the encryptions are correct.[4] In certain vote-by-mail scenarios and when ElectionGuard is used within an RLA, cast-vote records can be provided in batch without any interface between the voting equipment and ElectionGuard.

The encrypted ballots are published along with non-interactive zero-knowledge (NIZK) proofs of their well-formedness. The proofs assert that an encrypted ballot is well-formed, which means that it is a legitimate ballot and adheres to the limits imposed on selection options and contests. For example, they prove that a selection did not receive more votes than allowed and that no more than the allowed number of votes were received across the selection options in each contest. The encryption method used herein has a homomorphic property which allows the encrypted ballots to be combined into a single aggregate ballot which consists of encryptions of the election tallies.

Verifiable Decryption

In the final step, election guardians each independently use a share of the secret decryption key to jointly decrypt the election tallies and generate associated verification data. It is not necessary for all guardians to be available to complete this step. If some guardians are missing, a quorum of guardians is sufficient to complete decryption and generate the verification data. The number of guardians and the quorum required to decrypt are parameters fixed at the time of key generation.

Independent Verification

Observers can use this open specification and/or accompanying materials to write independent *election verifiers* that can confirm the well-formedness of each encrypted ballot, the correct aggregation of these ballots, and the accurate decryption of election tallies.

[3]The confirmation code is not necessary for RLA usage.

[4]A ballot that has been decrypted should be regarded as a test ballot and should not be included in an election tally. After a ballot is challenged, the voter should have an opportunity to cast a fresh ballot. In an E2E-verifiable election, a decrypted ballot should never be cast. However, in an RLA, some anonymized cast ballots may ultimately be challenged and decrypted.

The verification of an election consists of various verification steps that can be separated into three groupings.

Verification of Key Generation. The first group of verification steps pertain to the key generation process. The privacy of the votes depends on the guardians performing these verification steps, and guardians must verify the key generation before any voting operations start.

Under normal circumstances, guardians will engage in a key generation ceremony, produce the election public keys, and verify the generation without controversy; and there is no need for independent verification of this process. Still, independent verification can help to resolve conflicts should one or more parties to the key generation ceremony complain about the process.

These steps should therefore be considered to be optional for an independent verifier. The integrity of an election record can be fully verified by independent parties without any of the key verification steps. However, a "premium" verifier may wish to include these steps to verify the correctness of guardian actions during the key generation ceremony.

The key generation verification steps are highlighted in orange boxes in this document.

Verification of Ballot Correctness. As will be described in detail below, there are instances in both the E2E-verifiability application and the RLA application where it is desirable to verifiably decrypt individual ballots. In the former application, this allows voters to confirm that their selections have been correctly recorded within an encrypted ballot, and in the latter application, this allows encrypted electronic ballots that have been selected for audit to be compared with their associated paper ballots.

The steps required to verify the correct decryption of a ballot are grouped together to allow independent verifiers to be easily constructed whose sole purpose is to allow voters or observers to directly verify correct decryption of individual ballots.

The ballot correctness verification steps are highlighted in blue boxes.

Verification of the Election Record. The remaining verification steps apply to the election as a whole. These steps enable independent verification that every ballot in an election record is well-formed, that cast ballots have been correctly aggregated homorphically, and that these aggregated values have been correctly decrypted to produce election tallies.

It is critical that a complete verification of an election record also includes verification of the correct decryption of any individual ballots that have been decrypted as part of the process. A complete election verifier must therefore incorporate an individual ballot verifier as described above.

The election record verification steps are highlighted in green boxes.

Using this Specification

The principal purposes of this document are to specify the functionality of the ElectionGuard toolkit and to provide details necessary for independent parties to write election verifiers that consume the artifacts produced by the toolkit.

3 Components

This section describes the four principal components of ElectionGuard.

1. *Parameter Requirements:* These are properties required of parameters to securely and efficiently instantiate the cryptographic components of ElectionGuard. These *cryptographic parameters* are fixed ahead of every election and the same parameters can be used across multiple elections. A specific, recommended set of standard parameters is provided. In addition, there are properties required of the *election parameters* that define the election contests, selectable options, and ballot styles, among other information.
2. *Key Generation:* Prior to each individual election, guardians must generate individual public-secret key pairs and exchange shares of secret keys to enable completion of an election even if some guardians become unavailable. Although it is preferred to generate new keys for each election, it is permissible to use the same keys for a small number of elections so long as the set of guardians remains the same. A complete new set of keys must be generated if even a single guardian is replaced. Secret key material that is not in use anymore must be destroyed.
3. *Ballot Encryption:* While encrypting the contents of a ballot is a relatively simple operation, most of the work of ElectionGuard is the process of creating externally-verifiable artifacts to prove that each encrypted ballot is well-formed (i.e., its decryption is a legitimate ballot without overvotes or improper values).
4. *Verifiable Decryption:* At the conclusion of each election, guardians use their secret key shares to jointly produce election tallies together with verifiable artifacts that prove that the tallies are correct.

Notation

In the remainder of this specification, the following notation will be used.

- $\mathbb{Z} = \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$ is the set of integers.
- $\mathbb{Z}_n = \{0, 1, 2, \dots, n-1\}$ is the ring of integers modulo n for some positive integer n.
- $\mathbb{Z}_n^*$ is the multiplicative subgroup of $\mathbb{Z}_n$ that consists of all invertible elements modulo n. When p is a prime, $\mathbb{Z}_p^* = \{1, 2, 3, \dots, p-1\}$.
- $\mathbb{Z}_p^r$ is the set of r^{th} powers in $\mathbb{Z}_p^*$. Formally, $\mathbb{Z}_p^r = \{y \in \mathbb{Z}_p^*$ for which there exists $x \in \mathbb{Z}_p^*$ such that $y = x^r \bmod p\}$. When p is a prime for which $p - 1 = qr$, then $\mathbb{Z}_p^r$ is an order-q cyclic subgroup of $\mathbb{Z}_p^*$ and for each $y \in \mathbb{Z}_p^*$, $y \in \mathbb{Z}_p^r$ if and only if $y^q \bmod p = 1$.
- $x \equiv_n y$ is the predicate that is true if and only if $(x \bmod n) = (y \bmod n)$.
- The function HMAC(,) shall be used to denote the HMAC-SHA-256 keyed Hash Message Authentication Code (as defined in NIST PUB FIPS 198-1[5]) instantiated with SHA-256 (as defined in NIST PUB FIPS 180-4[6]). HMAC takes as input a key k and a message m of arbitrary length and returns a bit string $\mathrm{HMAC}(k, m)$ of length 256 bits.
- The ElectionGuard hash function $H(\, , \,)$ is instantiated with HMAC and thus has two inputs, both given as byte arrays. The first input to H is used to bind hash values to a specific election. The second is a byte array of arbitrary length and consists of domain separation

[5]NIST (2008) The Keyed-Hash Message Authentication Code (HMAC). In: FIPS 198-1. https://csrc.nist.gov/publications/detail/fips/198/1/final

[6]NIST (2015) Secure Hash Standard (SHS). In: FIPS 180-4. https://csrc.nist.gov/publications/detail/fips/180/4/final

bytes and the data being hashed. H outputs a digest of 256 bits, which is interpreted as a byte array of length 32. The detailed specification for H is given in Section 5.

- The symbol $\oplus$ denotes bitwise XOR.
- The symbol $\|$ denotes concatenation.
- In general, the variable pairs (α, β), (a, b), and (A, B) will be used to denote encryptions.[7] Specifically, (α, β) will be used to designate encryptions of votes (usually an encryption of a zero or a one), (A, B) will be used to denote aggregations of encryptions—which may be encryptions of larger values, and (a, b) will be used to denote encryption commitments used to prove properties of other encryptions.

Encryption of Votes

Encryption of votes in ElectionGuard is performed using a public key encryption method suggested by Devillez, Pereira, and Peters,[8] which is a variant exponential form of the ElGamal cryptosystem,[9] which is, in turn, a static form of the widely-used Diffie-Hellman(-Merkle) key exchange.[10] This encryption method is called *DPP vote encryption* or simply *vote encryption* in this document and rests on precisely the same security basis as Diffie-Hellman(-Merkle) key exchange—which is used to protect the confidentiality of the vast majority of Internet traffic.

Primes p and q are publicly fixed such that $qr = p - 1$. A generator g of the order q subgroup $\mathbb{Z}_p^r$ is also fixed. (Any $g = x^r \bmod p$ for which $x \in \mathbb{Z}_p^*$ suffices so long as $g \neq 1$.)

A public-secret key pair can be chosen by selecting a random $s \in \mathbb{Z}_q$ as a secret key and publishing $K = g^s \bmod p$ as the corresponding public key.[11]

A message $m \in \mathbb{Z}_q$ is then encrypted by selecting a random nonce $\xi \in \mathbb{Z}_q$ and forming the pair $(\alpha, \beta) = (g^\xi \bmod p, K^m \cdot K^\xi \bmod p) = (g^\xi \bmod p, K^{m+\xi} \bmod p)$. An encryption (α, β) can be decrypted by the holder of the secret s as $\beta/\alpha^s \bmod p = K^m \bmod p$ because

$$\frac{\beta}{\alpha^s} \equiv_p \frac{K^{m+\xi}}{(g^\xi)^s} \equiv_p \frac{K^m \cdot K^\xi}{(g^\xi)^s} \equiv_p \frac{K^m \cdot (g^s)^\xi}{(g^\xi)^s} \equiv_p \frac{K^m \cdot g^{\xi s}}{g^{\xi s}} \equiv_p K^m. \qquad (1)$$

The value of m can be computed from $K^m \bmod p$ as long as the message m is limited to a small, known set of options.[12]

[7]Note that as described below, the encryption method used herein requires a pair of values to represent each individual encryption.

[8]Devillez H., Pereira, O., Peters, T. (2022) *How to Verifiably Encrypt Many Bits for an Election?* in ESORICS 2022 Lecture Notes in Computer Science, vol 13555. Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-031-17146-8_32

[9]ElGamal T. (1985) *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*. In: Blakley G.R., Chaum D. (eds) Advances in Cryptology. CRYPTO 1984. Lecture Notes in Computer Science, vol 196. Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/3-540-39568-7_2.pdf

[10]Diffie W., Hellman, M. (1976) *New Directions in Cryptography* IEEE Transactions on Information Theory, vol 22

[11]As will be seen below, the actual public key used to encrypt votes will be a combination of separately generated public keys. So, no entity will ever be in possession of a secret key that can be used to decrypt votes.

[12]The simplest way to compute m from $K^m \bmod p$ is an exhaustive search through possible values of m. Alternatively, a table pairing each possible value of $K^m \bmod p$ with m can be pre-computed. A final option which can accommodate a larger space of possible values for m is to use Shanks's baby-step giant-step method as described in the 1971 paper *Class Number, a Theory of Factorization and Genera*, Proceedings of Symposium in Pure Mathematics, Vol. 20, American Mathematical Society, Providence, 1971, pp. 415-440.

The value of K^m and therefore m can also be computed from the encryption nonce ξ, namely via $\beta/K^\xi \equiv_p K^m$. The encryption nonce must therefore be securely protected. While the secret key s allows decryption of any ciphertext encrypted to the public key K, the encryption nonce only allows decryption of the specific ciphertext it was used to generate. Release of an encryption nonce can, when appropriate, serve as a fast and convenient method of verifiable decryption.

Usually, only two possible votes are encrypted in this way by ElectionGuard. An encryption of one is used to indicate that an option is selected, and an encryption of zero is used to indicate that an option is not selected. For some voting methods such as cumulative voting, Borda count, and a variety of cardinal voting methods like score voting and STAR-voting, it can be necessary to encrypt other small, positive integers.

Homomorphic Properties

A fundamental quality of the DPP vote encryption is its additively homomorphic property. If two messages m_1 and m_2 are respectively encrypted as $(A_1, B_1) = (g^{\xi_1} \bmod p, K^{m_1+\xi_1} \bmod p)$ and $(A_2, B_2) = (g^{\xi_2} \bmod p, K^{m_2+\xi_2} \bmod p)$, then the component-wise product

$$(A, B) = (A_1 A_2 \bmod p, B_1 B_2 \bmod p) = (g^{\xi_1+\xi_2} \bmod p, K^{(m_1+m_2)+(\xi_1+\xi_2)} \bmod p) \qquad (2)$$

is an encryption of the sum $m_1 + m_2$ using the nonce $\xi_1 + \xi_2$.[13]

This additively homomorphic property is used in two important ways in ElectionGuard. First, all of the encryptions of a single option across ballots can be multiplied to form an encryption of the sum of the individual values. Since the individual values are one (or some other integer for certain voting methods) on ballots that select that option and zero otherwise, the sum is the tally of votes for that option and the product of the individual encryptions is an encryption of the tally.

The other use is to sum all of the selections made in a single contest on a single ballot. In the simplest case, after demonstrating that each option is an encryption of either zero or one, the product of the encryptions indicates the number of options that are encryptions of one, and this can be used to show that no more ones than permitted are among the encrypted options—i.e. that no more options were selected than permitted. When larger integers are allowed, i.e. an option can receive multiple votes or weighted votes, the product of the ciphertexts then encrypts the total number of votes or the sum of weights and is used in the same way to ensure only the permitted number of votes or permitted sum of weights were used.

However, as will be described below, it is possible for a holder of a nonce ξ to prove to a third party that a pair (α, β) is an encryption of m without revealing the nonce ξ and without access to the secret s.

Non-Interactive Zero-Knowledge (NIZK) Proofs

ElectionGuard provides numerous proofs about encryption keys, encrypted ballots, and election tallies using the following four techniques.

[13] There is an implicit assumption here that $(m_1 + m_2) < q$, which is easily satisfied when m_1 and m_2 are both small. If $(\xi_1 + \xi_2) \geq q$, $(\xi_1 + \xi_2) \bmod q$ may be substituted without changing the equation since $g^q \equiv_p 1$.

1. A Schnorr proof[14] allows the holder of a secret key s to interactively prove possession of s without revealing s.
2. A Chaum-Pedersen proof[15] allows an encryption to be interactively proven to decrypt to a particular value without revealing the nonce used for encryption or the secret decryption key s. (This proof can be constructed with access to either the nonce used for encryption or the secret decryption key.)
3. The Cramer-Damgård-Schoenmakers technique[16] enables a disjunction to be interactively proven without revealing which disjunct is true.
4. The Fiat-Shamir heuristic[17] allows interactive proofs to be converted into non-interactive proofs.

Using a combination of the above techniques, it is possible for ElectionGuard to demonstrate that keys are properly chosen, that ballots are well-formed, and that decryptions match claimed values.[18]

Threshold Encryption

Threshold encryption is used for encryption of ballots and other data.[19] This form of encryption makes it very easy to combine individual guardian public keys into a single public key. The threshold encryption mechanism used in ElectionGuard also offers a homomorphic property that allows individual encrypted votes to be combined to form encrypted tallies.

The guardians of an election will each generate three public-secret key pairs. One of the public keys for each guardian will then be combined (as described in the following section) into a single vote encryption public key which is used to encrypt all selections made by voters in the election. The public keys of the second key pairs will be combined in the same way to form a single encryption key for other ballot data. The third key pairs will be used by guardians to privately communicate with each other.

At the conclusion of the election, each guardian will compute a verifiable partial decryption of each tally. These partial decryptions will then be combined to form full verifiable decryptions of the election tallies.

To accommodate the possibility that one or more of the guardians will not be available at the

[14]Schnorr C.P. (1990) Efficient Identification and Signatures for Smart Cards. In: Brassard G. (eds) Advances in Cryptology — CRYPTO' 89 Proceedings. CRYPTO 1989. Lecture Notes in Computer Science, vol 435. Springer, New York, NY. https://link.springer.com/content/pdf/10.1007%2F0-387-34805-0_22.pdf

[15]Chaum D., Pedersen T.P. (1993) *Wallet Databases with Observers*. In: Brickell E.F. (eds) Advances in Cryptology — CRYPTO' 92. CRYPTO 1992. Lecture Notes in Computer Science, vol 740. Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007%2F3-540-48071-4_7.pdf

[16]Cramer R., Damgård I., Schoenmakers B. (1994) *Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols*. In: Desmedt Y.G. (eds) Advances in Cryptology — CRYPTO' 94. CRYPTO 1994. Lecture Notes in Computer Science, vol 839. Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007%2F3-540-48658-5_19.pdf

[17]Fiat A., Shamir A. (1987) *How To Prove Yourself: Practical Solutions to Identification and Signature Problems*. In: Odlyzko A.M. (eds) Advances in Cryptology — CRYPTO' 86. CRYPTO 1986. Lecture Notes in Computer Science, vol 263. Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007%2F3-540-47721-7_12.pdf

[18]For all proof variants, ElectionGuard uses a compact representation that omits the commitments. Proofs consist of the challenge and response values only. When verifying proofs, the verification equations can be used to recompute the commitments and check their correctness via the challenge hash computation.

[19]Shoup, V., Gennaro, R. (2002) *Securing Threshold Cryptosystems against Chosen Ciphertext Attack*. In: J. Cryptology, vol 15. https://link.springer.com/content/pdf/10.1007/s00145-001-0020-9.pdf

conclusion of the election to form their partial decryptions, the guardians will cryptographically share[20] their secret keys amongst each other during key generation in a manner to be detailed in Section 3.2. Each guardian can then compute a share of the secret decryption key, which it uses to form the partial decryptions. A pre-determined threshold quorum value k out of the n guardians will be necessary to produce a full decryption.

Encryption of Other Data

ElectionGuard provides means to encrypt data other than votes that do not need to be homomorphically aggregated. Such data may include write-in information that needs to be attached to an encrypted selection and is encrypted to the public key used for other ballot data, or other auxiliary data attached to an encrypted ballot, either encrypted to this public key or to an administrative public key. The non-vote data do not need to be homomorphically encrypted and can use a more standard form of public-key encryption removing the data size restrictions imposed by the vote encryption. ElectionGuard encrypts such data with signed hashed ElGamal encryption, which employs a key derivation function (KDF) to generate a key stream that is then XORed with the data.[21] To implement the KDF, encryption makes use of the hash-based message authentication code HMAC. In ElectionGuard, HMAC is instantiated as HMAC-SHA-256 with the hash function SHA-256. Encryption also produces a Schnorr proof of knowledge of the random nonce used for the ElGamal encryption.

Guardians also need confidential communication channels between each other, for example to securely communicate the cryptographic shares of a guardian's secret key. This is the purpose of the third key pair generated by each guardian. Each guardian shares the public portion of its communication key with the other guardians.

3.1 Parameter Requirements

ElectionGuard uses integers to instantiate the encryption rather than elliptic curves in order to make construction of election verifiers as simple as possible without requiring special tools and dependencies.[22] The encryption scheme used to encrypt votes is defined by a prime p and a prime q which divides $(p-1)$. We use $r = (p-1)/q$ to denote the cofactor of q, and a generator g of the order q subgroup $\mathbb{Z}_p^r$ is fixed. We also require $r/2$ to be prime. The specific values for a 4096-bit prime p and a 256-bit prime q which divides $(p-1)$ and a generator g that define all standard baseline parameters are given below.

[20]Shamir A. (1979) *How to Share a Secret.* Communications of the ACM, vol 22. https://dl.acm.org/doi/10.1145/359168.359176

[21]Zheng Y., Seberry J. (1993) *Immunizing public key cryptosystems against chosen ciphertext attacks.* In: IEEE Journal on Selected Areas in Communications, vol 11. https://ieeexplore.ieee.org/document/223871

[22]Elliptic curves are commonly used in cryptographic applications because they are regarded as achieving the same security for lower cost. But they add complexity that ElectionGuard seeks to avoid.

3.1.1 Standard Baseline Cryptographic Parameters

Standard parameters for ElectionGuard begin with the largest 256-bit prime $q = 2^{256} - 189$. The (big endian) hexadecimal representation of q is as follows.

$$q = \texttt{0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFF43} \quad (3)$$

The modulus p is then set to be a 4096-bit prime with the following properties.

1. The first 256 bits of p are all ones.
2. The last 256 bits of p are all ones.
3. $p - 1$ is a multiple of q.
4. $(p - 1)/2q$ is also prime.

The middle 3584 bits of p are chosen by starting with the first 3584 bits of the constant ln(2) (the natural logarithm of 2).[23] After pre-pending and appending 256 ones, p is determined by finding the smallest prime larger than this value that satisfies the above properties.

This works out to $p = 2^{4096} - 2^{3840} + 2^{256}(\lfloor 2^{3584}\ln(2)\rfloor + \delta) + (2^{256} - 1)$ where the value of δ is given by

$$\delta = 287975203778583638958138611533602521491887169409704874643524560756486080635197037903.$$[24]

The hexadecimal representation of p is as follows.

```
p  =  0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
        B17217F7 D1CF79AB C9E3B398 03F2F6AF 40F34326 7298B62D 8A0D175B 8BAAFA2B
        E7B87620 6DEBAC98 559552FB 4AFA1B10 ED2EAE35 C1382144 27573B29 1169B825
        3E96CA16 224AE8C5 1ACBDA11 317C387E B9EA9BC3 B136603B 256FA0EC 7657F74B
        72CE87B1 9D6548CA F5DFA6BD 38303248 655FA187 2F20E3A2 DA2D97C5 0F3FD5C6
        07F4CA11 FB5BFB90 610D30F8 8FE551A2 EE569D6D FC1EFA15 7D2E23DE 1400B396
        17460775 DB8990E5 C943E732 B479CD33 CCCC4E65 9393514C 4C1A1E0B D1D6095D
        25669B33 3564A337 6A9C7F8A 5E148E82 074DB601 5CFE7AA3 0C480A54 17350D2C
        955D5179 B1E17B9D AE313CDB 6C606CB1 078F735D 1B2DB31B 5F50B518 5064C18B
        4D162DB3 B365853D 7598A195 1AE273EE 5570B6C6 8F969834 96D4E6D3 30AF889B
        44A02554 731CDC8E A17293D1 228A4EF9 8D6F5177 FBCF0755 268A5C1F 9538B982
        61AFFD44 6B1CA3CF 5E9222B8 8C66D3C5 422183ED C9942109 0BBB16FA F3D949F2
        36E02B20 CEE886B9 05C128D5 3D0BD2F9 62136319 6AF50302 0060E499 08391A0C
        57339BA2 BEBA7D05 2AC5B61C C4E9207C EF2F0CE2 D7373958 D7622658 90445744
        FB5F2DA4 B7510058 92D35689 0DEFE9CA D9B9D4B7 13E06162 A2D8FDD0 DF2FD608
        FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
```

The hexadecimal representation of the cofactor $r = (p - 1)/q$ is as follows.

[23] See https://oeis.org/A068426 for the integer sequence consisting of the bits of ln(2).

[24] Discovering this value δ required enumerating roughly 2.49 million values satisfying the first three of the above properties to find the first one for which both p and $(p - 1)/2q$ are prime.

```
r  =  0x01 00000000 00000000 00000000 00000000 00000000 00000000 00000000 000000BC
           B17217F7 D1CF79AB C9E3B398 03F2F6AF 40F34326 7298B62D 8A0D175B 8BAB857A
           E8F42816 5418806C 62B0EA36 355A3A73 E0C74198 5BF6A0E3 130179BF 2F0B43E3
           3AD86292 3861B8C9 F768C416 9519600B AD06093F 964B27E0 2D868312 31A9160D
           E48F4DA5 3D8AB5E6 9E386B69 4BEC1AE7 22D47579 249D5424 767C5C33 B9151E07
           C5C11D10 6AC446D3 30B47DB5 9D352E47 A53157DE 04461900 F6FE360D B897DF53
           16D87C94 AE71DAD0 BE84B647 C4BCF818 C23A2D4E BB53C702 A5C8062D 19F5E9B5
           033A94F7 FF732F54 12971286 9D97B8C9 6C412921 A9D86797 70F499A0 41C297CF
           F79D4C91 49EB6CAF 67B9EA3D C563D965 F3AAD137 7FF22DE9 C3E62068 DD0ED615
           1C37B4F7 4634C2BD 09DA912F D599F433 3A8D2CC0 05627DCA 37BAD43E 64A39631
           19C0BFE3 4810A21E E7CFC421 D53398CB C7A95B3B F585E5A0 4B790E2F E1FE9BC2
           64FDA810 9F6454A0 82F5EFB2 F37EA237 AA29DF32 0D6EA860 C41A9054 CCD24876
           C6253F66 7BFB0139 B5531FF3 01899612 02FD2B0D 55A75272 C7FD7334 3F7899BC
           A0B36A4C 470A64A0 09244C84 E77CEBC9 2417D5BB 13BF1816 7D8033EB 6C4DD787
           9FD4A7F5 29FD4A7F 529FD4A7 F529FD4A 7F529FD4 A7F529FD 4A7F529F D4A7F52A
```

Finally, the generator g is chosen to be $g = 2^r \bmod p$ and has the following hexadecimal representation.

```
g  =  0x36036FED 214F3B50 DC566D3A 312FE413 1FEE1C2B CE6D02EA 39B477AC 05F7F885
         F38CFE77 A7E45ACF 4029114C 4D7A9BFE 058BF2F9 95D2479D 3DDA618F FD910D3C
         4236AB2C FDD783A5 016F7465 CF59BBF4 5D24A22F 130F2D04 FE93B2D5 8BB9C1D1
         D27FC9A1 7D2AF49A 779F3FFB DCA22900 C14202EE 6C996160 34BE35CB CDD3E7BB
         7996ADFE 534B63CC A41E21FF 5DC778EB B1B86C53 BFBE9998 7D7AEA07 56237FB4
         0922139F 90A62F2A A8D9AD34 DFF799E3 3C857A64 68D001AC F3B681DB 87DC4242
         755E2AC5 A5027DB8 1984F033 C4D17837 1F273DBB 4FCEA1E6 28C23E52 759BC776
         5728035C EA26B44C 49A65666 889820A4 5C33DD37 EA4A1D00 CB62305C D541BE1E
         8A92685A 07012B1A 20A746C3 591A2DB3 815000D2 AACCFE43 DC49E828 C1ED7387
         466AFD8E 4BF19355 93B2A442 EEC271C5 0AD39F73 3797A1EA 11802A25 57916534
         662A6B7E 9A9E449A 24C8CFF8 09E79A4D 806EB681 119330E6 C57985E3 9B200B48
         93639FDF DEA49F76 AD1ACD99 7EBA1365 7541E79E C57437E5 04EDA9DD 01106151
         6C643FB3 0D6D58AF CCD28B73 FEDA29EC 12B01A5E B86399A5 93A9D5F4 50DE39CB
         92962C5E C6925348 DB54D128 FD99C14B 457F883E C20112A7 5A6A0581 D3D80A3B
         4EF09EC8 6F9552FF DA1653F1 33AA2534 983A6F31 B0EE4697 935A6B1E A2F75B85
         E7EBA151 BA486094 D68722B0 54633FEC 51CA3F29 B31E77E3 17B178B6 B9D8AE0F
```

Alternative parameter sets are possible and may be allowed in future versions of ElectionGuard.[25]

[25]If alternative parameters are allowed, election verifiers must confirm that p, q, r, and g are such that both p and q are prime (this may be done probabilistically using the Miller-Rabin algorithm), that $p - 1 = qr$ is satisfied, that $1 < g < p$, that $g^q \bmod p = 1$, and that generation of the parameters is consistent with the cited standard.

Note 3.1. As an example for alternative parameters, the Appendix provides a set of reduced parameters that offer better performance at a lower security level. It additionally provides various sets of very small and insecure parameters for testing purposes only. A good source for parameter generation is appendix A of FIPS 186-4.[26]Allowing alternate non-standard parameters would force election verifiers to recognize and check that parameters are correctly generated. Since these checks are very different from other checks that are required of a verifier, allowing such parameters would add substantial complexity to election verifiers. For this reason, this version of ElectionGuard fixes the parameters as above.

3.1.2 Parameter Base Hash

The prime modulus p, the subgroup order q, and the subgroup generator g are hashed together with the number n of guardians that participate in the election and the quorum value k using the ElectionGuard hash function H (as specified in detail in Section 5) to form the parameter base hash

$$\mathrm{H}_P = H(\mathtt{ver}; \mathtt{0x00}, p, q, g, n, k). \tag{4}$$

The symbol `ver` denotes the version byte array that encodes the used version of this specification. The array has length 32 and contains the UTF-8 encoding of the string "`v2.1.0`" followed by `0x00`-bytes, i.e. `ver` = `0x76322E312E30` $\parallel$ $\mathtt{b}(0, 26)$.[27] The `0x00`-byte at the beginning of the second argument of H is a domain separation byte (written in hexadecimal notation), see Section 5.1. This hash value binds together the basic cryptographic election parameters and is an input to all subsequent hash computations, either directly or indirectly, which binds all hash outputs to the baseline parameters and the version of the specification used for the election.

3.1.3 Election Parameters and the Election Manifest

Another parameter of an election is a public election manifest. The ElectionGuard manifest is a single file that describes the modalities of the election and must specify a wide range of data about it. The manifest file must contain data that makes it unique for each election. Most importantly, the manifest lists, among other information, all the contests in an election, the candidates (selectable options) for each contest together with associations between each option and its representation on a virtual ballot, and the contest selection limit—the number of options that a voter may select in that contest. It is assumed that each contest in the manifest has a unique label and that within each contest, each option also has a unique label. For instance, if Alice, Bob, and Carol are running for governor, and David and Ellen are running for senator, the election manifest file could enable the vector $\langle 0, 1, 0; 0, 1\rangle$ to be recognized as a ballot with votes for Bob as governor and Ellen as senator. This section defines how the manifest must uniquely specify the election parameters.

Labels. The election objects relevant for ElectionGuard such as contests, selectable options, and ballot styles are identified by unique labels. A *label* is a string, it should be a short, concise name

[26]NIST (2013) Digital Signature Standard (DSS). In: FIPS 186-4. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[27]The notation $\mathtt{b}(0, 26)$ denotes the concatenation of 26 `0x00` bytes, see Section 5 for details.

or identifier and should therefore not contain any long-form descriptions or line break characters, tabs, and similar special characters. Likewise, a label should not have leading or trailing whitespace characters.

The ElectionGuard manifest must contain a label that contains a unique and descriptive identifier for the election.

Indices. Instead of using the labels directly, ElectionGuard handles contests, selectable options, ballot styles, etc. in terms of their position in a unique ordered list. They can therefore be uniquely identified by an index value. An *index* is a 1-based ordinal in the range[28] $1 \leq i < 2^{31}$.

Contests and the contest index. The election manifest must contain a single ordered list of all the contests that can appear on any ballot generated for the election. Each contest must have a contest label $\lambda_{\mathcal{C}}$ that is unique within the election. The position of a given contest $\mathcal{C}$ in this list is the *contest index* $\mathtt{ind}_{\mathtt{c}}(\lambda_{\mathcal{C}})$. The contest list provides a bijective mapping between the unique contest labels and the corresponding contest index values. Note that the first contest in the list has contest index 1.

Selectable options and the option index. For each contest in the contest list, the election manifest must also contain a single ordered list of all the selectable options in this contest. Each selectable option $\mathcal{O}$ must have an option label $\lambda_{\mathcal{O}}$ that is unique within the contest. The position of a given selectable option $\mathcal{O}$ in this list is the *option index* $\mathtt{ind}_{\mathtt{o}}(\lambda_{\mathcal{O}})$. This option list provides a bijective mapping between the unique selectable option labels and the corresponding option index values. The first selectable option in the list has option index 1.

Ballot styles and the ballot style index. A *ballot style* is a single list of contest indices that defines which contests appear on a ballot generated according to this ballot style. If a contest index is contained in this list, the corresponding contest is present, otherwise it is not. The list is not ordered and only specifies whether a contest is present or not on a ballot of this ballot style. Each ballot style $\mathcal{S}$ must have a ballot style label $\lambda_{\mathcal{S}}$ that is unique within the election. The election manifest must contain a single, ordered list of all ballot styles that are possible in the election. The position of a given ballot style $\mathcal{S}$ in this list is the *ballot style index* $\mathtt{ind}_{\mathtt{s}}(\lambda_{\mathcal{S}})$. The ballot style list provides a bijective mapping between the unique ballot styles and the corresponding ballot style index values. Again, the first ballot style in the list has ballot style index 1.

Selections, option selection limits, and contest selection limits. A *selection* is the assignment of a value to an option by the voter. This could be the value 1, meaning that the voter selected this option, or the value 0, meaning that the voter did not select this option. To allow other voting methods, ElectionGuard allows a selection to be the assignment of a value in a range $\{0, 1, \ldots, R\}$, where R is the *option selection limit*, a positive integer that defines the maximal value allowed to be assigned to this option by the voter. For each contest, the election manifest must specify the option selection limit for the options in this contest, and must also specify a *contest selection limit*

[28]The upper bound of $2^{31} - 1$ is in consideration of languages and runtimes that do not have full support for unsigned integers.

L, which is the maximal total value for the sum of all selections made in that contest. In most elections, all of the option selection limits will be 1. Contest selection limits are usually 1 as well, although larger contest selection limits are not uncommon (e.g. "Choose up to 3").

Accompanying data fields. The election manifest offers optional data fields for accompanying data that can be attached to any of the above and to the manifest in general at the top level. They can be used to provide additional information on contests, selectable options, and ballot styles that go beyond the short unique labels described above. In addition, there can be data fields that describe general information about the election, jurisdiction, election device manufacturers, software versions, the date and location, etc. that help to interpret the accompanying data.

There are many other things that a manifest may specify. Some examples are listed here.

Undervotes. An undervote occurs in a contest if the number of selected options (or more generally, the total sum of selections assigned by the voter) is strictly less than the contest selection limit, the number (or total sum) of allowed selections a voter can make in that contest.

Counting undervoted contests. A manifest may specify whether the fact that a contest was undervoted should be recorded for public verification. If this is specified, the indicated contest should have an supplemental field that functions very much like a selectable option field.

This field is an undervote indicator field and is added to all ballots that contain this contest if the occurrence of an undervote is to be verifiably recorded. The field only indicates that an undervote has occurred on the specific ballot in this contest. A tally of this field will contain the total number of ballots that showed an undervote in this contest. The field does not record by how many votes the contest was undervoted, i.e., by how much the total number of selections by the voter is less than the contest selection limit. This number can be recorded in an undervote difference count field.

Undervote difference count. A manifest may specify whether the total number of undervotes, i.e., the difference between the number of selections the voter made and the contest selection limit (or in other words, the contest net undervote count) for each contest should be recorded for public verification. If this is specified, the indicated contest should have a supplemental field that functions very much like an option field but whose value is a count of the difference between the selection limit and the number of selections made by the voter on that ballot for that contest.

Overvotes. An overvote occurs in a contest if the voter selected more options than allowed, i.e., more than the contest selection limit for that contest specifies. A manifest may specify whether an overvoted contest should be recorded for public verification. If this is specified, the indicated contest should have a supplemental field whose value is set to one if the contest was overvoted on the ballot and to zero otherwise.

For some voting methods, it is possible that a voter marks an overvote for a single selection option, which means that the vote exceeds the option selection limit, while at the same time the contest selection limit is not exceeded. This case should be treated analogously to a contest overvote as described above.

Null votes. A null vote occurs in a contest if a voter does not select any option. As such, it is a special form of undervote. A manifest may specify whether the fact that a voter made a null vote in a contest should be counted.

If this is specified, the indicated contest should have a supplemental field that functions like the undervote and overvote indicator fields and indicates whether a null vote occurred.

Write-ins total. A manifest may specify whether the total number of write-ins selected in each contest is recorded for public verification. If this is specified, the indicated contest should have a single supplemental field that functions very much like an option field but whose value need not be limited to zero or one.

Any supplemental verifiable field functioning like an option field such as undervote or overvote indicator fields, undervote or overvote count fields, null vote fields, write-in fields, or write-in total count fields must be specified in the manifest per contest. In this specification, they are not discussed separately, but instead are treated like and listed with the option selection fields. Whenever this document lists option selection fields or their encryptions, it is assumed that these include all verifiable fields in that contest such as those defined above. Which of those fields are counted while ensuring adherence to the contest selection limit must also be specified in the manifest.

The data in the election manifest is written to a file `manifest` in a canonical representation that may be implementation specific.

3.1.4 Election Base Hash

The election manifest file `manifest` is hashed[29] using the ElectionGuard hash function with the parameter hash H_P to produce the *election base hash* H_B as

$$\mathrm{H}_B = H(\mathrm{H}_P; \mathtt{0x01}, \mathtt{manifest}). \tag{5}$$

Incorporating the byte array H_B into subsequent hash computations binds those hashes to the election parameters, the number of guardians, the threshold quorum value, and to a specific election by including the manifest.

[29] The file that constitutes the election manifest is input to H as an entire file. Again the `0x01` byte at the beginning of the second argument is a domain separation byte (in hex). In what follows, all uses of the function H include such domain separation bytes.

Verification 1 (Parameter validation)
An ElectionGuard election verifier must verify that it uses the correct version of the ElectionGuard specification, that it uses the standard baseline parameters, which may be hardcoded, and that the base hash values have been computed correctly.

(1.A) The ElectionGuard specification version used to generate the election record is the same as the ElectionGuard specification version used to verify the election record. This specification is version 2.1.0.
(1.B) The large prime is equal to the large modulus p defined in Section 3.1.1.
(1.C) The small prime is equal to the prime q defined in Section 3.1.1.
(1.D) The generator is equal to the generator g defined in Section 3.1.1.
(1.E) The parameter base hash has been computed correctly as

$$\mathrm{H}_P = H(\mathtt{ver}; \mathtt{0x00}, p, q, g, n, k)$$

using the version byte array $\mathtt{ver} = \mathtt{0x76322E312E30} \parallel \mathtt{b}(0, 26)$, which is the UTF-8 encoding of the version string "v2.1.0" padded with 0x00-bytes to length 32 bytes.
(1.F) The base hash has been computed correctly as

$$\mathrm{H}_B = H(\mathrm{H}_P; \mathtt{0x01}, \mathtt{manifest}).$$

3.2 Key Generation

Before an election, the number of guardians n is fixed together with a quorum value k that describes the number of guardians necessary to decrypt tallies and produce election verification data. The values n and k are integers subject to the constraint that $1 \leq k \leq n$. Canvassing board members can often serve the role of election guardians, and typical values for n and k could be 5 and 3—indicating that 3 of 5 guardians must cooperate to produce the artifacts that enable election verification. The reason for not setting the quorum value k too low is that it will also be possible for k guardians to jointly decrypt individual ballots.

Note 3.2. Decryption of individual ballots does not directly compromise voter privacy since links between encrypted ballots and the voters who cast them are not retained by the system. However, voters receive confirmation codes that can be associated with individual encrypted ballots, so any group that has the ability to decrypt individual ballots can also coerce voters by demanding to see their confirmation codes. Voters who decide to disclose or publish their confirmation codes in a non-anonymous way may also have their privacy impacted by unauthorized decryptions.

Threshold encryption is used for encryption of ballots. This form of encryption makes it very easy to combine individual guardian public keys into a single public key for encrypting votes and ballots. The encryption used herein also offers a homomorphic property that allows individual encrypted votes to be combined to form encrypted tallies.

The guardians of an election will each generate a public-secret key pair. The public keys will then be combined (as described in the following section) into a single vote encryption public key which is used to encrypt all selections made by voters in the election.

At the conclusion of the election, each guardian will compute a verifiable partial decryption of each tally. These partial decryptions will then be combined to form full verifiable decryptions of the election tallies.

To accommodate the possibility that one or more of the guardians will not be available at the conclusion of the election to form their partial decryptions, the guardians will cryptographically share[30] their secret keys amongst each other during key generation in a manner to be detailed in the next section. Each guardian will then compute a share of the secret decryption key, which it uses to form the partial decryptions. A pre-determined quorum value k out of the n guardians will be necessary to produce a full decryption. It is worth noting that fewer than k partial decryptions reveal nothing about the tally.

If the same set of n guardians supports multiple elections using the same threshold value k, the generated keys and key shares may be reused across a small number of elections, although it is preferred to generate new keys for each election.

3.2.1 Overview of Key Generation

The n guardians of an election are denoted by $G_1, G_2, \ldots, G_n$. Each guardian G_i generates an independent public-secret key pair by generating a random integer secret $s_i \in \mathbb{Z}_q$ and forming the public key $K_i = g^{s_i} \bmod p$. Each of these public keys will be published in the election record together with a non-interactive zero-knowledge Schnorr proof of knowledge of the associated secret key.

The joint vote encryption public key will be

$$K = \left(\prod_{i=1}^{n} K_i\right) \bmod p.$$

To enable robustness and allow for the possibility of missing guardians at the conclusion of an election, the ElectionGuard key generation includes a sharing of secret keys between guardians to enable decryption by any k guardians. This sharing is verifiable, so that receiving guardians can confirm that the shares they receive are correct; and the process allows for decryption without explicitly reconstructing secret keys of missing guardians.

Each guardian G_i generates $k-1$ random polynomial coefficients $a_{i,j}$ such that $0 < j < k$ and $a_{i,j} \in \mathbb{Z}_q$ and forms the polynomial

$$P_i(x) = \sum_{j=0}^{k-1} a_{i,j} x^j$$

by setting $a_{i,0}$ equal to its secret value s_i. Guardian G_i then publishes commitments $K_{i,j} = g^{a_{i,j}} \bmod p$ to each of its polynomial coefficients. As with the primary secret keys, each guardian should provide a Schnorr proof of knowledge of the secret coefficient value $a_{i,j}$ associated with each published commitment $K_{i,j}$. Since polynomial coefficients will be generated and managed in precisely the same fashion as secret keys, they will be treated together in a single step below.

Adding up all guardian polynomials $P_i(x)$ modulo q results in a polynomial that is a secret-sharing polynomial for the election secret key. This polynomial is denoted by $P(x)$ and computed

[30]Shamir A. (1979) *How to Share a Secret.* Communications of the ACM, vol 22.

as

$$P(x) = \left(\sum_{i=1}^{n} P_i(x)\right) \bmod q = \sum_{j=0}^{k-1} \big((a_{1,j} + a_{2,j} + \cdots + a_{n,j}) \bmod q\big)\ x^j. \tag{6}$$

The share of guardian G_i's secret key that is sent to guardian G_ℓ is $P_i(\ell) \bmod q$. Guardian G_i's share of the election secret key is the polynomial $P(x)$ evaluated at i, i.e., the sum $z_i = P(i) = (P_1(i)+\cdots+P_n(i)) \bmod q$ of all shares $P_j(i)$ that G_i receives from the other guardians G_j, $1 \leq j \leq n$, including its own $P_i(i)$.

At the end of the key generation process, each guardian must verify that the information that is being added to the election records is consistent with its own view of the key generation protocol execution. This verification step plays no role in the verifiability of the election but offers privacy protection against key substitution attacks by a malicious election administrator or network operator. At the conclusion of the election, individual encrypted ballots will be homomorphically combined into a single encrypted aggregate ballot—consisting of an encryption of the tally for each option offered to voters. Each guardian will use its share of the secret decryption key to generate a partial decryption of each encrypted tally value, and these partial decryptions will be combined into full decryptions. Any set of k partial decryptions suffices to compute a full decryption.

Decryption steps must only happen after verification that the ciphertexts that are decrypted are valid and independent of each other.

The above key generation process is run a second time to generate a second public key

$$\hat{K} = \left(\prod_{i=1}^{n} \hat{K}_i\right) \bmod p$$

as the product of a second set of guardian public keys $\hat{K}_i$. This second key generation can be done sequentially, or, concurrently with the first by duplicating each operation. The key $\hat{K}$ is used to encrypt additional data when homomorphic ciphertexts are not required, and is used in an encryption mode that supports efficient decryption of any data.

Because a guardian only uses its secret shares z_i and $\hat{z}_i$ of the joint secret keys for decryption, the actual guardian secret keys s_i and $\hat{s}_i$ are not needed any more and may be discarded once all shares have been successfully generated.

3.2.2 Details of Key Generation

Guardian coefficients and key pair for vote encryption. Guardians first need to generate a vote encryption public key and corresponding secret key shares. Each guardian G_i in an election with a decryption threshold of k generates k secret polynomial coefficients $a_{i,j}$, for $0 \leq j < k$, by sampling them uniformly, at random in $\mathbb{Z}_q$ and forms the polynomial

$$P_i(x) = \sum_{j=0}^{k-1} a_{i,j} x^j. \tag{7}$$

Guardian G_i then publishes commitments

$$K_{i,j} = g^{a_{i,j}} \bmod p \tag{8}$$

for each of its polynomial coefficients $a_{i,j}$, for $0 \leq j < k$. The constant term $a_{i,0}$ of this polynomial will serve as the secret key for guardian G_i, and for convenience we denote $s_i = a_{i,0}$, and its commitment $K_{i,0}$ will serve as the public key for guardian G_i and will also be denoted as $K_i = K_{i,0}$.[31]

Guardian coefficients and key pair for encrypting other ballot data. To generate keys that are used to encrypt other ballot data, the above process is run a second time, either in parallel or sequentially. Each guardian generates k coefficients $\hat{a}_{i,j}$, the corresponding polynomial $\hat{P}_i(x)$, and commitments $\hat{K}_{i,j}$ as above.

Additional key pair for sharing secret data between guardians. To share secret values amongst each other, guardians generate an additional ElGamal key pair that can be used for communication between guardians. Guardian G_i thus generates a uniformly random value $\zeta_i \in \mathbb{Z}_q$ as the secret communication key and computes the public communication key

$$\kappa_i = g^{\zeta_i} \bmod p. \tag{9}$$

In order to prove possession of the coefficient associated with each public commitment, for each $K_{i,j}$ and each $\hat{K}_{i,j}$ with $0 \leq j < k$, and to prove possession of the secret communication key, guardian G_i generates a Schnorr proof of knowledge for each of its coefficients and the secret communication key ζ_i. These proofs are grouped together for the coefficients $a_{i,j}$ and likewise for the coefficients $\hat{a}_{i,j}$ as follows.

NIZK proof: Guardian G_i proves knowledge of secrets $a_{i,j}$ such that $K_{i,j} = g^{a_{i,j}} \bmod p$ and of ζ_i such that $\kappa_i = g^{\zeta_i} \bmod p$.

For each $0 \leq j \leq k$, Guardian G_i generates a random integer value $u_{i,j}$ in $\mathbb{Z}_q$ and computes

$$h_{i,j} = g^{u_{i,j}} \bmod p. \tag{10}$$

Then, using the hash function H (as defined in Section 5 below), guardian G_i performs a single hash computation[32]

$$c_i = H_q(\mathrm{H}_P; \texttt{0x10}, \text{“}\texttt{pk\_vote}\text{”}, i, K_{i,0}, K_{i,1}, \ldots, K_{i,k-1}, \kappa_i, h_{i,0}, h_{i,1}, \ldots, h_{i,k-1}, h_{i,k}), \tag{11}$$

and publishes the proof $(c_i, v_{i,0}, v_{i,1}, \ldots, v_{i,k-1}, v_{i,k})$ consisting of the challenge value c_i and the response values $v_{i,j} = (u_{i,j} - c_i a_{i,j}) \bmod q$ $(0 \leq j < k)$ and $v_{i,k} = (u_{i,k} - c_i\zeta_i) \bmod q$ together with $(K_{i,0}, K_{i,1}, \ldots, K_{i,k-1})$ and κ_i. This proof is a compact version of the $k+1$ Schnorr proofs $(c_i, v_{i,j})$ for $0 \leq j \leq k$.

[31]Note that the detailed description here generates the secret guardian key s_i as one of the polynomial coefficients and does not treat it separately as indicated in the overview description.

[32]The hash function H_q hashes into $\mathbb{Z}_q$. In this version of the ElectionGuard specification, it is simply the hash function H concatenated with reduction modulo q, see Section 5 for details.

NIZK proof: Guardian G_i proves knowledge of secrets $\hat{a}_{i,j}$ such that $\hat{K}_{i,j} = g^{\hat{a}_{i,j}} \bmod p$ and of ζ_i such that $\kappa_i = g^{\zeta_i} \bmod p$.

Guardian G_i computes a proof for the coefficients $\hat{a}_{i,j}$ that is completely analogous to the above proof for the $a_{i,j}$, including the additional secret key ζ_i again. As above, for each $0 \leq j \leq k$, it generates random integers $\hat{u}_{i,j}$ in $\mathbb{Z}_q$ and computes

$$\hat{h}_{i,j} = g^{\hat{u}_{i,j}} \bmod p. \tag{12}$$

The challenge value is computed as

$$\hat{c}_i = H_q(\mathrm{H}_P; \texttt{0x10}, \text{“}\texttt{pk\_data}\text{”}, i, \hat{K}_{i,0}, \hat{K}_{i,1}, \ldots, \hat{K}_{i,k-1}, \kappa_i, \hat{h}_{i,0}, \hat{h}_{i,1}, \ldots, \hat{h}_{i,k-1}, \hat{h}_{i,k}), \tag{13}$$

which allows to publish the proof $(\hat{c}_i, \hat{v}_{i,0}, \hat{v}_{i,1}, \ldots, \hat{v}_{i,k-1}, \hat{v}_{i,k})$ consisting of the challenge value $\hat{c}_i$ and the response values $\hat{v}_{i,j} = (\hat{u}_{i,j} - \hat{c}_i \hat{a}_{i,j}) \bmod q$ $(0 \leq j < k)$ and $\hat{v}_{i,k} = (\hat{u}_{i,k} - \hat{c}_i \zeta_i) \bmod q$ together with $(\hat{K}_{i,0}, \hat{K}_{i,1}, \ldots, \hat{K}_{i,k-1})$ and κ_i.

Verification 2 (Guardian public-key validation)

For each guardian G_i, $1 \leq i \leq n$ an election verifier must compute the values

(2.1) $h_{i,j} = (g^{v_{i,j}} \cdot K_{i,j}^{c_i}) \bmod p$ for each $0 \leq j < k$,

(2.2) $h_{i,k} = (g^{v_{i,k}} \cdot \kappa_i^{c_i}) \bmod p$,

(2.3) $\hat{h}_{i,j} = (g^{\hat{v}_{i,j}} \cdot \hat{K}_{i,j}^{\hat{c}_i}) \bmod p$ for each $0 \leq j < k$,

(2.4) $\hat{h}_{i,k} = (g^{\hat{v}_{i,k}} \cdot \kappa_i^{\hat{c}_i}) \bmod p$,

and then must confirm the following.

(2.A) The values $K_{i,j}$ and $\hat{K}_{i,j}$, for $0 \leq j < k$, and κ_i are in $\mathbb{Z}_p^r$. (A value x is in $\mathbb{Z}_p^r$ if and only if x is an integer such that $0 \leq x < p$ and $x^q \bmod p = 1$ is satisfied.)

(2.B) The values $v_{i,j}$ and $\hat{v}_{i,j}$, for $0 \leq j \leq k$ are in $\mathbb{Z}_q$. (A value x is in $\mathbb{Z}_q$ if and only if x is an integer such that $0 \leq x < q$.)

(2.C) The challenge c_i is correctly computed as

$$c_i = H_q(\mathrm{H}_P; \texttt{0x10}, \text{“}\texttt{pk\_vote}\text{”}, i, K_{i,0}, K_{i,1}, \ldots, K_{i,k-1}, \kappa_i, h_{i,0}, \ldots, h_{i,k-1}, h_{i,k})$$

and the challenge $\hat{c}_i$ is correctly computed as

$$\hat{c}_i = H_q(\mathrm{H}_P; \texttt{0x10}, \text{“}\texttt{pk\_data}\text{”}, i, \hat{K}_{i,0}, \hat{K}_{i,1}, \ldots, \hat{K}_{i,k-1}, \kappa_i, \hat{h}_{i,0}, \ldots, \hat{h}_{i,k-1}, \hat{h}_{i,k}).$$

This verification box and some others below contain computation steps as well as verification conditions. The former are numbered with decimal numerals, while the latter are numbered with capital letters alphabetically.

It is worth noting here that for any fixed constant α, the value $g^{P_i(\alpha)} \bmod p$ (and similarly $g^{\hat{P}_i(\alpha)}$) can be computed entirely from the published commitments as

$$g^{P_i(\alpha)} \equiv_p g^{\sum_{j=0}^{k-1} a_{i,j} \alpha^j} \equiv_p \prod_{j=0}^{k-1} g^{a_{i,j} \alpha^j} \equiv_p \prod_{j=0}^{k-1} (g^{a_{i,j}})^{\alpha^j} \equiv_p \prod_{j=0}^{k-1} K_{i,j}^{\alpha^j}. \tag{14}$$

Note 3.3. Although this formula includes double exponentiation—raising a given value to the power α^j—in what follows, α and j will always be small values (bounded by n). The resulting exponents α^j will therefore be relatively small. Even for more general settings, the exponents can always be reduced to $\alpha^j \bmod q$ and therefore will never exceed q.

Share encryption. Each guardian G_i encrypts its shares for each other guardian G_ℓ ($1 \le \ell \le n$, $\ell \ne i$) using an encryption function E_ℓ that can be instantiated using that guardian's public communication key $\kappa_\ell = g^{\zeta_\ell} \bmod p$ as laid out in the section describing the encryption of other data above. This means that each guardian G_i publishes the encryption $E_\ell(P_i(\ell), \hat{P}_i(\ell))$ of its secret key shares $P_i(\ell)$ and $\hat{P}_i(\ell)$ for every other guardian G_ℓ as follows.

Guardian G_i uses guardian G_ℓ's public communication key κ_ℓ and a randomly-selected nonce $\xi_{i,\ell} \in \mathbb{Z}_q$ to compute

$$(\alpha_{i,\ell}, \beta_{i,\ell}) = (g^{\xi_{i,\ell}} \bmod p, \kappa_\ell^{\xi_{i,\ell}} \bmod p) \tag{15}$$

and the symmetric key

$$k_{i,\ell} = H(\mathrm{H}_P; \mathtt{0x11}, i, \ell, \kappa_\ell, \alpha_{i,\ell}, \beta_{i,\ell}). \tag{16}$$

Using the hash-based message authentication code HMAC instantiated with SHA-256, this key is used to derive[33] the encryption keys

$$k_{i,\ell,1} = \mathrm{HMAC}(k_{i,\ell}, \mathtt{0x01} \parallel \mathtt{Label} \parallel \mathtt{0x00} \parallel \mathtt{Context} \parallel \mathtt{0x0200}), \tag{17}$$

and

$$k_{i,\ell,2} = \mathrm{HMAC}(k_{i,\ell}, \mathtt{0x02} \parallel \mathtt{Label} \parallel \mathtt{0x00} \parallel \mathtt{Context} \parallel \mathtt{0x0200}), \tag{18}$$

both of which are 256 bits (32 bytes) long, and where $\mathtt{Label} = \mathtt{b}(\text{“}\mathtt{share\_enc\_keys}\text{”}, 14)$ and $\mathtt{Context} = \mathtt{b}(\text{“}\mathtt{share\_encrypt}\text{”}, 13) \parallel \mathtt{b}(i, 4) \parallel \mathtt{b}(\ell, 4)$.[34]

Since $P_i(\ell)$ and $\hat{P}_i(\ell)$ are integers modulo q, they can be encoded as byte arrays $\mathtt{b}(P_i(\ell), 32)$ and $\mathtt{b}(\hat{P}_i(\ell), 32)$ of exactly 32 bytes each as specified in Section 5.1.2. The encoded values are then encrypted as

$$E_\ell(P_i(\ell), \hat{P}_i(\ell)) = (C_{i,\ell,0}, C_{i,\ell,1}, C_{i,\ell,2}), \tag{19}$$

where

$$C_{i,\ell,0} = \alpha_{i,\ell} = g^{\xi_{i,\ell}} \bmod p, \tag{20}$$

$$C_{i,\ell,1} = (\mathtt{b}(P_i(\ell), 32) \oplus k_{i,\ell,1}) \parallel (\mathtt{b}(\hat{P}_i(\ell), 32) \oplus k_{i,\ell,2}), \tag{21}$$

and the third ciphertext component $C_{i,\ell,2} = (\bar{c}_{i,\ell}, \bar{v}_{i,\ell})$ is a Schnorr proof of knowledge of the encryption nonce $\xi_{i,\ell}$ computed as follows. Guardian G_i generates a uniform random $\bar{u}_{i,\ell} \in \mathbb{Z}_q$ and computes $\gamma_{i,\ell} = g^{\bar{u}_{i,\ell}} \bmod p$, then computes the challenge value

$$\bar{c}_{i,\ell} = H_q(H_P; \mathtt{0x12}, i, \ell, \gamma_{i,\ell}, C_{i,\ell,0}, C_{i,\ell,1}) \tag{22}$$

[33]NIST (2022) *Recommendation for Key Derivation Using Pseudorandom Functions.* In: SP 800-108r1 `https://csrc.nist.gov/pubs/sp/800/108/r1/upd1/final`.

[34]This key derivation uses the KDF in counter mode from SP 800-108r1. The second input to HMAC contains the counter in the first byte, the UTF-8 encoding of the string “`share_enc_keys`” as the *Label* (encoding is denoted by `b`(...), see Section 5.1.4), a separation `0x00` byte, the UTF-8 encoding of the string “`share_encrypt`” concatenated with encodings of the numbers i and ℓ of the sending and receiving guardians as the *Context*, and the final two bytes specifying the length of the output key material as 512 bits in total.

and the response $\bar{v}_{i,\ell} = (\bar{u}_{i,\ell} - \bar{c}_{i,\ell}\xi_{i,\ell}) \bmod q$.

Share decryption. After receiving the ciphertext $(C_{i,\ell,0}, C_{i,\ell,1}, C_{i,\ell,2})$ from G_i, guardian G_ℓ decrypts it by first verifying the Schnorr proof $C_{i,\ell,2} = (\bar{c}_{i,\ell}, \bar{v}_{i,\ell})$. Guardian G_ℓ first computes $\gamma_{i,\ell} = (g^{\bar{v}_{i,\ell}} \cdot C_{i,\ell,0}^{\bar{c}_{i,\ell}}) \bmod p$ and then verifies that $\bar{c}_{i,\ell} = H_q(H_P; \texttt{0x12}, i, \ell, \gamma_{i,\ell}, C_{i,\ell,0}, C_{i,\ell,1})$ according to Equation (22). If the Schnorr proof is valid, G_ℓ obtains $k_{i,\ell} = H(\mathrm{H}_P; \texttt{0x11}, i, \ell, \kappa_\ell, \alpha_{i,\ell}, \beta_{i,\ell})$ and computes the encryption keys $k_{i,\ell,1}$ and $k_{i,\ell,2}$ as above in Equations (17) and (18). The ciphertext can then be decrypted as

$$\mathtt{b}(P_i(\ell), 32) \parallel \mathtt{b}(\hat{P}_i(\ell), 32) = C_{i,\ell,1} \oplus (k_{i,\ell,1} \parallel k_{i,\ell,2}). \tag{23}$$

Secret key share computation. Each guardian G_i computes its share $z_i = P(i)$ of the vote encryption secret key, which is the value

$$z_i = P(i) = \left(\sum_{j=1}^{n} P_j(i)\right) \bmod q = (P_1(i) + P_2(i) + \cdots + P_n(i)) \bmod q. \tag{24}$$

for later use in decryption as will be described in Section 3.6.3. This requires computing its own share $P_i(i) \bmod q$. In the same way, guardian G_i computes its share $\hat{z}_i = \hat{P}(i)$ of the ballot data encryption secret key from the values $\hat{P}_\ell(i)$.

Vote encryption public key computation. The joint vote encryption public key K is computed as

$$K = \left(\prod_{i=1}^{n} K_i\right) \bmod p. \tag{25}$$

Ballot data encryption public key computation. The joint ballot data encryption public key $\hat{K}$ is computed as

$$\hat{K} = \left(\prod_{i=1}^{n} \hat{K}_i\right) \bmod p. \tag{26}$$

Verification 3 (Election public-key validation)
An election verifier must verify the correct computation of the joint vote encryption public key and the joint ballot data encryption public key.
(3.A) $K = (\prod_{i=1}^{n} K_i) \bmod p$.
(3.B) $\hat{K} = (\prod_{i=1}^{n} \hat{K}_i) \bmod p$.

Share verification and the guardian record. The election administrator produces a *preliminary* guardian record and makes it available to all guardians. This record contains the election parameters p, q, g, n, and k; the election public keys K and $\hat{K}$; all $K_{i,j}$ and $\hat{K}_{i,j}$ values from all guardians (for $1 \le i \le n$ and $0 \le j < k$), and the public communication keys κ_i, together with all Schnorr proofs $(c_i, v_{i,0}, v_{i,1}, \dots, v_{i,k})$ and $(\hat{c}_i, \hat{v}_{i,0}, \hat{v}_{i,1}, \dots, \hat{v}_{i,k})$.

Each guardian G_ℓ now performs the following verifications of the preliminary guardian record.

1. Guardian G_ℓ compares all key data in the preliminary guardian record with its own view of the data. This means G_ℓ checks whether the data that G_ℓ computed or obtained from other guardians is the same as the one displayed in the preliminary guardian record. In practice, this can be done by comparing a hash of all the key data as

$$\begin{aligned}\mathrm{H}_G \;=\; & H(\mathrm{H}_B; \texttt{0x13}, K, \hat{K}, \\ & K_{1,0}, K_{1,1}, \dots, K_{1,k-1}, K_{2,0}, K_{2,1}, \dots, K_{2,k-1}, \dots, K_{n,0}, K_{n,1}, \dots, K_{n,k-1}, \\ & \hat{K}_{1,0}, \hat{K}_{1,1}, \dots, \hat{K}_{1,k-1}, \hat{K}_{2,0}, \hat{K}_{2,1}, \dots, \hat{K}_{2,k-1}, \dots, \hat{K}_{n,0}, \hat{K}_{n,1}, \dots, \hat{K}_{n,k-1}, \\ & \kappa_1, \kappa_2, \dots, \kappa_n) \end{aligned} \tag{27}$$

   with the hash value computed in the same way from G_ℓ's own view of the record. Guardian G_ℓ also verifies the correctness of the base hash H_B by performing Verification 1.
2. Guardian G_ℓ verifies that all Schnorr proofs $(c_i, v_{i,j})$ and $(\hat{c}_i, \hat{v}_{i,j})$ are valid for $1 \le i \le n$ and $0 \le j \le k$. This means that guardian G_ℓ performs Verification 2.
3. Guardian G_ℓ verifies that the election public keys have been correctly computed from the guardian public keys as $K = (\prod_{i=1}^{n} K_i) \bmod p$ and $\hat{K} = (\prod_{i=1}^{n} \hat{K}_i) \bmod p$. This means that guardian G_ℓ performs Verification 3.
4. Guardian G_ℓ also verifies the validity of each share $P_i(\ell)$ against guardian G_i's commitments $K_{i,0}, K_{i,1}, \dots, K_{i,k-1}$ to its coefficients for all $1 \le i \le n$ by confirming that the following equation holds:

$$g^{P_i(\ell)} \bmod p = \left(\prod_{j=0}^{k-1} (K_{i,j})^{\ell^j} \right) \bmod p. \tag{28}$$

   The same equation is checked for the $\hat{P}_i(\ell)$ and the $\hat{K}_{i,j}$, i.e.,

$$g^{\hat{P}_i(\ell)} \bmod p = \left(\prod_{j=0}^{k-1} (\hat{K}_{i,j})^{\ell^j} \right) \bmod p. \tag{29}$$

If any of the above verification steps fails, guardian G_ℓ complains to the election administrator and all other guardians. This triggers an out-of-band investigation to identify the cause of the verification failure.

One possible way to investigate verification failures is to have all guardians release their secret information such that detailed checks can identify parties that provided false information or carried out erroneous computations. This does not necessarily allow identification of a misbehaving guardian, but might do so. At least, it may help to identify implementation bugs in the guardian or administrator software.

After possibly excluding or replacing a misbehaving guardian, the key generation procedure is started from scratch. It is the joint responsibility of the election administrator and the guardians to conclude key generation such that all guardians agree with the proposed preliminary election record.

Once all guardians confirm that all verification steps have passed successfully, the preliminary guardian record is validated and published as the guardian record as part of the election record. Guardians verify that the public keys K and $\hat{K}$ that they verified in the preliminary guardian record appear in the election record.

At this point, guardian G_i only needs to retain the secret values $z_i = P(i)$ and $\hat{z}_i = \hat{P}(i)$ that it needs for participating in the decryption process. All other secret values, in particular its initial secret keys s_i and $\hat{s}_i$ may be discarded.

3.2.3 Extended Base Hash

Once the baseline parameters and the election public keys have been produced and confirmed, the base hash H_B is hashed with the election public keys K and $\hat{K}$ to form an *extended base hash*

$$\mathrm{H}_E = H(\mathrm{H}_B; \mathtt{0x14}, K, \hat{K}) \tag{30}$$

that will form the basis of subsequent hash computations.

Verification 4 (Extended base hash validation)
An election verifier must verify the correct computation of the extended base hash.

(4.A) $\mathrm{H}_E = H(\mathrm{H}_B; \mathtt{0x14}, K, \hat{K})$.

3.3 Ballot Encryption

Although there are some exceptions, an ElectionGuard ballot is typically comprised entirely of encryptions of one (indicating selection made) and zero (indicating selection not made).

3.3.1 Selection Encryption

To encrypt a ballot entry σ, a random value $\xi \in \mathbb{Z}_q$ is selected, and σ is encrypted as

$$\mathrm{Enc}(\sigma, \xi) = (\alpha, \beta) = (g^{\xi} \bmod p,\ (K^{\sigma} \cdot K^{\xi}) \bmod p) = (g^{\xi} \bmod p,\ K^{\sigma+\xi} \bmod p). \tag{31}$$

Specifically, for the typical case of $\sigma \in \{0, 1\}$, one of the following two computations is performed.

- Zero (not selected, $\sigma = 0$) is encrypted as $\mathrm{Enc}(0, \xi) = (g^{\xi} \bmod p,\ K^{\xi} \bmod p)$.
- One (selected, $\sigma = 1$) is encrypted as $\mathrm{Enc}(1, \xi) = (g^{\xi} \bmod p,\ K^{1+\xi} \bmod p)$.

The notation $\mathrm{Enc}(\sigma, \xi)$ for this homomorphically additive encryption includes the encryption nonce ξ. In cases, where the nonce does not need to be referenced, $\mathrm{Enc}(\sigma)$ is used for clarity.

Note that if multiple encrypted votes $(g^{\xi_i} \bmod p,\ K^{\sigma_i+\xi_i} \bmod p)$ are formed, their component-wise product $(g^{\sum_i \xi_i} \bmod p,\ K^{\sum_i \sigma_i + \sum_i \xi_i} \bmod p)$ serves as an encryption of $\sum_i \sigma_i$—which is the tally of those votes.[35]

[35]The initial decryption actually forms the value $g^{\sum_i \sigma_i} \bmod p$. However, since $\sum_i \sigma_i$ is a relatively small value, it can be effectively computed from $g^{\sum_i \sigma_i} \bmod p$ by means of an exhaustive search or similar methods.

Some cardinal voting methods like cumulative voting, score voting, STAR-voting, and Borda count may require the encryption of small positive integers σ greater than 1 that represent multiple allowed votes or weighted votes. In all these cases, encryption is done as shown in Equation (31).

A ciphertext (α, β) as above can be decrypted using the encryption nonce ξ by computing $K^\sigma = \beta / K^\xi \bmod p$.

3.3.2 Selection Encryption Identifiers and Identifier Hash

For each ballot B, a 256-bit *selection encryption identifier* $\mathrm{id}_B \in \{0, \ldots, 2^{256}-1\}$ is chosen uniformly at random. This value is a unique identifier that is publicly released with the selection encryptions of the ballot and stored with the encrypted ballot in the election record. A *selection encryption identifier hash* H_I is computed as

$$\mathrm{H}_I = H(H_E; \mathtt{0x20}, \mathrm{id}_B) \tag{32}$$

and is used in all subsequent hash computations relating to the ballot B such as the generation of encryption nonces and challenge computation for zero-knowledge proofs.

Verification 5 (Uniqueness of selection encryption identifiers)
An election verifier must verify the following.

(5.A) There are no duplicate selection encryption identifiers, i.e., among the set of submitted (cast and challenged) ballots, no two have the same selection encryption identifiers.

For each ballot (cast and challenged), an election verifier must verify the following.

(5.B) The selection encryption identifier hash H_I has been correctly computed as

$$\mathrm{H}_I = H(\mathrm{H}_E; \mathtt{0x20}, \mathrm{id}_B).$$

3.3.3 Generation of the Ballot Nonce and Encryption Nonces

The "random" nonces used for vote encryption on a single ballot B are all derived from the selection encryption identifier hash H_I and a single, secret 256-bit *ballot nonce* $\xi_B \in \{0, \ldots, 2^{256} - 1\}$ that is generated uniformly at random. As detailed in Section 3.1.3, it is assumed that each contest Λ has a unique index $i = \mathtt{ind_c}(\Lambda)$ and that each possible option λ within a contest also has a unique index $j = \mathtt{ind_o}(\lambda)$. The nonce $\xi_{i,j}$ used to encrypt the j-th selection of the i-th contest is derived as

$$\xi_{i,j} = H_q(\mathrm{H}_I; \mathtt{0x21}, i, j, \xi_B). \tag{33}$$

Ballot nonces must be independent across different ballots, and only the nonces used to encrypt ballot selections need to be derived from the ballot nonce. The use of a single ballot nonce for each ballot allows the entire ballot encryption to be re-derived from the contents of a ballot, the selection encryption identifier, and the ballot nonce. It also allows the encrypted ballot to be fully decrypted with the single ballot nonce.

3.3.4 Encryption of Ballot Nonces

Since access to a ballot nonce ξ_B enables decryption of the ballot which it encrypted, ballot nonces must be protected carefully. Still, they can be very useful in several scenarios. For instance, challenged ballots and ballots that are selected in the context of a risk limiting audit need to be decrypted and it may be much more efficient to decrypt a single ballot nonce and release the derived encryption nonces $\xi_{i,j}$ than to verifiably decrypt every encrypted ballot selection on the ballot.[36] Some elections may also make use of offline ballot marking devices that print a ballot and a ballot confirmation code for the voter. In such cases, printing an encrypted ballot nonce on the ballot may offer an efficient solution enabling a scanner to recompute the encrypted ballot selections matching the confirmation code provided to the voter.

In order to accommodate such scenarios, every ElectionGuard ballot contains an encryption of the ballot nonce using the ballot data encryption public key $\hat{K}$. This encryption is computed using the same encryption mode that is used for encrypting guardian secret key shares. Concretely, a random nonce $\hat{\xi}_B \in \mathbb{Z}_q$ is selected and used to compute

$$(\alpha_B, \beta_B) = (g^{\hat{\xi}_B} \bmod p, \hat{K}^{\hat{\xi}_B} \bmod p) \tag{34}$$

and the 256-bit secret key

$$h = H(\mathrm{H}_I; \mathtt{0x22}, \alpha_B, \beta_B), \tag{35}$$

which is used to derive an encryption key k_1 by computing

$$k_1 = \mathrm{HMAC}(h, \mathtt{0x01} \parallel \mathtt{Label} \parallel \mathtt{0x00} \parallel \mathtt{Context} \parallel \mathtt{0x0100}), \tag{36}$$

which is 256 bits (32 bytes) long and where $\mathtt{Label} = \mathtt{b}(\text{“}\mathtt{ballot\_nonce}\text{”}, 12)$ and $\mathtt{Context} = \mathtt{b}(\text{“}\mathtt{ballot\_nonce\_encrypt}\text{”}, 20)$.

The ballot nonce ξ_B is then encrypted as $C_{\xi_B} = (C_{\xi_B,0}, C_{\xi_B,1}, C_{\xi_B,2})$ where

$$\begin{aligned} C_{\xi_B,0} &= \alpha_B = g^{\hat{\xi}_B} \bmod p, \\ C_{\xi_B,1} &= \mathtt{b}(\xi_B, 32) \oplus k_1. \end{aligned} \tag{37}$$

The third component $C_{\xi_B,2}$ is a Schnorr proof of knowledge for the encryption nonce $\hat{\xi}_B$ and is generated by selecting a uniform random integer $u_B \in \mathbb{Z}_q$, computing the commitment $a_B = g^{u_B} \bmod p$, the challenge

$$c_B = H_q(H_I; \mathtt{0x23}, a_B, C_{\xi_B,0}, C_{\xi_B,1}) \tag{38}$$

and the response value $v_B = (u_B - c_B\hat{\xi}_B) \bmod q$. The third component is $C_{\xi_B,2} = (c_B, v_B)$.

3.3.5 Ballot Well-Formedness

Contest and option selection limits. A contest in an election consists of a set of options together with a selection limit that indicates the number of selections that are allowed to be made

[36] ElectionGuard releases the encryption nonces $\xi_{i,j}$ rather than the ballot nonce ξ_B in order to offer the flexibility to decrypt only some of the encrypted ballot selections on a ballot, which can be useful to preserve privacy in the case of a risk limiting audit for instance. Furthermore, it may offer some (limited) protection against a weak PRG that would be used to generate the ballot nonces of multiple ballots.

in that contest and an option selection limit that indicates the maximal value that is allowed to be assigned to an individual option. In most elections, most contests have a selection limit of one and an option selection limit of one. However, a larger selection limit (e.g., select up to three) is not uncommon in some elections. Approval voting can be achieved by setting the selection limit to the total number of options in a contest. Ranked choice voting is not supported in this version of ElectionGuard, but it may be enabled in a future version.[37] Write-ins are assumed to be explicitly registered or allowed to be lumped into a single "write-ins" category for the purpose of verifiable tallying. Verifiable tallying of free-form write-ins may be best done with a mixnet[38] design.

Undervotes. A legitimate vote in a contest consists of a set of selections with cardinality not exceeding the selection limit of that contest. To accommodate legitimate undervotes, in previous versions of ElectionGuard (up to v1.1), the internal representation of a contest was augmented with "placeholder" options equal in number to the selection limit. Placeholder options were selected as necessary to force the total number of selections made in a contest to be equal to the selection limit. When the selection limit is one, for example, ElectionGuard used a single placeholder option that can be thought of as a "none of the above" option. The current version of ElectionGuard adopts a different approach to accomodate undervotes and allows the total number of selected options to lie in a range between zero and the selection limit. This has lower computational cost and leads to smaller election records.

Overvotes. When the number of selections made by the voter exceeds the contest selection limit or when the selection assigned to a single option in a contest exceeds its option selection limit, the votes in the contest become invalid as an overvote. To not affect the election tallies, all selectable options in the contest are set to zero (unselected). To record that an overvote has occurred and what specific selections were made by the voter in this contest, this information may be encoded into a contest data field, together with other data such as write-in text. This data is then encrypted with the ballot data encryption public key $\hat{K}$ using the hashed ElGamal encryption described in Section 3.3.10.

Ballot well-formedness. Two things must now be proven about the encryption of each vote to ensure a ballot is well-formed.

1. The encryption associated with each option is an encryption of a legitimate value not exceeding the option selection limit—usually either an encryption of zero or an encryption of one.
2. The sum of all encrypted selections in each contest lies in the range between 0 and the contest selection limit L for that contest (usually $L = 1$), i.e., it is one of the values $0, 1, \ldots, L$.

The use of DPP vote encryption enables efficient zero-knowledge proofs of these requirements, and the Fiat-Shamir heuristic can be used to make these proofs non-interactive. Chaum-Pedersen proofs are used to demonstrate that an encryption is that of a specified value, and these are combined with the Cramer-Damgård-Schoenmakers technique to show that an encryption is that of one of a specified set of values—usually that a value is an encryption of either zero or one. The encryptions

[37]Benaloh J., Moran. T, Naish L., Ramchen K., and Teague V. (2009) *Shuffle-Sum: Coercion-Resistant Verifiable Tallying for STV Voting* in Transactions of Information Forensics and Security.

[38]Chaum D. (1981) *Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms* Communications of the ACM.

of selections in a contest are homomorphically combined, and the result is shown to be an encryption of a non-negative value that is at most the contest's selection limit—again using Chaum-Pedersen proofs combined with the Cramer-Damgård-Schoenmakers technique.[39]

Cardinal voting methods. A range proof, i.e., a proof that a ciphertext is an encryption of one of the values $0, 1, \ldots, R$ is a generalization of the proof that a ciphertext is an encryption of zero or one. It can be used to prove well-formedness of an individual selection when it is allowed for options to receive multiple votes in a contest by a single voter. Using range proofs with a range up to a certain option selection limit for the individual option as well as the contest selection sum therefore enables cardinal voting methods such as cumulative voting, score voting, STAR-voting, and Borda count in ElectionGuard.

3.3.6 Outline for Proofs of Ballot Correctness

This section provides the outlines for different types of encryption proofs that aim to give an understanding of how they work. It starts with the proof that a given ciphertext is an encryption of the value 0. Based on that, it explains how it can be proved that a ciphertext is an encryption of the value 1. It then combines those to arrive at the base case used in ElectionGuard, namely that a ciphertext is an encryption of 0 or 1. Finally, it shows how the latter is generalized to generate range proofs, i.e., proofs that a ciphertext encrypts a value in a range $0, 1, \ldots, R$.

NIZK proof that (α, β) is an encryption of zero. To prove that ciphertext (α, β) is an encryption of zero, the Chaum-Pedersen protocol proceeds as follows. This proof assumes knowledge of the secret encryption nonce ξ such that $(\alpha, \beta) = (g^\xi \bmod p, K^\xi \bmod p)$. The prover selects a random value u in $\mathbb{Z}_q$ and commits by computing the pair $(a, b) = (g^u \bmod p,\ K^u \bmod p)$. A hash computation is then performed (using the Fiat-Shamir heuristic) to create a pseudo-random challenge value $c = H_q(\mathrm{H}_I; \alpha, \beta, a, b)$,[40] and the prover responds with $v = (u - c\xi) \bmod q$. A verifier can now confirm the claim by checking that both $g^v \cdot \alpha^c \equiv_p a$ and $K^v \cdot \beta^c \equiv_p b$ are true.

NIZK proof that (α, β) is an encryption of one. To prove that (α, β) is an encryption of one, $\beta/K \bmod p$ is substituted for β in the above. The verifier can be relieved of the need to perform a modular division by computing $\beta K^{q-1} \bmod p$ rather than $\beta/K \bmod p$. As an alternative, the verifier can confirm that $K^{(v-c) \bmod q} \cdot \beta^c \equiv_p b$ instead of $K^v \cdot (\beta/K)^c \equiv_p b$.[41]

As with many zero-knowledge protocols, if the prover knows a challenge value prior to making its commitment, it can create a false proof. For example, if a particular challenge c is known to be forthcoming, a prover can generate a random $v \in \mathbb{Z}_q$ and commit via $(a, b) = ((g^v \alpha^c) \bmod p,\ (K^v \beta^c) \bmod p)$. This selection will satisfy the required checks for (α, β) to appear as an encryption of zero regardless of the values of (α, β). Similarly, setting $(a, b) = ((g^v \alpha^c) \bmod p,\ (K^{(v-c) \bmod q} \beta^c) \bmod p)$ will satisfy the required checks for (α, β) to appear as an encryption of one regardless of the values

[39]For approval voting, when the selection limit is equal to the number of available options, the proof of satisfying the selection limit is not necessary because it is implied by the proofs for the individual selections.

[40]The actual proof would add to the hash inputs a domain separation byte and other elements of context, like the contest and option for which the ciphertext is computed. They are omitted here in favor of a simpler description.

[41]If it is more convenient, the verifier could instead check that β^c is congruent to $K^{c-v}b$.

of (α, β). This quirk is what enables the proof of a disjunction of two predicates as described in the next paragraph.

Sketch of NIZK proof that (α, β) is an encryption of zero or one. The Cramer-Damgard-Schoenmakers technique enables the proof of a disjunction by giving the prover one degree of freedom in proving two assertions. Instead of being required to answer two challenges exactly, the prover is constrained only by the sum of the two challenges it must answer. It can therefore freely choose one of the two challenges that it will answer.

More concretely, the prover will compute a challenge value c that it does not yet know, and it will have to answer challenges c_0 and c_1 subject to the constraint that $c = (c_0 + c_1) \bmod q$. The prover starts with whichever statement is *not* true, generates a random challenge for it, and then derives the commitment from it to be able to produce a false proof, as described in the previous paragraph. For example, if (α, β) is an encryption of one, the prover chooses a challenge c_0 and a response at random, then derives values for (a_0, b_0) so that the 'response' satisfies the 'challenge'; if (α, β) is an encryption of zero, the prover chooses c_1 and a response at random and derives satisfying values for (a_1, b_1). The prover then works on the true part. It generates a truthful commitment (a_0, b_0) like the first part of a proof of an encryption of zero (or a truthful commitment (a_1, b_1) like the first part of a proof of an encryption of one, whichever is true). A single challenge value c is selected by hashing all commitments and baseline parameters. The prover must produce challenge values c_0 and c_1 s.t. $c = (c_0 + c_1) \bmod q$. It answers its false claim with the challenge it chose in the beginning (c_0 if (α, β) is an encryption of one, c_1 if it is an encryption of zero), then derives the other challenge by subtraction ($c_1 = (c - c_0) \bmod q$ or $c_0 = (c - c_1) \bmod q$ as needed). It then finishes the proof of the true part as usual, using this challenge. An observer can see that one of the claims must be true but cannot tell which.

Sketch of NIZK proof that (α, β) is an encryption of an integer ℓ such that $0 \leq \ell \leq R$. The above method to prove that (α, β) is an encryption of zero or one can be extended to more than two values in an analogous way. Now, the prover needs to produce false proofs that verify for all values that are not encrypted in the ciphertext. This means that for each i with $0 \leq i \leq R$ and $i \neq \ell$, the prover selects a challenge value c_i and a response at random and uses them to create commitments (a_i, b_i) that make the response verify the challenge for the false claim that (α, β) is an encryption of the value $i \neq \ell$. For the true assertion that (α, β) is an encryption of ℓ, the prover makes honest commitments (a_ℓ, b_ℓ). At this point, there exist commitments (a_i, b_i) for all $0 \leq i \leq R$ and a single challenge value c is selected by hashing all these commitments. The remaining challenge value c_ℓ is then uniquely determined by the constraint $c = (c_0 + \cdots + c_\ell + \cdots + c_R) \bmod q$ and can be computed by subtracting from c all other challenge values that were chosen for the false proofs. The challenge c_ℓ must be used to answer the true assertion that (α, β) is an encryption of ℓ. The proof now consists of all challenge and response pairs, including the honestly generated one for the true assertion and R faked pairs for the false assertions.

3.3.7 Details for Proofs of Ballot Correctness

This subsection begins with the special but common case where a selection is either a 0 or a 1, which means that the range bound is $R = 1$. Both cases—unselected (encryption of 0) and selected (encryption of 1)—are spelled out in detail before the general case range proof is specified. An ElectionGuard implementation can directly implement the general range proof and use it in the special case $R = 1$. There is no need to implement the case $R = 1$ separately.

Unselected option. To encrypt an "unselected" option on a ballot, a pseudo-random nonce $\xi \in \mathbb{Z}_q$ for the selection is derived from the ballot nonce ξ_B as described in Section 3.3.3, and an encryption of zero is formed as $(\alpha, \beta) = (g^{\xi} \bmod p,\ K^{\xi} \bmod p)$.

NIZK proof: Proves that (α, β) is an encryption of zero or one.
(Requires knowledge of encryption nonce ξ for which (α, β) is an encryption of zero.)

To create the proof that (α, β) is an encryption of a zero or a one, randomly select u_0, u_1, and c_1 from $\mathbb{Z}_q$ and compute

$$(a_0, b_0) = (g^{u_0} \bmod p,\ K^{u_0} \bmod p) \tag{39}$$

and

$$(a_1, b_1) = (g^{u_1} \bmod p,\ K^{u_1 - c_1} \bmod p). \tag{40}$$

When computing a proof for option λ of contest Λ, a challenge value c is formed by hashing the identifier hash H_I together with $\mathtt{ind}_{\mathsf{c}}(\Lambda)$, $\mathtt{ind}_{\mathsf{o}}(\lambda)$, α, β, a_0, b_0, a_1, and b_1, namely

$$c = H_q(\mathrm{H}_I; \mathtt{0x24}, \mathtt{ind}_{\mathsf{c}}(\Lambda), \mathtt{ind}_{\mathsf{o}}(\lambda), \alpha, \beta, a_0, b_0, a_1, b_1). \tag{41}$$

The proof consists of the four values c_0, c_1, v_0, and v_1, where c_1 has been selected at random above and

$$c_0 = (c - c_1) \bmod q, \tag{42}$$
$$v_0 = (u_0 - c_0 \cdot \xi) \bmod q, \tag{43}$$
$$v_1 = (u_1 - c_1 \cdot \xi) \bmod q. \tag{44}$$

These values satisfy the proof equations as follows.

$$c = ((c - c_1) + c_1) \bmod q = (c_0 + c_1) \bmod q, \tag{45}$$
$$(g^{v_0} \alpha^{c_0}, K^{v_0} \beta^{c_0}) \equiv_p (g^{u_0 - c_0\xi} g^{c_0\xi}, K^{u_0 - c_0\xi} K^{c_0\xi}) \equiv_p (a_0, b_0), \tag{46}$$
$$(g^{v_1} \alpha^{c_1}, K^{v_1 - c_1} \beta^{c_1}) \equiv_p (g^{u_1 - c_1\xi} g^{c_1\xi}, K^{u_1 - c_1\xi - c_1} K^{c_1\xi}) \equiv_p (a_1, b_1). \tag{47}$$

Selected option. To encrypt a "selected" option on a ballot, a pseudo-random nonce $\xi \in \mathbb{Z}_q$ for the selection is derived from the ballot nonce ξ_B as described in Section 3.3.3, and an encryption of one is formed as $(\alpha, \beta) = (g^{\xi} \bmod p,\ K^{\xi+1} \bmod p)$.

NIZK proof: Proves that (α, β) is an encryption of zero or one.
(Requires knowledge of encryption nonce ξ for which (α, β) is an encryption of one.)

To create the proof that (α, β) is an encryption of a zero or a one, randomly select u_0, u_1, and c_0 from $\mathbb{Z}_q$ and compute

$$(a_0, b_0) = (g^{u_0} \bmod p,\ K^{u_0 + c_0} \bmod p) \tag{48}$$

and

$$(a_1, b_1) = (g^{u_1} \bmod p,\ K^{u_1} \bmod p). \tag{49}$$

When computing a proof for option λ of contest Λ, a challenge value c is formed by hashing the identifier hash H_I together with α, β, a_0, b_0, a_1, and b_1, namely

$$c = H_q(\mathrm{H}_I; \mathtt{0x24}, \mathtt{ind_c}(\Lambda), \mathtt{ind_o}(\lambda), \alpha, \beta, a_0, b_0, a_1, b_1). \tag{50}$$

The proof consists of the four values c_0, c_1, v_0, and v_1, where c_0 has been selected at random above and

$$c_1 = (c - c_0) \bmod q, \tag{51}$$
$$v_0 = (u_0 - c_0 \cdot \xi) \bmod q, \tag{52}$$
$$v_1 = (u_1 - c_1 \cdot \xi) \bmod q. \tag{53}$$

These values satisfy the proof equations as follows.

$$c = (c_0 + (c - c_0)) \bmod q = (c_0 + c_1) \bmod q, \tag{54}$$
$$(g^{v_0}\alpha^{c_0}, K^{v_0}\beta^{c_0}) \equiv_p (g^{u_0 - c_0\xi} g^{c_0\xi}, K^{u_0 - c_0\xi}(K^{\xi+1})^{c_0}) \equiv_p (g^{u_0}, K^{u_0 + c_0}) \equiv_p (a_0, b_0), \tag{55}$$
$$(g^{v_1}\alpha^{c_1}, K^{v_1 - c_1}\beta^{c_1}) \equiv_p (g^{u_1 - c_1\xi} g^{c_1\xi}, K^{u_1 - c_1\xi - c_1}(K^{\xi+1})^{c_1}) \equiv_p (a_1, b_1). \tag{56}$$

The remainder of this section specifies the general case of a range proof used for encrypting a selection. It is a straightforward generalization of the method for proofs of encryptions of 0 or 1 described above. Any zero-knowledge proof that asserts a ciphertext (α, β) encrypts an integer value σ in the range $0, 1, \ldots, R$ for some integer R is computed in detail as follows.

NIZK proof: Proves that (α, β) is an encryption of an integer in the range $0, 1, \ldots, R$. (Requires knowledge of encryption nonce ξ for which $(\alpha, \beta) = (g^{\xi} \bmod p, K^{\xi+\ell} \bmod p)$ with $0 \leq \ell \leq R$.)

A disjunctive Chaum-Pedersen range proof of (α, β) being an encryption of an integer in the range $0, 1, \ldots, R$ is produced as follows. First, for each $0 \leq j \leq R$, a random $u_j \in \mathbb{Z}_q$ is selected. The commitment

$$(a_\ell, b_\ell) = (g^{u_\ell} \bmod p,\ K^{u_\ell} \bmod p) \tag{57}$$

for $j = \ell$ is computed from u_ℓ alone. In addition, for each $0 \leq j \leq R$ with $j \neq \ell$, random $c_j \in \mathbb{Z}_q$ are selected and commitments are computed as

$$(a_j, b_j) = (g^{u_j} \bmod p,\ K^{t_j} \bmod p), \tag{58}$$

where $t_j = (u_j + (\ell - j)c_j) \bmod q$. Next, all the a_j, b_j values are hashed together with the contest and option indices $\mathtt{ind_c}(\Lambda)$ and $\mathtt{ind_o}(\lambda)$, the ciphertext and the selection encryption identifier hash H_I to form a pseudo-random challenge

$$c = H_q(\mathrm{H}_I; \mathtt{0x24}, \mathtt{ind_c}(\Lambda), \mathtt{ind_o}(\lambda), \alpha, \beta, a_0, b_0, a_1, b_1, \ldots, a_R, b_R). \tag{59}$$

The remaining challenge value c_ℓ is determined as

$$c_\ell = \left(c - \sum_{j \neq \ell} c_j\right) \bmod q = (c - (c_0 + \cdots + c_{\ell-1} + c_{\ell+1} + \cdots + c_R)) \bmod q. \tag{60}$$

Finally, responses are computed for all $0 \leq j \leq R$ as

$$v_j = (u_j - c_j\xi) \bmod q \tag{61}$$

and the proof consists of the challenge values $c_0, c_1, \ldots, c_R$ and the response values $v_0, v_1, \ldots, v_R$. Note that a range proof can be performed directly by the entity performing the public key encryption of a ballot without access to the decryption key(s). All that is required is the nonce ξ used for the selection encryption.

Specializing the general range proof with $R = 1$ and $\ell = 0$ is identical to the proof of the unselected option above, i.e., the ciphertext is an encryption of 0 or 1 given that it is an encryption of 0. Likewise, setting $R = 1$ and $\ell = 1$ is identical to the proof of the selected option.

Note 3.4. One may observe that this proof makes no use of the fact that the encrypted integer ℓ is within a range $0, 1, \ldots, R$ of consecutive integers. It is instead simply a membership proof that a ciphertext is an encryption of a value in the finite subset $\{0, 1, \ldots, R\}$ of the set of integers modulo q. This set can be replaced by any other small, finite subset, for example, $\{0, R\}$ or $\{0, 2, \ldots, 10\}$, if the used voting method only allows integers from such a subset to be assigned to a selection option. As such, the proof can easily be adapted to contests in which voters are allowed to make a null vote (no option selected) or to select exactly R candidates, with undervotes being declared invalid, that is, $\ell \in \{0, R\}$. In such a setting it is sufficient to only compute the commitments (a_j, b_j), challenges c_j and responses v_j for the values of the index j that are within the authorized set and to ignore the remaining values of j.

Note 3.5. Because all the exponentiations required to encrypt and prove ballot components have a base of either g or K, implementations can be optimized by pre-computing tables of powers of these two bases. Use of these tables can be further optimized by computing and storing the table values in Montgomery form. This technique can reduce the time to encrypt a ballot by an order of magnitude or more.

Verification 6 (Well-formedness of selection encryptions)
For each selectable option λ within each contest Λ on each cast ballot, an election verifier must, on input the encrypted selection (α, β), compute the values

(6.1) $a_j = (g^{v_j} \cdot \alpha^{c_j}) \bmod p$ for all $0 \leq j \leq R$,
(6.2) $b_j = (K^{w_j} \cdot \beta^{c_j}) \bmod p$, where $w_j = (v_j - jc_j) \bmod q$ for all $0 \leq j \leq R$,
(6.3) $c = H_q(\mathrm{H}_I; \mathtt{0x24}, \mathtt{ind_c}(\Lambda), \mathtt{ind_o}(\lambda), \alpha, \beta, a_0, b_0, a_1, b_1, \ldots, a_R, b_R)$,

where R is the option selection limit. An election verifier must then confirm the following:

(6.A) The given values α and β are in the set $\mathbb{Z}_p^r$.
(A value x is in $\mathbb{Z}_p^r$ if and only if x is an integer such that $0 \leq x < p$ and $x^q \bmod p = 1$.)
(6.B) The given values c_j are each in the set $\mathbb{Z}_q$ for all $0 \leq j \leq R$.
(A value x is in $\mathbb{Z}_q$ if and only if x is an integer such that $0 \leq x < q$.)
(6.C) The given values v_j are each in the set $\mathbb{Z}_q$ for all $0 \leq j \leq R$.
(6.D) The equation $c = (c_0 + c_1 + \cdots + c_R) \bmod q$ is satisfied.

3.3.8 Proof of Satisfying the Contest Selection Limit

The final step in proving that a ballot is well-formed is demonstrating that the contest selection limits for each contest have not been exceeded. This is accomplished by homomorphically combining all of the relevant (α_i, β_i) values for a contest (as specified in the manifest) by forming the aggregate contest encryption $(\bar{\alpha}, \bar{\beta}) = (\prod_i \alpha_i \bmod p,\ \prod_i \beta_i \bmod p)$ and proving that $(\bar{\alpha}, \bar{\beta})$ is an encryption of a value that is not larger than the total number of votes allowed for that contest (usually one). The simplest way to complete this proof is to combine all of the pseudo-random nonces ξ_i that were used to form each $(\alpha_i, \beta_i) = (g^{\xi_i} \bmod p,\ K^{\xi_i+\sigma_i} \bmod p)$. The aggregate nonce $\xi = (\sum_i \xi_i) \bmod q$ matches the aggregate contest encryption as $(\bar{\alpha}, \bar{\beta}) = (\prod_i \alpha_i \bmod p, \prod_i \beta_i \bmod p) = (g^{\xi} \bmod p,\ K^{\xi+\ell} \bmod p)$, where ℓ is the total number of votes cast by the voter in the contest. A range proof can then be used as above to show that $0 \leq \ell \leq L$, where L is the selection limit for the contest.

NIZK proof: Proves that $(\bar{\alpha}, \bar{\beta})$ is an encryption of an integer in the range $0, 1, \ldots, L$. (Requires knowledge of aggregate encryption nonce ξ for which $(\bar{\alpha}, \bar{\beta})$ is an encryption of ℓ.)

This proof is a disjunctive Chaum-Pedersen range proof as described in detail in Section 3.3.7. The range bound L is equal to the contest selection limit and the ciphertext (α, β) from Section 3.3.7 is replaced by the aggregate contest encryption $(\bar{\alpha}, \bar{\beta})$.

Commitments (a_i, b_i) and challenge values c_j for $j \neq \ell$ are computed as shown in Section 3.3.7. The challenge value for contest Λ is then computed as

$$c = H_q(\mathrm{H}_I; \texttt{0x24}, \texttt{ind}_{\texttt{c}}(\Lambda), \bar{\alpha}, \bar{\beta}, a_0, b_0, a_1, b_1, \ldots, a_L, b_L). \qquad (62)$$

The remaining challenge value c_ℓ can then be determined and the responses v_i computed as shown in Section 3.3.7. The proof consists of the challenge values $c_0, c_1, \ldots, c_L$ and the response values $v_0, v_1, \ldots, v_L$. Note that all of the above proofs can be performed directly by the entity performing the public key encryption of a ballot without access to the decryption key(s). All that is required is the aggregate nonce ξ, i.e., the sum of the nonces ξ_i used for the individual selection encryptions.

Verification 7 (Adherence to vote limits)
For each contest Λ on each cast ballot, an election verifier must compute the contest totals
(7.1) $\bar{\alpha} = (\prod_i \alpha_i) \bmod p$,
(7.2) $\bar{\beta} = (\prod_i \beta_i) \bmod p$,
where the (α_i, β_i) represent all possible selections for the contest, as well as the values
(7.3) $a_j = (g^{v_j} \cdot \bar{\alpha}^{c_j}) \bmod p$ for all $0 \leq j \leq L$,
(7.4) $b_j = (K^{w_j} \cdot \bar{\beta}^{c_j}) \bmod p$, where $w_j = (v_j - jc_j) \bmod q$ for all $0 \leq j \leq L$,
(7.5) $c = H_q(\mathrm{H}_I; \texttt{0x24}, \texttt{ind}_{\texttt{c}}(\Lambda), \bar{\alpha}, \bar{\beta}, a_0, b_0, a_1, b_1, \ldots, a_L, b_L)$,
where L is the contest selection limit. An election verifier must then confirm the following:
(7.A) The given values α_i and β_i are each in $\mathbb{Z}_p^r$.
(7.B) The given values c_j are each in $\mathbb{Z}_q$ for all $0 \leq j \leq L$.
(7.C) The given values v_j are each in $\mathbb{Z}_q$ for all $0 \leq j \leq L$.
(7.D) The equation $c = (c_0 + c_1 + \cdots + c_L) \bmod q$ is satisfied.

3.3.9 Supplemental Verifiable Fields (Optional)

ElectionGuard offers compatibility with several supplemental fields that may be desirable in some applications. Examples are counter fields that indicate whether an undervote occurred in the contest, whether a null vote occurred, whether the contest was overvoted, or a counter for the number of used write-in fields. These supplemental fields allow verifiable tallies that show the total numbers of undervotes, null votes, overvotes, or used write-in fields in a specific contest. Any supplemental fields that are used must be included in the election manifest. In most respects, supplemental fields are treated like ordinary option fields; however, the proofs for meeting selection limits may differ because of the semantics of the various supplemental fields. An implementation of ElectionGuard is not required to support supplemental fields.

Details of the above examples for supplemental verifiable fields are discussed below. In general, ElectionGuard allows verifiable fields that contain any other data as long as it can be represented by small positive integers. Depending on the properties that must be proved for these fields, it might be necessary to prove conjunctions or disjunctions of certain membership set proofs. These proofs are not described in detail in this specification.

Undervote. When the manifest requires counting undervoted contests, an encrypted undervote indicator is provided. This indicator is an encryption of one if the contest was undervoted, that is, if and only if the number of option fields that are set to one (or the sum of the selections assigned by the voter) is strictly less than the contest selection limit. It is an encryption of zero otherwise. Just as for every indicator, the encryption is performed using the vote encryption public key K.

In order to make this encrypted undervote indicator verifiable, an additional proof must be provided. This proof can demonstrate that one of the two following statements is true: either (1) the encrypted undervote indicator is an encryption of zero and the sum of the selections assigned by the voter matches the contest selection limit, or (2) the encrypted undervote indicator is an encryption of one and the sum of the selections assigned by the voter lies in the range between zero and the contest selection limit minus one. (We note that this proof also implies that the encrypted undervote indicator is an encryption of zero or one, which is needed for counting the number of undervoted contests.)

The manifest may also require an undervote difference count. The value of this field is an encryption of the number of undervotes on that ballot for that contest. The validity of the corresponding ciphertext can be verified by checking that the difference between the contest selection limit for that contest and the undervote difference count exactly matches the sum of all selections assigned by the voter.

Overvote. When the manifest requires counting overvoted contests, an encrypted overvote indicator is provided. Similarly to the encrypted undervote indicator, this indicator is an encryption of one if the contest was overvoted, that is, when and only when the number of option fields that are set to one (or the sum of the selections assigned by the voter) is strictly more than the contest selection limit. It is an encryption of zero otherwise.

In the case of an overvote, ElectionGuard sets all selectable options in the contest to zero, i.e., ElectionGuard generates encryptions of zero (unselected option) for all selectable options. This makes sure that an overvote does not affect the tallies.

The validity of the encrypted overvote indicator can be enforced by checking that this indicator encrypts 0 or 1 and that the sum of the selection fields—when added to the product of the contest selection limit with the overvote field content does not exceed the selection limit for that contest.[42] This verification step also guarantees that the selection limit is satisfied for the contest. We also note that this validity check is compatible with a situation in which the selections assigned by the voters are all zero and the overvote indicator is zero as well: this reflects a null vote and not an overvote that has been neutralized.

ElectionGuard does not provision for an overvote difference counter, given that overvotes are treated as invalid votes anyway.

Null vote. A null vote occurs if no selection has been made, which is a special case of an undervote. When the manifests requires counting the null votes, an encrypted null vote indicator is provided, which is set to an encryption of one in case of a null vote and to an encryption of zero otherwise, similarly to the encrypted undervote and overvote indicators.

The validity of the encrypted null vote indicator can be enforced just as the validity of the encrypted overvote indicator. When all the selections are set to zero as a consequence of an overvote, the null vote indicator should be set to zero.

In order to make this encrypted null vote indicator verifiable, an additional proof must be provided. This proof can demonstrate that one of the two following statements is true: either (1) the encrypted null vote indicator is an encryption of zero and the sum of the selections assigned by the voter lies in the range between one and the selection limit, or (2) the encrypted null vote indicator is an encryption of one and the sum of the selections assigned by the voter is zero. Providing such a proof forces setting the null vote indicator to one for every overvote ballot as well. Note that the functionality of a null vote indicator is similar to that of an overvote indicator. But the requirement here is that the null vote indicator *must* be set if no other selection has been made. Therefore, the proof is somewhat different from that used for the overvote indicator.

Contest write-in. When the manifest requires counting the number of write-in fields utilized in a contest, an encrypted write-in indicator is provided. If a contest offers more than one write-in field, the indicator may take a value larger than one. Range proofs should be included to show that each encrypted write-in indicator encrypts a value between zero and the number of write-in fields that are offered.

For all write-in selections in a contest, ElectionGuard captures the text written into each write-in field. This text can be collected in the contest data field for that contest, after any information about overvotes, undervotes, or null votes, and encrypted as described below. The number of write-ins should be incorporated into the proof of meeting the selection limit.

[42] The encrypted product can be computed by exponentiating the ciphertext encrypting the overvote field with the contest selection limit. An alternative implementation would be to always set the overvote field to either zero or the contest selection limit. This avoids the exponentiation during the check but the publicly verified number of overvotes for that contest would be scaled up by a factor of the contest selection limit and the selection limit would then need to be divided out for reporting purposes.

3.3.10 Contest Data (Optional)

For each contest Λ, ElectionGuard allows the inclusion of a contest data field in addition to the optional encrypted indicators discussed above. This data field may include any text written into one or more write-in text fields, information about overvotes, undervotes, and null votes, and possibly other data about voter selections. An implementation of ElectionGuard is not required to support contest data fields. If such a field is provided and utilized, all non-numerical data associated with the contest is encrypted with the ballot data encryption public key in a single hashed ElGamal encryption as follows.

Fixed-length encoding. The information held in this field must be encoded in a byte array D_Λ of a fixed length of $32 \cdot b_\Lambda$ bytes, where b_Λ is specified in the election manifest to be the smallest value large enough for D_Λ to capture all additional information about this contest from the ballot that needs to be retained and stored in encrypted form as part of the encrypted ballot. It is incumbent upon the device calling ElectionGuard to fill these $32 \cdot b_\lambda$ bytes unambiguously. In particular, if padding is used, it should not be possible to confuse a padding character with a trailing blank or other character that is part of the data.

Encryption. Write the contest data field as a concatenation of blocks

$$D_\Lambda = D_1 \parallel D_2 \parallel \cdots \parallel D_{b_\Lambda}, \tag{63}$$

where the D_i, $1 \le i \le b_\Lambda$, consist of 32 bytes each. To encrypt it, ElectionGuard uses the same encryption mode as for encrypting secret key shares and the ballot nonces above. Therefore, ElectionGuard uses the public ballot data encryption key $\hat{K}$. A pseudo-random nonce ξ is derived from the ballot nonce ξ_B and the contest label Λ as

$$\xi = H_q(\mathrm{H}_I; \mathtt{0x25}, \mathtt{ind_c}(\Lambda), \xi_B) \tag{64}$$

to compute $(\alpha, \beta) = (g^\xi \bmod p, \hat{K}^\xi \bmod p)$. A 256-bit secret key is derived as the hash

$$h = H(\mathrm{H}_I; \mathtt{0x26}, \mathtt{ind_c}(\Lambda), \alpha, \beta). \tag{65}$$

Next, a KDF in counter mode[43] based on HMAC is used to generate encryption keys $k_1 \parallel k_2 \parallel \cdots \parallel k_{b_\Lambda}$ by computing

$$k_i = \mathrm{HMAC}(h, \mathtt{b}(i,4) \parallel \mathtt{Label} \parallel \mathtt{0x00} \parallel \mathtt{Context} \parallel \mathtt{b}(b_\Lambda \cdot 256, 4)), \tag{66}$$

for $1 \le i \le b_\Lambda$. The label and context byte arrays are $\mathtt{Label} = \mathtt{b}(\text{“}\mathtt{data\_enc\_keys}\text{”}, 13)$ and $\mathtt{Context} = \mathtt{b}(\text{“}\mathtt{contest\_data}\text{”}, 12) \parallel \mathtt{b}(\mathtt{ind_c}(\Lambda), 4)$. Each k_i is a 256-bit key, and $\mathtt{b}(i,4)$ and $\mathtt{b}(b_\Lambda \cdot 256, 4)$ are byte arrays of the fixed length of 4 bytes that encode the integers i and $b_\Lambda \cdot 256$. Therefore, i and $b_\Lambda \cdot 256$ must be less than 2^{32}, i.e., $0 \le i < 2^{32}$ and $1 \le b_\Lambda < 2^{24}$.

[43] NIST (2022) *Recommendation for Key Derivation Using Pseudorandom Functions.* In: SP 800-108r1 https://csrc.nist.gov/pubs/sp/800/108/r1/upd1/final. Note that the secret session key k changes for every encryption because a fresh encryption nonce ξ is pseudo-randomly generated every time. The second input to HMAC consists of the byte encoding of the counter i, the UTF-8 encoding of the string “data_enc_keys” as the fixed *Label* field specified in SP 800-108r1, the separating 0x00 byte, the UTF-8 encoding of the string “contest_data” as the *Context* field, and the 4-byte encoding of the length of the total key material output in bits.

The first two components of the ciphertext encrypting D_Λ are

$$C_0 = \alpha = g^\xi \bmod p, \tag{67}$$

$$C_1 = D_1 \oplus k_1 \parallel D_2 \oplus k_2 \parallel \cdots \parallel D_{b_\Lambda} \oplus k_{b_\Lambda}. \tag{68}$$

The component C_1 is computed by bitwise XOR (here denoted by $\oplus$) of each data block D_i with the corresponding key k_i.

Next, a Schnorr proof of knowledge of the nonce ξ is generated by selecting a uniform random element $u \in \mathbb{Z}_q$ and computing $a = g^u \bmod p$. A challenge value is computed as

$$c = H_q(\mathrm{H}_I; \texttt{0x27}, \texttt{ind}_\texttt{c}(\Lambda), a, C_0, C_1) \tag{69}$$

and the corresponding response value as $v = (u - c\xi) \bmod q$. The full ciphertext encrypting D_Λ is $C = (C_0, C_1, C_2)$, where $C_2 = (c, v)$.

3.4 Confirmation Codes

Upon completion of the encryption of each ballot, a *confirmation code* is prepared for each voter.[44] The code is a hash value (an output of the function H, which is specified in detail in Section 5) whose inputs must include the encrypted ballot and the selection encryption identifier hash H_I. Inputs to the hash may optionally include other information such as an identifier for the voting device, the location of the voting device, as well as the date and time that the ballot was encrypted.

The data that constitutes the encrypted ballot for the purpose of computing its confirmation code consists of all encrypted selections on that ballot. This includes a ciphertext (α, β) for each selection in each contest on the ballot. The number of contests on the ballot B is denoted by m_B and contests have a specified order defined in the election manifest file. Therefore, each contest has a unique contest index l, where $1 \le l \le m_B$. The remainder of this section details the process for computing a confirmation code for a given ballot B.

3.4.1 Contest Hash

For each contest, ElectionGuard computes a *contest hash*, which is a hash value of an input that contains encryptions of all verifiable fields in the contest. These verifiable fields include all selectable option fields, i.e., the choices made by the voters, and, if the election manifest specifies any, all additional verifiable data fields such as the counters discussed in Section 3.3.9. In what follows, additional fields are treated just like selectable option fields. The encryptions are hashed in order specified by the election manifest file. Let $E_i = (\alpha_i, \beta_i)$ denote the ciphertext encrypting the data in the i-th verifiable field ($1 \le i \le m_l$) of the l-th contest ($1 \le l \le m_B$, $l = \texttt{ind}_\texttt{c}(\Lambda_l)$).

The contest hash value χ_l is computed from the contest index l, the sequence of ciphertexts $E_1, E_2, \ldots, E_{m_l}$, and, if present, the encrypted contest data $C_\Lambda = (C_0, C_1, C_2)$ as

$$\chi_l = H(\mathrm{H}_I; \texttt{0x28}, l, \alpha_1, \beta_1, \alpha_2, \beta_2 \ldots, \alpha_{m_l}, \beta_{m_l}, C_0, C_1, C_2). \tag{70}$$

If no contest data is present, C_0, C_1, and C_2 are omitted from the input to H in Equation (70).

[44] Confirmation codes may be omitted when ElectionGuard is used for post-election audits.

3.4.2 Confirmation Code

The *ballot confirmation code* H_C is a *ballot hash* that is computed from all contest hashes as

$$\mathrm{H}_C = H(\mathrm{H}_I; \mathtt{0x29}, \chi_1, \chi_2, \ldots, \chi_{m_B}, \mathrm{B}_C). \tag{71}$$

The input contains the contest hashes in the order of the contests as specified in the election manifest file. Besides the contest hashes, there is an additional input byte array B_C, the *chaining field*. The byte array B_C is 36 bytes long. It is used to enable ballot chaining and in that case may contain the confirmation code of a previous ballot as described below. It can also be used to link confirmation codes to the voting device they are generated on.

3.4.3 Voting Device Information Hash

The chaining field is used to include information about the voting device encoded in a hash value H_{DI}, the *device information hash.* The value H_{DI} is a hash

$$\mathrm{H}_{DI} = H(H_E; \mathtt{0x2A}, S_{\mathrm{device}}), \tag{72}$$

of a string S_{device} that may contain a unique voting device identifier or a unique voting location identifier and possibly other encoded voting device information or the time the ballot was processed as specified in the election manifest file. The manifest may specify that S_{device} does not contain voting device information. The string is hashed by first encoding it as a byte array using UTF-8. The array starts with a 4-byte field specifying the length of the UTF-8 encoding of S_{device} in bytes concatenated with the encoding itself.[45]

3.4.4 Ballot Chaining

Additional input to the confirmation code $\mathrm{H}_j = \mathrm{H}_{C,j}$ of the j-th ballot computed on a specific device can be included. In its simplest form, this input is the confirmation code $\mathrm{H}_{j-1} = \mathrm{H}_{C,j-1}$ of the previous ballot processed on the device. Hash chaining can be implemented in different modes, for example using simple sequential chaining as above or allowing a tree of hash dependencies. This section only specifies the no chaining and simple chaining modes in detail, other modes must be uniquely identified by a 4-byte identifier and specified in the election manifest. In any case, the chaining field B_C is a byte array of 36 bytes. It contains the *chaining mode identifier* in its first four bytes and has space for a hash value in the remaining 32 bytes.

The benefit of a chain is that it makes it more difficult for a malicious insider to selectively delete ballots and confirmation codes after an election without detection.

No chaining. The simplest chaining mode is the *no chaining mode*, which is identified by the chaining mode identifier `0x00000000`. Since there is no dependency on previous confirmation codes, the remaining 32 bytes of B_C are the same for every confirmation code computation. To include the voting device information, the chaining field contains the device information hash H_{DI} and is set to

$$B_C = \mathtt{0x00000000} \parallel \mathrm{H}_{DI}. \tag{73}$$

[45] See Section 5.1.4.

Simple chaining. The chaining mode identified by `0x00000001` is the *simple chaining mode*, where a confirmation code only depends on the confirmation code of the previous ballot (except for the first ballot voted on the device) and the voting device information. It is specified in detail as follows.

The simple chaining mode hash chain is initialized by computing

$$\mathrm{H}_0 = H(\mathrm{H}_E; \mathtt{0x29}, \mathrm{B}_{C,0}), \tag{74}$$

where the hash input

$$B_{C,0} = \mathtt{0x00000001} \parallel \mathrm{H}_{DI}, \tag{75}$$

binds confirmation codes in the chain to the voting device information.

The confirmation code for the j-th ballot when $j > 0$ is then computed as above via $\mathrm{H}_j = H(\mathrm{H}_I; \mathtt{0x29}, \chi_1, \chi_2, \ldots, \chi_{m_B}, \mathrm{B}_{C,j})$ and the chaining field byte array

$$\mathrm{B}_{C,j} = \mathtt{0x00000001} \parallel \mathrm{H}_{j-1} \tag{76}$$

contains the confirmation code H_{j-1} of the previous ballot (or the initialization code H_0 when computing H_1). The chain is closed at the end of an election by forming and publishing

$$\overline{\mathrm{H}} = H(\mathrm{H}_E; \mathtt{0x29}, \overline{\mathrm{B}}_C), \tag{77}$$

using

$$\overline{\mathrm{B}}_C = \mathtt{0x00000001} \parallel H(\mathrm{H}_E; \mathtt{0x2B}, \mathrm{H}_\ell, \mathrm{B}_{C,0}), \tag{78}$$

where H_ℓ is the final confirmation code in the chain.

Ballot casting or challenging. Once in possession of a confirmation code (*and never before*), a voter is afforded an option to either cast the associated ballot or challenge it and restart the ballot preparation process. The precise mechanism for voters to make this choice may vary depending upon the instantiation, but this choice would ordinarily be made immediately after a voter is presented with the confirmation code, and the status of the ballot would be undetermined until the decision is made. It is possible, for instance, for a voter to make the decision directly on the voting device, or a voter may instead be afforded an option to deposit the ballot in a receptacle or to take it to a poll worker to be challenged. For vote-by-mail scenarios, a voter can be sent (hashes of) two complete sets of encryptions for each selectable option and can effect a ballot challenge implicitly by choosing which encryptions to return (see Section 4).

Verification 8 (Validation of confirmation codes)
An election verifier must confirm the following for each ballot.

(8.A) The contest hash χ_l for the contest with contest index l for all $1 \leq l \leq m_B$ has been correctly computed from the contest's m_l selection encryptions (α_i, β_i), $1 \leq i \leq m_l$, as

$$\chi_l = H(\mathrm{H}_I; \texttt{0x28}, l, \alpha_1, \beta_1, \alpha_2, \beta_2 \ldots, \alpha_{m_l}, \beta_{m_l}, C_0, C_1, C_2).$$

(8.B) The ballot confirmation code H_C has been correctly computed from the contest hashes and the chaining field byte array B_C as

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

(8.C) The voting device information hash has been correctly computed from the string S_{device} for the device the ballot was processed on as specified in the election manifest as

$$\mathrm{H}_{DI} = H(H_E; \texttt{0x2A}, S_{\text{device}}).$$

(8.D) If the no-chaining mode was used on the device, the chaining field is $B_C = \texttt{0x00000000} \parallel \mathrm{H}_{DI}$.

If the simple chaining mode was used on the device, an election verifier must confirm the following three items.

(8.E) If the ballot is the j^{th} ballot processed on this device, $1 \leq j \leq \ell$, the chaining field byte array used to compute H_j is equal to $\mathrm{B}_{C,j} = \texttt{0x00000001} \parallel \mathrm{H}_{j-1}$.

The following items need only be verified once per voting device.

(8.F) The initial hash code H_0 satisfies $\mathrm{H}_0 = H(\mathrm{H}_E; \texttt{0x29}, \mathrm{B}_{C,0})$ and $\mathrm{B}_{C,0} = \texttt{0x00000001} \parallel \mathrm{H}_{DI}$.

(8.G) The final input byte array is $\overline{\mathrm{B}}_C = \texttt{0x00000001} \parallel H(\mathrm{H}_E; \texttt{0x2B}, \mathrm{H}_\ell, \mathrm{B}_{C,0})$, where H_ℓ is the final confirmation code on this device, and the closing hash is correctly computed as $\overline{\mathrm{H}} = H(\mathrm{H}_E; \texttt{0x29}, \overline{\mathrm{B}}_C)$.

3.5 Ballot Aggregation

At the conclusion of voting, all of the ballot encryptions are published in the election record together with the proofs that the ballots are well-formed. Additionally, all of the encryptions of each option are homomorphically combined to form an encryption of the sum of the values that were individually encrypted. The encryptions (α_i, β_i) of each individual option are combined by forming the product

$$(A, B) = \left(\left(\prod_i \alpha_i\right) \bmod p, \left(\prod_i \beta_i\right) \bmod p\right). \tag{79}$$

This aggregate encryption (A, B), which represents an encryption of the tally of that option, is published in the election record for each option.

Verification 9 (Correctness of ballot aggregation)
An election verifier must confirm for each option in each contest in the election manifest that the aggregate encryption (A, B) satisfies
(9.A) $A = (\prod_j \alpha_j) \bmod p$,
(9.B) $B = (\prod_j \beta_j) \bmod p$,
where the (α_j, β_j) are the corresponding encryptions on all cast ballots in the election record.

In some applications, such as shareholder or stakeholder voting, different votes may each have a different *weight* or *scaling factor* W associated with each voter. ElectionGuard can accommodate weighted votes if each encrypted ballot in the election record is explicitly associated with an identified voter and weight. Weights must be small positive integers to allow correct decryption of tallies. The default value of a weight should be 1. The *weighted* aggregation is then computed as

$$(A, B) = \left(\left(\prod_i \alpha_i^{W_i} \right) \bmod p, \left(\prod_i \beta_i^{W_i} \right) \bmod p \right), \tag{80}$$

where W_i is the weight associated with ballot B_i. Setting all weights to the default value of 1 results in the un-weighted case.

If weights are used, the equations in Verification 9 need to be adjusted correspondingly to include weights as above.

3.6 Verifiable Decryption

3.6.1 Preliminary Verification Steps

When the election administrator has populated the election record with the sets of aggregated ballots and ballots marked to be challenged, and prior to starting any decryption operation, each guardian must verify that all the ciphertexts marked for decryption are correct. This is necessary to guarantee that the privacy of the votes is preserved and that the decryption data that is produced contributes to the verifiability of the election.

In effect, Verification steps 1, 2, and 3 have been performed by the guardians as part of the validation of the key generation steps. Guardians must now ensure that Verification steps 4, 5, 6, 7, 8, and 9 are successful, which confirms the set of ciphertexts that need to be decrypted and their validity.[46]

3.6.2 Verifiable Decryption Strategy

To decrypt an aggregate encryption (A, B), guardians work together in a protocol, where each guardian produces a partial decryption and contributes to an accumulated Chaum-Pedersen proof of correct decryption.

[46]Guardians can perform these verification steps themselves or rely on parties they trust to perform these verification steps in their stead.

As long as at least k guardians are present for decryption, the partial decryptions produced by the guardians are used to compute the value

$$M = A^s \bmod p, \tag{81}$$

without ever computing the secret key s. This value is then used to obtain

$$T = (B \cdot M^{-1}) \bmod p. \tag{82}$$

This T has the property that $T = K^t \bmod p$ where t is the tally of the associated option.

In general, this final computation of the tally t from T is computationally intractable as it requires computation of a discrete logarithm. However, in this application, t is relatively small—usually bounded by the number of votes cast. This tally value t can be determined from T by exhaustive search, by precomputing a table of all possible T values in the allowable range and then performing a single look-up, or by a combination in which some exponentiations are precomputed and a small search is used to find the value of t (e.g., a partial table consisting of $K^{100} \bmod p$, $K^{200} \bmod p$, $K^{300} \bmod p$, $\ldots$ is precomputed and the value T is repeatedly divided (or multiplied) by K until a value is found that is in the partial table). The value t is published in the election record, and verifiers should check both that $T = K^t \bmod p$ and that $B = (T \cdot M) \bmod p$.[47]

3.6.3 Partial Decryption by Available Guardians

During the key generation process (as described in Section 3.2.2), each guardian G_i has received from each other guardian G_j a share $P_j(i)$ of G_j's secret key s_j ($1 \leq i, j \leq n$ and $i \neq j$), has computed its own share $P_i(i)$, and with all shares $P_j(i)$ for $1 \leq j \leq n$ has computed

$$z_i = P(i) = \left(\sum_{j=1}^{n} P_j(i)\right) \bmod q = (P_1(i) + P_2(i) + \cdots + P_n(i)) \bmod q. \tag{83}$$

The value z_i is G_i's share of the implicit secret key $s = (s_1 + s_2 + \cdots + s_n) \bmod q$.[48]

Each guardian G_i available for decryption uses z_i to compute the partial decryption

$$M_i = A^{z_i} \bmod p \tag{84}$$

that contributes to computing the value $M = A^s \bmod p$.

3.6.4 Combination of Partial Decryptions

The partial decryptions are combined to obtain M with the help of the Lagrange coefficients that correspond to the available guardians. Let $U \subseteq \{1, 2, \ldots, n\}$ be the set of indices such that $\{G_i :$

[47] For larger values of t (more than 20 bits), Shanks' baby-step giant step method can be used as described in Footnote 12.

[48] Guardian G_i's secret key s_i is the constant coefficient of the polynomial $P_i(x)$ and therefore the implicit secret key $s = (s_1 + s_2 + \cdots + s_n) \bmod q$ is the constant coefficient of the polynomial $P(x) = (P_1(x) + P_2(x) + \cdots + P_n(x)) \bmod q$. The collection of the $z_i = P(i)$ are the shares for a k-out-of-n sharing of the secret key s. It is important to note that this secret key s is never actually computed; instead each guardian uses its share z_i of the implicit secret key s to form a share of any decryption that needs to be performed.

$i \in U\}$ is the set of available guardians. For decryption to succeed, a quorum of at least k guardians is needed, i.e., $|U| = h \geq k$. All available guardians participate in the reconstruction even if $h > k$, i.e., there are more guardians available than strictly necessary for having a quorum.

The Lagrange coefficients corresponding to U are computed as

$$w_i = \left(\prod_{\ell \in (U \setminus \{i\})} \frac{\ell}{\ell - i} \right) \bmod q. \tag{85}$$

These coefficients are used to combine the M_i for $i \in U$ provided by the available guardians to obtain

$$M = \left(\prod_{i \in U} (M_i)^{w_i} \right) \bmod p. \tag{86}$$

Note 3.6. The decryption $M = A^s \bmod p$ can be computed as shown in Equation (86) because $s = (\sum_{i \in U} w_i z_i) \bmod q = (\sum_{i \in U} w_i P(i)) \bmod q$. Likewise, a missing secret s_j could be computed directly as $s_j = (\sum_{i \in U} w_i P_j(i)) \bmod q$. However, it is preferable to not release any missing secret s_j (or the secret s) and instead only release the partial decryptions that the secret would have produced. This prevents the secret from being used for additional decryptions without the cooperation of at least k guardians.

As an example, consider an election with five guardians and a threshold of three. If two guardians are missing at the time of decryption, the remaining three can perform any required decryptions as described in the text above. If, instead, they take the shortcut of simply reconstructing and then using the two missing secrets, then any of the three could, at a later time, use these missing secrets together with its own secret to perform additional decryptions without the cooperation of any other guardian.

3.6.5 Proof of Correctness

The available guardians work together to produce a Chaum-Pedersen proof that M was computed correctly, which means that $M = A^s \bmod p$. The proof is computed as follows, for each option λ in each contest Λ.

NIZK proof: The available guardians jointly prove that they have shared knowledge of $s \in \mathbb{Z}_q$ such that $M = A^s \bmod p$ and $K = g^s \bmod p$.

Each available guardian G_i, $i \in U$, selects a random value u_i in $\mathbb{Z}_q$ and computes the commitment pair

$$(a_i, b_i) = (g^{u_i} \bmod p,\ A^{u_i} \bmod p). \tag{87}$$

Guardian G_i then commits to (a_i, b_i) by computing the hash value

$$d_i = H(\mathrm{H}_E; \texttt{0x30}, \texttt{ind}_{\texttt{c}}(\Lambda), \texttt{ind}_{\texttt{o}}(\lambda), i, A, B, a_i, b_i, M_i, U) \tag{88}$$

and, using the administrator as a mediator, sends it to every other guardian G_j, $j \in U$, $j \neq i$. Only after having received a value d_j from every other participating guardian G_j, guardian G_i sends the pair (a_i, b_i) to every other guardian. Guardian G_i verifies that the values d_j have been correctly computed via equation (88) for every $j \in U$. If G_i does not receive (a_j, b_j) from a participating guardian G_j or if equation (88) does not hold for any $j \in U$, then guardian G_i halts the protocol and complains.

If none of the guardians complains, the a_i and b_i obtained from each guardian are used to compute the accumulated commitments as

$$a = \left(\prod_{i\in U} a_i\right) \bmod p,\ \ b = \left(\prod_{i\in U} b_i\right) \bmod p. \tag{89}$$

This means $a = g^u \bmod p$ and $b = A^u \bmod p$, where $u = (\sum_{i\in U} u_i) \bmod q$ and u is not computed explicitly.

The ciphertext (A, B), the commitments (a, b), and the combined value M are then hashed together with the extended base hash value H_E to form a challenge

$$c = H_q(\mathrm{H}_E; \texttt{0x31}, \texttt{ind}_\texttt{c}(\Lambda), \texttt{ind}_\texttt{o}(\lambda), A, B, a, b, M). \tag{90}$$

The challenge c is adjusted by the i-th Lagrange coefficient to produce a challenge c_i for available guardian G_i $(i \in U)$ as

$$c_i = (c \cdot w_i) \bmod q. \tag{91}$$

Next, each available guardian G_i responds to the challenge c_i with

$$v_i = (u_i - c_i z_i) \bmod q. \tag{92}$$

An accumulated response

$$v = \left(\sum_{i\in U} v_i\right) \bmod q \tag{93}$$

is computed and the decrypted value $T = (B \cdot M^{-1}) \bmod p$ is published along with the proof (c, v) in the election record. Note that $v = (u - c \cdot s) \bmod q$.

Note 3.7. Individual responses v_i can be verified by recomputing the commitments from the responses, the individual challenge values c_i, the guardians' commitments $K_{j,m}$ to the coefficients $a_{j,m}$ of their secret sharing polynomials P_j (see 3.2.2), the ciphertext value A, and the partial decryptions M_i as

$$a'_i = \left(\left(\prod_{j=1}^{n}\prod_{m=0}^{k-1} K_{j,m}^{i^m}\right)^{c_i} g^{v_i}\right) \bmod p, \tag{94}$$

$$b'_i = (A^{v_i} M_i^{c_i}) \bmod p, \tag{95}$$

and by checking that $a'_i = a_i$ and $b'_i = b_i$. Should the accumulated proof be invalid, this mechanism may help identify whether one of the guardians produced an invalid proof.

Verification 10 (Correctness of tally decryptions)
For each option λ in each contest Λ on each tally, an election verifier must confirm the correctness of the decrypted accumulated value t. It must compute the values

(10.1) $M = (B \cdot T^{-1}) \bmod p$,
(10.2) $a = (g^v \cdot K^c) \bmod p$,
(10.3) $b = (A^v \cdot M^c) \bmod p$.

An election verifier must then confirm the following:

(10.A) The given value v is in the set $\mathbb{Z}_q$.
(10.B) The challenge value c satisfies $c = H_q(\mathrm{H}_E; \mathtt{0x31}, \mathtt{ind}_{\mathtt{c}}(\Lambda), \mathtt{ind}_{\mathtt{o}}(\lambda), A, B, a, b, M)$.
(10.C) The decrypted tally value t satisfies $T = K^t \bmod p$.

Tally verification. The final step is to verify that the tallies contain the correct contests and options.

Verification 11 (Validation of contents of tallies)
An election verifier must confirm that the text labels listed in the election record tallies match the corresponding text labels in the election manifest. For each contest in a decrypted tally, an election verifier must confirm the following.

(11.A) The contest text label occurs as a contest label in the list of contests in the election manifest.
(11.B) For each option in the contest, the option text label occurs as an option label for the contest in the election manifest.
(11.C) For each option text label listed for this contest in the election manifest, the option label occurs for an option in the decrypted tally contest.

An election verifier must also confirm the following.

(11.D) For each contest text label that occurs in at least one submitted ballot, that contest text label occurs in the list of contests in the corresponding tally.

3.6.6 Decryption of Contest Data (Optional)

For each contest Λ, an encrypted ballot may contain a ciphertext $C_E = (C_0, C_1, C_2)$ encrypting contest data such as overvote and write-in text fields. The ciphertext has been generated as described in Section 3.3.10. Such data may need to be decrypted if the tallies record a significant number of votes in write-in selections. If contest data of actual cast votes is decrypted, it will likely be necessary to use a mixnet[49] to keep decrypted contest data from being associated with individual voters. Details of mixnets are outside the scope of this document. The use of a mixnet may not be necessary for auditing applications.

Decryption can be done by a quorum of guardians similarly to the decryption of tallies explained above. However, before each available guardian G_i, $i \in U$, computes a partial decryption, it verifies that the Schnorr proof is valid, i.e., it parses $C_2 = (c, v)$, then computes $a = (g^v \cdot C_0^c) \bmod p$ and

[49] Chaum D. (1981) *Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms* Communications of the ACM.

verifies that $c = H_q(H_I; \texttt{0x27}, \texttt{ind}_\texttt{c}(\Lambda), a, C_0, C_1)$ according to Equation (69). Only if this is the case, guardian G_i proceeds to compute the partial decryption

$$m_i = C_0^{\hat{z}_i} \bmod p \tag{96}$$

using its precomputed ballot data encryption key share $\hat{z}_i$. The partial decryptions are combined using the Lagrange coefficients for the set U of available guardians to obtain

$$\beta = \left(\prod_{i \in U} m_i^{w_i}\right) \bmod p \tag{97}$$

Again, the available guardians work together to produce and publish the following proof.

NIZK proof: The available guardians jointly prove that they have shared knowledge of $\hat{s} \in \mathbb{Z}_q$ such that $\beta = C_0^{\hat{s}} \bmod p$ and $\hat{K} = g^{\hat{s}} \bmod p$.

This proof is exactly the same as the one in Section 3.6.5, where the ciphertext (A, B) is replaced by the contest data ciphertext (C_0, C_1, C_2) as follows. Guardian G_i selects a random value u_i in $\mathbb{Z}_q$ and computes the pair

$$(a_i, b_i) = (g^{u_i} \bmod p,\ C_0^{u_i} \bmod p). \tag{98}$$

Guardian G_i then commits to (a_i, b_i) by computing the hash value

$$d_i = H(\mathrm{H}_I; \texttt{0x32}, \texttt{ind}_\texttt{c}(\Lambda), i, C_0, C_1, C_2, a_i, b_i, m_i, U) \tag{99}$$

and sends it to every other guardian G_j, $j \in U$, $j \neq i$. Only after having received a value d_j from every other participating guardian G_j, guardian G_i sends the pair (a_i, b_i) to every other guardian. Guardian G_i verifies that the values d_j have been correctly computed via equation (99) for every $j \in U$. If G_i does not receive (a_j, b_j) from a participating guardian G_j or if equation (99) does not hold for any $j \in U$, then guardian G_i halts the protocol and complains.

If none of the guardians complains, the values a_i and b_i are accumulated into

$$a = \left(\prod_{i \in U} a_i\right) \bmod p,\ \ b = \left(\prod_{i \in U} b_i\right) \bmod p \tag{100}$$

and the joint challenge value c is obtained as

$$c = H_q(\mathrm{H}_I; \texttt{0x33}, \texttt{ind}_\texttt{c}(\Lambda), C_0, C_1, C_2, a, b, \beta). \tag{101}$$

Each available guardian G_i responds to its challenge $c_i = (c \cdot w_i) \bmod q$ with

$$v_i = (u_i - c_i \hat{z}_i) \bmod q. \tag{102}$$

An accumulated response

$$v = \left(\sum_{i \in U} v_i\right) \bmod q \tag{103}$$

is computed and the decryption value β is published along with the proof (c, v).

Then decryption proceeds by computing the key $h = H(\mathrm{H}_I; \mathtt{0x26}, \mathtt{ind_c}(\Lambda), C_0, \beta)$ and then the encryption keys $k_1, k_2, \ldots, k_{b_\Lambda}$ as

$$k_i = \mathrm{HMAC}(h, \mathtt{b}(i, 4) \parallel \mathtt{Label} \parallel \mathtt{0x00} \parallel \mathtt{Context} \parallel \mathtt{b}(b_\Lambda \cdot 256, 4)) \tag{104}$$

with $\mathtt{Label} = \mathtt{b}(\text{“}\mathtt{data\_enc\_keys}\text{”}, 13)$ and $\mathtt{Context} = \mathtt{b}(\text{“}\mathtt{contest\_data}\text{”}, 12) \parallel \mathtt{b}(\mathtt{ind_c}(\Lambda), 4)$, and the byte array D representing the contest data string is decrypted by parsing C_1 in 32-byte blocks as

$$C_1 = C_{1,1} \parallel C_{1,2} \parallel \cdots \parallel C_{1,b_\Lambda} \tag{105}$$

and obtaining

$$D = C_{1,1} \oplus k_1 \parallel C_{1,2} \oplus k_2 \parallel \cdots \parallel C_{1,b_\Lambda} \oplus k_{b_\Lambda}. \tag{106}$$

The byte array D can now be parsed to reveal the captured overvote, undervote, null vote data, and write-in text fields.

Verification 12 (Correctness of decryptions of contest data)
An election verifier must confirm the correct decryption of the contest data field for each contest Λ by verifying the conditions analogous to Verification 10 for the corresponding NIZK proof with (A, B) replaced by (C_0, C_1, C_2) and M by β as follows. An election verifier must compute the following values.

(12.1) $a = (g^v \cdot \hat{K}^c) \bmod p$,
(12.2) $b = (C_0^v \cdot \beta^c) \bmod p$.

An election verifier must then confirm the following.

(12.A) The given value v is in the set $\mathbb{Z}_q$.
(12.B) The challenge value c satisfies $c = H_q(\mathrm{H}_I; \mathtt{0x33}, \mathtt{ind_c}(\Lambda), C_0, C_1, C_2, a, b, \beta)$.

An election verifier now must compute the values

(12.3) $h = H(\mathrm{H}_I; \mathtt{0x26}, \mathtt{ind_c}(\Lambda), C_0, \beta)$,
(12.4) $k_i = \mathrm{HMAC}(h, \mathtt{b}(i, 4) \parallel \mathtt{Label} \parallel \mathtt{0x00} \parallel \mathtt{Context} \parallel \mathtt{b}(b_\Lambda \cdot 256, 4))$

for $1 \leq i \leq b_\Lambda$ and verify correct decryption by using $C_1 = C_{1,1} \parallel C_{1,2} \parallel \cdots \parallel C_{1,b_\Lambda}$ and confirming that

(12.C) $D = C_{1,1} \oplus k_1 \parallel C_{1,2} \oplus k_2 \parallel \cdots \parallel C_{1,b_\Lambda} \oplus k_{b_\Lambda}$.

3.6.7 Decryption of Challenged Ballots

Each and every challenged ballot must be verifiably decrypted. Guardians could use their shares of the election secret key in precisely the way they do to decrypt election tallies. However, it is more efficient and makes it easier for a verifier if the available guardians instead use their shares of the ballot data encryption key to decrypt the ballot nonce ξ_B, which was encrypted as described in Section 3.3.4. Once the ballot nonce is available, the individual encryption nonces $\xi_{i,j}$ that were used to encrypt each selection on the challenged ballot as described in Section 3.3.3 (or the encryption nonce ξ used to encrypt contest data as described in Section 3.3.10) can be derived and published. The ballot nonce ξ_B should not be published.

Decryption of ballot nonces. Given the ciphertext $C_{\xi_B} = (C_{\xi_B,0}, C_{\xi_B,1}, C_{\xi_B,2})$, which is an encryption of the ballot nonce ξ_B, each available guardian G_i, $i \in U$, first verifies the validity of the Schnorr proof $C_{\xi_B,2} = (c_B, v_B)$. This is done by computing $a_B = (g^{v_B} \cdot C_{\xi_B,0}^{c_B}) \bmod p$ and verifying that $c_B = H_q(H_I; \mathtt{0x23}, a_B, C_{\xi_B,0}, C_{\xi_B,1})$ according to Equation (38).

Only if the proof is verified as correct, does guardian G_i use its ballot data encryption key share $\hat{z}_i$ to compute the partial decryption

$$m_i = C_{\xi_B,0}^{\hat{z}_i} \bmod p. \tag{107}$$

Partial decryptions from the available guardians are combined via the Lagrange coefficients to obtain

$$\beta_B = \left(\prod_{i \in U} m_i^{w_i}\right) \bmod p. \tag{108}$$

The ballot nonce can be decrypted by computing $h = H(\mathrm{H}_I; \mathtt{0x22}, C_{\xi_B,0}, \beta_B)$ via Equation (35) using β_B and from that deriving the encryption key via Equation (36) as $k_1 = \mathrm{HMAC}(h, \mathtt{0x01} \parallel \mathtt{Label} \parallel \mathtt{0x00} \parallel \mathtt{Context} \parallel \mathtt{0x0100})$, where $\mathtt{Label} = \mathtt{b}(\text{“}\mathtt{ballot\_nonce}\text{”}, 12)$ and $\mathtt{Context} = \mathtt{b}(\text{“}\mathtt{ballot\_nonce\_encrypt}\text{”}, 20)$. The ballot nonce can be computed as $\xi_B = C_{\xi_B,1} \oplus k_1$.

The desired encryption nonces are derived from ξ_B via Equation (33) for option encryptions (yielding $\xi_{i,j}$) or via Equation (64) for contest data (yielding ξ).

Decryption with encryption nonces. An encryption nonce $\xi_{i,j} \in \mathbb{Z}_q$ that was used to encrypt a selection σ as $(\alpha_{i,j}, \beta_{i,j}) = (g^{\xi_{i,j}} \bmod p, K^{\sigma + \xi_{i,j}} \bmod p)$ can be used to decrypt the ciphertext $(\alpha_{i,j}, \beta_{i,j})$ by computing

$$K^{\sigma} = (\beta_{i,j} \cdot K^{-\xi_{i,j}}) \bmod p. \tag{109}$$

Then, the small discrete logarithm problem is solved to obtain σ as is done for all other decryptions of ciphertexts that were encrypted usind DPP vote encryption.

To decrypt contest data given the contest data encryption nonce ξ, one computes the value

$$\beta = \hat{K}^{\xi} \bmod p, \tag{110}$$

then derives the secret key h using Equation (65), and computes the encryption keys $k_1, k_2, \ldots, k_{b_\Lambda}$ using Equation (66). These keys are then used to decrypt the contest data from the contest data ciphertext (C_0, C_1, C_2), where $C_1 = C_{1,1} \parallel C_{1,2} \parallel \cdots \parallel C_{1,b_\Lambda}$, by computing

$$D = C_{1,1} \oplus k_1 \parallel C_{1,2} \oplus k_2 \parallel \cdots \parallel C_{1,b_\Lambda} \oplus k_{b_\Lambda}. \tag{111}$$

Verifying decryption with nonces. When the ballot nonce ξ_B that was used to encrypt a ballot B is decrypted, a verifiable decryption can be produced by deriving the encryption nonces $\xi_{i,j}$ that have been used to encrypt the selections, then publishing these encryption nonces. When specific selections or contests are challenged, as it is common in the context of risk limiting audits targeting specific contests for instance, the publication can be restricted to the desired subset of encryption nonces, improving the privacy of the votes (this would not be possible if the ballot nonce were published instead). This approach can drastically reduce the amount of computation required of both guardians and verifiers.

Specifically, when possessing encryption nonces $\xi_{i,j}$ and the corresponding selections made on a ballot, a verification application can repeat the vote encryption process as described in §3.3 as follows.

- Using each available encryption nonce $\xi_{i,j}$, it recomputes the selection encryptions as shown in Equation (31) of Section 3.3.1.
- It derives the contest hashes as shown in Equation (70) of Section 3.4.1 using the selection encryptions, which are either recomputed when encryption nonces are available, or are given when the encryption nonces are not available.
- Finally, it recomputes the confirmation code H_C via the contest hashes as described in Equation (71) of Section 3.4.2. If the chaining field is not the byte array consisting of only `0x00`, more information must be provided to the verifier as specified by the chaining mode, see Section 3.4.4.

If the resulting confirmation code matches the confirmation code provided to the voter, then this is verification that the confirmation code corresponds to a ballot with votes for the indicated selections.

Note that confirmation codes *do not* include any of the zero-knowledge proofs of ballot correctness. So a verifier does not need to reproduce these zero-knowledge proofs. Only the actual encryptions of the selections must be regenerated and confirmed. As such, the success of the verification process described here *does not* guarantee that a ballot with a given confirmation code is valid: the valid encryptions must be accompanied by valid zero-knowledge proofs of ballot correctness.

Also, $\xi_{i,j}$ or ξ values that are inconsistent with recorded encryptions *do not* mean that the selection encryptions or contest data encryptions encrypt incorrect selections or contest data: they may only reflect that the ballot nonce encryption was incorrect. This, alone, would not prevent a correct tallying process, should the ballot be included in the tally.

An election verifier must confirm Verifications 13 and 14.

Verification 13 (Correctness of decryptions for challenged ballots)
For each challenged ballot B (which has m_B contests), an election verifier must confirm the correct decryption of the selection values and contest data as follows.
For each contest Λ_i (for $1 \leq i \leq m_B$ with m_i verifiable option fields) on the challenged ballot, an election verifier must verify correctness of the decrypted values $\sigma_{i,j}$ using the corresponding encryption nonces $\xi_{i,j}$ and correctness of the decrypted contest data D using the contest data encryption nonce ξ_i.
For all $1 \leq j \leq m_i$, it must recompute the selection encryptions

(13.1) $\alpha_{i,j} = g^{\xi_{i,j}} \bmod p$,
(13.2) $\beta_{i,j} = K^{\sigma_{i,j}+\xi_{i,j}} \bmod p$,

and from those (together with the encrypted contest data (C_0, C_1, C_2)), the contest hash

(13.3) $\chi_i = H(\mathrm{H}_I; \texttt{0x28}, \texttt{ind}_{\texttt{c}}(\Lambda_i), \alpha_{i,1}, \beta_{i,1}, \alpha_{i,2}, \beta_{i,2} \ldots, \alpha_{i,m_i}, \beta_{i,m_i}, C_0, C_1, C_2)$.

It must also recompute the encryption keys for the contest data via

(13.4) $\alpha = g^{\xi_i} \bmod p$,
(13.5) $\beta = \hat{K}^{\xi_i} \bmod p$,
(13.6) $h = H(\mathrm{H}_I; \texttt{0x26}, \texttt{ind}_{\texttt{c}}(\Lambda), \alpha, \beta)$.
(13.7) $k_l = \mathrm{HMAC}(h, \texttt{b}(l, 4) \parallel \texttt{Label} \parallel \texttt{0x00} \parallel \texttt{Context} \parallel \texttt{b}(b_\Lambda \cdot 256, 4))$ for $0 \leq l < b_\Lambda$.

An election verifier must then verify correct decryption of the contest data by using the decrypted data $D = D_1 \parallel D_2 \parallel \cdots \parallel D_{b_\Lambda}$ and the ciphertext C_1 and confirming that

(13.A) $C_1 = D_1 \oplus k_1 \parallel D_2 \oplus k_2 \parallel \cdots \parallel D_{b_\Lambda} \oplus k_{b_\Lambda}$.

Finally, it must confirm that the provided confirmation code is correct, i.e.,

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

where B_C is the chaining field for ballot B.
If only selected contests have been decrypted—as might be the case in an RLA setting—Verification step (13.B) uses contest hashes provided with or recomputed from the encrypted ballot for the contests that have not been decrypted.

Verification 14 (Validation of well-formedness and content of challenged ballots)
An election verifier must confirm that for each decrypted challenged ballot, the selections listed in text match the corresponding text in the election manifest.

(14.A) The contest text label occurs as a contest label in the list of contests for the ballot's ballot style in the election manifest.
(14.B) For each contest in the list of contests for the ballot's ballot style in the election manifest, the contest label appears as a contest label on the uncast pre-encrypted ballot.
(14.C) For each option in the contest, the option text label occurs as an option label for the contest in the election manifest.
(14.D) For each option text label listed for this contest in the election manifest, the option label occurs for an option in the decrypted challenged ballot.

An election verifier must also confirm that the challenged ballot is well-formed, i.e., for each contest on the challenged ballot, it must confirm the following.

(14.E) For each option in the contest, the selection σ is a valid value—usually either a 0 or a 1.
(14.F) The sum of all selections in the contest is at most the selection limit L for that contest.

3.7 The Election Record

The record of an election should be a full accounting of all of the election artifacts. Specifically, it should contain the following.

- Information sufficient to uniquely identify and describe the election, such as date, location, election type, etc. (not otherwise included in the election manifest).
- The election manifest file.
- The baseline parameters:
  - primes p and q and integer r such that $p = qr + 1$,
  - a generator g of the order q multiplicative subgroup $\mathbb{Z}_p^*$,
  - the number n of election guardians,
  - the quorum threshold k of guardians required to complete verification.
- The parameter base hash H_P computed from the parameters.
- The base hash value H_B computed from the above.
- The commitments from each election guardian to each of their polynomial coefficients.
- The proofs from each guardian of possession of each of the associated coefficients.
- The election public keys K and $\hat{K}$.
- The additional public key κ_i from each election guardian together with the proof of knowledge of the corresponding secret key.
- The extended base hash value H_E computed from the above.
- Every encrypted ballot prepared in the election (whether cast or challenged):
  - The selection encryption identifier id_B,
  - the selection encryption identifier hash H_I,
  - all of the encrypted selections on each ballot,
  - the proofs that each such value is an encryption of either zero or one (or more generally, the proofs that these values satisfy the respective option selection limits),

  - the selection limit for each contest,
  - the proof that the number of selections made does not exceed the selection limit,
  - the ballot weight if given—if no weight is given, the weight is assumed to be 1,
  - the ballot style,
  - the device information for the device that encrypted the ballot,
  - the date and time of the ballot encryption,
  - the confirmation code produced for the ballot,
  - the status of the ballot (cast or challenged).
- The decryption of each challenged ballot:
  - The selections made on the ballot,
  - the plaintext representation of the selections,
  - proofs of each decryption or decryption nonces.
- Tallies of each option in an election:
  - The encrypted tally of each option,
  - full decryptions of each encrypted tally,
  - plaintext representations of each tally,
  - proofs of correct decryption of each tally.
- Ordered lists of the ballots encrypted by each device.

The election record should also contain the encrypted contest data when such data is available.

An election record should be digitally signed by election administrators together with the date of the signature. The entire election record and its digital signature should be published and made available for full download by any interested individuals. Tools should also be provided for easy look up of confirmation codes by voters.

The exact organizational structure of the election record will be specified in a separate document.

4 Pre-Encrypted Ballots (Optional)

In typical use, ElectionGuard ballots are encrypted after the voter selections are known. However, there are several common scenarios in which it is desirable to perform the encryption before the selections of a voter are known. These include Vote-by-Mail, pre-printed ballots for use in precincts with central count or minimal in-precinct equipment, and back-ups for precincts which ordinarily perform encryption on demand. Ordinary and pre-encrypted ballots can be tallied together so that it is not revealed which votes came from which mode. Support for pre-encrypted ballots should be regarded as optional and may not be included in all implementations of ElectionGuard.

With pre-encrypted ballots, each possible selection on a ballot is individually encrypted in advance; and the selections made by the voter indicate which encryptions are used.

ElectionGuard requires two applications to support pre-encrypted ballots: a *ballot encrypting tool* to provide data to enable printing of blank ballots and a companion *ballot recording tool* to receive information about selections made on pre-printed ballots and produce data compatible with the ElectionGuard election record.

As Section 3.3.2 shows for regular ballots, each pre-encrypted ballot also has a 256-bit selection encryption identifier id_B that is chosen uniformly at random and a corresponding identifier hash $\mathrm{H}_I = H(\mathrm{H}_E; \mathtt{0x20}, \mathrm{id}_B)$.

4.1 Format of Pre-Encrypted Ballots

Each selectable option within each contest is associated with a vector Ψ of encryptions E_j—with one encryption for each selectable option within the contest. Selectable options in a contest have a unique order determined by their option indices in increasing order. Let i be the position of the selectable option in this ordering (not necessarily identical to its option index). In its *normal* form and for a contest with m selection options, this vector

$$\Psi_{i,m} = \langle E_1, E_2, \ldots, E_m \rangle \tag{112}$$

includes an encryption $E_i = (\alpha_i, \beta_i) = \mathrm{Enc}(1; \xi_i)$ of one in the vector position i associated with the selection made and encryptions $E_j = (\alpha_j, \beta_j) = \mathrm{Enc}(0, \xi_j)$ of zero in every other position $1 \leq j \leq m$, $j \neq i$, where ξ_i and the ξ_j are (pseudo-)random encryption nonces. For example, the vector $\Psi_{2,4}$ associated with selecting the second option in a contest with a total of $m = 4$ selection options is $\Psi_{2,4} = \langle \mathrm{Enc}(0; \xi_1), \mathrm{Enc}(1; \xi_2), \mathrm{Enc}(0; \xi_3), \mathrm{Enc}(0; \xi_4) \rangle$. This corresponds precisely to the standard ElectionGuard encryption of a vote for the same option. There is also a *null* form $\Psi_{0,m} = \langle \mathrm{Enc}(0; \xi_1), \mathrm{Enc}(0; \xi_2), \ldots, \mathrm{Enc}(0; \xi_m) \rangle$ which is the same form except that all values are encryptions of zero.

The principal difference between a pre-encrypted ballot and a standard ElectionGuard ballot is that while standard ElectionGuard has a single vector of encryptions for each contest, here we have a vector of encryptions for each selectable option in each contest (generally including the possibility of an undervote in which no selections are made). Another difference is that while a standard ElectionGuard contest encryption can contain multiple selections, each vector of pre-encryptions represents at most one selection. However, in a contest where a voter is allowed to make multiple selections, multiple pre-encryption vectors can be combined to form a single contest encryption vector with multiple encryptions of one that precisely matches the standard ElectionGuard format.

4.1.1 Selection Hash

Each pre-encrypted vector of a pre-encrypted ballot is hashed using the ElectionGuard hash function H (specified in detail in Section 5) to form a *selection hash.* For all selectable options, i.e., for each option at position i in the contest with $1 \leq i \leq m$, the hash value ψ_i of the selection vector $\Psi_{i,m} = \langle E_1, E_2, \ldots, E_m \rangle$ is computed as

$$\psi_i = H(\mathrm{H}_I; \mathtt{0x40}, \alpha_1, \beta_1, \alpha_2, \beta_2 \ldots, \alpha_m, \beta_m), \tag{113}$$

where $E_i = (\alpha_i, \beta_i)$ is an encryption of one and $E_j = (\alpha_j, \beta_j)$ is an encryption of zero for $j \neq i$.

In a contest with a selection limit of L, an additional L null vectors are hashed to obtain

$$\psi_{m+\ell} = H(\mathrm{H}_I; \mathtt{0x40}, \alpha_1, \beta_1, \alpha_2, \beta_2 \ldots, \alpha_m, \beta_m), \tag{114}$$

where all $E_i = (\alpha_i, \beta_i)$ are encyptions of zero and $1 \leq \ell \leq L$.

4.1.2 Contest Hash

All of the selection hashes within each contest will ultimately be hashed *in sorted order*[50] to form the *contest hash* of that contest. Each contest on a ballot has a unique position determined by the position of its contest index in the list of contest indices in increasing order. The contest hash for the l-th contest (with label Λ_l) on the ballot is computed as

$$\chi_l = H(\mathrm{H}_I; \mathtt{0x41}, \mathtt{ind}_{\mathtt{c}}(\Lambda_l), \psi_{\pi(1)}, \psi_{\pi(2)}, \ldots, \psi_{\pi(m+L)}), \tag{115}$$

where π is a permutation that represents the sorting of the selection hashes. This means that contests are *not* hashed in the order given by the contest indices, but instead $\pi(i) < \pi(j)$ implies that $\psi_{\pi(i)} < \psi_{\pi(j)}$ (when hash values are interpreted as integers in big endian byte order). The sorting is required so that the order of the selection hashes $\psi_1, \psi_2, \ldots, \psi_m$ does not reveal the contents of the encryptions that are used to generate the hashes.

4.1.3 Confirmation Code

While contest hashes for pre-encrypted ballots are computed from selection hashes, which differs from the standard scenario for ElectionGuard described in Section 3.4, the computation of confirmation codes aligns with the previous case. The *confirmation code* H_C of a pre-encrypted ballot is generated as the hash of all the contest hashes on the ballot in sequential order. If there are m_B contests on the ballot (in sequential order specified by their contest indices in the election manifest file), its confirmation code is computed as

$$\mathrm{H}_C = H(\mathrm{H}_I; \mathtt{0x42}, \chi_1, \chi_2, \ldots, \chi_{m_B}, \mathrm{B}_C). \tag{116}$$

[50] It is critical that these selection hashes be sorted in order to purge the information about which hash is associated with each selection.

4.1.4 Ballot Chaining

Use of the chaining field input byte array B_C is the same as described in Section 3.4.4 to enable ballot chaining on the device generating pre-encrypted ballots. In particular, if simple chaining is used, the chain is initialized with

$$\mathrm{H}_0 = H(\mathrm{H}_E; \texttt{0x42}, \mathrm{B}_{C,0}) \tag{117}$$

and the chain is closed with

$$\overline{\mathrm{H}} = H(\mathrm{H}_E; \texttt{0x42}, \overline{\mathrm{B}}_C), \tag{118}$$

where $\mathrm{B}_{C,0}$ and $\overline{\mathrm{B}}_C$ are defined as in Section 3.4.4. Note that the device information hash H_{DI} is computed as in Section 3.4.3, only with a different domain separation byte, i.e.,

$$\mathrm{H}_{DI} = H(H_E; \texttt{0x43}, S_{\mathrm{device}}). \tag{119}$$

Likewise, the hash computation for $\overline{B}_C$ has a different domain separation byte as well, i.e.,

$$\overline{\mathrm{B}}_C = \texttt{0x00000001} \parallel H(\mathrm{H}_E; \texttt{0x44}, \mathrm{H}_\ell, \mathrm{B}_{C,0}). \tag{120}$$

A pre-encrypted ballot's hash will typically be printed directly on the ballot. Ideally, two copies of the ballot hash will be included on each ballot with one remaining permanently with the ballot and the other in an immediately adjacent location on a removable tab. These removable ballot codes are intended to be retained by voters and can be used as an identifier to allow voters to look up their (encrypted) ballots in the election record and confirm that they include the proper *short codes* as described below.

4.1.5 Short Codes

In any instantiation of pre-encrypted ballots, an additional *hash trimming function* Ω must be provided. The hash trimming function takes as its input a selection hash, and its output is a *short code*. As an example, Ω could produce the last byte of its input in a specified form. For instance, a short code representation could be a pair of hex characters, a pair of letters from a 16-letter alphabet, a letter followed by a digit, or a three-digit number. The size of a short code does not need to be a single byte, but this is a convenient choice. Different vendors or jurisdictions might choose to distinguish themselves by using their own preferred short code formats, so the details of the short code format are intentionally left open. However, Ω must be completely specified in the election manifest so that a verifier can match its functionality.

The hash trimming function Ω associates each selection on a ballot with a short code. *The short codes on a ballot need not be unique.* However, it is required that the short codes within a contest be unique. If there is a collision of short codes within a contest, the ballot nonce should be changed to generate a new ballot. When a pre-encrypted ballot is presented to a voter, the short codes associated with each selection should be displayed beside the selection. If the ballot is cast by a voter, the short codes associated with selections made by the voter will be published as part of the election record.

Undervotes. In addition to the short codes for each possible selection, short codes are provided to indicate undervotes.[51] A short code for a null vote is generated from a vector of encryptions of zero.

[51] It may not be necessary to print undervote short codes on ballots.

In a contest in which the voter may select only one option there will be a single pre-encrypted null vote and associated short code to indicate that the voter did not make a selection in that contest. In general, the number of null votes and short codes for a contest should match the selection limit of the contest. So, for example, a contest in which a voter may make three selections should have three null short codes. The short codes corresponding to the selections made by each voter will be published in the election record, so the use of null short codes allows the election record to not reveal undervotes.

4.2 The Ballot Encrypting Tool

The encrypting tool for pre-encrypted ballots takes as input parameters

- the election manifest,
- a ballot style index,
- the cryptographic parameters and the election vote encryption key K,
- and a nonce encryption key (usually the election data encryption key $\hat{K}$).

The tool produces the following outputs—which can be used to construct a single pre-encrypted ballot.[52]

- An encryption of the ballot nonce used to encrypt the ballot, encrypted using the nonce encryption key,
- a selection hash value for each possible selection in the ballot style,
- for each contest, additional null hash values corresponding in number to the contest selection limit,
- a contest hash for each contest computed from the sorted list of selection hashes and null hashes for that contest,
- and a confirmation code consisting of a hash of all of the contest hashes on the ballot.

The encrypting tool operates as follows.

First, it samples uniformly, at random, a 256-bit selection encryption identifier id_B and computes the corresponding identifier hash $\mathrm{H}_I = H(\mathrm{H}_E; \texttt{0x20}, \mathrm{id}_B)$, just like for regular ballots as described in Section 3.3.2. Then, it generates a 256-bit ballot nonce ξ_B for the ballot and encrypts this nonce with the nonce encryption key provided as shown in Section 3.3.4.

Next, for each contest on the indicated ballot style, an encryption vector is produced for each selection within the contest (see Equation (112)). This encryption vector is deterministically derived from the ballot nonce ξ_B and consists of an encryption of one in the position corresponding to the selection and an encryption of zero in all other positions in the contest (see Equation (113)). Additionally, one or more null vectors consisting entirely of encryptions of zeros are produced (again deterministically from the ballot nonce) as in Equation (114). The number of null vectors should match the selection limit L of the contest. The selection hashes and null hashes (computed as described in Section 4.1.1) are then sorted numerically and hashed together (in sorted order) to produce the contest hash as shown in Equation (115) in Section 4.1.2.

Finally, the contest hashes are themselves hashed sequentially to form the ballot's confirmation

[52]Note that it will likely be desirable to construct a wrapper that can be called to produce data for a specified number of pre-encrypted ballots.

code according to Equation (116) in Section 4.1.3.

4.2.1 Deterministic Nonce Derivation

The process of deterministic encryption is guided by the ballot nonce ξ_B. Following the manifest discussed in Section 3.1.3, each contest has a unique index i and, within each contest, each possible selection has a unique index j. The nonce used within the i-th contest and within that the j-th selection vector to form the k-th encryption is

$$\xi_{i,j,k} = H_q(\mathrm{H}_I; \texttt{0x45}, i, j, k, \xi_B). \tag{121}$$

Note that the nonce $\xi_{i,j,k}$ will be used to encrypt a one whenever $j = k$ and a zero whenever $j \neq k$. Note also that some of the selection labels will represent null votes. If labels for null votes are not included within the manifest file, the sequence of indices should be extended accordingly.

The output of the encryption tool includes the encrypted ballot nonce (computed as described in Section 3.3.4), the selection hash corresponding to each selection on the ballot (including nulls), the contest hashes, and the confirmation code.

4.2.2 Using the Ballot Encrypting Tool

It is the responsibility of the entity that calls the encryption tool to produce short codes from selection hashes. This must be done in a deterministic, repeatable fashion using a hash trimming function Ω. As suggested above, one option is to use a human-friendly representation of the final byte (or bytes) of each selection hash. Another would be to use an ordinal integer to indicate where in the sorted order of selection hashes within each contest each selection falls. (For example, if a contest has four possible selections, the integers 1, 2, 3, 4, 5 could be placed beside each selection – including "none" to indicate the sorted position of each of the five selection hashes.) This flexibility allows vendors to distinguish themselves with different ballot presentations and for vendors and jurisdictions to choose presentations that best accommodate their voters.

Unless the short codes in a particular instantiation are quite long, it is likely that there will be occasional collisions of short codes within a contest. It is the responsibility of the entity that calls the ballot encryption tool to ensure that no ballot is produced with a short code that is repeated within a contest. If a collision is found, the caller simply discards this ballot data and calls the ballot encryption tool again. The caller may also choose to discard ballot data if a short code is repeated anywhere within a ballot or if two short codes—within a contest or across a ballot-—meet some definition of similarity. The caller is free to discard data and obtain a new ballot pre-encryption as often as it likes.[53]

4.3 The Ballot Recording Tool

The ballot recording tool receives an election manifest, an identifier for a ballot style, the decrypted ballot nonce ξ_B, and, for a cast ballot, all the selections made by the voter. The recording tool

[53]Note that a malicious encryption wrapper could bias the printed ballots by, for instance, only using encryptions in which a particular selection's hash is always numerically first within a contest. However, a malicious wrapper already knows the associations between the selection hashes, the short codes, and the actual selections, so it is not clear how a malicious wrapper could benefit from creating a bias.

uses the ballot nonce ξ_B to regenerate all of the encryptions on the ballot. For a cast ballot, the tool then isolates the pre-encryptions corresponding to the selections made by the voter and, using the encryption nonces derived from the ballot nonce, generates proofs of ballot-correctness as in standard ElectionGuard section 3.3.7.

Note that if a contest selection limit is greater than one, the recording tool homomorphically combines the selected pre-encryption vectors corresponding to the selections made to produce a single vector of encrypted selections. The selected pre-encryption vectors are combined by componentwise multiplication (modulo p), and the derived encryption nonces $\xi_{i,j,k}$ are added (modulo q) to create suitable nonces for this combined pre-encryption vector. These derived nonces will be necessary to form zero-knowledge proofs that the associated encryption vectors are well-formed.

For each uncast (implicitly or explicitly challenged) ballot, the recording tool returns the encryption nonces that enable the encryptions to be opened and checked. Releasing the individual encryption nonces instead of the ballot nonce enables selective decryption of specific contests only as explained for the standard ElectionGuard case in Section 3.6.7.

4.3.1 Using the Recording Tool

A wrapper for the recording tool takes one or more encrypted ballot nonces and obtains the decryption(s) by interacting with guardians, an administrator, or a local database and then calls the recording tool with each decrypted nonce and ballot style – and for a cast ballot, the voter selections.

If the ballot is a cast ballot, the wrapper then uses the hash trimming function Ω to compute the short codes for the selections made by the voter and posts in the election record the full set of selection hashes generated from *all* pre-encryption vectors (whether or not selected by the voter), the full pre-encryption vectors corresponding to the voter selections, the proofs that these selected pre-encryption vectors are well-formed, and the short codes for the selections made by the voter.

For an uncast ballot, the wrapper computes the short codes for all possible selections and posts in the election record the full set of pre-encryption vectors, selection hashes, and short codes for each possible selection.

4.4 The Election Record

Selection vectors generated from pre-encrypted ballots are indistinguishable from those produced by standard ElectionGuard.[54] However, the election record for each pre-encrypted ballot includes a significant amount of additional information. Specifically, for each cast ballot, the election record should contain

- the standard ElectionGuard encrypted ballot data consisting of selection vectors for each contest together with all the standard associated zero-knowledge proofs that the ballot is well-formed,

[54] Pre-encrypted ballots in an election record can easily be distinguished from ordinary ballots. However, ordinary ballots and pre-encrypted ballots contribute to the same tallies; so, no information about which votes came from which mode is revealed.

- the selection hashes for every option on the ballot (including null options) – sorted numerically within each contest, and
- the short codes and pre-encryption selection vectors associated with all selectable options (including null options) on the ballot made by the voter.

Note that in a contest with a selection limit of one, the selection vector will be identical to one of the pre-encryption selection vectors. However, when a contest has a selection limit greater than one, the resulting selection vector will be a product of multiple pre-encryption selection vectors.

For each uncast ballot, the ballot nonce for that ballot is published in the encryption record.

While the basic pre-encrypted ballots are identical to standard ElectionGuard ballots, their confirmation codes are computed differently. Unlike standard ElectionGuard ballots, pre-encrypted ballot confirmation codes are computed before any selections are made. The confirmation codes on pre-encrypted ballots are computed from the full set of pre-encryptions. This is the same whether the pre-encrypted ballot is cast or not.

4.4.1 Election Record Presentation

The presentation of data in the election record should be cognizant of the fact that there are two very different uses that may be made of this record. Individual voters will want to look up their own cast and uncast ballots and to easily review that they match their expectations. Election verifiers will want to verify the cryptographic artifacts associated with individual cast and uncast ballots and check their consistency as well as the consistency of the reported tallies.

It should therefore be possible for voters to see, in as clean a presentation as is feasible, the short codes associated with selections made on their cast ballots and the short codes associated with all possible selections on uncast ballots. The presentation of uncast ballots should match, as closely as feasible, the appearance of the physical uncast ballot as this presentation would facilitate comparison between the two.

For election verifiers, all of the cryptographic artifacts should be made available for verification. Verifiers should not only confirm the consistency of this additional data but also that this additional data is consistent with the cleaner data views made available to individual voters.

4.5 Verification of Pre-Encrypted Ballots

Every step of verification that applies to traditional ElectionGuard ballots also applies to pre-encrypted ballots – with the exception of the process for computing confirmation codes. However, there are some additional verification steps that must be applied to pre-encrypted ballots. Specifically, the following verifications should be done for every pre-encrypted cast ballot contained in the election record.

- The ballot confirmation code correctly matches the hash of all contest hashes on the ballot (listed sequentially).
- Each contest hash correctly matches the hash of all selection hashes (including null selection hashes) within that contest (sorted within each contest).
- All short codes shown to voters are correctly computed from selection hashes in the election record which are, in turn, correctly computed from the pre-encryption vectors published in

the election record.

- For contests with selection limit greater than 1, the selection vectors published in the election record match the product of the pre-encryptions associated with the short codes listed as selected.

The following verifications should be done for every pre-encrypted ballot listed in the election record as uncast.

- The ballot confirmation code correctly matches the hash of all contest hashes on the ballot (listed sequentially).
- Each contest hash correctly matches the hash of all selection hashes (including null selection hashes) within that contest (sorted within each contest).
- All short codes on the ballot are correctly computed from the selection hashes in the election record which are, in turn, correctly computed from the pre-encryption vectors published in the election record.
- The decryptions of all pre-encryptions correspond to the plaintext values indicated in the election manifest.

This means that an election including pre-encrypted ballots must be verified with the following verification items:

Verifications 1, 2, 3, and 4 pertain to verifying the election parameters and key generation. They remain the same, independent of whether pre-encrypted ballots are used for an election or not. As indicated in Section 2, these verification should be considered optional for an independent verifier.

Verification 5 must include all ballots in an election, i.e., if pre-encrypted ballots are used, this step must verify uniqueness of selection encryption identifiers for the full set of ballots, including the pre-encrypted ballots.

Verification 6 must be validated for all selection encryptions on all ballots, including all individual selection encryptions within the selection vectors on pre-encrypted ballots using the appropriate option selection limits.

Verification 7 confirms adherence to selection limits and must be validated for all contests on all ballots, including contests on pre-encrypted ballots with the corresponding contest selection limits. In addition, if the contest selection limit is greater than 1, the selection vectors that are published in the election record must be validated because they may be accumulated from several pre-encrypted selection vectors with corresponding short codes. Therefore, **Verification 15** must be validated.

Verification 15 (Validation of correct accumulation of selection vectors)
An election verifier must confirm the following for each contest on each pre-encrypted ballot.

(15.A) If the contest selection limit is greater than 1, the selection vector Ψ published in the election record for this contest has been correctly computed as a product of the individual selection vectors $\Psi_{i,m}$ that correspond to the short codes as selected by the voter.

Verification 8 is only used for regular ElectionGuard ballots. Confirmation codes for pre-encrypted ballots must be validated with **Verification 16**.

Verification 16 (Validation of confirmation codes in pre-encrypted ballots)
An election verifier must confirm the following for each pre-encrypted ballot.

(16.A) For each selection in each contest on the ballot and the corresponding selection vector $\Psi_{i,m} = \langle E_1, E_2, \ldots, E_m \rangle$ consisting of the selection encryptions $E_j = (\alpha_j, \beta_j)$, the selection hash ψ_i satisfies

$$\psi_i = H(\mathrm{H}_I; \texttt{0x40}, \alpha_1, \beta_1, \alpha_2, \beta_2, \ldots, \alpha_m, \beta_m).$$

(16.B) The contest hash χ_l for the contest with context index l for all $1 \leq l \leq m_B$ has been correctly computed from the selection hashes ψ_i as

$$\chi_l = H(\mathrm{H}_I; \texttt{0x41}, l, \psi_{\pi(1)}, \psi_{\pi(2)}, \ldots, \psi_{\pi(m+L)}),$$

where π is a permutation and $\psi_{\pi(1)} < \psi_{\pi(2)} < \cdots < \psi_{\pi(m+L)}$.

(16.C) The ballot confirmation code H_C has been correctly computed from the (sequentially ordered) contest hashes and the chaining field byte array B_C as

$$\mathrm{H}_C = H(\mathrm{H}_I; \texttt{0x42}, \chi_1, \chi_2, \ldots, \chi_{m_B}, \mathrm{B}_C).$$

(16.D) The voting device information hash has been correctly computed from the string S_{device} for the device the ballot was processed on as specified in the election manifest as

$$\mathrm{H}_{DI} = H(H_E; \texttt{0x43}, S_{\text{device}}).$$

(16.E) If the device used the no-chaining mode, the chaining field is $B_C = \texttt{0x00000000} \parallel \mathrm{H}_{DI}$.

If the device used the simple chaining mode, an election verifier must confirm the following three items.

(16.F) If the ballot is the j^{th} ballot processed on this device, $1 \leq j \leq \ell$, the chaining field byte array used to compute H_j is equal to $\mathrm{B}_{C,j} = \texttt{0x00000001} \parallel \mathrm{H}_{j-1}$.

The following items need only be verified once per voting device.

(16.G) The initial hash code H_0 satisfies $\mathrm{H}_0 = H(\mathrm{H}_E; \texttt{0x42}, \mathrm{B}_{C,0})$ and $\mathrm{B}_{C,0} = \texttt{0x00000001} \parallel \mathrm{H}_{DI}$.

(16.H) The final input byte array is $\overline{\mathrm{B}}_C = \texttt{0x00000001} \parallel H(\mathrm{H}_E; \texttt{0x44}, \mathrm{H}_\ell, \mathrm{B}_{C,0})$, where H_ℓ is the final confirmation code on this device, and the closing hash is correctly computed as $\overline{\mathrm{H}} = H(\mathrm{H}_E; \texttt{0x42}, \overline{\mathrm{B}}_C)$.

Additionally, for all pre-encrypted ballots, an election verifier must validate the correctness of the short codes on these ballots, i.e., it must validate **Verification 17**.

Verification 17 (Validation of short codes in pre-encrypted ballots)
An election verifier must confirm for every selectable option on every pre-encrypted ballot in the election record that the short code ω displayed with the selectable option satisfies

(17.A) $\omega = \Omega(\psi)$ where ψ is the selection hash associated with the selectable option.

Specifically, for cast ballots, this includes all short codes that are published in the election record whose associated selection hashes correspond to selection vectors that are accumulated to form tallies. For uncast ballots, this includes all selection vectors on the ballot.

Verification 9 of correct aggregation must be validated for each option in each contest including regular and pre-encrypted ballots.

Verifications 10 and 11 validate the correct decryptions of the tallies and the correctness of the tally contents and must be performed for all tallies, including those comprising accumulations of regular ballots, pre-encrypted ballots, or both.

Verification 12 is not validated for pre-encrypted ballots because contests on pre-encrypted ballots do not have encrypted contest data.

Finally, uncast pre-encrypted ballots take the place of challenged ballots and their decryptions are "opened" by releasing the encryption nonces $\xi_{i,j,k}$, which are derived from the ballot nonce ξ_B via Equation 121 after it has been decrypted as specified in Section 3.6.7. **Verification 13** is replaced by **Verification 18**. And to verify well-formedness of the uncast ballot, **Verification 14** is replaced by **Verification 19**.

Verification 18 (Correctness of encryptions for uncast pre-encrypted ballots)
For each uncast pre-encrypted ballot B (which has m_B contests), an election verifier must confirm that all encryptions are correct encryptions of the corresponding selections as follows.
For each contest Λ_i (for $1 \leq i \leq m_B$ with m_i verifiable option fields) on the uncast ballot, an election verifier must verify correctness of the encrypted selections using the corresponding encryption nonces $\xi_{i,j,k}$.
For all $1 \leq j \leq m_i$, it must recompute the selection encryption vector of the j^{th} selection, i.e., for all $1 \leq k \leq m_i$ it must recompute

(18.1) $\alpha_{i,j,k} = g^{\xi_{i,j,k}} \bmod p$,

(18.2) $\beta_{i,j,k} = K^{\delta_{j,k}+\xi_{i,j,k}} \bmod p$, where $\delta_{j,k} = 1$ if and only if $j = k$ and $\delta_{j,k} = 0$, otherwise.

From those ciphertexts it must compute the selection hashes

(18.3) $\psi_{i,j} = H(\mathrm{H}_I; \mathtt{0x40}, \alpha_{i,j,1}, \beta_{i,j,1}, \ldots, \alpha_{i,j,m_i}, \beta_{i,j,m_i})$,

and from those, the contest hash with selection hashes in sorted order as

(18.4) $\chi_i = H(\mathrm{H}_I; \mathtt{0x41}, \mathtt{ind}_{\mathtt{c}}(\Lambda_i), \psi_{i,\pi(1)}, \psi_{i,\pi(2)}, \ldots, \psi_{i,\pi(m_i)})$.

Finally, it must confirm that the provided confirmation code is correct, i.e.,

(18.A) $\mathrm{H}_C = H(\mathrm{H}_I; \mathtt{0x42}, \chi_1, \chi_2, \ldots, \chi_{m_B}, \mathrm{B}_C)$,

where B_C is the chaining field for ballot B.

Verification 19 (Validation of content of uncast pre-encrypted ballots)
An election verifier must confirm that for each uncast ballot, the selections listed in text match the corresponding text in the election manifest. For each contest on the ballot, it must verify the following.

(19.A) The contest text label occurs as a contest label in the list of contests for the ballot's ballot style in the election manifest.

(19.B) For each contest in the list of contests for the ballot's ballot style in the election manifest, the contest label appears as a contest label on the uncast pre-encrypted ballot.

(19.C) For each option in the contest, the option text label occurs as an option label for the contest in the election manifest.

(19.D) For each option text label listed for this contest in the election manifest, the option label occurs for an option on the uncast pre-encrypted ballot.

4.6 Hash-Trimming Functions

To allow vendors and jurisdictions to present distinct formats to their voters, the details of the hash-trimming function that produces short codes from full-sized hashes are not explicitly provided. However, to facilitate verification, a variety of possible hash-trimming functions are pre-specified here.

Two Hex Characters

- $\Omega_1(x) =$ final byte of x expressed as two hexadecimal characters.

Four Hex Characters

- $\Omega_2(x) =$ final two bytes of x expressed as four hexadecimal characters.

Letter-Digit

- $\Omega_3(x) =$ final byte of x expressed as a letter followed by a digit with $\{0, 1, \ldots, 255\}$ mapping to $\{A0, A1, ..., A9, B0, B1, ...B9, ..., Z0, Z1, ..., Z5\}$.

Digit-Letter

- $\Omega_4(x) =$ final byte of x expressed as a digit followed by a letter with $\{0, 1, \ldots, 255\}$ mapping to $\{0A, 0B, ..., 0Z, 1A, 1B, ...1Z, ..., 9A, 9B, ..., 9V\}$.

Number: 0-255

- $\Omega_5(x) =$ final byte of x expressed as a number with $\{0, 1, \ldots, 255\}$ mapping to $\{0, 1, \ldots, 255\}$ using the identity function.

Number: 1-256

- $\Omega_6(x) =$ final byte of x expressed as a number with $\{0, 1, \ldots, 255\}$ mapping to $\{1, 2, \ldots, 256\}$ by adding 1.

Number: 100-355

- $\Omega_7(x) =$ final byte of x expressed as a number with $\{0, 1, \ldots, 255\}$ mapping to $\{100, 101, \ldots, 355\}$ by adding 100.

Number: 101-356

- $\Omega_8(x) =$ final byte of x expressed as a number with $\{0, 1, \ldots, 255\}$ mapping to $\{101, 102, \ldots, 356\}$ by adding 101.

5 Hash Computation

The function H that is used throughout ElectionGuard is instantiated based on the hashed message authentication code HMAC. This section defines how to evaluate H. It first defines how inputs are represented as byte arrays and then how these inputs are used to compute hash values.

5.1 Input Data Representation

All inputs to the function H are byte arrays. A *byte* is a non-negative integer less than 2^8, i.e., an integer in the set $\mathcal{B} = \{0, 1, \dots, 255\}$. Its binary form consists of at most 8 bits. Its hexadecimal form consists of at most two hexadecimal characters. Here, the leading 0 characters are written out such that a byte always has exactly two hexadecimal characters. Therefore, $\mathcal{B}$ is represented as $\{\texttt{0x00}, \texttt{0x01}, \texttt{0x02}, \dots \texttt{0xFE}, \texttt{0xFF}\}$.

A *byte array* B of length m is an array of m bytes $\mathrm{b}_0, \mathrm{b}_1, \dots, \mathrm{b}_m \in \mathcal{B}$. It is represented by the concatenation[55] of the byte values as $\mathrm{B} = \mathrm{b}_0 \parallel \mathrm{b}_1 \parallel \cdots \parallel \mathrm{b}_m$. A byte array of length m consists of $8m$ bits. For $0 \leq i < 8m$, the i-th bit of the byte array B is the (i mod 8)-th bit of the byte $\mathrm{b}_{\lfloor i/8 \rfloor}$.[56] The set of all byte arrays of length exactly m is denoted by $\mathcal{B}^m$ and the set of all byte arrays of any finite length is denoted by $\mathcal{B}^*$.

Any byte array B of length m represents a non-negative (i.e., unsigned) integer less than 2^{8m} by interpreting the bytes of B as the digits of the representation in base 2^8 with the most significant bytes to the left, i.e., in big endian format. For example, the byte array $\mathrm{B} = \texttt{0x1F} \parallel \texttt{0xFF}$ of length 2 represents the integer 0x1FFF in hexadecimal form, which corresponds to the integer $2^{13} - 1 = 8191$. In general, let $\mathrm{b}_0 \parallel \mathrm{b}_1 \parallel \cdots \parallel \mathrm{b}_{m-1}$ be a byte array of length m, the integer

$$\mathrm{b}_0 \cdot 2^{(m-1)\cdot 8} + \mathrm{b}_1 \cdot 2^{(m-2)\cdot 8} + \cdots + \mathrm{b}_{m-1} \tag{122}$$

is a non-negative integer less than 2^{8m}. Vice versa, if $0 \leq a < 2^{8m}$, then the byte array of length m representing a is given as

$$\mathsf{b}(a, m) = \mathrm{b}_0 \parallel \mathrm{b}_1 \parallel \cdots \parallel \mathrm{b}_{m-1}, \text{ where } \mathrm{b}_i = \lfloor a/2^{8(m-1-i)} \rfloor \bmod 2^8. \tag{123}$$

In this document and if not specified otherwise, a byte array and big endian non-negative integers are used synonymously. However, byte arrays representing integer data types have a fixed length and must be padded with 0x00 bytes to the left. The byte array $\texttt{0x00} \parallel \texttt{0x1F} \parallel \texttt{0xFF}$ represents the same integer as $\mathrm{B} = \texttt{0x1FFF}$, but has length 3 instead of 2. For the integer data types used in ElectionGuard, this is laid out in detail in the following sections.

5.1.1 Integers Modulo the Large Prime p

Most inputs to H like public keys, vote encryptions and commitments for NIZK proofs consist of big integers modulo the large prime p, i.e., integers in the set $\mathbb{Z}_p = \{0, 1, \dots, p-1\}$. Any such element is represented in big endian format by a fixed size byte array of length exactly l_p, the length

[55] The symbol $\parallel$ simply denotes concatenation and does not mean that this symbol is inserted as a separator into the array in any way.

[56] Here, $\lfloor \cdot \rfloor$ denotes the floor function, which means that the result is obtained by rounding down. For a real number x, $\lfloor x \rfloor$ is the largest integer that is not larger than x.

of the byte array representing p. If p has 4096 bits like the prime given in the standard parameters in Section 3.1.1, this length is exactly 512. The conversion from an integer to a byte array works explicitly as follows. For $a \in \mathbb{Z}_p$, the byte array $\mathtt{b}(a, l_p)$ of length l_p representing a is defined as

$$\mathtt{b}(a, l_p) = \mathrm{b}_0 \parallel \mathrm{b}_1 \parallel \cdots \parallel \mathrm{b}_{l_p-1}, \tag{124}$$

where

$$\mathrm{b}_i = \lfloor a/2^{8(l_p-1-i)} \rfloor \bmod 2^8 \in \mathcal{B} \text{ for } i \in \{0, 1, \ldots, l_p - 1\}. \tag{125}$$

The bytes b_i are the coefficients of a when it is written in base 2^8, i.e.,

$$a = \sum_{i=0}^{l_p-1} \mathrm{b}_i 2^{8(l_p-1-i)} = \mathrm{b}_0 \cdot 2^{(l_p-1)\cdot 8} + \mathrm{b}_1 \cdot 2^{(l_p-2)\cdot 8} + \cdots + \mathrm{b}_{l_p-1}. \tag{126}$$

Any byte array $\mathtt{b}(a, l_p)$ that represents an integer a modulo p always has length l_p. This means that, for the standard parameters, where $l_p = 512$, we have $\mathtt{b}(a, 512) = \mathrm{b}_0 \parallel \mathrm{b}_1 \parallel \cdots \parallel \mathrm{b}_{511}$, where

$$\mathrm{b}_i = \lfloor a/2^{8(511-i)} \rfloor \bmod 2^8 \in \mathcal{B} \text{ for } i \in \{0, 1, \ldots, 511\}.$$

For example, $\mathtt{b}(0, 512) =$ `0x0000...000000000000` is a byte array of 512 `0x00`-bytes, $\mathtt{b}(15, 512) =$ `0x0000...00000000000F`, $\mathtt{b}(8572345, 512) =$ `0x0000...00000082CDB9`, and $\mathtt{b}(p - 1, 512)$ is the array shown in Section 3.1.1 for p, but ending with `...FFFFFFFFFFFFFE`.

5.1.2 Integers Modulo the Small Prime q

Other inputs are integers modulo the smaller prime q, such as response values in NIZK proofs and encryption nonces. They are integers in the set $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$ and are represented by a fixed size byte array of length exactly l_q in big endian format. For the standard parameters, $l_q = 32$. The conversion from integer to byte array works as above. If $a \in \mathbb{Z}_q$, the byte array $\mathtt{b}(a, 32)$ representing a is defined as

$$\mathtt{b}(a, 32) = \mathrm{b}_0 \parallel \mathrm{b}_1 \parallel \cdots \parallel \mathrm{b}_{31}, \tag{127}$$

where

$$\mathrm{b}_i = \lfloor a/2^{8(31-i)} \rfloor \bmod 2^8 \in \mathcal{B} \text{ for } i \in \{0, 1, \ldots, 31\}. \tag{128}$$

Again, the bytes are coefficients of a in its 2^8-adic form, namely

$$a = \sum_{i=0}^{31} \mathrm{b}_i 2^{8(31-i)} = \mathrm{b}_0 \cdot 2^{31\cdot 8} + \mathrm{b}_1 \cdot 2^{30\cdot 8} + \cdots + \mathrm{b}_{31}. \tag{129}$$

All byte arrays that represent integers modulo q have length 32. For example,

$$\begin{aligned}
\mathtt{b}(0, 32) &= \mathtt{0x0000000000000000000000000000000000000000000000000000000000000000},\\
\mathtt{b}(16, 32) &= \mathtt{0x0000000000000000000000000000000000000000000000000000000000000010},\\
\mathtt{b}(q - 1, 32) &= \mathtt{0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF42}.
\end{aligned}$$

5.1.3 Small Integers

Other integers such as *indices* are much smaller and are encoded as fixed length byte arrays in big endian format in the same way, but have much smaller lengths. In ElectionGuard, all such small integers are assumed to be smaller than 2^{31}. They can therefore be encoded with 4 bytes, i.e., with 32 bits and the most significant bit set to 0.[57]

The number of guardians n and the quorum threshold value k are examples of such small integers that require even less than 4 bytes for all reasonable use cases, i.e., they are represented as $\mathtt{b}(n, 4)$ and $\mathtt{b}(k, 4)$. For example, $\mathtt{b}(5, 4) =$ `0x00000005` and $\mathtt{b}(3, 4) =$ `0x00000003`.

5.1.4 Strings

When an input to the function H is a string s, it is encoded as a byte array $\mathtt{b}(s, \mathrm{len}(s))$ using UTF-8 encoding. Here, $\mathrm{len}(s)$ is the length of the UTF-8 encoding of the string s in bytes. For example, the string "ElectionGuard" is encoded as $\mathtt{b}(\text{"ElectionGuard"}, 13) =$ `0x456C656374696F6E4775617264`. Some string inputs may be of variable length that cannot be specified in advance by this document. Such inputs to the ElectionGuard hash function H start with a 4-byte encoding in big endian format of the byte length of the string encoding followed by the encoding itself.[58]

5.1.5 Files

When an input to the function H is a file `file`, it is parsed as input entirely, meaning that all bytes of the file are parsed to H as a byte array $\mathtt{b}(\mathtt{file}, \mathrm{len}(\mathtt{file}))$ that is a concatenation of all the bytes of the file in order. File lengths are not specified in this document and file inputs to H must start with a 4-byte encoding[59] in big endian format of the file byte length $\mathrm{len}(\mathtt{file})$ followed by the file bytes, i.e., $\mathtt{b}(\mathrm{len}(\mathtt{file}), 4) \parallel \mathtt{b}(\mathtt{file}, \mathrm{len}(\mathtt{file}))$. An example of this use case in ElectionGuard is the computation of the base hash H_B from the `manifest` file.

5.2 Hash Function

The hash function H used in ElectionGuard is HMAC-SHA-256, i.e., HMAC[60] instantiated with SHA-256[61]. Therefore, H takes two byte arrays as inputs.

The first input corresponds to the key in HMAC. The HMAC-SHA-256 specification allows arbitrarily long keys, but preprocesses the key to make it exactly 64 bytes (512 bits) long, which is the block size for the input to the SHA-256 hash function. If the given key is smaller, HMAC-SHA-256 pads it with `0x00` bytes at the end, if it is longer, HMAC-SHA-256 hashes it first with SHA-256

[57] Setting the most significant bit to 0 is done in consideration of languages and runtimes that do not have full support for unsigned integers.

[58] The maximal length of strings that can be encoded for input to H is therefore $2^{31} - 1$ bytes.

[59] Input files to H are restricted to a length of $2^{31} - 1$ bytes.

[60] NIST (2008) The Keyed-Hash Message Authentication Code (HMAC). In: FIPS 198-1. https://csrc.nist.gov/publications/detail/fips/198/1/final

[61] NIST (2015) Secure Hash Standard (SHS). In: FIPS 180-4. https://csrc.nist.gov/publications/detail/fips/180/4/final

and then pads it to 64 bytes. In ElectionGuard, all inputs that are used as the HMAC key, i.e., all inputs to the first argument of H, have a fixed length of exactly 32 bytes.

The second input can have arbitrary length and is only restricted by the maximal input length for SHA-256 and HMAC. Hence we view the function H formally as follows (understanding that HMAC implementations pad the 32-byte keys to exactly 64 bytes by appending 32 `0x00` bytes):

$$H : \mathcal{B}^{32} \times \mathcal{B}^{*} \to \mathcal{B}^{32},\ (\mathrm{B}_0, \mathrm{B}_1) \mapsto \text{HMAC-SHA-256}(\mathrm{B}_0, \mathrm{B}_1). \tag{130}$$

ElectionGuard uses HMAC not as a keyed hash function with a secret key or a message authentication code, but instead uses it as a general purpose hash function to implement a random oracle.[62]

The first input is used to bind hash values to a specific election by including the ElectionGuard version identifier `ver`, the parameter base hash H_P, the election base hash H_B, the extended base hash H_E, or the selection encryption identifier hash H_I. The second input consists of domain separation tags for different use cases and the actual data that is being hashed.

5.3 Hashing Multiple Inputs

ElectionGuard often requires that multiple input elements are hashed together. In fact, the second input to the function H specified in the previous section always consists of multiple parts. The input byte array B_1 is then simply the concatenation of the byte arrays that represent the multiple input elements as specified in Section 5.1 in the order specified in each case. As all byte arrays that represent input elements have a fixed length, there is no need for a separator byte or character.

The notation in this document lists the multiple input elements in order separated by commas. The first input element forming the byte array B_0 and the list of elements forming the second byte array B_1 are separated by a semicolon.

To illustrate the notation, consider, for example, the computation of the parameter hash $\mathrm{H}_P = H(\mathtt{ver}; \mathtt{0x00}, p, q, g, n, k)$, where $\mathtt{ver} = \mathtt{0x76322E312E30} \parallel \mathtt{b}(0, 26)$ (see Equation (4)). This notation means that HMAC-SHA-256 is called as HMAC-SHA-256$(\mathrm{B}_0, \mathrm{B}_1)$ with the byte arrays[63]

$$\mathrm{B}_0 = \mathtt{0x76322E312E30} \parallel \mathtt{b}(0, 26),\ \mathrm{B}_1 = \mathtt{0x00} \parallel \mathtt{b}(p, l_p) \parallel \mathtt{b}(q, l_q) \parallel \mathtt{b}(g, l_p) \parallel \mathtt{b}(n, 4) \parallel \mathtt{b}(k, 4)$$

as inputs. The first argument is simply the ElectionGuard version identifier `ver`. The second argument is the byte array that is the result of concatenating the domain separator byte `0x00` that identifies the use case and the byte arrays representing the large integers p, q, and g, and the small integers n and k.

5.4 Hash Function Outputs and Hashing to $\mathbb{Z}_q$

The output of SHA-256 and therefore H is a 256-bit string, which can be interpreted as a byte array of 32 bytes. As such, outputs of H are directly used as inputs to H again without any modifications.

[62] Dodis Y., Ristenpart T., Steinberger J., Tessaro S. (2012) *To Hash or Not to Hash Again? (In)differentiability Results for H^2 and* HMAC. This paper shows that HMAC used as a general purpose hash function with keys of a fixed length shorter than $d - 1$, where d is the block length in bits of the underlying hash function, is indifferentiable from a random oracle.

[63] The symbol $\parallel$ does not represent a separator symbol and is simply used to denote concatenation like for byte arrays as mentioned above.

In some cases, ElectionGuard requires a hash function with outputs that lie in the exponent space, i.e., in the set $\mathbb{Z}_q$. In particular, these outputs must be (close to) uniformly distributed in $\mathbb{Z}_q$. Such a hash function H_q can be defined based on the hash function H. To that end, an output byte array $\mathrm{b}_0 \parallel \mathrm{b}_1 \parallel \cdots \parallel \mathrm{b}_{31}$ is interpreted as a big endian base-2^8 representation of an integer a, which means that

$$a = \sum_{i=0}^{31} \mathrm{b}_i 2^{8(31-i)} = \mathrm{b}_0 \cdot 2^{31 \cdot 8} + \mathrm{b}_1 \cdot 2^{30 \cdot 8} + \cdots + \mathrm{b}_{31}. \tag{131}$$

With this interpretation, hash function outputs correspond to integers in the interval $[0, 2^{256} - 1]$. Note that, although this byte representation is identical to the byte representation of integers modulo q as described above, the integers corresponding to hash function outputs do not have to be smaller than $q = 2^{256} - 189$. However, in practice, it is highly unlikely that an integer in the interval $[2^{256} - 189, 2^{256} - 1]$ will occur. When picking a 256-bit integer uniformly at random, the probability that it is not smaller than q is negligibly small, namely $189/2^{256} < 2^{-248}$.

This allows the hash function H_q to be defined as[64]

$$H_q : \mathcal{B}^{32} \times \mathcal{B}^* \to \mathbb{Z}_q,\ (\mathrm{B}_0, \mathrm{B}_1) \mapsto H(\mathrm{B}_0, \mathrm{B}_1) \bmod q \tag{132}$$

and notation for hashing multiple inputs is identical in this document.

5.5 Domain Separation

The tables below list every use of the hash function H (and H_q) in this specification together with the specific domain separation input for the second argument. They explicitly provide the byte arrays B_0 and B_1 that are passed to HMAC as inputs to evaluate HMAC-SHA-256$(\mathrm{B}_0, \mathrm{B}_1)$ together with the length $\mathrm{len}(\mathrm{B}_1)$ of B_1 in bytes (note that always $\mathrm{len}(\mathrm{B}_0) = 32$). The tables use the standard parameters such that $l_p = 512$ and $l_q = 32$. For other parameters, the values l_p and l_q can be different.

[64]This construction of H_q by simply reducing the output from H modulo q is tailored to the specific choice of $q = 2^{256} - 189$. Should ElectionGuard be used with other parameters, the group order q might be much further away from 2^{256} and care must be taken to ensure that hashing to $\mathbb{Z}_q$ is done securely with a different approach.

5.5.1 Parameter Base and Base Hashes

Computation of the parameter base hash H_P §3.1.2

$\mathrm{H}_P = H(\mathtt{ver}; \mathtt{0x00}, p, q, g, n, k)$ (4)

$\mathrm{B}_0 = \mathtt{ver} = \mathtt{0x76322E312E30} \parallel \mathtt{b}(0, 26)$

$\mathrm{B}_1 = \mathtt{0x00} \parallel \mathtt{b}(p, 512) \parallel \mathtt{b}(q, 32) \parallel \mathtt{b}(g, 512) \parallel \mathtt{b}(n, 4) \parallel \mathtt{b}(k, 4)$

$\mathrm{len}(\mathrm{B}_1) = 1065$

Computation of the base hash H_B §3.1.4

$\mathrm{H}_B = H(\mathrm{H}_P; \mathtt{0x01}, \mathtt{manifest})$ (5)

$B_0 = \mathrm{H}_P$

$B_1 = \mathtt{0x01} \parallel \mathtt{b}(\mathrm{len}(\mathtt{manifest}), 4) \parallel \mathtt{b}(\mathtt{manifest}, \mathrm{len}(\mathtt{manifest}))$

$\mathrm{len}(B_1) = 5 + \mathrm{len}(\mathtt{manifest})$

5.5.2 Key Generation and Extended Base Hash

NIZK proof of knowledge of polynomial coefficients in key generation §3.2.2

$c_i = H_q(\mathrm{H}_P; \mathtt{0x10}, \text{“}\mathtt{pk\_vote}\text{”}, i, K_{i,0}, K_{i,1}, \ldots, K_{i,k-1}, \kappa_i, h_{i,0}, h_{i,1}, \ldots, h_{i,k-1}, h_{i,k})$ (11)

$\mathrm{B}_0 = \mathrm{H}_P$, $\mathrm{B}_1 = \mathtt{0x10} \parallel \mathtt{706B5F766F7465} \parallel \mathtt{b}(i, 4) \parallel \mathtt{b}(K_{i,0}, 512) \parallel \cdots \parallel \mathtt{b}(K_{i,k-1}, 512) \parallel$
$\mathtt{b}(\kappa_i, 512) \parallel \cdots \parallel \mathtt{b}(h_{i,k}, 512)$

$\mathrm{len}(\mathrm{B}_1) = 12 + 2(k+1) \cdot 512$

$\hat{c}_i = H_q(\mathrm{H}_P; \mathtt{0x10}, \text{“}\mathtt{pk\_data}\text{”}, i, \hat{K}_{i,0}, \hat{K}_{i,1}, \ldots, \hat{K}_{i,k-1}, \kappa_i, \hat{h}_{i,0}, \hat{h}_{i,1}, \ldots, \hat{h}_{i,k-1}, \hat{h}_{i,k})$ (13)

$\mathrm{B}_0 = \mathrm{H}_P$, $\mathrm{B}_1 = \mathtt{0x10} \parallel \mathtt{706B5F64617461} \parallel \mathtt{b}(i, 4) \parallel \mathtt{b}(\hat{K}_{i,0}, 512) \parallel \cdots \parallel \mathtt{b}(\hat{K}_{i,k-1}, 512) \parallel$
$\mathtt{b}(\kappa_i, 512) \parallel \cdots \parallel \mathtt{b}(\hat{h}_{i,k}, 512)$

$\mathrm{len}(\mathrm{B}_1) = 12 + 2(k+1) \cdot 512$

Encryption of key shares in key generation §3.2.2

$k_{i,\ell} = H(\mathrm{H}_P; \mathtt{0x11}, i, \ell, \kappa_\ell, \alpha_{i,\ell}, \beta_{i,\ell})$ (16)

$\mathrm{B}_0 = \mathrm{H}_P$, $\mathrm{B}_1 = \mathtt{0x11} \parallel \mathtt{b}(i, 4) \parallel \mathtt{b}(\ell, 4) \parallel \mathtt{b}(\kappa_\ell, 512) \parallel \mathtt{b}(\alpha_{i,\ell}, 512) \parallel \mathtt{b}(\beta_{i,\ell}, 512)$

$\mathrm{len}(\mathrm{B}_1) = 1545$

Challenge computation for encryption proof

$c = H_q(H_P; \mathtt{0x12}, i, \ell, \gamma_{i,\ell}, C_{i,\ell,0}, C_{i,\ell,1})$ (22)

$\mathrm{B}_0 = \mathrm{H}_P$, $\mathrm{B}_1 = \mathtt{0x12} \parallel \mathtt{b}(i, 4) \parallel \mathtt{b}(\ell, 4) \parallel \mathtt{b}(\gamma_{i,\ell}, 512) \parallel \mathtt{b}(C_{i,\ell,0}, 512) \parallel C_{i,\ell,1}$

$\mathrm{len}(\mathrm{B}_1) = 1097$

Comparison of different views of preliminary guardian record §3.2.2

$\mathrm{H}_G = \mathrm{H}_G = H(\mathrm{H}_B; \mathtt{0x13}, K, \hat{K}, K_{1,0}, \ldots, K_{1,k-1}, K_{2,0}, \ldots, K_{n,k-1},$ (27)
$\hat{K}_{1,0}, \ldots, \hat{K}_{1,k-1}, \hat{K}_{2,0}, \ldots, \hat{K}_{n,k-1}, \kappa_1, \kappa_2, \ldots, \kappa_n)$

$\mathrm{B}_0 = \mathrm{H}_B$

$\mathrm{B}_1 = \mathtt{0x13} \parallel \mathtt{b}(K, 512) \parallel \mathtt{b}(\hat{K}, 512) \parallel \mathtt{b}(K_{1,0}, 512) \parallel \cdots \parallel \mathtt{b}(\kappa_1, 512) \parallel \cdots \parallel \mathtt{b}(\kappa_n, 512)$

$\mathrm{len}(\mathrm{B}_1) = 1 + (2 + 2nk + n) \cdot 512$

Computation of the extended base hash H_E §3.2.3

$\mathrm{H}_E = H(\mathrm{H}_B; \mathtt{0x14}, K, \hat{K})$ (30)

$\mathrm{B}_0 = \mathrm{H}_B$, $\mathrm{B}_1 = \mathtt{0x14} \parallel \mathtt{b}(K, 512) \parallel \mathtt{b}(\hat{K}, 512)$

$\mathrm{len}(\mathrm{B}_1) = 1025$

5.5.3 Ballot Encryption and Confirmation Codes

Computation of the selection encryption identifier hash §3.3.2

$\mathrm{H}_I = H(H_E; \texttt{0x20}, \mathrm{id}_B)$ (32)

$\mathrm{B}_0 = \mathrm{H}_E$

$\mathrm{B}_1 = \texttt{0x20} \parallel \texttt{b}(\mathrm{id}_B, 32)$

$\mathrm{len}(\mathrm{B}_1) = 33$

Computation of encryption nonces §3.3.3

$\xi_{i,j} = H_q(\mathrm{H}_I; \texttt{0x21}, i, j, \xi_B)$ (33)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \texttt{0x21} \parallel \texttt{b}(i, 4) \parallel \texttt{b}(j, 4) \parallel \texttt{b}(\xi_B, 32)$

$\mathrm{len}(\mathrm{B}_1) = 41$

Key derivation for ballot nonce encryption § 3.3.4

$h = H(\mathrm{H}_I; \texttt{0x22}, \alpha_B, \beta_B)$ (35)

$\mathrm{B}_0 = \mathrm{H}_I,$

$\mathrm{B}_1 = \texttt{0x22} \parallel \texttt{b}(\alpha_B, 512) \parallel \texttt{b}(\beta_B, 512)$

$\mathrm{len}(\mathrm{B}_1) = 1025$

Challenge for ballot nonce encryption proof

$c = H_q(H_I; \texttt{0x23}, a_B, C_{\xi_B,0}, C_{\xi_B,1})$ (38)

$\mathrm{B}_0 = \mathrm{H}_I,$

$\mathrm{B}_1 = \texttt{0x23} \parallel \texttt{b}(a_B, 512) \parallel \texttt{b}(C_{\xi_B,0}, 512) \parallel C_{\xi_B,1}$

$\mathrm{len}(\mathrm{B}_1) = 1057$

Challenge computation for 0/1-range proofs for option λ in contest Λ §3.3.7

$c = H_q(\mathrm{H}_I; \texttt{0x24}, \texttt{ind}_\texttt{c}(\Lambda), \texttt{ind}_\texttt{o}(\lambda), \alpha, \beta, a_0, b_0, a_1, b_1)$ (41), (50)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \texttt{0x24} \parallel \texttt{b}(\texttt{ind}_\texttt{c}(\Lambda), 4) \parallel \texttt{b}(\texttt{ind}_\texttt{o}(\lambda), 4) \parallel \texttt{b}(\alpha, 512) \parallel \texttt{b}(\beta, 512) \parallel \texttt{b}(a_0, 512) \parallel \texttt{b}(b_0, 512) \parallel \texttt{b}(a_1, 512) \parallel \texttt{b}(b_1, 512)$

$\mathrm{len}(\mathrm{B}_1) = 3081$

Challenge computation for R-range proofs for contest Λ and option λ

$c = H_q(\mathrm{H}_I; \texttt{0x24}, \texttt{ind}_\texttt{c}(\Lambda), \texttt{ind}_\texttt{o}(\lambda), \alpha, \beta, a_0, b_0, a_1, b_1, \ldots, a_R, b_R)$ (59)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \texttt{0x24} \parallel \texttt{b}(\texttt{ind}_\texttt{c}(\Lambda), 4) \parallel \texttt{b}(\texttt{ind}_\texttt{o}(\lambda), 4) \parallel \texttt{b}(\alpha, 512) \parallel \texttt{b}(\beta, 512) \parallel \texttt{b}(a_0, 512) \parallel \texttt{b}(b_0, 512) \parallel \texttt{b}(a_1, 512) \parallel \texttt{b}(b_1, 512) \parallel \cdots \parallel \texttt{b}(a_R, 512) \parallel \texttt{b}(b_R, 512)$

$\mathrm{len}(\mathrm{B}_1) = 9 + (2R + 4) \cdot 512$

Challenge computation for selection limit proofs in contest Λ §3.3.8

$c = H_q(\mathrm{H}_I; \texttt{0x24}, \texttt{ind}_\texttt{c}(\Lambda), \bar{\alpha}, \bar{\beta}, a_0, b_0, a_1, b_1, \ldots, a_L, b_L)$ (62)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \texttt{0x24} \parallel \texttt{b}(\texttt{ind}_\texttt{c}(\Lambda), 4) \parallel \texttt{b}(\bar{\alpha}, 512) \parallel \texttt{b}(\bar{\beta}, 512) \parallel \texttt{b}(a_0, 512) \parallel \texttt{b}(b_0, 512) \parallel \cdots \parallel \texttt{b}(a_L, 512) \parallel \texttt{b}(b_L, 512)$

$\mathrm{len}(\mathrm{B}_1) = 5 + (2L + 4) \cdot 512$

Computation of encryption nonces for contest data for contest Λ §3.3.10

$\xi = H_q(\mathrm{H}_I; \texttt{0x25}, \texttt{ind}_\texttt{c}(\Lambda), \xi_B)$ (64)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \texttt{0x25} \parallel \texttt{b}(\texttt{ind}_\texttt{c}(\Lambda), 4) \parallel \texttt{b}(\xi_B, 32)$

$\mathrm{len}(\mathrm{B}_1) = 37$

Key derivation for contest data encryption

$h = H(\mathrm{H}_I; \texttt{0x26}, \texttt{ind}_\texttt{c}(\Lambda), \alpha, \beta)$ (65)

$\mathrm{B}_0 = \mathrm{H}_I,$

$\mathrm{B}_1 = \texttt{0x26} \parallel \texttt{b}(\texttt{ind}_\texttt{c}(\Lambda), 4) \parallel \texttt{b}(\alpha, 512) \parallel \texttt{b}(\beta, 512)$

$\mathrm{len}(\mathrm{B}_1) = 1029$

Challenge computation for contest data encryption proof

$c = H_q(H_I; \texttt{0x27}, \texttt{ind}_\texttt{c}(\Lambda), a, C_0, C_1)$ (69)

$\mathrm{B}_0 = \mathrm{H}_I,$

$\mathrm{B}_1 = \texttt{0x27} \parallel \texttt{b}(\texttt{ind}_\texttt{c}(\Lambda), 4) \parallel \texttt{b}(a, 512) \parallel \texttt{b}(C_0, 512) \parallel C_1$

$\mathrm{len}(\mathrm{B}_1) = 1029 + b_\Lambda \cdot 32$

Computation of contest hashes §3.4.1

$\chi_l = H(\mathrm{H}_I; \texttt{0x28}, l, \alpha_1, \beta_1, \alpha_2, \beta_2 \ldots, \alpha_{m_l}, \beta_{m_l}, C_0, C_1, C_2)$ (70)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \texttt{0x28} \parallel \texttt{b}(l, 4) \parallel \texttt{b}(\alpha_1, 512) \parallel \ldots \parallel \texttt{b}(\beta_{m_l}, 512) \parallel \texttt{b}(C_0, 512) \parallel C_1 \parallel \texttt{b}(C_2, 64)$

$\mathrm{len}(\mathrm{B}_1) = 69 + (2m_l + 1) \cdot 512 + b_\Lambda \cdot 32$

Computation of confirmation codes §3.4.2

$\mathrm{H}_C = H(\mathrm{H}_I; \texttt{0x29}, \chi_1, \chi_2, \ldots, \chi_{m_B}, \mathrm{B}_C)$ (71)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \texttt{0x29} \parallel \chi_1 \parallel \chi_2 \parallel \ldots \parallel \chi_{m_B} \parallel \mathrm{B}_C$

$\mathrm{len}(\mathrm{B}_1) = 37 + m_B \cdot 32$

Voting device information hash §3.4.3

$\mathrm{H}_{DI} = H(H_E; \texttt{0x2A}, S_{\mathrm{device}})$ (72)

$\mathrm{B}_0 = \mathrm{H}_E$

$\mathrm{B}_1 = \texttt{0x2A} \parallel \texttt{b}(\mathrm{len}(S_{\mathrm{device}}), 4) \parallel \texttt{b}(S_{\mathrm{device}}, \mathrm{len}(S_{\mathrm{device}}))$

$\mathrm{len}(\mathrm{B}_1) = 5 + \mathrm{len}(S_{\mathrm{device}})$

Hash chain initialization for ballot chaining §3.4.4

$\mathrm{H}_0 = H(\mathrm{H}_E; \texttt{0x29}, \mathrm{B}_{C,0})$ (74)

$\mathrm{B}_0 = \mathrm{H}_E$

$\mathrm{B}_1 = \texttt{0x29} \parallel \mathrm{B}_{C,0}$

$\mathrm{len}(\mathrm{B}_1) = 37$

Hash chain closing for ballot chaining

$\overline{\mathrm{H}} = H(\mathrm{H}_E; \texttt{0x29}, \overline{\mathrm{B}}_C)$ (77)

$\mathrm{B}_1 = \texttt{0x29} \parallel \overline{\mathrm{B}}_C = \texttt{0x29} \parallel \texttt{0x00000001} \parallel H(\mathrm{H}_E; \texttt{0x2B}, \mathrm{H}_\ell, \mathrm{B}_{C,0})$

$\mathrm{len}(\mathrm{B}_1) = 37$

Computation of $\overline{\mathrm{B}}_C$

$\overline{\mathrm{B}}_C = \texttt{0x00000001} \parallel H(\mathrm{H}_E; \texttt{0x2B}, \mathrm{H}_\ell, \mathrm{B}_{C,0})$ (78)

$\mathrm{B}_0 = \mathrm{H}_E$

$\mathrm{B}_1 = \texttt{0x2B} \parallel \mathrm{H}_\ell \parallel \mathrm{B}_{C,0}$

$\mathrm{len}(\mathrm{B}_1) = 69$

5.5.4 Verifiable Decryption

Guardian commitments for tally decryption proof for option λ in contest Λ §3.6.5

$d_i = H(\mathrm{H}_E; \texttt{0x30}, \texttt{ind}_\texttt{c}(\Lambda), \texttt{ind}_\texttt{o}(\lambda), i, A, B, a_i, b_i, M_i, U)$ (88)

$\mathrm{B}_0 = \mathrm{H}_E$

$\mathrm{B}_1 = \texttt{0x30} \parallel \texttt{b}(\texttt{ind}_\texttt{c}(\Lambda), 4) \parallel \texttt{b}(\texttt{ind}_\texttt{o}(\lambda), 4) \parallel \texttt{b}(i, 4) \parallel \texttt{b}(A, 512) \parallel \texttt{b}(B, 512) \parallel \texttt{b}(a_i, 512) \parallel \texttt{b}(b_i, 512) \parallel \texttt{b}(M_i, 512) \parallel \texttt{b}(\#U, 4) \parallel \texttt{b}(j_1, 4) \parallel \cdots \parallel \texttt{b}(j_{\#U}, 4)$

$\mathrm{len}(\mathrm{B}_1) = 2577 + 4 \cdot \#U$

Challenge computation for proofs of correct tally decryption for option λ in contest Λ

$c = H_q(\mathrm{H}_E; \texttt{0x31}, \texttt{ind}_\texttt{c}(\Lambda), \texttt{ind}_\texttt{o}(\lambda), A, B, a, b, M)$ (90)

$\mathrm{B}_0 = \mathrm{H}_E$

$\mathrm{B}_1 = \texttt{0x31} \parallel \texttt{b}(\texttt{ind}_\texttt{c}(\Lambda), 4) \parallel \texttt{b}(\texttt{ind}_\texttt{o}(\lambda), 4) \parallel \texttt{b}(A, 512) \parallel \texttt{b}(B, 512) \parallel \texttt{b}(a, 512) \parallel \texttt{b}(b, 512) \parallel \texttt{b}(M, 512)$

$\mathrm{len}(\mathrm{B}_1) = 2569$

Guardian commitments for decryption proofs of contest data for contest Λ §3.6.6

$d_i = H(\mathrm{H}_I; \texttt{0x32}, \texttt{ind}_\texttt{c}(\Lambda), i, C_0, C_1, C_2, a_i, b_i, m_i, U)$ (99)

$\mathrm{B}_0 = \mathrm{H}_E$

$\mathrm{B}_1 = \texttt{0x32} \parallel \texttt{b}(\texttt{ind}_\texttt{c}(\Lambda), 4) \parallel \texttt{b}(i, 4) \parallel \texttt{b}(C_0, 512) \parallel C_1 \parallel \texttt{b}(C_2, 64) \parallel \texttt{b}(a_i, 512) \parallel \texttt{b}(b_i, 512) \parallel \texttt{b}(m_i, 512) \parallel \texttt{b}(\#U, 4) \parallel \texttt{b}(j_1, 4) \parallel \cdots \parallel \texttt{b}(j_{\#U}, 4)$

$\mathrm{len}(\mathrm{B}_1) = 2125 + 32 \cdot b_\Lambda + 4 \cdot \#U$

Challenge computation for proofs of correct decryption of contest data

$c = H_q(\mathrm{H}_I; \texttt{0x33}, \texttt{ind}_\texttt{c}(\Lambda), C_0, C_1, C_2, a, b, \beta)$ (101)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \texttt{0x33} \parallel \texttt{b}(\texttt{ind}_\texttt{c}(\Lambda), 4) \parallel \texttt{b}(C_0, 512) \parallel C_1 \parallel \texttt{b}(C_2, 64) \parallel \texttt{b}(a, 512) \parallel \texttt{b}(b, 512) \parallel \texttt{b}(\beta, 512)$

$\mathrm{len}(\mathrm{B}_1) = 2117 + 32 \cdot b_\Lambda$

5.5.5 Pre-Encrypted Ballots

Computation of selection hashes for pre-encrypted ballots §4.1.1

$\psi_i = H(\mathrm{H}_I; \mathtt{0x40}, \alpha_1, \beta_1, \alpha_2, \beta_2 \ldots, \alpha_m, \beta_m)$ (113)

$\psi_{m+\ell} = H(\mathrm{H}_I; \mathtt{0x40}, \alpha_1, \beta_1, \alpha_2, \beta_2 \ldots, \alpha_m, \beta_m)$ (114)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \mathtt{0x40} \parallel \mathtt{b}(\alpha_1, 512) \parallel \mathtt{b}(\beta_1, 512) \parallel \mathtt{b}(\alpha_2, 512) \parallel \ldots \parallel \mathtt{b}(\alpha_m, 512) \parallel \mathtt{b}(\beta_m, 512)$

$\mathrm{len}(\mathrm{B}_1) = 1 + 2m \cdot 512$

Computation of contest hashes for pre-encrypted ballots for contest Λ_l §4.1.2

$\chi_l = H(\mathrm{H}_I; \mathtt{0x41}, \mathtt{ind_c}(\Lambda_l), \psi_{\pi(1)}, \psi_{\pi(2)}, \ldots, \psi_{\pi(m+L)})$ (115)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \mathtt{0x41} \parallel \mathtt{b}(\mathtt{ind_c}(\Lambda_l), 4) \parallel \psi_{\pi(1)} \parallel \psi_{\pi(2)} \parallel \ldots \parallel \psi_{\pi(m+L)}$

$\mathrm{len}(\mathrm{B}_1) = 5 + (m + L) \cdot 32$

Computation of ballot hashes for pre-encrypted ballots §4.1.3

$\mathrm{H}_C = H(\mathrm{H}_I; \mathtt{0x42}, \chi_1, \chi_2, \ldots, \chi_{m_B}, \mathrm{B}_C)$ (116)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \mathtt{0x42} \parallel \chi_1 \parallel \chi_2 \parallel \ldots \parallel \chi_{m_B} \parallel \mathrm{B}_C$

$\mathrm{len}(\mathrm{B}_1) = 37 + m_B \cdot 32$

Hash chain initialization for ballot chaining §4.1.4

$\mathrm{H}_0 = H(\mathrm{H}_E; \mathtt{0x42}, \mathrm{B}_{C,0})$ (117)

$\mathrm{B}_0 = \mathrm{H}_E$

$\mathrm{B}_1 = \mathtt{0x42} \parallel \mathrm{B}_{C,0}$

$\mathrm{len}(\mathrm{B}_1) = 37$

Hash chain closing for ballot chaining

$\overline{\mathrm{H}} = H(\mathrm{H}_E; \mathtt{0x42}, \overline{\mathrm{B}}_C)$ (118)

$\mathrm{B}_1 = \mathtt{0x42} \parallel \overline{\mathrm{B}}_C = \mathtt{0x42} \parallel \mathtt{0x00000001} \parallel H(\mathrm{H}_E; \mathtt{0x44}, \mathrm{H}_\ell, \mathrm{B}_{C,0})$

$\mathrm{len}(\mathrm{B}_1) = 37$

Device information hash

$\mathrm{H}_{DI} = H(H_E; \mathtt{0x43}, S_{\mathrm{device}})$ (119)

$\mathrm{B}_0 = \mathrm{H}_E$

$\mathrm{B}_1 = \mathtt{0x43} \parallel \mathtt{b}(\mathrm{len}(S_{\mathrm{device}}), 4) \parallel \mathtt{b}(S_{\mathrm{device}}, \mathrm{len}(S_{\mathrm{device}}))$

$\mathrm{len}(\mathrm{B}_1) = 5 + \mathrm{len}(S_{\mathrm{device}})$

Computation of $\overline{\mathrm{B}}_C$

$\overline{\mathrm{B}}_C = \mathtt{0x00000001} \parallel H(\mathrm{H}_E; \mathtt{0x44}, \mathrm{H}_\ell, \mathrm{B}_{C,0})$ (120)

$\mathrm{B}_0 = \mathrm{H}_E$

$\mathrm{B}_1 = \mathtt{0x44} \parallel \mathtt{0x4C4F434B} \parallel \mathrm{H}_\ell \parallel \mathrm{B}_{C,0}$

$\mathrm{len}(\mathrm{B}_1) = 69$

Computation of encryption nonces §4.2.1

$\xi_{i,j,k} = H_q(\mathrm{H}_I; \mathtt{0x45}, i, j, k, \xi_B)$ (121)

$\mathrm{B}_0 = \mathrm{H}_I$

$\mathrm{B}_1 = \mathtt{0x45} \parallel \mathtt{b}(i, 4) \parallel \mathtt{b}(j, 4) \parallel \mathtt{b}(k, 4) \parallel \mathtt{b}(\xi_B, 32)$

$\mathrm{len}(\mathrm{B}_1) = 45$

6 Verifier Construction

While it is desirable for anyone who may construct an ElectionGuard verifier to have as complete an understanding as possible of the ElectionGuard design, this section isolates the items which must be verified and maps the variables used in the specification equations herein to the labels provided in the artifacts produced in an election.

6.1 Implementation Details

There are four operations which must be performed—all on very large integer values: modular addition, modular multiplication, modular exponentiation, and hash computations, i.e., evaluations of the hash function H. These operations can be performed using a programming language that provides native support, by importing tools to perform these large integer operations, or by implementing these operations from scratch.

Modular Addition

To compute $(a+b) \bmod n$, one can compute $((a \bmod n)+(b \bmod n)) \bmod n$. However, this is rarely beneficial. If it is known that $a, b \in \mathbb{Z}_n$, then one can choose to avoid the division normally inherent in the modular reduction and just use $(a+b) \bmod n = a+b$ (if $a+b < n$) or $a+b-n$ (if $a+b \geq n$).

Modular Multiplication

To compute $(a \cdot b) \bmod n$, one can compute $((a \bmod n) \cdot (b \bmod n)) \bmod n$. Unless it is already known that $a, b \in \mathbb{Z}_n$, it is usually beneficial to perform modular reduction on these intermediate values before computing the product. However, it is still necessary to perform modular reduction on the result of the multiplication.

Modular Exponentiation

To compute $a^b \bmod n$, one can compute $(a \bmod n)^b \bmod n$, but one should not perform a modular reduction on the exponent.[65] One should, however, never attempt to compute the exponentiation a^b before performing a modular reduction as the number a^b would likely contain more digits than there are particles in the universe. Instead, one should use a special-purpose modular exponentiation method such as a repeated square-and-multiply algorithm which prevents intermediate values from growing excessively large. Some languages include a native modular exponentiation primitive, but when this is not available a specialized modular exponentiation tool can be imported or written from scratch.

[65] In general, if n is prime, one can compute $a^b \bmod n$ as $(a \bmod n)^{(b \bmod (n-1))} \bmod n$. In the particular instance of this specification, if $a \in \mathbb{Z}_p^r$, then one can compute $a^b \bmod p$ as $a^{(b \bmod q)} \bmod p$.

6.2 Validation Steps

This section collects the steps to validate an election and, for each step, lists the involved variables together with an explanation and pointers to how they are introduced in the specification.

6.2.1 Parameter Verification

Verification 1 (Parameter validation)
An ElectionGuard election verifier must verify that it uses the correct version of the ElectionGuard specification, that it uses the standard baseline parameters, which may be hardcoded, and that the base hash values have been computed correctly.

(1.A) The ElectionGuard specification version used to generate the election record is the same as the ElectionGuard specification version used to verify the election record. This specification is version 2.1.0.

(1.B) The large prime is equal to the large modulus p defined in Section 3.1.1.

(1.C) The small prime is equal to the prime q defined in Section 3.1.1.

(1.D) The generator is equal to the generator g defined in Section 3.1.1.

(1.E) The parameter base hash has been computed correctly as

$$\mathrm{H}_P = H(\texttt{ver}; \texttt{0x00}, p, q, g, n, k)$$

using the version byte array $\texttt{ver} = \texttt{0x76322E312E30} \parallel \texttt{b}(0, 26)$, which is the UTF-8 encoding of the version string "v2.1.0" padded with 0x00-bytes to length 32 bytes.

(1.F) The base hash has been computed correctly as

$$\mathrm{H}_B = H(\mathrm{H}_P; \texttt{0x01}, \texttt{manifest}).$$

variable	description	
p	4096-bit prime modulus	§3.1.1
q	256-bit prime order of subgroup of $\mathbb{Z}_p^*$	
r	Cofactor of q in $p-1$	
g	Generator of subgroup of $\mathbb{Z}_p^*$ of order q	
n	Total number of guardians	§3.1.2
k	Quorum of guardians required to decrypt	
H_P	Parameter base hash	
`manifest`	Election manifest	§3.1.3
H_B	Base hash	§3.1.4

6.2.2 Key Ceremony Verification

Verification 2 (Guardian public-key validation)
For each guardian G_i, $1 \le i \le n$ an election verifier must compute the values

(2.1) $h_{i,j} = (g^{v_{i,j}} \cdot K_{i,j}^{c_i}) \bmod p$ for each $0 \le j < k$,
(2.2) $h_{i,k} = (g^{v_{i,k}} \cdot \kappa_i^{c_i}) \bmod p$,
(2.3) $\hat{h}_{i,j} = (g^{\hat{v}_{i,j}} \cdot \hat{K}_{i,j}^{\hat{c}_i}) \bmod p$ for each $0 \le j < k$,
(2.4) $\hat{h}_{i,k} = (g^{\hat{v}_{i,k}} \cdot \kappa_i^{\hat{c}_i}) \bmod p$,

and then must confirm the following.

(2.A) The values $K_{i,j}$ and $\hat{K}_{i,j}$, for $0 \le j < k$, and κ_i are in $\mathbb{Z}_p^r$. (A value x is in $\mathbb{Z}_p^r$ if and only if x is an integer such that $0 \le x < p$ and $x^q \bmod p = 1$ is satisfied.)
(2.B) The values $v_{i,j}$ and $\hat{v}_{i,j}$, for $0 \le j \le k$ are in $\mathbb{Z}_q$. (A value x is in $\mathbb{Z}_q$ if and only if x is an integer such that $0 \le x < q$.)
(2.C) The challenge c_i is correctly computed as

$$c_i = H_q(\mathrm{H}_P; \texttt{0x10}, \text{“}\texttt{pk\_vote}\text{”}, i, K_{i,0}, K_{i,1}, \ldots, K_{i,k-1}, \kappa_i, h_{i,0}, \ldots, h_{i,k-1}, h_{i,k})$$

and the challenge $\hat{c}_i$ is correctly computed as

$$\hat{c}_i = H_q(\mathrm{H}_P; \texttt{0x10}, \text{“}\texttt{pk\_data}\text{”}, i, \hat{K}_{i,0}, \hat{K}_{i,1}, \ldots, \hat{K}_{i,k-1}, \kappa_i, \hat{h}_{i,0}, \ldots, \hat{h}_{i,k-1}, \hat{h}_{i,k}).$$

variable	description	
p	4096-bit prime modulus	§3.1.1
q	256-bit prime order of subgroup of $\mathbb{Z}_p^*$	
r	Cofactor of q in $p-1$	
g	Generator of subgroup of $\mathbb{Z}_p^*$ of order q	
n	Total number of guardians	§3.1.2
k	Quorum of guardians required to decrypt	
H_P	Parameter base hash	
i	Sequence number of sharing guardian G_i	§3.2.2
j	Sequence number of receiving guardian G_j	
$K_{i,j}$, $\hat{K}_{i,j}$	Public forms of random coefficients $a_{i,j}$, $\hat{a}_{i,j}$	
κ_i	Additional public key	
$h_{i,j}$, $\hat{h}_{i,j}$	Commitment in Schnorr proof of knowledge of coefficient $a_{i,j}$, $\hat{a}_{i,j}$ or ζ_i	
c_i, $\hat{c}_i$	Challenge value in proof	
$v_{i,j}, \hat{v}_{i,j}$	Response value in proof	

Verification 3 (Election public-key validation)
An election verifier must verify the correct computation of the joint vote encryption public key and the joint ballot data encryption public key.
(3.A) $K = (\prod_{i=1}^{n} K_i) \bmod p$.
(3.B) $\hat{K} = (\prod_{i=1}^{n} \hat{K}_i) \bmod p$.

variable	description	
p	4096-bit prime modulus	§3.1.1
n	Total number of guardians	§3.1.2
K	Joint vote encryption public key	§3.2.2
$\hat{K}$	Joint ballot data encryption public key	
i	Sequence number of sharing guardian G_i	
K_i	Public vote encryption key of guardian G_i	
$\hat{K}_i$	Public ballot data encryption key of guardian G_i	

6.2.3 Extended Base Hash Verification

Verification 4 (Extended base hash validation)
An election verifier must verify the correct computation of the extended base hash.
(4.A) $\mathrm{H}_E = H(\mathrm{H}_B; \texttt{0x14}, K, \hat{K})$.

variable	description	
H_B	Base hash	§3.1.4
K	Joint vote encryption public key	§3.2.2
$\hat{K}$	Joint ballot data encryption public key	
H_E	Extended base hash	§3.2.3

6.2.4 Ballot Verification

Verification 5 (Uniqueness of selection encryption identifiers)
An election verifier must verify the following.

(5.A) There are no duplicate selection encryption identifiers, i.e., among the set of submitted (cast and challenged) ballots, no two have the same selection encryption identifiers.

For each ballot (cast and challenged), an election verifier must verify the following.

(5.B) The selection encryption identifier hash H_I has been correctly computed as

$$\mathrm{H}_I = H(\mathrm{H}_E; \texttt{0x20}, \mathrm{id}_B).$$

variable	description	
H_E	Extended base hash	§3.2.3
id_B	Selection encryption identifier for ballot B	§3.3.2
H_I	Selection encryption identifier hash	

Verification 6 (Well-formedness of selection encryptions)
For each selectable option λ within each contest Λ on each cast ballot, an election verifier must, on input the encrypted selection (α, β), compute the values

(6.1) $a_j = (g^{v_j} \cdot \alpha^{c_j}) \bmod p$ for all $0 \leq j \leq R$,
(6.2) $b_j = (K^{w_j} \cdot \beta^{c_j}) \bmod p$, where $w_j = (v_j - jc_j) \bmod q$ for all $0 \leq j \leq R$,
(6.3) $c = H_q(\mathrm{H}_I; \mathtt{0x24}, \mathtt{ind}_{\mathtt{c}}(\Lambda), \mathtt{ind}_{\mathtt{o}}(\lambda), \alpha, \beta, a_0, b_0, a_1, b_1, \ldots, a_R, b_R)$,

where R is the option selection limit. An election verifier must then confirm the following:

(6.A) The given values α and β are in the set $\mathbb{Z}_p^r$.
(A value x is in $\mathbb{Z}_p^r$ if and only if x is an integer such that $0 \leq x < p$ and $x^q \bmod p = 1$.)
(6.B) The given values c_j are each in the set $\mathbb{Z}_q$ for all $0 \leq j \leq R$.
(A value x is in $\mathbb{Z}_q$ if and only if x is an integer such that $0 \leq x < q$.)
(6.C) The given values v_j are each in the set $\mathbb{Z}_q$ for all $0 \leq j \leq R$.
(6.D) The equation $c = (c_0 + c_1 + \cdots + c_R) \bmod q$ is satisfied.

variable	description	
p	4096-bit prime modulus	§3.1.1
q	256-bit prime order of subgroup of $\mathbb{Z}_p^*$	
r	Cofactor of q in $p - 1$	
g	Generator of subgroup of $\mathbb{Z}_p^*$ of order q	
K	Joint vote encryption public key	§3.2.2
H_I	Selection encryption identifier hash	§3.3.2
(α, β)	Encryption of vote	§3.3.1
R	Option selection limit	§3.3.7
j	Index running through the range $0, 1, \ldots, R$	
c_j	Derived challenge for value j	
v_j	Response to challenge for value j	

Verification 7 (Adherence to vote limits)
For each contest Λ on each cast ballot, an election verifier must compute the contest totals

(7.1) $\bar{\alpha} = (\prod_i \alpha_i) \bmod p$,
(7.2) $\bar{\beta} = (\prod_i \beta_i) \bmod p$,

where the (α_i, β_i) represent all possible selections for the contest, as well as the values

(7.3) $a_j = (g^{v_j} \cdot \bar{\alpha}^{c_j}) \bmod p$ for all $0 \leq j \leq L$,
(7.4) $b_j = (K^{w_j} \cdot \bar{\beta}^{c_j}) \bmod p$, where $w_j = (v_j - jc_j) \bmod q$ for all $0 \leq j \leq L$,
(7.5) $c = H_q(\mathrm{H}_I; \mathtt{0x24}, \mathtt{ind_c}(\Lambda), \bar{\alpha}, \bar{\beta}, a_0, b_0, a_1, b_1, \ldots, a_L, b_L)$,

where L is the contest selection limit. An election verifier must then confirm the following:

(7.A) The given values α_i and β_i are each in $\mathbb{Z}_p^r$.
(7.B) The given values c_j are each in $\mathbb{Z}_q$ for all $0 \leq j \leq L$.
(7.C) The given values v_j are each in $\mathbb{Z}_q$ for all $0 \leq j \leq L$.
(7.D) The equation $c = (c_0 + c_1 + \cdots + c_L) \bmod q$ is satisfied.

variable	description	
p	4096-bit prime modulus	§3.1.1
q	256-bit prime order of subgroup of $\mathbb{Z}_p^*$	
r	Cofactor of q in $p-1$	
g	Generator of subgroup of $\mathbb{Z}_p^*$ of order q	
K	Joint vote encryption public key	§3.2.2
H_I	selection encryption identifier hash	§3.3.2
L	Contest selection limit	§3.3.8
i	Sequence number of selections in the contest (i-th selection)	
(α_i, β_i)	Encryption of vote on i-th selection	
j	Index running through the range $0, 1, \ldots, L$	
c_j	Selection limit challenge values	
v_j	Responses to selection limit challenges	

Verification 8 (Validation of confirmation codes)
An election verifier must confirm the following for each ballot.

(8.A) The contest hash χ_l for the contest with contest index l for all $1 \leq l \leq m_B$ has been correctly computed from the contest's m_l selection encryptions (α_i, β_i), $1 \leq i \leq m_l$, as

$$\chi_l = H(\mathrm{H}_I; \mathtt{0x28}, l, \alpha_1, \beta_1, \alpha_2, \beta_2 \ldots, \alpha_{m_l}, \beta_{m_l}, C_0, C_1, C_2).$$

(8.B) The ballot confirmation code H_C has been correctly computed from the contest hashes and the chaining field byte array B_C as

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

(8.C) The voting device information hash has been correctly computed from the string S_{device} for the device the ballot was processed on as specified in the election manifest as

$$\mathrm{H}_{DI} = H(H_E; \mathtt{0x2A}, S_{\text{device}}).$$

(8.D) If the no-chaining mode was used on the device, the chaining field is $B_C = \mathtt{0x00000000} \parallel \mathrm{H}_{DI}$.

If the simple chaining mode was used on the device, an election verifier must confirm the following three items.

(8.E) If the ballot is the j^{th} ballot processed on this device, $1 \leq j \leq \ell$, the chaining field byte array used to compute H_j is equal to $\mathrm{B}_{C,j} = \mathtt{0x00000001} \parallel \mathrm{H}_{j-1}$.

The following items need only be verified once per voting device.

(8.F) The initial hash code H_0 satisfies $\mathrm{H}_0 = H(\mathrm{H}_E; \mathtt{0x29}, \mathrm{B}_{C,0})$ and $\mathrm{B}_{C,0} = \mathtt{0x00000001} \parallel \mathrm{H}_{DI}$.

(8.G) The final input byte array is $\overline{\mathrm{B}}_C = \mathtt{0x00000001} \parallel H(\mathrm{H}_E; \mathtt{0x2B}, \mathrm{H}_\ell, \mathrm{B}_{C,0})$, where H_ℓ is the final confirmation code on this device, and the closing hash is correctly computed as $\overline{\mathrm{H}} = H(\mathrm{H}_E; \mathtt{0x29}, \overline{\mathrm{B}}_C)$.

variable	description	
H_E	Extended base hash	§3.2.3
H_I	Selection encryption identifier hash	§3.3.2
l	Sequence number of contests on the ballot	§3.4.1
(α_i, β_i)	Encryption of vote on i-th selection	
χ_l	Contest hash of the contest with contest index l	
(C_0, C_1, C_2)	Encrypted contest data	
H_C	Confirmation code of the ballot	§3.4.2
B_C	Chaining field byte array	
H_{DI}	Device information hash	§3.4.3
S_{device}	Device information string	
$\mathrm{B}_{C,0}$	Initial input byte array for ballot chaining	§3.4.4
j	Sequence number of ballots in the hash chain	
$\mathrm{B}_{C,j}$	Chaining field input byte array for ballot chaining	
ℓ	Sequence number of last ballot in the hash chain	
$\overline{\mathrm{B}}_C$	Closing input byte array for ballot chaining	

6.2.5 Tally Verification

Verification 9 (Correctness of ballot aggregation)
An election verifier must confirm for each option in each contest in the election manifest that the aggregate encryption (A, B) satisfies
(9.A) $A = (\prod_j \alpha_j) \bmod p$,
(9.B) $B = (\prod_j \beta_j) \bmod p$,
where the (α_j, β_j) are the corresponding encryptions on all cast ballots in the election record.

variable	description	
p	4096-bit prime modulus	§3.1.1
(α_j, β_j)	Encryption of selection on the j-th ballot	§3.5
(A, B)	Encrypted aggregate total of votes for this option	

Verification 10 (Correctness of tally decryptions)
For each option λ in each contest Λ on each tally, an election verifier must confirm the correctness of the decrypted accumulated value t. It must compute the values

(10.1) $M = (B \cdot T^{-1}) \bmod p$,
(10.2) $a = (g^v \cdot K^c) \bmod p$,
(10.3) $b = (A^v \cdot M^c) \bmod p$.

An election verifier must then confirm the following:

(10.A) The given value v is in the set $\mathbb{Z}_q$.
(10.B) The challenge value c satisfies $c = H_q(\mathrm{H}_E; \texttt{0x31}, \texttt{ind}_\texttt{c}(\Lambda), \texttt{ind}_\texttt{o}(\lambda), A, B, a, b, M)$.
(10.C) The decrypted tally value t satisfies $T = K^t \bmod p$.

variable	description	
p	4096-bit prime modulus	§3.1.1
q	256-bit prime order of subgroup of $\mathbb{Z}_p^*$	
r	Cofactor of q in $p-1$	
g	Generator of subgroup of $\mathbb{Z}_p^*$ of order q	
K	Joint vote encryption public key	§3.2.2
H_E	Extended base hash	§3.2.3
(A, B)	Encrypted aggregate total of votes for option	§3.5
T	Decrypted value of (A, B)	§3.6.2
t	Decrypted tally value	
c	Challenge to decryption	§3.6.5
v	Response to challenge	

Verification 11 (Validation of contents of tallies)
An election verifier must confirm that the text labels listed in the election record tallies match the corresponding text labels in the election manifest. For each contest in a decrypted tally, an election verifier must confirm the following.

(11.A) The contest text label occurs as a contest label in the list of contests in the election manifest.

(11.B) For each option in the contest, the option text label occurs as an option label for the contest in the election manifest.

(11.C) For each option text label listed for this contest in the election manifest, the option label occurs for an option in the decrypted tally contest.

An election verifier must also confirm the following.

(11.D) For each contest text label that occurs in at least one submitted ballot, that contest text label occurs in the list of contests in the corresponding tally.

6.2.6 Verification of Correct Contest Data Decryption

Verification 12 (Correctness of decryptions of contest data[66])
An election verifier must confirm the correct decryption of the contest data field for each contest Λ by verifying the conditions analogous to Verification 10 for the corresponding NIZK proof with (A, B) replaced by (C_0, C_1, C_2) and M by β as follows. An election verifier must compute the following values.

(12.1) $a = (g^v \cdot \hat{K}^c) \bmod p$,
(12.2) $b = (C_0^v \cdot \beta^c) \bmod p$.

An election verifier must then confirm the following.

(12.A) The given value v is in the set $\mathbb{Z}_q$.
(12.B) The challenge value c satisfies $c = H_q(\mathrm{H}_I; \mathtt{0x33}, \mathtt{ind_c}(\Lambda), C_0, C_1, C_2, a, b, \beta)$.

An election verifier now must compute the values

(12.3) $h = H(\mathrm{H}_I; \mathtt{0x26}, \mathtt{ind_c}(\Lambda), C_0, \beta)$,
(12.4) $k_i = \mathrm{HMAC}(h, \mathtt{b}(i, 4) \parallel \mathtt{Label} \parallel \mathtt{0x00} \parallel \mathtt{Context} \parallel \mathtt{b}(b_\Lambda \cdot 256, 4))$

for $1 \leq i \leq b_\Lambda$ and verify correct decryption by using $C_1 = C_{1,1} \parallel C_{1,2} \parallel \cdots \parallel C_{1,b_\Lambda}$ and confirming that

(12.C) $D = C_{1,1} \oplus k_1 \parallel C_{1,2} \oplus k_2 \parallel \cdots \parallel C_{1,b_\Lambda} \oplus k_{b_\Lambda}$.

variable	description	
p	4096-bit prime modulus	§3.1.1
q	256-bit prime order of subgroup of $\mathbb{Z}_p^*$	
r	Cofactor of q in $p-1$	
g	Generator of subgroup of $\mathbb{Z}_p^*$ of order q	
$\hat{K}$	Joint ballot data encryption public key	§3.2.2
H_I	Selection encryption identifier hash	§3.3.2
(C_0, C_1, C_2)	Encrypted contest data	§3.6.6
b_Λ	Byte length of encrypted contest data	
β	Decryption factor of (C_0, C_1, C_2)	
c	Challenge to decryption	
v	Response to challenge	

6.2.7 Verification of Challenged Ballots

Verification 13 (Correctness of decryptions for challenged ballots)
For each challenged ballot B (which has m_B contests), an election verifier must confirm the correct decryption of the selection values and contest data as follows.
For each contest Λ_i (for $1 \leq i \leq m_B$ with m_i verifiable option fields) on the challenged ballot, an election verifier must verify correctness of the decrypted values $\sigma_{i,j}$ using the corresponding encryption nonces $\xi_{i,j}$ and correctness of the decrypted contest data D using the contest data encryption nonce ξ_i.
For all $1 \leq j \leq m_i$, it must recompute the selection encryptions

(13.1) $\alpha_{i,j} = g^{\xi_{i,j}} \bmod p$,
(13.2) $\beta_{i,j} = K^{\sigma_{i,j}+\xi_{i,j}} \bmod p$,

and from those (together with the encrypted contest data (C_0, C_1, C_2)), the contest hash

(13.3) $\chi_i = H(\mathrm{H}_I; \texttt{0x28}, \texttt{ind}_\texttt{c}(\Lambda_i), \alpha_{i,1}, \beta_{i,1}, \alpha_{i,2}, \beta_{i,2} \ldots, \alpha_{i,m_i}, \beta_{i,m_i}, C_0, C_1, C_2)$.

It must also recompute the encryption keys for the contest data via

(13.4) $\alpha = g^{\xi_i} \bmod p$,
(13.5) $\beta = \hat{K}^{\xi_i} \bmod p$,
(13.6) $h = H(\mathrm{H}_I; \texttt{0x26}, \texttt{ind}_\texttt{c}(\Lambda), \alpha, \beta)$.
(13.7) $k_l = \mathrm{HMAC}(h, \texttt{b}(l, 4) \parallel \texttt{Label} \parallel \texttt{0x00} \parallel \texttt{Context} \parallel \texttt{b}(b_\Lambda \cdot 256, 4))$ for $0 \leq l < b_\Lambda$.

An election verifier must then verify correct decryption of the contest data by using the decrypted data $D = D_1 \parallel D_2 \parallel \cdots \parallel D_{b_\Lambda}$ and the ciphertext C_1 and confirming that

(13.A) $C_1 = D_1 \oplus k_1 \parallel D_2 \oplus k_2 \parallel \cdots \parallel D_{b_\Lambda} \oplus k_{b_\Lambda}$.

Finally, it must confirm that the provided confirmation code is correct, i.e.,

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

where B_C is the chaining field for ballot B.
If only selected contests have been decrypted—as might be the case in an RLA setting—Verification step (13.B) uses contest hashes provided with or recomputed from the encrypted ballot for the contests that have not been decrypted.

variable	description	
p	4096-bit prime modulus	§3.1.1
g	Generator of subgroup of $\mathbb{Z}_p^*$ of order q	
K	Joint vote encryption public key	§3.2.2
$\sigma_{i,j}$	Selection values	§3.3.1
H_I	Selection encryption identifier hash	§3.3.2
$\xi_{i,j}$	Encryption nonces used for selection encryptions	§3.3.3
(C_0, C_1, C_2)	Encrypted contest data	§3.6.7

Verification 14 (Validation of well-formedness and content of challenged ballots)
An election verifier must confirm that for each decrypted challenged ballot, the selections listed in text match the corresponding text in the election manifest.

(14.A) The contest text label occurs as a contest label in the list of contests for the ballot's ballot style in the election manifest.
(14.B) For each contest in the list of contests for the ballot's ballot style in the election manifest, the contest label appears as a contest label on the uncast pre-encrypted ballot.
(14.C) For each option in the contest, the option text label occurs as an option label for the contest in the election manifest.
(14.D) For each option text label listed for this contest in the election manifest, the option label occurs for an option in the decrypted challenged ballot.

An election verifier must also confirm that the challenged ballot is well-formed, i.e., for each contest on the challenged ballot, it must confirm the following.

(14.E) For each option in the contest, the selection σ is a valid value—usually either a 0 or a 1.
(14.F) The sum of all selections in the contest is at most the selection limit L for that contest.

variable	description	
L	Contest selection limit	§3.3.8
σ	Selection value	

6.2.8 Verification of Pre-Encrypted Ballots

Verification 15 (Validation of correct accumulation of selection vectors)
An election verifier must confirm the following for each contest on each pre-encrypted ballot.

(15.A) If the contest selection limit is greater than 1, the selection vector Ψ published in the election record for this contest has been correctly computed as a product of the individual selection vectors $\Psi_{i,m}$ that correspond to the short codes as selected by the voter.

variable	description	
$\Psi_{i,m}$	Selection vector	§4.1.1
Ψ	Accumulated contest selection vector	

Verification 16 (Validation of confirmation codes in pre-encrypted ballots)
An election verifier must confirm the following for each pre-encrypted ballot.

(16.A) For each selection in each contest on the ballot and the corresponding selection vector $\Psi_{i,m} = \langle E_1, E_2, \ldots, E_m \rangle$ consisting of the selection encryptions $E_j = (\alpha_j, \beta_j)$, the selection hash ψ_i satisfies

$$\psi_i = H(\mathrm{H}_I; \texttt{0x40}, \alpha_1, \beta_1, \alpha_2, \beta_2, \ldots, \alpha_m, \beta_m).$$

(16.B) The contest hash χ_l for the contest with context index l for all $1 \le l \le m_B$ has been correctly computed from the selection hashes ψ_i as

$$\chi_l = H(\mathrm{H}_I; \texttt{0x41}, l, \psi_{\pi(1)}, \psi_{\pi(2)}, \ldots, \psi_{\pi(m+L)}),$$

where π is a permutation and $\psi_{\pi(1)} < \psi_{\pi(2)} < \cdots < \psi_{\pi(m+L)}$.

(16.C) The ballot confirmation code H_C has been correctly computed from the (sequentially ordered) contest hashes and the chaining field byte array B_C as

$$\mathrm{H}_C = H(\mathrm{H}_I; \texttt{0x42}, \chi_1, \chi_2, \ldots, \chi_{m_B}, \mathrm{B}_C).$$

(16.D) The voting device information hash has been correctly computed from the string S_{device} for the device the ballot was processed on as specified in the election manifest as

$$\mathrm{H}_{DI} = H(H_E; \texttt{0x43}, S_{\text{device}}).$$

(16.E) If the device used the no-chaining mode, the chaining field is $B_C = \texttt{0x00000000} \parallel \mathrm{H}_{DI}$.

If the device used the simple chaining mode, an election verifier must confirm the following three items.

(16.F) If the ballot is the j^{th} ballot processed on this device, $1 \le j \le \ell$, the chaining field byte array used to compute H_j is equal to $\mathrm{B}_{C,j} = \texttt{0x00000001} \parallel \mathrm{H}_{j-1}$.

The following items need only be verified once per voting device.

(16.G) The initial hash code H_0 satisfies $\mathrm{H}_0 = H(\mathrm{H}_E; \texttt{0x42}, \mathrm{B}_{C,0})$ and $\mathrm{B}_{C,0} = \texttt{0x00000001} \parallel \mathrm{H}_{DI}$.

(16.H) The final input byte array is $\overline{\mathrm{B}}_C = \texttt{0x00000001} \parallel H(\mathrm{H}_E; \texttt{0x44}, \mathrm{H}_\ell, \mathrm{B}_{C,0})$, where H_ℓ is the final confirmation code on this device, and the closing hash is correctly computed as $\overline{\mathrm{H}} = H(\mathrm{H}_E; \texttt{0x42}, \overline{\mathrm{B}}_C)$.

variable	description	
H_I	Selection encryption identifier hash	§3.3.2
i	Sequence number of selections in the contest	§4.1.1
λ_i	Selection label	
(α_i, β_i)	Encryption of vote on i-th selection	
$\Psi_{i,m}$	Selection vector	
ψ_i	Selection hash	
π	Permutation reflecting sorting in order	
l	Sequence number of contests on the ballot	§4.1.2
χ_l	Contest hash of the l-th contest	
H_C	Confirmation code of the ballot	§4.1.3
B_C	Chaining field input byte array	
H_{DI}	Device information hash	§3.4.3
S_{device}	Device information string	
$\mathrm{B}_{C,0}$	Initial input byte array for ballot chaining	§3.4.4
j	Sequence number of ballots in the hash chain	
$\mathrm{B}_{C,j}$	Chaining field input byte array for ballot chaining	
ℓ	Sequence number of last ballot in the hash chain	
$\overline{\mathrm{B}}_C$	Closing input byte array for ballot chaining	

Verification 17 (Validation of short codes in pre-encrypted ballots)
An election verifier must confirm for every selectable option on every pre-encrypted ballot in the election record that the short code ω displayed with the selectable option satisfies

(17.A) $\omega = \Omega(\psi)$ where ψ is the selection hash associated with the selectable option.

Specifically, for cast ballots, this includes all short codes that are published in the election record whose associated selection hashes correspond to selection vectors that are accumulated to form tallies. For uncast ballots, this includes all selection vectors on the ballot.

variable	description	
ψ	Selection hash	§4.1.1
ω	Short code of selection hash	§4.1.5
Ω	Hash trimming function	

Verification 18 (Correctness of encryptions for uncast pre-encrypted ballots)
For each uncast pre-encrypted ballot B (which has m_B contests), an election verifier must confirm that all encryptions are correct encryptions of the corresponding selections as follows.
For each contest Λ_i (for $1 \leq i \leq m_B$ with m_i verifiable option fields) on the uncast ballot, an election verifier must verify correctness of the encrypted selections using the corresponding encryption nonces $\xi_{i,j,k}$.
For all $1 \leq j \leq m_i$, it must recompute the selection encryption vector of the j^{th} selection, i.e., for all $1 \leq k \leq m_i$ it must recompute

(18.1) $\alpha_{i,j,k} = g^{\xi_{i,j,k}} \bmod p$,
(18.2) $\beta_{i,j,k} = K^{\delta_{j,k}+\xi_{i,j,k}} \bmod p$, where $\delta_{j,k} = 1$ if and only if $j = k$ and $\delta_{j,k} = 0$, otherwise.

From those ciphertexts it must compute the selection hashes

(18.3) $\psi_{i,j} = H(\mathrm{H}_I; \mathtt{0x40}, \alpha_{i,j,1}, \beta_{i,j,1}, \ldots, \alpha_{i,j,m_i}, \beta_{i,j,m_i})$,

and from those, the contest hash with selection hashes in sorted order as

(18.4) $\chi_i = H(\mathrm{H}_I; \mathtt{0x41}, \mathtt{ind_c}(\Lambda_i), \psi_{i,\pi(1)}, \psi_{i,\pi(2)}, \ldots, \psi_{i,\pi(m_i)})$.

Finally, it must confirm that the provided confirmation code is correct, i.e.,

(18.A) $\mathrm{H}_C = H(\mathrm{H}_I; \mathtt{0x42}, \chi_1, \chi_2, \ldots, \chi_{m_B}, \mathrm{B}_C)$,

where B_C is the chaining field for ballot B.

variable	description	
p	4096-bit prime modulus	§3.1.1
g	Generator of subgroup of $\mathbb{Z}_p^*$ of order q	
K	Joint vote encryption public key	§3.2.2
H_I	Selection encryption identifier hash	§3.3.2
$\psi_{i,j}$	Selection hash	§4.1.1
l	Sequence number of contests on the ballot	§4.1.2
χ_i	Contest hash of the i-th contest	
π	Permutation reflecting sorting in order	
H_C	Confirmation code of the ballot	§4.1.3
B_C	Chaining field input byte array	
$\xi_{i,j,k}$	Encryption nonce	§4.2.1

Verification 19 (Validation of content of uncast pre-encrypted ballots)
An election verifier must confirm that for each uncast ballot, the selections listed in text match the corresponding text in the election manifest. For each contest on the ballot, it must verify the following.

(19.A) The contest text label occurs as a contest label in the list of contests for the ballot's ballot style in the election manifest.

(19.B) For each contest in the list of contests for the ballot's ballot style in the election manifest, the contest label appears as a contest label on the uncast pre-encrypted ballot.

(19.C) For each option in the contest, the option text label occurs as an option label for the contest in the election manifest.

(19.D) For each option text label listed for this contest in the election manifest, the option label occurs for an option on the uncast pre-encrypted ballot.

7 Applications to End-to-End Verifiability and Risk-Limiting Audits

The methods described in this specification can be used to enable either end-to-end (E2E) verifiability or enhanced risk-limiting audits (RLAs). In both cases, the ballots are individually encrypted and proofs are provided to allow observers to verify that the set of encrypted ballots is consistent with the announced tallies in an election.

In the case of E2E-verifiability, voters are given confirmation codes to enable them to confirm that their individual ballots are correctly recorded amongst the set of encrypted ballots. In the case of RLAs, encrypted ballots are randomly selected and compared against physical ballots to obtain confidence that the physical records match the electronic records.

To support enhanced risk-limiting audits (RLAs), it may be desirable to encrypt the ballot nonce of each ballot with a simple administrative key rather than the "heavyweight" ballot data encryption public key. This streamlines the process for decrypting an encrypted ballot that has been selected for audit. It should be noted that the privacy risks of revealing decrypted ballots are substantially reduced in the RLA case since voters are not given confirmation codes that could be used to associate them with individual ballots. The primary risk is a coercion threat (e.g., via pattern voting) that only manifests if the full set of ballots were to be decrypted.

While the administratively encrypted nonce can be stored in an electronic record alongside each encrypted ballot, one appealing RLA instantiation is for the administrative encryption of a ballot's nonce to be printed directly onto the physical ballot. This allows an RLA to proceed by randomly selecting an encrypted ballot, fetching the associated physical ballot, extracting the nonce from its encryption on the physical ballot, using the nonce to decrypt the electronic record, and then comparing the physical ballot contents with those of the electronic record. A malicious actor with an administrative decryption key would need to go to each individual physical ballot to obtain the nonces necessary to decrypt all of the encrypted ballots, and the access to do so would enable this malicious actor to obtain all of the open ballots without necessitating the administrative decryption key.

If E2E-verifiability and enhanced RLAs are both provided in the same election, there must be separate ballot encryptions (ideally, but not necessarily, using separate election encryption keys) of each ballot. The E2E-verifiable data set must be distinguished from the enhanced RLA data set. Using the same data set for both applications would compromise voter privacy for voters whose ballots are selected for auditing.

Acknowledgments

The authors are happy to thank Eion Blanchard, Aleš Bizjak, Nicholas Boucher, John Caron, Henri Devillez, Joey Dodds, Felix Doerre, Gerald Doussot, Aleks Essex, Keith Fung, Rainbow Huang, Chris Jeuell, Anunay Kulshrestha, Moses Liskov, Shreyas Minocha, Arash Mirzaei, Luke Myers, Karan Newatia, Thomas Peters, John Ramsdell, Marsh Ray, Dan Shumow, Vanessa Teague, Aaron Tomb, Daniel Tschudi, Daniel Wagner, Jake Waksbaum, Dan Wallach, Matt Wilhelm, and Greg Zaverucha for many helpful comments and suggestions on earlier versions of this specification.

Appendix

Reduced Parameters—Using a 3072-Bit Prime

Starting from the same 256-bit prime $q = 2^{256} - 189$ for the group order, the procedure outlined in Section 3.1.1 on the standard baseline parameters can be used to find a 3072-bit prime p_{3072}. This results in the following parameters that are provided here as an alternative. The parameters satisfy all the same requirements, but the smaller prime offers faster arithmetic in the base field and thus improved performance. This comes at the cost of a somewhat lower security level.

When using the reduced parameters in this section, the number of bytes needed to represent integers modulo p is $l_p = 384$ instead of 512 for the standard parameters. This means that when interpreting integers modulo p as byte arrays for hashing as described in Section 5.1, representations will have exactly $l_p = 384$ bytes and are thus shorter than for the standard parameters.

The prime p_{3072} is given as $p_{3072} = 2^{3072} - 2^{2816} + 2^{256}(\lfloor 2^{2560}\ln(2)\rfloor + \delta_{3072}) + 2^{256} - 1$ with

$$\delta_{3072} = 2987074079534379958763006253707499063253226635980367563918676629265692139358095775 93.$$

The hexadecimal representation of p_{3072} is as follows.

```
p3072 = 0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
          B17217F7 D1CF79AB C9E3B398 03F2F6AF 40F34326 7298B62D 8A0D175B 8BAAFA2B
          E7B87620 6DEBAC98 559552FB 4AFA1B10 ED2EAE35 C1382144 27573B29 1169B825
          3E96CA16 224AE8C5 1ACBDA11 317C387E B9EA9BC3 B136603B 256FA0EC 7657F74B
          72CE87B1 9D6548CA F5DFA6BD 38303248 655FA187 2F20E3A2 DA2D97C5 0F3FD5C6
          07F4CA11 FB5BFB90 610D30F8 8FE551A2 EE569D6D FC1EFA15 7D2E23DE 1400B396
          17460775 DB8990E5 C943E732 B479CD33 CCCC4E65 9393514C 4C1A1E0B D1D6095D
          25669B33 3564A337 6A9C7F8A 5E148E82 074DB601 5CFE7AA3 0C480A54 17350D2C
          955D5179 B1E17B9D AE313CDB 6C606CB1 078F735D 1B2DB31B 5F50B518 5064C18B
          4D162DB3 B365853D 7598A195 1AE273EE 5570B6C6 8F969834 96D4E6D3 30D6E582
          CAB40D66 550984EF 0C42A457 4280B378 45189610 AE3E4BB2 2590A08F 6AD27BFB
          FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
```

The hexadecimal representation of the cofactor $r_{3072} = (p_{3072} - 1)/q$ is shown below.

```
r3072 = 0x01 00000000 00000000 00000000 00000000 00000000 00000000 00000000 000000BC
             B17217F7 D1CF79AB C9E3B398 03F2F6AF 40F34326 7298B62D 8A0D175B 8BAB857A
             E8F42816 5418806C 62B0EA36 355A3A73 E0C74198 5BF6A0E3 130179BF 2F0B43E3
             3AD86292 3861B8C9 F768C416 9519600B AD06093F 964B27E0 2D868312 31A9160D
             E48F4DA5 3D8AB5E6 9E386B69 4BEC1AE7 22D47579 249D5424 767C5C33 B9151E07
             C5C11D10 6AC446D3 30B47DB5 9D352E47 A53157DE 04461900 F6FE360D B897DF53
             16D87C94 AE71DAD0 BE84B647 C4BCF818 C23A2D4E BB53C702 A5C8062D 19F5E9B5
             033A94F7 FF732F54 12971286 9D97B8C9 6C412921 A9D86797 70F499A0 41C297CF
```

F79D4C91 49EB6CAF 67B9EA3D C563D965 F3AAD137 7FF22DE9 C3E62068 DD0ED615
1C37B4F7 4634C2BD 09DA912F D599F433 3A8D2CC0 05627DCA 37BAD43E 64CAF318
9FD4A7F5 29FD4A7F 529FD4A7 F529FD4A 7F529FD4 A7F529FD 4A7F529F D4A7F52A

And the generator $g_{3072} = 2^{r_{3072}} \bmod p_{3072}$ has the following hexadecimal representation.

g_{3072} = 0x4A1523CB 0111B381 04EBCDE5 163F581E EEDD9163 7AC57544 C1D22832 34272732
FF0CD85F 38539544 3F573701 32A237FF 38702AB0 37F35E7C 7003669D 83697BA1
3BED69B6 3C88BD61 0D33C6A8 9E4882EE 6F849F05 06A4A8F0 B169E5CA 000A21DC
16D7DCEC C69E593C 65967739 3B6CE260 D3D6A578 E74E42A1 B2ADE1ED 8627050C
FB59E604 CAC389E9 9161DA6E 6E9407DF 94517864 01003A8B 7626AC5E 90B888EA
BB5E07E9 96B18662 9B17165F D630E139 788F674D FF4978A6 B74C6D02 0A6570CC
7C7A9E38 21283571 BA3FA1FC C6901A8C 28D02EF8 B8C4B019 F7DDADE5 1A089C57
EF90C2CE 50761754 D778BC9A BFD84809 5C4A0ED0 FA7B7AE5 2CDA4BD6 E2CB16F3
8EDC033F 32F259C5 13DD9E0D 1F780886 D71D7DB8 35F3F08D B11CC9CD 41EB0D5A
37AC6DBA 1A1EBA55 C378BC06 95B9D93A A59903EB A1CE5288 6A0BAAFB 15354863
1BCEAC52 07B97205 BE8FDF83 0F27348C 7AE852F9 F8876887 D23B8054 A077DC8A
EC0BF615 A1FA74BC 727014CF AC40E20E A194489F 63A6C224 27CB999C 9D04AA61

Toy Parameters for Testing Purposes Only

The following parameter sets are provided for testing purposes only and must not be used in any real-world scenario. The involved primes are too small to provide any security at all.

- 7-bit q, 16-bit p

$$q = 127 = \texttt{0x007F},$$
$$p = 59183 = \texttt{0xE72F},$$
$$r = 466 = \texttt{0x01D2},$$
$$g = 32616 = \texttt{0x7F68}$$

- 16-bit q, 32-bit p

$$q = 65521 = \texttt{0xFFF1},$$
$$p = 4214179679 = \texttt{0xFB2B 475F},$$
$$r = 64318 = \texttt{0xFB3E},$$
$$g = 496451214 = \texttt{0x1D97 3E8E}$$

- 16-bit q, 48-bit p

$$q = 65521 = \texttt{0xFFF1},$$
$$p = 281010572049407 = \texttt{0xFF93 DF53 3BFF},$$
$$r = 4288862686 = \texttt{0xFFA2 D9DE},$$
$$g = 109132885510074 = \texttt{0x6341 7ADF C7BA}$$

- 24-bit q, 64-bit p

$$\begin{aligned}
q &= 16777213 = \mathtt{0xFFFFFFFD}, \\
p &= 18444843247520538623 = \mathtt{0xFFF93F35\ 6A395FFF}, \\
r &= 1099398526294 = \mathtt{0x000000FF\ F9423556}, \\
g &= 14757355607624201864 = \mathtt{0xCCCCA8BC\ C08F3688}
\end{aligned}$$

- 32-bit q, 96-bit p

$$\begin{aligned}
q &= 4294967291 = \mathtt{0xFFFFFFFB}, \\
p &= 79227651399410325621583970303 \\
&= \mathtt{0xFFFF93C4\ 6882B6AA\ F57CFFFF}, \\
r &= 18446625091983808922 \\
&= \mathtt{0xFFFF93C9\ 6880999A}, \\
g &= 60786014689883675535506158858 \\
&= \mathtt{0xC469034B\ 2CE5EC6E\ 1970350A}
\end{aligned}$$

- 32-bit q, 128-bit p

$$\begin{aligned}
q &= 4294967291 = \mathtt{0xFFFFFFFB}, \\
p &= 340282366887442052436576802921059975167 \\
&= \mathtt{0xFFFFFFFF\ 93C46B0F\ B6C381D8\ FFFFFFFF}, \\
r &= 79228162598699067120922761626 \\
&= \mathtt{0x00000001\ 00000004\ 93C46B26\ 9999999A}, \\
g &= 55628101181055236817878380639043675517 \\
&= \mathtt{0x29D99524\ 0DFB12B3\ 6FD0F8CCE06B657D}
\end{aligned}$$

- 48-bit q, 192-bit p

$$\begin{aligned}
q &= \mathtt{0x0000FFFF\ FFFFFFC5}, \\
p &= \mathtt{0xFFFFFFFF\ FFFFFFFF\ 9ECB7796\ 49D9A82D\ FFFFFFFF\ FFFFFFFF}, \\
r &= \mathtt{0x00010000\ 0000003A\ FFFF9ECB\ 852F49C3\ 4115B1E6}, \\
g &= \mathtt{0x0B5DA090\ 0B367E3C\ 92A11019\ 54DB5E3C\ 873E929A\ 0E324F00}
\end{aligned}$$

- 64-bit q, 256-bit p

$$\begin{aligned}
q &= \mathtt{0xFFFFFFFF\ FFFFFFC5}, \\
p &= \mathtt{0xFFFFFFFF\ FFFFFFFF\ 93C467E3\ 7DB1212B\ 89995855\ 493FF059\ FFFFFFFF\ FFFFFFFF}, \\
r &= \mathtt{0x00000001\ 00000000\ 0000003A\ 93C467E3\ 7DB12EAB\ 97DD49C3\ 4115B1E6}, \\
g &= \mathtt{0x3B543166\ 9E3E4893\ DF745C67\ CDCFD95C\ CDDA2248\ 78A3CD5D\ 3226F75C\ C5A95638}
\end{aligned}$$

Glossary of Typical Variable Usage

variable	description
(a, b), (a_i, b_i)	Commitments to an encryption
(A, B)	Aggregation of multiple encrypted votes
$a_{i,j}$, $\hat{a}_{i,j}$	Polynomial coefficients of secret polynomials of guardian G_i
b_i	Bytes
B	Ballot
B, B_i	Byte arrays
c, c_i, $c_{i,j}$, $\hat{c}_{i,j}$, $\bar{c}_{i,j}$	Challenge values
C_i	Ciphertext components
d_i	Commitment of guardian G_i to decryption proof commitments
D, D_i	Contest data field and components
E_i	Encryptions, ciphertexts
G_i	Guardians
g	Generator of order-q group of $\mathbb{Z}_p^*$
h	Encryption key
H	Hash function
H	Hash value
i, j	Index values
id_B	Selection encryption identifier
k	Decryption threshold
k_i	Encryption and MAC keys
K, $\hat{K}$	Joint vote encryption and joint ballot data encryption public keys
K_i, $\hat{K}_i$	Public keys of i^{th} guardian
$K_{i,j}$, $\hat{K}_{i,j}$	Public commitment to polynomial coefficents $a_{i,j}$ and $\hat{a}_{i,j}$
l, ℓ	Index values
L	Selection or range limit

variable	description
m, m_i	Messages
M	Exponentiated decryption
M_i	Partial decryption of i^{th} guardian
n	Number of guardians
p	Large prime modulus
P, P_i	Polynomials for secret sharing
q	Small prime order of subgroup of $\mathbb{Z}_p^*$
r	Cofactor of q in $p - 1$
R	Range limit or option selection limit
s	Secret election key
s_i	Secret guardian keys
S	Encoded selection value
t	Tally value
T	Encoded tally value
u_i, $u_{i,j}$, $\hat{u}_{i,j}$, $\bar{u}_{i,j}$	Random values in proofs
v_i, $v_{i,j}$, $\hat{v}_{i,j}$, $\bar{v}_{i,j}$	Proof response values
w_i	Lagrange coefficients
W_i	Ballot weights
x	Polynomial indeterminate
z_i, $\hat{z}_i$	Guardian G_i's shares of the decryption keys K and $\hat{K}$

variable	description
(α, β)	Encryption of individual vote
γ	Proof component
ζ_i	Secret communiation keys
κ_i	Public communication keys
λ, Λ	Selection and contest labels
ξ, ξ_i, $\xi_{i,j}$	Random nonces
π	Sorting permutation
σ	Selection value
χ, χ_i	Contest hashes
ψ	Selection hash
Ψ	Selection vector
ω	Short code of selection hash
Ω	Hash trimming function

List of Changes Over Version 2.0.0

Section 3.1

- The version byte array is now an encoding of the string `"v2.1.0"` and the parameter base hash H_P now includes the small integers n and k as hash inputs. See Equation (4) in Section 3.1.2. Therefore, H_P is not a fixed hash value anymore and the old Equation (5) has been removed accordingly.
- The manifest hash H_M has been removed, i.e., the old section 3.1.4 with the old Equation (6) defining H_M has been removed.
- The election base hash H_B is now directly computed from the manifest the same way H_M was computed before. See Equation (5) in the current Section 3.1.4.
- Verification 1 has been changed accordingly to reflect the above modifications. The verification of the cofactor r has been removed from Verification 1.

Section 3.2

- The challenge values $c_{i,j}$ for the Schnorr proofs in the old Equation (12) in Section 3.2.2 have been replaced by a single challenge value c_i per guardian, see Equation (11) in Section 3.2.2. Verification 2 has been adjusted.
- The key generation process is now run twice to generate not only an election public key K but also a ballot data encryption key $\hat{K}$ to obtain separate keys for different encryption modes. The key K is used to encrypt votes and other verifiable fields that are homomorphically aggregated, the key $\hat{K}$ is used to encrypt data on the ballot such as the ballot nonce and contest data. See Section 3.2.2.
- Each guardian now generates an additional ElGamal public/secret key pair (κ_i, ζ_i) that is used for encrypting secret shares that are sent between guardians, see Equation (9) in Section 3.2.2.
- The Schnorr proofs of knowledge of the coefficients $a_{i,j}$ now also include a proof of knowledge of the additional secret key ζ_i. See Equation (11) and the adjusted Verification 2.
- Share encryption now uses the additional public keys, see Equations (15) and (16). Share encryption is now done with signed ElGamal encryption, see the end of the paragraph *Share encryption* in Section 3.2.2.
- The end of Section 3.2.2 now describes the verification steps that guardians perform to confirm that key shares have been successfully distributed and that the key generation protocol was correctly and consistently executed. These steps are the prerequisites for publishing the guardian record and for enabling the use of ElectionGuard in the election.
- The domain separator byte in the computation of the extended base hash has changed to `0x14`, see Equation (30).

Section 3.3

- The new Section 3.3.2 introduces the selection encryption identifier id_B for each ballot and the selection encryption identifier hash H_I, see Equation (32). A new verification step has been introduced in Verification 5. Subsequent numbering has changed accordingly.
- The order of inputs to H_q when computing encryption nonces has changed and the first input is now the selection encryption identifier hash H_I instead of the extended base hash, see

Equation (33).

- The new Section 3.3.4 now describes how to encrypt ballot nonces using the ballot data encryption key. Encryption is done using signed ElGamal.
- Challenge computation for proofs of ballot well-formedness have changed. In equations (41), (50), and (59), the first input is now the selection encryption identifier hash H_I, the domain separator byte has changed to `0x24`, and the contest and option indices have been added as inputs. The new Verification 6 (previously Verification 5) has been adjusted.
- The new Note 3.4 discusses the adaptation of the range proof technique to the proof of membership in a discrete set within a range. The old Note 3.4 has become the new Note 3.5.
- Computations of the pseudo-random nonce ξ and the secret key h for encrypting contest data have changed. They now include H_I as the first argument, domain separation bytes have changed, and the public key K for h has been replaced by the contest index, see Equations (64) and (65).
- Challenge computation for the proof of satisfying the selection limit has changed. The first input is now H_I, the public key has been replaced by the contest index, and the domain separation byte has changed, see Equation (62). Verification 7 has been adjusted.
- The contest data field is now an optional field and does not have to be included, see the updated Section 3.3.10.
- Contest data is now encrypted using signed ElGamal, counters in the KDF have changed, see the updated Section 3.3.10.

Section 3.4

- The computation of the contest hash has changed. The first input is now H_I, the domain separation byte has changed, the public key has been omitted, and the encrypted contest data is included as an input, see Equation (70).
- Notation of the confirmation code has changed to H_C. Computation of the confirmation code has changed. The first input is now H_I and the domain separation byte has changed, see Equation (71).
- The new Section 3.4.3 defines the newly introduced device information hash that is an input to the confirmation code computation.
- Section 3.4.4 has been updated. Domain separation bytes in hash computations have changed. Hash inputs that facilitate chaining have been simplified and two chaining modes are now described explicitly, the no chaining mode and the simple chaining mode. Verification 8 has been adjusted accordingly.

Section 3.5

- Section 3.5 on ballot aggregation now describes how to use ballot weights.

Section 3.6

- The new Section 3.6.1 describes the verification steps all available guardians must confirm before participating in any decryption operation.
- When participating in the generation of the verifiable decryption proof, each guardian must now commit to the commitment pair (a_i, b_i) by computing a hash value d_i as shown in

Equation (88). A guardian should only publish the commitment pair after all d_i have been verified. This step adds a round of communication and a verification step to the verifiable decryption protocol.
- Challenge computation the proof has changed, the domain separation byte has changed and the public key has been replaced by the contest and option indices, see Equation (90). Verification 10 has been adjusted accordingly.
- Due to the above additional commitment step, the check in the new Note 3.7 is now an optional tool to be used when trying to resolve disputes. It does not need to be verified during the proof generation protocol.
- Contest data decryption in Section 3.6.6 now also has an additional commitment step before publishing the commitment pairs and the challenge computation has been modified. See the new Equation (99) and modified Equation (101). Verification 12 has been adjusted.
- Challenged ballots are always decrypted verifiably by decrypting the ballot nonce and by generating and releasing all or a set of selected encryption nonces. Section 3.6.7 now describes this in more detail. The previous approach of verifiably decrypting all selections by the available guardians as is used for decrypting the tallies has been removed.

Section 3.7

- The newly introduced keys and identifier values have been added as required data in the election record as well as the status of the ballot, i.e., whether it was cast or not (challenged).

Section 4

- Pre-encrypted ballots now also have a selection encryption identifier and the corresponding identifier hash.

Section 4.1

- The first input to the selection and contest hashes has changed to H_I, see Equations (113), (114), and (115).
- Notation of the confirmation code has changed to H_C. The first input to the confirmation code has changed to H_I and the chaining description has been adjusted as defined in Section 3.4.4, see Equations (116), (117), and (118).

Section 4.5

- Section 4.5 now contains an explicit list of verification steps that must be validated for pre-encrypted ballots, pointing to steps that are unchanged from regular ballots and listing differing and new verification steps.

Section 5.4

- Section 5.4 now explicitly defines a hashfunction H_q that maps into $\mathbb{Z}_q$. It is obtained by reducing the output of the function H modulo q, see Equation (132).

Section 5.5

- Many of the domain separation bytes have changed because several hash computations have been added at different positions in the specification. Domain separation bytes might have to be adjusted even if not mentioned explicitly in this list of changes.