2022 Re-evaluation of the Swiss Post e-Voting System (Addendum II)

Audit Scope 1: Cryptographic Protocol

Aleksander Essex

Department of Electrical and Computer Engineering
Western University, Canada
aessex@uwo.ca

February 13th, 2023

Submitted to the Swiss Federal Chancellery

Management Summary

In cooperation with the Chancellery, I performed a re-evaluation of the Swiss Post e-voting system in 2022. This effort examined Swiss Post's responses to recommendations made in my in original 2021 evaluation. Of 73 issues identified in the original evaluation, Swiss Post has addressed (and hence resolved) 59.

Based on positive and constructive conversations with Swiss Post and the Chancellery over the past year, I remain optimistic that they will continue to work toward addressing the remaining issues. In the meantime, this document (Addendum II) expands on my 2022 re-evaluation with several new findings and recommendations:

- In the case of digital identity, Swiss Post created a new section on digital certificates and digital signatures. Unfortunately, the specification suffers from simultaneously trying to be general and specific, which leads to several ambiguous areas of practical importance.

- In the case of primality testing, Swiss Post changed their proposal to a somewhat non-standard combination of algorithms, which is unsupported by a concrete security bound. In the case of parameter generation, Swiss Post modified its proposal for choosing generators. In both cases, the motivation and justification for the changes need to be explained.

- As part of this examination, I wrote a script to verify the Swiss Post-provided test vectors for the verifiable parameter generation algorithm. In the process of this exercise, I observed that the verifiable parameter generation did not enforce domain separation between entities in the hash pre-image. This appears to give the election administrators higher-than-intended degrees of freedom in selecting discrete-logarithm domain parameters.

1 Description

This document is a second follow-up addendum (referred to as Addendum II) to the report entitled "2022 Re-evaluation of the Swiss Post e-Voting System (Addendum)," dated November 21st, 2022 (referred to as Addendum I). My re-evaluation report (Nov. 2022) and the accompanying Addendum I (Dec. 2022) focused on commenting on progress made in addressing existing JIRA-catalogued issues arising from my Final Report (Dec. 2021). This report (Addendum II) focuses on new observations in the cryptographic primitives specification (version 1.2.0).

1.1 Documents Examined

Below is a list of the versions of my reports submitted to the Chancellery, the version of the primitives specification examined in my report, and the date that version was published by Swiss Post.

Primitives Specification

Description: Pseudocode specifications of cryptographic functions used by the Swiss Post system. Referred to throughout this document as the *primitives specification*.

Report	Version Examined	Date Published
2021 Preliminary Report	0.9.5	2021-06-22
2021 Final Report	0.9.8	2021-10-15
2022 Re-Examination	1.0.0	2022-06-24
2022 Re-Examination (Addendum I)	1.0.0	2022-06-24
2023 Addendum II	1.2.0	2022-12-09

Available: https://gitlab.com/swisspost-evoting/crypto-primitives/crypto-primitives/-/blob/master/Crypto-Primitives-Specification.pdf

2 Findings and Recommendations

In this section, I comment on new findings in the primitives specification (version 1.2.0).

2.1 Resolved Issues

Of the 73 outstanding issues identified in my Re-evaluation Report Addendum I (Nov. 2022), 14 issues remained unresolved as of primitives specification version 1.0.0. Although I have not conducted an exhaustive re-evaluation of my *re-evaluation*, as of version 1.2.0, at least one of the issues has been resolved:

VE-4961: Recursive Hash – Infinite Input (Resolved)
Summary: Unusual wording about avoiding "infinite input in practice" (pg. 14 [1]).
Action Taken: Wording addressed in primitives specification (version 1.2.0).

2.2 Digital signatures

This section examines and remarks on the newly added Section 6 (Digital signatures) of the primitives specification. Swiss Post acknowledges that this section departs somewhat from the rest of their document in terms of presenting a high-level description of the functionality while leaving the details to "well-established standards."

This approach seems reasonable, but a tension exists between Swiss Post's attempt to be simultaneously general and specific.

- The GetCertificate function is left completely unspecified, which leaves some ambiguity about key fields/inputs. For example:
  - The X.509 standard requires a serial number to be designated by the issuer.[1] GenKeysAndCert (Algorithm 6.1), makes no mention of serial numbers. Are they produced by GetCertificate? Should the serial numbers be unique? How is this enforced?
  - The X.509 `signatureAlgorithm` field contains the identifier for the cryptographic algorithm used to sign the certificate. This is not specified as an input to GetCertificate, which suggests GetCertificate infers from the key pair. This is ambiguous, especially with regard to the hash function, which cannot be inferred from a digital signature key pair alone. Note that Table 2 in Section 2 does not specify the hash function to be used in the signature algorithm.
  - Broadly citing X.509 is ambiguous. A version should be specified. For example, GenKeysAndCert specifies an explicit usage field in Line 4. The X.509 standard did not have a Key Usage field until version 3 when certificate extensions were introduced.[2]
  - If certificate extensions are available, then it should be clarified whether the Basic Constraints field is needed to permit self-signed certificates.
- GetCertificate is defined as a function that specifically returns a *self-signed* certificate. In keeping with clear function naming, the self-signed aspect should be communicated in the function name to distinguish it from certificates issued based on 3rd-party certificate signing requests.
- Since certificate validation is being discussed (Section 6.2), the issue of revocation should be addressed, if only to clarify whether revocation should be part of the threat model or not.

[1] https://www.ietf.org/rfc/rfc2459.txt
[2] https://datatracker.ietf.org/doc/html/rfc5280

– The GenSignature function (Section 6.3, Algorithm 6.2) specifies the hash operation as taking place *outside* of the signing function, which conflicts with, e.g., FIPS 186-4.[3]

– Pursuant to the previous point, RecursiveHash is not a valid X.509 hashing algorithm[4] although the underlying SHA3/SHAKE hash algorithms specified in Table 2 is X.509 now are.[5]

– The previous point also applies to VerifySignature in Algorithm 6.3.

– GenSignature outputs "the signature for the message $\in \mathcal{B}^*$." Clarify if the message is $\in \mathcal{B}^*$ or if the signature is $\in \mathcal{B}^*$. Note that GenSignature defines the message $\in \mathcal{V}$, and VerifySignature on the following page defines the signature as $s \in \mathcal{B}^*$. However, similar to a notation issue identified in my 2021 report, the Kleene star denotes the signature can be of arbitrary length, meaning this notation would permit short (and hence insecure) signatures.

2.3 Primality Testing

The description of primality testing has changed again in the primitives specification: "We use a probabilistic variant of the Baillie-PSW deterministic test based on two rounds of probabilistic Miller-Rabin [and . . .] deterministic Lucas-Lehmer testing." This change appears to be an attempt to address my comments in issue VE-4969 and my subsequent re-examination (2022) where I recommended retaining Miller-Rabin because of well-defined bounds on probabilities or, if Baillie-PSW was retained, that it be used to replace *some* rounds of Miller-Rabin (as in GMP). This new description, however, still has some issues:

– Lucas-Lehmer is a primality test for Mersenne primes p, q of the form $p = 2^q - 1$. This form does not apply to the Swiss Post context where $p = 2q + 1$. I believe they mean *strong Lucas probable prime test.*

– Swiss Post refers to the Lucas (Lucas-Lehmer [*sic.*]) test as deterministic, implying it always correctly identifies whether a number is prime (or not). However, the Lucas test is probabilistic. A number that passes the test is a Lucas *probable prime*, as there exist composite integers (Lucas pseudoprimes) which pass the test.

– The standard version of the Baillie-PSW test begins with a base-2 strong pseudoprime test.[6] Changing this to two rounds of Miller-Rabin (i.e., strong pseudoprime tests on two *randomly* chosen bases) seems unusual. For example, GMP's implementation of the Miller-Rabin test actually performs a Baillie-PSW test first (which includes the strong pseudoprime test base-2) and only performs additional repetitions of Miller-Rabin (i.e., strong pseudoprime tests with random bases) if the function is called with a parameterization *above* the default 25 repetitions (a recommendation which the GMP designers attribute to Knuth).[7]

[3] https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
[4] https://datatracker.ietf.org/doc/html/rfc5758
[5] https://datatracker.ietf.org/doc/html/rfc8692
[6] Base-2 is not required but is a common and well-studied parameterization.
[7] https://github.com/alisw/GMP/blob/master/mpz/millerrabin.c

In essence, Swiss Post has modified its primality testing algorithm to something that is non-standard. It is neither (a) the standard Baillie-PSW test nor (b) a proper Miller-Rabin test. I am not sure whether any existing library implements this particular version.

Recommendation: Reference a well-established primality testing algorithm with concrete parameters (and, ideally, a concrete implementation). Whatever primality testing algorithm Swiss Post ultimately picks, the motivation and security guarantees should be stated. For example, Miller-Rabin has concrete asymptotic upper bounds on the probability that a composite number will (falsely) pass the test. The Baillie-PSW test does not have such a bound. Although no counter-example (pseudoprime) has ever been found, and their existence is only conjectured, it means their distribution is, similarly, unknown.

2.4 Parameters Generation

I re-examined changes to the GetEncryptionParameters function in the primitives specification.

Generator selection. The function outputs a generator of $\mathbb{G}_q$, which was originally specified to be either $g = 2$ or $g = 4$. In this situation, $g = 4$ is guaranteed to be in $\mathbb{G}_q$.

This function has now been changed to instead output either $g = 2$ or $g = 3$. Swiss Post observes that if p is a safe-prime, and $2 \notin \mathbb{G}_q$, it necessarily implies $3 \in \mathbb{G}_q$. I accept this observation. However, the explanation at the beginning of Section 7.2 (Parameters Generation) does not explain how it follows from quadratic reciprocity. The reasons for this property should be made explicit, perhaps by adding a short theorem in an appendix. Alternatively, since picking between $g = 2$ and $g = 3$ offers no obvious security benefit (for reasons argued in my previous submissions), they could simply pick $g = 4$ and skip the theorem.

Test Vectors. The recent versions offer a set of test vectors in a file (get-encryption-parameters.json), which is embedded in the PDF of the primitives specification document. I wrote a short Python program (See Appendix A), which confirmed the test vectors' validity. I also used it to verify my understanding of the generation of q from the *seed* value, which is relevant to the following section.

2.5 Unintended Degrees of Freedom in Parameters Generation

The hashing operation in Line 3 of GetEncryptionParameters does not enforce domain separation, which could provide a malicious election authority with greater than intended degrees of freedom in picking domain parameters.

The domain parameters p, g, q are functionally dependent on the election's name. Here, the hash pre-image is a concatenation of an election name (*seed*) and an iteration count (i). However, neither the *seed* nor the iteration count have a fixed length, which makes it possible for different (distinct) combinations of *seed*/i pairs to create hash collisions and hence identical domain parameters across distinct elections.

First, observe value $seed \in \mathbb{A}_{UCS}$ is an ISO/IEC10646 string. *StringToByteArray* in Algorithm 3.11 specifies this as UTF-8. There appears to be no stated format for an election name (*seed*) in the primitives, protocol or system specifications. For example,

the two sets of test vectors in get-encryption-parameters.json were given as string-encoded integer values, i.e., "20782" and "65684," which are not semantically descriptive as an election name.

Consider an example where the *seed* value is manipulated by adding an additional character, such as the space character. First, observe the space character "␣" is encoded as the byte `0x20` in UTF-8. Now consider the following two cases:

1. *seed* = "`Vote 2020`␣" and $i = 0$ (i.e., `0x00`):
   - $(seed \,||\, i)$ = `\x56\x6F\x74\x65\x32\x30\x32\x30\x20\x00`
2. *seed* = "`Vote 2020`" and $i = 8192$ (i.e., `0x2000`):
   (a) $(seed \,||\, i)$ =`\x56\x6F\x74\x65\x32\x30\x32\x30\x20\x00`

Hence, even though seeds "`Vote 2020`␣" $\neq$ "`Vote 2020`," there exist reasonably small iteration counts ($i = 0$, resp. $i = 8192$) for which both seeds have the same SHAKE128 pre-image, and hence produce the same candidate prime q.

The probability that an odd k-bit integer q is a Sofie-Germain prime (i.e., a prime for which $p = 2q + 1$ is also prime) is given by the following heuristic estimate [2]:

$$P(p, q \in \mathbb{P}) = \frac{1}{2} \cdot \left(\frac{1}{\log^2(2^k)} \right).$$

At the extended security interval where $|q| = 3071$, this implies that execution of the loop of GetEncryptionParameters (Algorithm 7.1) will see i increment from 0 to an average value of $2,265,586$.

In byte-level terms, the IntegerToByteArray function will return values ranging from `\x00` to `\x22\x91\xf2`, meaning i ranges from 1 to 3 bytes in length in an average execution. This means the second last and third last bytes of the SHAKE128 pre-image are ambiguous, belonging either to *seed*, or to i. Given a particular *seed* value as input, this means an adversary has multiple opportunities to find another distinct but valid UTF-8 *seed* value producing an identical q in a single execution of GetEncryptionParameters. Running this experiment on polynomially many seed values would eventually lead to a q for which $2q + 1$ was also prime.

Hence it is computationally feasible to find two different election names $seed_1 \neq seed_2$ for which GetEncryptionParameters returns the same domain parameters.

Recommendation: Enforcing a format on the *seed* value may not be sufficient to prevent this issue. Instead, I would recommend enforcing domain separation between the *seed* and counter (i) by using an approach similar to RecursiveHash, which is adapted to the larger output produced by SHAKE128. As an example, Line 3 of GetEncryptionParameters could use SHA256 on each domain individually and SHAKE128 on their concatenation:

$$\hat{q} \leftarrow \mathsf{SHAKE128}\Big(\mathsf{SHA256}(\mathsf{StringToByteArray}(seed)) \,||\, \mathsf{SHA256}(\mathsf{IntegerToByteArray}(i)), \frac{|p|}{8}\Big).$$

This approach creates domain separation by pre-hashing the *seed* and counter i to their own separate fixed-length byte arrays. It is proposed for illustrative purposes and without a security analysis.

References

1. A. Essex. Analysis of the Swiss Post e-Voting System. Audit Scope 1: Cryptographic Protocol. In *Report Submitted to the Swiss Federal Chancellery*, 2021.
2. V. Shoup. *A Computational Introduction to Number Theory and Algebra.* Cambridge University Press, 2009.

A Validation of GetEncryptionParameters Output

```python
import hashlib
import gmpy2
import json

max_counter = 2 ** 20  # Enforce a halting condition
f = open('get-encryption-parameters.json')
params = json.load(f)

for param_instance in params:
    bit_len = int(param_instance["input"]["bit_len"])
    seed = param_instance["input"]["seed"].encode('utf-8')
    q_asserted = param_instance["output"]["q"]
    p_asserted = param_instance["output"]["p"]
    g_asserted = param_instance["output"]["g"]

    if q_asserted[0:2] == "0x":
        q_asserted = q_asserted[2:]
    if p_asserted[0:2] == "0x":
        p_asserted = p_asserted[2:]

    q_asserted = int(q_asserted, 16)
    p_asserted = int(p_asserted, 16)
    g_asserted = int(g_asserted, 16)

    i = 0
    while True:
        i_byte_len = (i.bit_length() + 7) // 8
        ctr = i.to_bytes(i_byte_len, byteorder='big')
        # No domain separation of variable length seed/ctr values
        qb_hat = hashlib.shake_128(seed + ctr).hexdigest(bit_len // 8)
        qb = '01' + qb_hat
        q = int(qb, 16) >> 2
        q = q + 1 - (q % 2)
        i += 1

        if q == q_asserted:
            break

        if i > max_counter:
            assert q == q_asserted

    p = 2 * q + 1
    if gmpy2.powmod(2, q, p) == 1:
        g = 2
    else:
        g = 3

    assert p_asserted.bit_length() == bit_len
    assert bit_len % 8 == 0
    assert p == p_asserted
    assert q == q_asserted
    assert g == g_asserted
    assert gmpy2.is_prime(q)
    assert gmpy2.is_prime(p)

print("Parameters successfully verified")
```