How not to prove your election outcome

The use of non-adaptive zero knowledge proofs in the Scytl-SwissPost Internet voting system, and its implications for decryption proof soundness

Sarah Jamie Lewis[1], Olivier Pereira[2], and Vanessa Teague[3]

[1]Open Privacy Research Society, sarah@openprivacy.ca
[2]UCLouvain – ICTeam, B-1348 Louvain-la-Neuve, Belgium,
olivier.pereira@uclouvain.be
[3]The University of Melbourne, Parkville, Australia,
vjteague@unimelb.edu.au

March 25, 2019

We show that a weakness in the SwissPost-Scytl implementation of the Fiat-Shamir transform allows the creation of false decryption proofs, which verify perfectly but actually "prove" a decryption that is different from the true plaintext.

This could, for instance, be used by a cheating decryption service to change valid votes into nonsense that would not be counted. This attack could have a political effect if the attacker knew which votes supported a party it wanted to harm.

Although it would be informally apparent that something had gone wrong, the formal verification process would pass. This contradicts the complete verifiability property that this voting system is supposed to offer.

If the decryption proofs were mistakenly believed to be sound, it seems that our exploit would put the system in an "impossible state", which would make it difficult to define a meaningful investigation process.

We have provided two cheating decryption proof transcripts with this report, which verify but do not claim the correct plaintext.

SwissPost have not yet confirmed our analysis, and NSWEC claim that this problem does not affect the iVote system.

We also list a collection of other issues in the implementation of non-interactive zero knowledge proofs. These cause concern, though it is not immediately obvious how they could be exploited.

1 Introduction

In the SwissPost-Scytl sVote voting system, there are two main parts to the proof that the votes received by the system are properly counted. The first is a series of proofs that the votes are properly shuffled—a sequence of mixing servers permutes and re-randomises them, then provides a proof that the set of votes represented by the input ciphertexts matches those in the output. The second is decryption: the mixed ciphertexts must be decrypted, and proven to contain the plaintexts that the authorities claim. In sVote, each mixer performs a partial decryption after it shuffles.

In earlier work [LPT19] (found independently by Haines and also Haenni [Hae19]), we showed that the SwissPost-Scytl implementation of the shuffle proof contained a trapdoor that permitted mixers to manipulate votes but pass verification.

In this work, we show a similar result for the decryption proof: an error in the implementation of the Fiat-Shamir heuristic allows a cheating authority to produce a proof of proper decryption, which passes verification, but declares something other than the true plaintext.

This voids the soundness of the decryption proof and, in effect, the arguments that sVote audit offers complete verifiability: since the verification procedure is based on an assumption that we show to be false, no conclusion can be made from its successful completion. Interestingly, we observe that the decryption proof described in the Verifiability Security Proof report [Scy18b] is different from the one that appears in the sVote protocol specification [Scy18c], which is the one implemented in the system under review. The proof that is described in that Security Proof report appears to be correct.

In order to demonstrate one possible impact of the lack of soundness of this decryption proof, we exhibit an exploit in which a malicious authority (e.g., the CCM_1 of the system) modifies selected votes during the (partial) decryption procedure and forges decryption proofs that are indistinguishable from valid ones, and would therefore pass verification.

This specific exploit has two limitations, but we do not rule out that there are other and possibly more dangerous ways of exploiting the same weakness.

1. In order to fake the decryption proof and also complete a valid proof of shuffle, the cheating CCM needs to know the randomness used to encrypt the votes that it wants to modify. This can be accomplished by corrupting a voting client, or by a poor random number generator.

2. The cheating authority cannot declare an arbitrary false plaintext while also making the shuffle proof work. But it can, for any ciphertext, prove that it decrypts

to something other than the truth. The exploit produces an output vote that will probably be nonsense rather than a valid vote.[1] This exploit could then be used to political advantage to nullify only those votes with which the cheater disagreed.

We should emphasise that SwissPost have not yet confirmed our findings, and that NSWEC claims that their decryption proofs are not affected.[2]

1.1 Is this detectable or attributable to the cheating prover?

Because of these invalid votes, this exploit will probably leave evidence that something went wrong. According to the sVote audit specification [Scy18a, Section 5, Step 2, p.57], the invalid votes are stored in a `auditableVotes.csv` file, and the audit verifies that all the ballots included in that file are invalid indeed. So, the ballots for which fake decryption proofs have been produced will be written in that file and, according to the audit specification, the verification will formally pass.

If someone wishes to push investigations further, one may wonder how invalid ballots were accepted in the ballot box and tallied. The sVote spec [Scy18c, p.117] states, "Usually these errors should not happen since the value encrypted by the voting client is the product of valid prime numbers included in the ballot." It is not clear what "usually" means, or what would be inferred if these errors happened.

For regular (i.e., non write-in) votes, it appears that this should just not happen under the proposed trust model. The zero knowledge proofs of valid vote construction [Scy18c, Section 5.4.1], produced in the voting client, are expected to prove some internal consistency in the ballots (even if they do not include a proof that the vote is the product of the prime numbers it should be). However, there is another step [Scy18c, Section 5.4.4, Step 1] in which the Vote Verification Context derives the Choice Return Codes, and that step would normally fail if the vote is not the product of the expected primes. As a result, it seems that our exploit would put the system in an "impossible state", which would make it difficult to define a meaningful investigation process. If the possibility that the cryptographic algorithms are broken is considered (but possibly without really knowing which ones), then it might eventually be possible to identify the cheater by requiring the CCM's to release their secret key. It is certainly unclear how to run such an exceptional investigation without breaking the privacy of some votes.

For write-in votes, the individual verifiability mechanisms do not offer any guarantee that the submitted write-ins make any sense (this would be complicated, since these can essentially be anything). So, our exploit would offer a way to transform valid write-ins into senseless votes, and such a situation would be consistent with a voter willing to express a senseless write-in, or with a corrupted voting client.

[1] Note that we are not sure whether this is also true for other elections, such as New South Wales, that express votes differently from Switzerland.

[2] `https://www.elections.nsw.gov.au/NSWEC/media/NSWEC/Media\%20Release/190322-NSW-Electoral-Commission-iVote-and-Swiss-Post-e-voting-update.pdf`

1.1.1 Summary

Formally, verification would pass. Informally, it would be apparent there was a problem. But, if the weakness that we identified in the Fiat-Shamir transform were not known, the path towards a proper diagnosis of this problem would be quite difficult to execute, in particular without violating the privacy of some votes.

1.2 Implementation

We have provided with this report two examples of decryption proofs that pass verification but change the plaintext—the method is described in Sections 3.1 and 3.2.

We have not implemented the inclusion of the fake decryption proof into the sVote mixing and decryption sequence (Section 3.3).

This document points to several other issues in the zero knowledge proofs implemented in sVote. We do not know if these can lead to other exploits.

2 Pitfalls of the Fiat-Shamir transform

The Fiat-Shamir transform [FS86] is a standard method of turning an interactive proof into a non-interactive one. Informally, the idea is simple: rather than waiting for the verifier to generate a random challenge, the prover generates a challenge by hashing the prover's initial commitments. This can be proven to be secure assuming that the hash function behaves as a random oracle.

As such, this strategy only works in a non-adaptive setting, in which the proof statement is given in advance to the the prover and verifier. However, when moving to an adaptive setting, in which the prover can choose the statement about which it wants to make a proof, it becomes crucial to also include that statement, in full, into the inputs of the hash function—soundness collapses otherwise [BPW12].

The requirement of adaptive security is quite common in voting systems and, as we demonstrate here, it is needed for the sVote protocol.

The problem In the sVote system, neither the protocol specification, nor the code, always includes the full statement to be proven in the inputs to the hash function that is used to generate the challenges in zero knowledge proofs.

For example, the proof of proper decryption of a ciphertext (C_0, C_1) does not hash C_0. This allows a cheating prover to compute a valid proof, then choose a statement as a function of that proof, which breaks the soundness of the proof.

Fixing the problem All uses of the Fiat-Shamir transform should include all data, including the statement to be proven, as input to the hash. In particular, the proof of proper decryption should include C_0 (and the base group generator g used to compute C_0 would be good too).

This is not only true for the decryption proof, but also for the other proofs that are present in the system.

3 Producing a false decryption proof

Remember that an ElGamal encryption of message m with public key pk is a pair $(C_0, C_1) = (g^r, m(pk)^r)$. A proof of proper decryption—that the ciphertext (C_0, C_1) decrypts to message m—can be constructed by anyone who knows the secret key x s.t. $pk = g^x \bmod p$. It consists of a proof that

$$\mathsf{dlog}_g pk = \mathsf{dlog}_{C_0}(C_1'), \text{ where } C_1' = C_1/m. \tag{1}$$

The Chaum-Pedersen proof sVote's Decryption proof [Scy18c, Section 9.1.8] uses a non-interactive version of Maurer's Unified Proofs Prover [Scy18c, Section 9.1.14] to prove Equation 1. The method follows a well-known proof method due to Chaum and Pedersen [CP92].

Suppose the prover (who knows x) has an ElGamal ciphertext (C_0, C_1), computed with generator g and public key $pk = g^x$. She wishes to prove that this ciphertext decrypts to m, so she computes $C_1' = C_1/m$ and proves that C_1' is a correct decryption factor for that ciphertext, that is $C_1' = C_0^x$, or equivalently that (g, pk, C_0, C_1, m) satisfy Equation 1. She computes a Chaum-Pedersen proof as follows:

1. Pick a random a.
2. set $B_0 = g^a$ and $B_1 = C_0^a$.
3. **Compute $\mathbf{c = H(pk, C_1', B_0, B_1)}$,** where H is a cryptographic hash function.
4. Compute $z = a + cx$.

The proof is (c, z). The verification proceeds by recomputing $B_0 = g^z(pk)^{-c}$ and $B_1 = C_0^z(C_1')^{-c}$, then verifying that $c = H(pk, C_1', B_0, B_1)$.

3.1 Exploiting the problem

The problem is that in Step 3, C_0 is not included in the hash (and nor is g). This allows an adaptive cheating prover to generate a fake proof by first calculating c, then choosing C_0 afterwards. Here is how this can work.

1. Pick a random a, a random s, and a random t.
2. Set $B_0 = g^a$, $C_1' = g^s$ and $B_1 = g^t$.
3. Compute $c = H(pk, C_1', B_0, B_1)$ (as expected).
4. Compute $z = a + cx$ (as expected).

5. Set $C_0 = (B_1(C'_1)^c)^{\frac{1}{z}} = g^{(t+sc)/z}$.

It can be observed that (c, z) pass verification as a decryption proof that (g, pk, C_0, C'_1) satisfy Equation 1. However, it is highly unlikely that this is truthful, *i.e.* that $C'_1 = C_0^x$. Taking logarithms base g, this equation would be satisfied only if $s = x(t + sc)/z \pmod q$, *i.e.*, if $s(z - cx) = xt \pmod q$ or, using the definition of z, if $sa = xt \pmod q$. But a, s and t are independent values chosen from $\mathbb{Z}_q$ (where q is the size of the ElGamal group, around 2^{2047} for the proposed parameters), so this coincidence occurs with negligible probability. Hence we have a valid proof for a fact that is not true.

We note that the cheating strategy described above does not depend on a, s and t to be random: any value could be picked for them, and the proof would still be considered to be valid. It is just unclear whether specific choices could lead to a more dangerous attack.

We also want to stress that this issue is not present in Maurer's framework [Mau09]: this framework focuses on the interactive proof setting, and therefore does not rely on the Fiat-Shamir transform.

3.2 Transforming to a set of fake decryption proofs that pass verification

A decryption proof [Scy18c, Section 9.1.8] proceeds by stating a ciphertext (C_0, C_1), declaring a plaintext P_1 and then performing a Chaum-Pedersen proof on $(g, pk, C_0, C_1/P_1)$. But there is nothing that forces a malicious prover to do things in that order.

For instance, we can start from a cheating Chaum-Pedersen proof as above and use it to produce a set of cheating decryption proof transcripts: given a forged proof (c, z) and the corresponding pair (C_0, C'_1), simply set $C_1 = C'_1 P$ and declare it to be a valid encryption of P. Whether this is true or not, the fake proof will support it.

3.3 Incorporating a fake decryption proof into the sVote mixing and decryption sequence

An attacker can exploit the flaw in the Chaum-Pedersen protocol described above, because sVote has a very specific feature: each mixer performs a shuffle and a (partial) decryption of the output of that shuffle. This means that a mixer can proceed exactly as above: compute a fake decryption proof and a matching ciphertext (C_0, C'_1), then define its shuffle in such a way that a ciphertext with the right C_0 comes out of it.

More precisely, suppose that CCM_1 has, as part of its list of input ciphertexts for mixing and partial decryption, a ciphertext (D_0, D_1). It needs to include this ciphertext in its shuffle and output a partially decrypted version of it, together with proofs that these operations were performed correctly. But it wishes to modify the contents of that ciphertext, so that it becomes invalid and will not be taken into account in the tally. It will do this by declaring a false decryption and proving it to be correct using the cheating proof described above.

First, it produces a fake decryption proof and a matching ciphertext (c, z, C_0, C'_1), as described in Section 3.1.

Now, in order to make it possible to produce a proof of shuffle (assuming that this proof is sound), it needs to define C_1 such that the pair (C_0, C_1) is actually a re-encryption of (D_0, D_1). To this purpose, it computes $E_0 = C_0/D_0$ and $E_1 = E_0^x$. Now (E_0, E_1) is an encryption of 1 such that $(D_0, D_1) \cdot (E_0, E_1) = (C_0, D_1 E_1)$. So, it can simply set $C_1 = D_1 E_1$, which is a true re-encryption of the vote. Then setting P to C_1/C'_1 makes (c, z) a valid proof that (C_0, C_1) decrypts to P, though this is (almost certainly) not true.

One last difficulty is to make the proof of shuffle work. The ciphertext (C_0, C_1) is a re-encryption of (D_0, D_1), so the shuffle is still valid. However, the proof requires knowing the randomness used in the re-encryption factor, which CCM_1 does not know, unless it has the discrete log of D_0 in base g. Indeed, if $D_0 = g^r$, then $E_0 = g^{(t+sc)/z-r}$, and the reencryption factor needed to complete the proof of shuffle is $\frac{t+sc}{z} - r$.

This (like the first example in our previous paper) would require information leakage from the voting client to the server (CCM_1). This does not seem an excessive requirement, when both are implemented by the same corporation and administered by the same authority. It is certainly inconsistent with the claim of complete verifiability that such information leakage should allow electoral manipulation.

3.4 On the possibility to cheat and produce valid ballots.

In the attack described here, assuming the attacker must also generate a true shuffle proof, the attacker gets an effectively random plaintext. The attack would of course be much more stealthy if that random plaintext corresponded to the encoding of a valid vote. In the Swiss system, this seems highly unlikely, since only a negligible fraction of plaintexts are valid votes, and the resulting vote will just be invalid.

In other places, it depends critically on the method for encoding the votes. For example, the set of valid votes in New South Wales is very large, because a valid vote may be a permutation of more than 340 candidates. Since $\log_2(340!) > 2047$, there must be some vote encoding using more than one 2048-bit ElGamal ciphertext pair. So we do not know whether a randomly selected plaintext has a non-negligible chance of matching a valid vote. If it did, and if the NSW decryption proof were implemented in the same way as sVote's, the attack could change a valid vote into another, unrelated, valid vote.

3.5 Summary

Suppose that the first mixer (CCM_1) is corrupted and that he can obtain the randomness used for encryption by some voters. Then this mixer can produce a valid shuffle proof and a fake decryption proof for the ciphertexts produced by these voters, so that their votes become invalid. If this mixer knows the randomness, he will of course focus on invalidating votes for candidates that he does not like.

4 Other issues affecting proof soundness

Here is a short list of other problems we have noticed.

4.1 The Fiat-Shamir transform in other proofs

The Fiat-Shamir heuristic is used throughout the sVote code base, so there may be numerous other examples in which the proofs are not sound. We did not check most of them, and the impact of a lack of soundness may vary quite a lot: the errors described in our previous report [LPT19] and in the current document break soundness of proofs in both cases, but they lead to very different exploits. In particular, all the sVote proofs based on a non-interactive variant of Maurer's generic protocol (including Schnorr's proof, the Exponentiation proof, ...) appear to not be adaptively secure, and these proofs are used by the voting clients and by the CCRs. It is plausible that this weakness could be exploited.

To illustrate our concern, we show a second brief example with the Schnorr proof. The sVote Audit document [Scy18a, Section 11.1.1 & 11.1.2] describes the construction of Schnorr proofs, which are proofs of knowledge of r such that $C = g^r$. The proof is (roughly) computed by computing $B = g^a$, then $c = H(C|B)$, then $z = a + cr$. The verification proceeds by verifying that $g^z = BC^c$ and $c = H(C|B)$. But the proof itself contains no reference to g (g is not input to the hash function), even though g is definitely part of the statement.

As a result, for any given C and B in the group, we can compute $c = H(C|B)$ and pick a random z, then decide that $g = (BC^c)^{1/z}$. This would make (according to the protocol specification) a valid Schnorr proof of knowledge of the discrete log of C in base g, even though there is no reason to think that the prover truly knows that discrete log.

It is unclear how this alone would lead to an attack on the system. The Schnorr proof is used by the CreateVote algorithm to produce proofs w.r.t. the standard group generator g, which is not picked by the prover in this case. Is is still uncomfortable that the soundness of this proofs depends on external factors.

4.2 Missing verification steps in OR Proof

The code base defines a 1-out-of-n zero knowledge proof construction (ORProof)[3] that contains a critical flaw which would allow a malicious prover to trick the verifier into accepting an element that does not belong to the required set of n elements.

The protocol implemented appears to follow the disjunctive proof approach of Cramer, Damgård and Schoenmakers [CDS94] for Sigma protocols (among which the Schnorr and the Chaum-Pedersen protocols discussed above). In well defined 1-out-of-n protocols, there is one challenge c sent by the verifier (or provided by a random oracle), and the prover needs to answer with n more challenges $c_1, \ldots, c_n$, such that $c = \sum_{i=1}^{n} c_i$. The

[3] The proof appears in scytl-cryptolib/cryptolib-proofs/src/main/java/com/scytl/cryptolib/proofs/maurer/factory/ORProofGenerator.java and its verification process in scytl-cryptolib/cryptolib-proofs/src/main/java/com/scytl/cryptolib/proofs/maurer/factory/ORProofVerifier.java

verifier must then verify this equality: if it is not satisfied, then the soundness of the protocol completely breaks, and the prover can make a proof that passes verification even if all of the n statements are false.

In the Swiss Post/Scytl code base this check is not performed, the result of which is that the Verifier can be tricked into verifying proofs that do not encode any of the elements that the OR proof is supposed to check against.

This ORProof is not used by the sVote protocol, and we have confirmed with Scytl that the ORProof construction is not used in any active voting system, and has only been used in internal prototypes. Even with this caveat, the existence of another broken zero knowledge protocol construct, in a code base submitted for review for national elections, raises further doubts about the integrity of this code.

4.3 Non-collision-resistant hash function

The hash function inputs numbers in a sequence of characters without describing the lengths or the types.[4] This would mean, for example, that $31, 7$ would hash to the same thing as 317 and $3, 17$. It is not immediately clear how this could be used to generate a false proof, but it breaks the main cryptographic assumption behind the secure hash function—it certainly does not behave like a random oracle. At the very least, this seems to invalidate an assumption of the formal proofs.

There is again an apparently-correct implementation, in `RandomOracleHash.java`. That hash is used in the mixnet, but for some reason the non-collision-resistant one is used in the proof library based on the Maurer framework.

5 Conclusion

We have shown in this report a second issue with the complete election verifiability process of the sVote protocol, which again opens the way for undetected electoral fraud in the Scytl-SwissPost system. A cheating mixer can forge decryption proofs and claim that ciphertexts decrypt to something other than the truth, in a way that passes verification in a formal sense.

It would probably be observed that something unusual occurred (the presence of invalid votes), but formally, verification would still pass.

We do not know what assumptions would be made or what investigations might be performed if this happens in practice. There may be some way of identifying the cheater, but the process to do this looks quite complicated, and we are not sure this could be done without risking the privacy of voters.

We emphasise that SwissPost have not confirmed this new finding. Also NSWEC claim that their system is not affected.

We are a small team of researchers investigating this code base for the first time. In a few weeks, and while spending a small fraction of our time on this investigation, we have

[4] We thank Andrew Conway for this observation.

found critical breaks of both the main components of the proof that there is no server-side fraud – the complete verifiability property. We only inspected a small fraction of this voting system, and we therefore have no reason to believe that it does not contain other critical issues.

The aim of verifiable election software is verifiable election outcomes, not proofs that pass. Independently of the impact of the attacks that we showed, our analysis of the ZK proofs shows that they do not offer the security guarantees that are assumed in the "complete verifiability security proof" report [Scy18b]. Therefore, successfully passing the current sVote audit process [Scy18a] cannot be used to draw any conclusion regarding the correctness of the election outcome.

We believe that this study confirms the importance for any democracy of enforcing openness such as that mandated by the Swiss Federal Ordinance 161.116. The source code for the system must be made public, and it must be "easily obtainable, free of charge, on the internet. [...] Anyone is entitled to examine, modify, compile and execute the source code for ideational purposes, and to write and publish studies thereon."

Thanks to this process, at least one issue that we publicly described was found to be present also in code related to a running election in New South Wales (NSW), Australia, code that was not available for open review and discussion. Apparently, the software used in NSW has a large fraction of its code in common with the Swiss Post/Scytl system. This might have meant that opportunities for undetectable electoral fraud would otherwise have gone unnoticed through a running election.

References

[BPW12] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios. In X. Wang and K. Sako, editors, *ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 626–643. Springer, 12 2012.

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Advances in Cryptology - CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 1994.

[CP92] David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *Advances in Cryptology - CRYPTO '92*, pages 89–105. Springer, 1992.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 186–194. Springer, 1986.

[Hae19] Rolf Haenni. Swiss Post Public Intrusion Test: Undetectable attack against vote integrity and secrecy. `https://e-voting.bfh.ch/app/download/7833162361/PIT2.pdf?t=1552395691`, March 2019.

[LPT19] Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. Ceci n'est pas une preuve: The use of trapdoor commitments in bayer-groth proofs and the implications for the verifiabilty of the scytl-swisspost internet voting system, 2019. https://people.eng.unimelb.edu.au/vjteague/UniversalVerifiabilitySwissPost.pdf.

[Mau09] Ueli M. Maurer. Unifying zero-knowledge proofs of knowledge. In *Progress in Cryptology - AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2009.

[Scy18a] Scytl. Scytl svote – audit of the process with control components - software version 2.1 - document 3.1, 2018.

[Scy18b] Scytl. Scytl svote – complete verifiability security proof report - software version 2.1 - document 1.0. https://www.post.ch/-/media/post/evoting/dokumente/complete-verifiability-security-proof-report.pdf, 2018.

[Scy18c] Scytl. Scytl sVote protocol specifications – software version 2.1 – document version 5.1, 2018.