Requirements for ElectionGuard version 1.91 Verifiers*

Moses Liskov
Matt Wilhelm
The ElectionGuard Partners mliskov@mitre.org,matt.wilhelm@infernored.com

November 3, 2023

Abstract

This document will serve as the requirements/specification for any Verifier for the ElectionGuard pilot occurring in College Park, Maryland on Sunday, November 5, 2023 using "version 1.91" of the ElectionGuard open source Software Development Toolkit (SDK), developed by InfernoRed Technologies.

This document is a joint publication of the ElectionGuard Partners and the MITRE Corporation National Election Security Lab (NESL).

Introduction

This document will serve as the requirements/specification for any Verifier for the ElectionGuard pilot occurring in College Park, Maryland on Sunday, November 5, 2023 using "version 1.91" of the ElectionGuard open source SDK, developed by InfernoRed Technologies.

This pilot election is based on an "in between" version of ElectionGuard, incorporating some elements of ElectionGuard version 2.0 [1] and some elements of ElectionGuard version 1.53 [2], and in some cases differing from both specifications.

The ElectionGuard team has articulated a scope for verification in this pilot [4], and some verifications are not included in any form here for this reason. Specifically, of the eighteen verifications described in the ElectionGuard version 2.0 specification [1], twelve are required, excluding Verifications 6, 11, 14, 15, 16, 17, and 18, and Verification 7 is reduced in scope. This document does not attempt to speak on the value of this reduced-scope verification; its goal is merely to describe how to verify all the non-excluded verifications.

This document serves two purposes: first, to articulate exactly which checks should be performed in order to verify the election record that will be published in this pilot, and second, to specify exactly where the data to be used in performing these checks should be obtained within the published election record.

This document, combined with the specifications [2, 1] should be sufficient to build a correct ElectionGuard verifier for this pilot.

Global data values and notes

First we define the source for important constants across multiple verification requirements:

$$p = \texttt{constants.json} \rightarrow \texttt{large\_prime} \tag{1}$$
$$q = \texttt{constants.json} \rightarrow \texttt{small\_prime} \tag{2}$$
$$g = \texttt{constants.json} \rightarrow \texttt{generator} \tag{3}$$
$$r = \texttt{constants.json} \rightarrow \texttt{cofactor} \tag{4}$$
$$n = \texttt{context.json} \rightarrow \texttt{number\_of\_guardians} \tag{5}$$
$$k = \texttt{context.json} \rightarrow \texttt{quorum} \tag{6}$$
$$Q = \texttt{context.json} \rightarrow \texttt{crypto\_base\_hash} \tag{7}$$
$$\bar{Q} = \texttt{context.json} \rightarrow \texttt{crypto\_extended\_base\_hash} \tag{8}$$
$$K = \texttt{context.json} \rightarrow \texttt{elgamal\_public\_key} \tag{9}$$

Hash serialization

The verifier shall calculate hash values according to the requirements below. This is how hash calculations were to be performed in version 1.1 of ElectionGuard [3], omitting the case of hashing sublists (e.g. $H(A, (B, C)))$ which are no longer part of any verification requirement.

The ElectionGuard specification requires, in many places, that a SHA-256 calculation be performed on some ordered, structured values. For instance:

$$H_P = H(\texttt{ver}; \texttt{"00"}, p, q, g)$$

Such an instruction is treated as a requirement to perform a particular SHA-256 calculation *modulo* q on some input string derived from the inputs to the H in the given notation.

Serializing a large integer. A large integer is converted to hexadecimal notation with at most 1 leading zero, so as to represent the number as an even number of hexadecimal digits, and represented as a utf-8 encoded string. The hexadecimal digits beyond 9 shall be represented as capital letters A through

F. For instance, the number 1234567891234567 89 would be represented as the string "01B69B4BACD05F15".

Serializing a string. Strings are merely utf-8 encoded and not otherwise altered when serialized.

Serializing a small integer. Small integers are converted to decimal notation and then encoded as a utf-8 string. "Small" integers are ones such as the selection limit or indices, things that don't need to have values approaching the parameters p or q.

Serializing a list. The serialization of a list is calculated by serializing each of the list elements in the given order, with the pipe character (`'|'`) as a separator. The pipe character occurs before each element and also after the final element, so $H(1, 2, 3, 4, 5)$ would be calculated based on the serialization `|1|2|3|4|5|`.

Where a semicolon separator is used in any formula for a hash (e.g. $H(a; b)$) it shall be regarded as no different from a comma separator.

Integerizing a SHA-256 output. A SHA-256 output is treated as a large integer, and ultimately represented as a string. The SHA-256 output is first treated as an array of bytes, then converted to an integer, big-endian style. That integer is then reduced modulo q.

All values described using $H()$ notation are ultimately considered to be numbers modulo $q - 1$ in this sense. When such values are subsequently used as an input to another $H()$ expression, they are treated as large integers and represented in hex as described above.

Examples.

- $H($`"hello world"`$) :=$ SHA256(`|hello world|`) modulo $q =$ 0x3658724c7b35cb1130e4896acfe5903d78bf219e68cf50a3252bf35800174ec6.
- $H(1, 2, 3) :=$ SHA256(`|1|2|3|`) modulo $q =$ 0xe132dc90d35f9705f47bbabf0105c0bf1f10ae13ac463d02067b7ac47955797b.
- $H(12) :=$ SHA256(`|12|`) modulo $q =$ 0xeca46619243c31b97422e995c44293a2fc08e63a0d1d0dbf17e49b462d450ad9
- $H(1,$ `"2|3"`$) :=$ SHA256(`|1|2|3|`) modulo $q = H(1, 2, 3) =$ 0xe132dc90d35f9705f47bbabf0105c0bf1f10ae13ac463d02067b7ac47955797b.

Verification 1: Parameter validation

The verifier shall verify Verifications 1.A through 1.E of [1]. In addition, the verifier shall calculate:

(1.1) $H_P = H(\texttt{ver}; \texttt{"00"}, p, q, g)$

(1.2) $H_M = H(H_P; \texttt{"01"}, \texttt{manifest\_hash})$,

and then check Verification 1.I:

(1.I) The election base hash Q satisfies $Q = H(H_P; \texttt{"02"}, H_M, n, k)$.

Data required.

- For Verification 1.A, the "ElectionGuard specification version used to generate the election record" is `manifest.json` $\rightarrow$ `spec_version`, and the expected value shall be `"1.0"`.
- `ver` shall be the ASCII string `"v2.0"` padded with zeros afterwards to form a 32 byte value. That is,

$$\begin{aligned}\texttt{ver} = \texttt{0x76322E30}\ &\texttt{00000000}\ \texttt{00000000}\ \texttt{00000000}\\ &\texttt{00000000}\ \texttt{00000000}\ \texttt{00000000}\ \texttt{00000000}\end{aligned}$$

- `manifest_hash` = `context.json` $\rightarrow$ `manifest_hash`.

Verification 2: Guardian public-key validation

The verifier shall verify Verifications 2.A and 2.B of [1] and Verification 2.A of [2], which shall be labeled "2.D".

Data required. For each i in $[1, n]$ and each j in $[0, k-1]$, we need:

$$\begin{aligned}v_{i,j} &= \texttt{guardians/guardians\_}n\texttt{.json}\\ &\rightarrow \texttt{coefficient\_proofs}[j]\\ &\rightarrow \texttt{response}\\ c_{i,j} &= \texttt{guardians/guardians\_}n\texttt{.json}\\ &\rightarrow \texttt{coefficient\_proofs}[j]\\ &\rightarrow \texttt{challenge}\\ K_{i,j} &= \texttt{guardians/guardians\_}n\texttt{.json}\\ &\rightarrow \texttt{coefficient\_proofs}[j]\\ &\rightarrow \texttt{public\_key}\end{aligned}$$

Where n and k are globally-defined values, see equations 5 and 6.

Verification 3: Election public-key validation

The verifier shall verify Verifications 3.A and 3.B of [1].

Data required. For each i in $[1, n]$, we need:

$$K_i = \texttt{guardians/guardians\_n.json} \rightarrow \texttt{key}$$

Verification 4: Extended base hash validation

The verifier shall check Verification 4.B:

(4.B) The extended base hash value $\bar{Q}$ satisfies

$$\bar{Q} = H(Q, \texttt{"12"}, K, H_C)$$

Data required. We need $H_C = \texttt{context.json} \rightarrow \texttt{commitment\_hash}$.

Verification 5: Well-formedness of selection encryptions

The verifier shall compute calculations 4.1 through 4.4 of [2] and verify Verifications 4.A through 4.B of [2], which shall be labeled "5.D" through "5.E", respectively. In addition, the verifier shall check Verification 5.F:

(5.F) The equation

$$(c_0 + c_1) \bmod q = H(\texttt{"21"}, \bar{Q}, K, \alpha, \beta, a_0, a_1, b_0, b_1)$$

is satisfied.

Data required. For each ballot in `submitted_ballots`, each contest in that ballot, and each selection in that contest:

$$
\begin{aligned}
v_0 &= \mathit{ballot} \rightarrow \texttt{contests}[] \rightarrow \texttt{ballot\_selections}[] \\
&\quad \rightarrow \texttt{proof} \rightarrow \texttt{proof\_zero\_response} \\
c_0 &= \mathit{ballot} \rightarrow \texttt{contests}[] \rightarrow \texttt{ballot\_selections}[] \\
&\quad \rightarrow \texttt{proof} \rightarrow \texttt{proof\_zero\_challenge} \\
v_1 &= \mathit{ballot} \rightarrow \texttt{contests}[] \rightarrow \texttt{ballot\_selections}[] \\
&\quad \rightarrow \texttt{proof} \rightarrow \texttt{proof\_one\_response} \\
c_1 &= \mathit{ballot} \rightarrow \texttt{contests}[] \rightarrow \texttt{ballot\_selections}[] \\
&\quad \rightarrow \texttt{proof} \rightarrow \texttt{proof\_one\_challenge} \\
\alpha &= \mathit{ballot} \rightarrow \texttt{contests}[] \rightarrow \texttt{ballot\_selections}[] \\
&\quad \rightarrow \texttt{ciphertext} \rightarrow \texttt{pad} \\
\beta &= \mathit{ballot} \rightarrow \texttt{contests}[] \rightarrow \texttt{ballot\_selections}[] \\
&\quad \rightarrow \texttt{ciphertext} \rightarrow \texttt{data}
\end{aligned}
$$

Clarifications. In the formula (4.5) of [2] for the value c, `04` should be interpreted as a string.

Verification 6: Adherence to vote limits

The verifier shall perform no verifications relating to Verification 6 per EG statement of goals [4].

Verification 7: Validation of confirmation codes

The verifier shall perform Verification 7.C of [1].

The verifier shall perform no other verifications related to Verification 7 per EG statement of goals [4].

Data required. For each ballot, $\mathit{ballot} \rightarrow \texttt{code}$ is required.

Verification 8: Correctness of ballot aggregation

The verifier shall perform Verifications 8.A and 8.B of [1].

Data required. For each contest (index cidx) and each selection (index sidx) in `encrypted_tally.json`, we require the values A and B, and also the correct

set of ballot-specific values $\{\alpha_j, \beta_j\}$.

$$A = \texttt{encrypted\_tally.json} \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[sidx] \rightarrow \texttt{ciphertext} \rightarrow \texttt{pad}$$
$$B = \texttt{encrypted\_tally.json} \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[sidx] \rightarrow \texttt{ciphertext} \rightarrow \texttt{data}$$

A ballot *ballot* in the `submitted_ballots` directory shall be considered a "cast" ballot if and only if $ballot \rightarrow \texttt{code}$ does not match $spoiled \rightarrow \texttt{name}$ for any spoiled ballot *spoiled* in the `spoiled_ballots` directory.

Where *ballot* is a *cast* ballot, and $cidx'$ and $sidx'$ are indices such that

$$ballot \rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{object\_id} = \texttt{encrypted\_tally.json} \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{object\_id}$$

and

$$ballot \rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{ballot\_selections}[sidx'] \rightarrow \texttt{object\_id} = \texttt{encrypted\_tally.json} \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[sidx] \rightarrow \texttt{object\_id},$$

include α_j, β_j in the products where

$$\alpha_j = ballot \rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{ballot\_selections}[sidx'] \rightarrow \texttt{ciphertext} \rightarrow \texttt{pad}$$
$$\beta_j = ballot \rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{ballot\_selections}[sidx'] \rightarrow \texttt{ciphertext} \rightarrow \texttt{data}$$

Verification 9: Correctness of tally decryptions

The verifier shall perform Verification 9.A of [1], and shall also check Verification 9.C:

(9.C) The challenge value c satisfies $c = h_8$ where

- $h_1 = H(\texttt{"30"})$,
- $h_2 = H(h_1, \bar{Q})$,
- $h_3 = H(h_2, K)$,
- $h_4 = H(h_3, A)$,
- $h_5 = H(h_4, B)$,
- $h_6 = H(h_5, a)$,
- $h_7 = H(h_6, b)$, and
- $h_8 = H(h_7, M)$.

Data required. For each contest (index $cidx$) within `tally.json` $\rightarrow$ `contests`[] and each selection (index $sidx$) within `tally.json` $\rightarrow$ `contests`[$cidx$] $\rightarrow$ `selections`[], find $cidx'$ be such that

$$\begin{aligned}&\texttt{encrypted\_tally.json} \rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{object\_id} = \\ &\texttt{tally.json} \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{object\_id}\end{aligned}$$

and find $sidx'$ such that

$$\begin{aligned}&\texttt{encrypted\_tally.json} \rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{selections}[sidx'] \\ &\qquad \rightarrow \texttt{object\_id} \\ = \ &\texttt{tally.json} \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[sidx] \\ &\qquad \rightarrow \texttt{object\_id}.\end{aligned}$$

Failure to find $cidx'$ or $sidx'$ shall result in a rejection of these Verifications. We need A, B, T, c, v:

$$\begin{aligned}A = &\texttt{encrypted\_tally.json} \rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{selections}[sidx'] \\ &\rightarrow \texttt{ciphertext} \rightarrow \texttt{pad} \\ B = &\texttt{encrypted\_tally.json} \rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{selections}[sidx'] \\ &\rightarrow \texttt{ciphertext} \rightarrow \texttt{data} \\ T = &\texttt{tally.json} \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[sidx] \\ &\rightarrow value \\ c = &\texttt{tally.json} \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[sidx] \\ &\rightarrow \texttt{proof} \rightarrow \texttt{challenge} \\ v = &\texttt{tally.json} \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[sidx] \\ &\rightarrow \texttt{proof} \rightarrow \texttt{response}\end{aligned}$$

Verification 10: Validation of correct decryption of tallies

The verifier shall perform Verifications 10.A through 10.E of [1].

Data required. For each contest and each selection within that contest, we need T, t where:

$$\begin{aligned}T &= \texttt{tally.json} \rightarrow \texttt{contests}[] \rightarrow \texttt{selections}[] \rightarrow \texttt{value} \\ t &= \texttt{tally.json} \rightarrow \texttt{contests}[] \rightarrow \texttt{selections}[] \rightarrow \texttt{tally}\end{aligned}$$

For Verifications 10.B through 10.E, the "text labels" are:

- contest text labels in tally: `tally.json` $\rightarrow$ `contests`[] $\rightarrow$ `object_id`

- contest text labels in manifest: `manifest.json` → `contests[]` → `object_id`
- option text labels in tally: `tally.json` → `contests[]` → `selections[]` → `object_id`
- option text labels in manifest: `manifest.json` → `contests[]` → `selections[]` → `object_id`
- contest text labels in a submitted ballot: *ballot* → `contests[]` → `object_id`

Verification 11: Correctness of decryptions of contest data

The verifier shall perform no checks relating to Verification 11 per EG statement of goals [4].

Verification 12: Correctness of decryptions for challenged ballots

The verifier shall perform Verification 12.A of [1], and shall also verify Verification 12.C:

(12.C) The challenge value c satisfies $c = h_8$ where

- $h_1 = H(\texttt{"30"})$,
- $h_2 = H(h_1, \bar{Q})$,
- $h_3 = H(h_2, K)$,
- $h_4 = H(h_3, \alpha)$,
- $h_5 = H(h_4, \beta)$,
- $h_6 = H(h_5, a)$,
- $h_7 = H(h_6, b)$, and
- $h_8 = H(h_7, M)$.

Data required. For each challenged ballot *spoiled* and each contest (index *cidx*) within that ballot, and each selection (index *sidx*) within that contest, we need the input values α, β, c, v, S.

The verifier first must find $B(spoiled)$ - a ballot in the `submitted_ballots` directory such that $B(spoiled) \rightarrow \texttt{code} = spoiled \rightarrow \texttt{name}$.

Failure to locate $B(spoiled)$ shall result in a failure of this Verification for this spoiled ballot.

Then we can calculate:

$$\begin{aligned}
\alpha = B(spoiled) &\rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{ballot_selections}[sidx'] \\
&\rightarrow \texttt{ciphertext} \rightarrow \texttt{pad} \\
\beta = B(spoiled) &\rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{ballot_selections}[sidx'] \\
&\rightarrow \texttt{ciphertext} \rightarrow \texttt{data} \\
S = spoiled &\rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[sidx] \\
&\rightarrow \texttt{value} \\
c = spoiled &\rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[sidx] \\
&\rightarrow \texttt{proof} \rightarrow \texttt{challenge} \\
v = spoiled &\rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[sidx] \\
&\rightarrow \texttt{proof} \rightarrow \texttt{response}
\end{aligned}$$

Verification 13: Validation of correct decryption of challenged ballots

The verifier shall perform Verifications 13.A through 13.F of [1].

Data required. For Verifications 13.A-C, we require the following for each spoiled ballot *spoiled*, each contest (index *cidx*) in that ballot, and each selection within that contest: S, σ, L.

The S and σ values are:

$$\begin{aligned}
S &= spoiled \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[] \rightarrow \texttt{value} \\
\sigma &= spoiled \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{selections}[] \rightarrow \texttt{tally}
\end{aligned}$$

The vote limit L is determined by the manifest. First we must find $cidx'$ such that

$$\begin{aligned}
&\texttt{manifest.json} \rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{object_id} = \\
&spoiled \rightarrow \texttt{contests}[cidx] \rightarrow \texttt{object_id}
\end{aligned}$$

Failure to find any such *cidx* shall result in a failure of this Verification.

Then,

$$L = \texttt{manifest.json} \rightarrow \texttt{contests}[cidx'] \rightarrow \texttt{votes\_allowed}.$$

Again, do not assume that the range limit in the corresponding encrypted ballot accurately reflects the vote limit.

For Verifications 13.D through 13.F, the "text labels" are:

- contest text labels in spoiled ballot: $spoiled \rightarrow \texttt{contests}[] \rightarrow \texttt{object\_id}$
- contest text labels in manifest: $\texttt{manifest.json} \rightarrow \texttt{contests}[] \rightarrow \texttt{object\_id}$

- option text labels in spoiled ballot: *spoiled* → `contests`[] → `selections`[] → `object_id`
- option text labels in manifest: `manifest.json` → `contests`[] → `selections`[] → `object_id`

Clarifications. In all cases for Verification 13.B, the range of "valid values" shall be the set $\{0, 1\}$.

Verifications 14-18

The verifier shall perform no verifications corresponding to Verifications 14, 15, 16, 17, or 18, per EG statement of goals [4].

References

[1] Josh Benaloh and Michael Naehrig. ElectionGuard Design Specification. Version 2.0.0, August 15, 2023.

[2] Josh Benaloh and Michael Naehrig. ElectionGuard Design Specification. Version 1.53, January 20, 2023.

[3] Josh Benaloh and Michael Naehrig. ElectionGuard Specification v.1.1.

[4] The ElectionGuard Partners. College Park, Maryland, 2023 information page on electionguard.vote.