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*

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 12, 2019

The implementation of the commitment scheme in the SwissPost-Scytl mixnet uses a trapdoor commitment scheme, which allows an authority who knows the trapdoor values to generate a shuffle proof transcript that passes verification but actually alters votes. We give two examples of details of how this could be used. The first example allows the first mix to use the trapdoors to substitute votes for which it knows the randomness used to generate the encrypted vote. The second example does not even require knowledge of the random factors used to generate the votes, and could be used by the last mix in the sequence.

*Since making this work public, we have learned that the same issue was identified independently by Thomas Haines of NTNU, and also by Rolf Haenni of the Bern University of Applied Sciences, https://e-voting.bfh.ch/publications/2019/

1. Introduction

1.1. Universal and Complete Verifiability

Verifiability is a critical part of the trustworthiness of e-voting systems. Universal verifiability means that a proof of proper election conduct should be verifiable by any member of the public. The authorities who conduct the election produce a mathematical proof transcript as evidence that they have conducted the election properly, then any member of the public can download and inspect the verification software (or write their own) to check that the election outcome is correct.

The Swiss sVote voting system claims to offer a form of verifiability, called "complete verifiability", which aims at offering the same guarantees as universal verifiability under the extra assumption that at least one of the components on the server-side, i.e., the people running the voting system, behaves honestly [Scy18]. (Universal verifiability offers guarantees even if all server-side components are malicious.)

In order to achieve complete verifiability, the sVote system produces audit data. One component of those audit data, which is used to demonstrate that the votes that are received are actually counted, is a sequence of proofs of shuffle—each mix server is supposed to prove that the set of input votes it received correspond exactly to the differently-encrypted votes it output.

These proofs can be complicated because they need to protect voter privacy. However, their trust assumptions are simple: it should not be possible for any collusion of authorities, whether those who hold the decryption keys, those who write the software, or those who mix the votes, to provide a proof transcript that passes verification but alters votes.

1.2. Summary of our Contribution

We show that the SwissPost-Scytl mixnet specification and code recently made available for analysis does not meet the assumptions of a sound shuffle proof and hence does not provide universal or complete verifiability.

The problem derives from the use of a trapdoor commitment scheme in the shuffle proof—if a malicious authority knows the trapdoors for the cryptographic commitments, it can provide an apparently-valid proof, which passes verification, while actually having manipulated votes. There is no modification of the audit process that would make it possible to detect if a manipulation happened. Instead, the key generation process for the commitment scheme should be modified in such a way that it offers evidence that no trapdoor has been produced, and the audit process should include the verification of this new evidence.

We give two examples of how knowledge of the commitment trapdoors could be used to provide a perfectly-verifying transcript while actually manipulating votes.

The first example allows the first mix to use the trapdoors to substitute votes for which it knows the randomness used to generate the encrypted vote. While this requires some violation of privacy, it is consistent with the requirements of the system, which state that an attacker shall not be able to change a vote even if voting clients are compromised [Scy18], and such a compromise could violate privacy. (We believe that the assumption that voting clients may be compromised is sound too: the voting system cannot do anything to guarantee that the computer of the voter does not contain any malware.)

The second example allows the last mix to use the same trapdoors to modify votes and does not require any violation of privacy, but has some constraints on the candidates for which votes could be added or removed. If, for some reason, these constraints are not satisfied, then the same strategy can still be used to render some chosen votes invalid.

We have attached example cheating transcripts to this report and encourage the public to verify them.

2. The soundness of the shuffle proof

The Scytl-Swisspost mixnet uses a provable shuffle due to Bayer and Groth [BG12]. We describe here an important implementation detail that allows the forging of apparently-verifying Bayer-Groth proofs. It is *not* a fault in the B-G proof mechanism, but rather in this specific implementation of it.

The issue concerns the soundness of the commitments. A core security requirement of commitment schemes is that they be *binding*, meaning that once someone has committed to a particular value, they can open the commitment only to that value.

The Bayer-Groth proof uses a generalisation of Pedersen commitments with multiple generators $H, G_1, G_2, \dots G_n$. They describe the scheme as “computationally binding under the discrete logarithm assumption,” (p.5). This phrasing is slightly confusing to the naive reader—it would be clearer to say that the scheme is a *trapdoor commitment scheme.* Trapdoor commitment schemes have various uses in cryptography (see [Fis01] for an excellent survey), because they are binding only on the assumption that certain secrets (the “trapdoors”) are not know to the committer.

The crucial point for the shuffle proof is then to guarantee that no one can learn the discrete logarithm of any generator H or G_i to base G_j (or of any non-trivial product of other generators). If someone knows the discrete log of G_i wrt G_j, they can create a commitment that they can open in multiple ways.

The system should prove, and the verifiers should check, that these generators are selected properly, that is, without the possibility for anyone to learn a trapdoor except by computing discrete logs.

In the Scytl-Swisspost code, the commitment parameters are just randomly generated without a proof of how they arose. Indeed, each mixer generates its own commitment parameters as follows:

```
public CommitmentParams(final ZpSubgroup group, final int n) {
    this.group = group;
    this.h = GroupTools.getRandomElement(group);
    this.commitmentlength = n;
    this.g = GroupTools.getVectorRandomElement(group, this.commitmentlength);
}
```

The implementation of `getVectorRandomElement` gathers random group elements without proving where they came from. Even more worryingly, `getVectorRandomElement` calls `getRandomElement`, which proceeds as follows:

```
Exponent randomExponent = ExponentTools.getRandomExponent(group.getQ());
return group.getGenerator().exponentiate(randomExponent);
```

This `randomExponent`, which is used to generate the random group element, is precisely the trapdoor that is needed to break the binding property of the commitment scheme. As a result, the binding property completely relies on the expectation that this randomExponent variable is properly erased from the memory.

These commitment parameters are eventually used in `ShuffleProofGenerator.java` to build the shuffle proof.

In summary: the implementation does not provide a proof, and the verifier cannot check, that the important assumption of discrete log hardness made by Bayer and Groth is valid here. It is possible for a malicious authority to generate the perfectly random $G_1, G_2, \ldots$ in a way that, at the same time, gives it a trapdoor that falsifies an assumption that is central to the security of the Bayer-Groth mixnet construction.

We will show how this can be used to produce a proof of a shuffle that passes verification but actually manipulates votes.

2.1. Details about the commitment scheme

The commitment scheme works over a group $\mathbb{G}$ of prime order q. The authority is supposed to choose $n+1$ commitment parameters $ck = H, G_1, G_2, \ldots, G_n$ at random from $\mathbb{G}$. To commit to n values $a_1, a_2, \ldots, a_n$, it chooses a random exponent r and computes

$$\mathsf{com}_{ck}(\vec{a}; r) = H^r \Pi_{i=1}^n G_i^{a_i}.$$

Commitment opening consists simply of reporting $\vec{a}$ and r.

Bayer and Groth say clearly that the commitment parameters should be generated at random and that the soundness of the commitment scheme depends on the hardness of computing discrete logs in the group. It's quite obvious that this assumption is necessary. For example, suppose that a cheating authority generates commitment parameters $ck = H, H^{e_1}, H^{e_2} \ldots, H^{e_n}$ for some H. That is, $G_i = H^{e_i}$ for $i = 1..n$. Then it can open commitments arbitrarily. A commitment $\mathsf{com}_{ck}(\vec{a}; r)$ can be opened as $\mathsf{com}_{ck}(\vec{b}; r')$ by setting

$$r' = r + \sum_{i=1}^{n} e_i(a_i - b_i) \tag{1}$$

because

$$\begin{aligned}
\mathsf{com}_{ck}(\vec{a}; r) &= H^r \Pi_{i=1}^n G_i^{a_i} \\
&= H^r \Pi_{i=1}^n H^{a_i e_i} \\
&= H^{r+\sum_{i=1}^n (a_i - b_i) e_i} \Pi_{i=1}^n H^{b_i e_i} \\
&= H^{r'} \Pi_{i=1}^n G_i^{b_i} \\
&= \mathsf{com}_{ck}(\vec{b}; r').
\end{aligned}$$

2.2. Details about the shuffle proof

Now consider how an ability to open commitments arbitrarily could be used to produce a shuffle proof that verifies but is false.

2.2.1. Faking a proof of ciphertexts with known randomness

Our demonstration shows how an attacker who knows the trapdoor can manipulate any votes for which it learns the randomness used to generate the vote ciphertext. This would allow the first mixer, in collusion with voting clients, to manipulate votes undetectably. A working demonstration transcript is submitted together with this report. Here we explain how it was generated.

The group $\mathbb{G}$ is defined as the subgroup of quadratic residues modulo a large prime, and each message is a (small quadratic residue) prime, (or the product of such primes, mod p, but let's leave out that case for now). We write the primes used to encode the messages as $q_1, q_2, \ldots$. The prover commits to applying permutation (shuffle) π.

Suppose we have three input ciphertexts $C_1 = \mathcal{E}_{pk}(M_1, \rho_1'), C_2 = \mathcal{E}_{pk}(M_2, \rho_2')$, $C_3 = \mathcal{E}_{pk}(M_3, \rho_3')$ with known messages M_1, M_2, M_3 and randomness $\rho_1', \rho_2', \rho_3'$, and one input ciphertext C_4 whose contents and randomness are unknown.

The idea of the cheat is, for each prime q_k, to accumulate all the votes for q_k, for which the attacker knows the contents and randomness, into one $\pi(i)$. The attacker can then substitute all the other votes (for which it know the randomness) with arbitrary votes of its own choice.

This attack succeeds with arbitrarily many known and unknown votes, as long as the number of known votes is larger than the number of candidates that received at least one vote—the attacker can substitute the votes for which it knows the randomness, and must honestly shuffle those for which it does not know the randomness.

We illustrate with a small example. Suppose $M_1 = M_2 = q_1$ and $M_3 = q_2$. M_4 is unknown. The cheating prover will apply the identity permutation (just for clarity here, this has no impact on the attack) and set

$$\begin{array}{rclcl}
C_1' &=& \mathcal{E}_{pk}(1; \rho_1) C_1 &=& \mathcal{E}_{pk}(M_1, \rho_1 + \rho_1') \\
\hline
C_2' &=& \mathcal{E}_{pk}(\mathbf{1}; \boldsymbol{\rho_2}) \mathbf{C_3} &=& \mathcal{E}_{pk}(\mathbf{M_3}, \boldsymbol{\rho_2} + \boldsymbol{\rho_3'}) \\
\hline
C_3' &=& \mathcal{E}_{pk}(1; \rho_3) C_3 &=& \mathcal{E}_{pk}(M_3, \rho_3 + \rho_3') \\
\text{and } C_4' &=& \mathcal{E}_{pk}(1; \rho_4) C_4 &=& \mathcal{E}_{pk}(M_4, \rho_4 + \rho_4')
\end{array}$$

If C_4 is an encryption of q_4 (neither q_1 nor q_2), the substitution of M_3 for M_2 in the second vote changes the winner: it used to be q_1; now it's q_2. The cheating prover knows M_1, M_2, M_3 but not M_4. It also knows ρ'_i for $i = 1, 2, 3$ but not ρ'_4.

The high-level protocol is described in Bayer & Groth p.8.

Input: $m = 2, n = 2, N = 4, \vec{C} = \{C_1, C_2, C_3, C_4\}, \vec{C'}$ as above; permutation π. We will compute ρ carefully later.

Suppose the mix has generated the trapdoored commitment key as in Section 2.1. The cheating shuffler's initial message $\vec{c}_A$ is a (truthful) commitment to π. That is,

$$\vec{c}_A = \mathsf{com}_{ck}(\vec{A}_1; r_1), \mathsf{com}_{ck}(\vec{A}_2, r_2) \text{ where } \vec{A}_1 = (\pi(1), \pi(2)) \text{ and } \vec{A}_2 = (\pi(3), \pi(4))$$

It then commits honestly to $\vec{B}$ as

$$\vec{c}_B = \mathsf{com}_{ck}(\vec{B}_1; s_1), \mathsf{com}_{ck}(\vec{B}_2, s_2) \text{ where } \vec{B}_1 = (x^{\pi(1)}, x^{\pi(2)}) \text{ and } \vec{B}_2 = (x^{\pi(3)}, x^{\pi(4)})$$

Now consider how the cheating shuffler responds to the second challenge y, z and generates a convincing answer for both parts. In the first part of the challenge, when it generates answer 1 in response to y, z, it treats $\vec{c}_B$ as a commitment to x^π and answers the product argument (Bayer & Groth Section 5) honestly.

Cheating on the multi-exponentiation argument In the second part of the challenge, it generates a cheating permutation π_{cheat}, which isn't actually a permutation, as follows:

$$\begin{aligned} \pi_{cheat}(1) &= x + x^2 \\ \pi_{cheat}(2) &= 0 \\ \pi_{cheat}(3) &= x^3 \\ \pi_{cheat}(4) &= x^4. \end{aligned}$$

The attacker then runs the multi-exponentiation argument from Section 4 of BG exactly as given, except for the following changes.

- It sets
$$\rho = -\rho_1 x - (\rho_1 + \rho'_1)x^2 + x^2\rho'_2 - \rho_3 x^3 - \rho_4 x^4. \tag{2}$$
(See Appendix A.1 for why this works.)

- It treats $\vec{c}_B = \mathsf{com}_{ck}(\vec{B}_1; s_1), \mathsf{com}_{ck}(\vec{B}_2, s_2)$ as a commitment to $\pi_{cheat} = ((x + x^2, 0)(x^3, x^4))$.

- It computes commitment openings $\vec{s}$ for π_{cheat} using Equation 1 and the random values s_1 and s_2.

This produces a proof that passes verification, though the election outcome has been changed. An example transcript, which passess verification, is attached with this report.

2.2.2. Faking a proof of ciphertexts with unknown randomness

As a second example, we exploit the trapdoor in the commitment scheme to break the soundness of the proof of shuffle, even in a situation in which we do not know the randomness or the content of any vote.

In this case, the malicious party could be the last mixer. This mixer indeed has the advantage of being able to perform the final decryption step, which means that it may know the content of the votes that it mixes before actually mixing them. (It could also be the first mixnet if it has some other way of learning the contents of the votes.)

We make the following assumption (many variants are possible): We know how to express the prime quadratic residue used to encode a candidate as a power of the generator G used for ElGamal encryption.[1] For instance, the secure 2047-bit subgroup of quadratic residues provided in the sources is generated by 2, which is used as a the ElGamal encryption generator and may very well also be chosen to encode a candidate. In the following example, we use that case for simplicity.

We note that the system specification does not require that at least one of the primes used to represent candidate should coincide with the generator used for ElGamal encryption. This is however permitted and plausible: the candidate encoding mechanism used in the system is more efficient when the prime quadratic residues that are used are as small as possible.

For concreteness, suppose that voters can support as many candidates as they want and that the last mixer receives input ciphertexts $C_1 = \mathcal{E}_{pk}(M_1, \rho'_1), C_2 = \mathcal{E}_{pk}(M_2, \rho'_2)$, $C_3 = \mathcal{E}_{pk}(M_3, \rho'_3)$, $C_4 = \mathcal{E}_{pk}(M_4, \rho'_4)$ such that the candidate “2” does not win the election.

The last mixer can now perform the final decryption step in order to identify which of these ciphertexts do not contain a vote for “2”. It does not learn the randomness $\rho'_1, \rho'_2, \rho'_3, \rho'_4$. Again, for simplicity, let us assume that the mixer finds out that nobody voted for “2”.

In order to manipulate the outcome, the mixer defines the output ciphertexts as $C'_i = \mathcal{E}_{pk}(2, \rho_i)C_i$. By the homomorphic property of ElGamal, We have added a vote for “2” to each ciphertext. (For ease of exposition we use the identity permutation on the list of ciphertexts, but any permutation is possible.)

We play the Bayer-Groth shuffle perfectly honestly, except for the multi-exponentiation argument. Indeed, that argument raises a difficulty because the statement equation $\vec{C}^{\vec{x}} = \mathcal{E}_{pk}(1;\rho)\vec{C'}^{\vec{b}}$ does not hold. Instead, the equation $\vec{C}^{\vec{x}} = \mathcal{E}_{pk}(2^{-x-x^2-x^3-x^4};\rho)\vec{C'}^{\vec{b}}$ holds, for $\rho = -\rho_1 x - \rho_2 x^2 - \rho_3 x^3 - \rho_4 x^4$, which is known to the mixer. In order to make the proof pass the verification despite this, we will use the trapdoor of the commitments in the multi-exponentiation argument.

We follow the notation in Bayer & Groth, Section 4. In the initial message, we cheat on the commitment $c_{B_m} = \mathsf{com}_{ck}(b_m, s_m)$: instead of setting $b_m = s_m = 0$, we set

[1]This assumption guarantees that we can actually modify votes in a chosen way. If it is not satisfied, the strategy discussed here would still make it possible for the last mixer to pick ballots that contain votes that it does not like, and completely rerandomize them in order to render them invalid. This targeted modification could also change an election outcome.

$b_m = -x - x^2 - x^3 - x^4$ and use the trapdoors to compute s_m such that $\mathsf{com}_{ck}(b_m; s_m) = \mathsf{com}_{ck}(0; 0)$. This choice makes sure that $c_{B_m} = \mathsf{com}_{ck}(0; 0)$ and $E_m = \vec{C}^{\vec{x}}$, as required in the first two steps of the proof verification steps.

All the other verification steps pass, as we did not break the truthfullness of any of the underlying proofs.

3. Discussion

Ease of exploiting the problem The first attack requires knowing the randomness used to generate the vote ciphertexts that will be manipulated. There are several ways this could be achieved. For example, an attacker could compromise the clients used for voting. Weak randomness generation (such as that which affected the Norwegian Internet voting system) would allow the attack to be performed without explicit collusion.

The second attack does not require any extra information at all, though it does rely on the election parameters having been set up in a particular way, and on multiple selections being accepted as valid votes.

Correcting the problem The issue needs to be corrected by ensuring that the commitment parameters are generated in a way that prevents any entity from knowing the discrete logs.

There are various techniques to do this—they are sometimes called "nothing up my sleeve numbers." A standard solution is to derive these group elements directly from applying a PRG based on a cryptographic hash function, the outputs of which are then mapped to group elements.[2]

Every verifier then needs to check the generation of the commitment parameters as well as the rest of the proof transcript.

We understand that SwissPost and Scytl have corrected the issue by generating the commitment parameters according to NIST FIPS 186-4, Appendix 2.3. Although we have not seen the implementation, we consider this approach to be appropriate for generating the commitment parameters. However, generating the commitment parameters properly might not completely resolve the problem. The FIPS standard should also be used to generate the group parameters p, q. This issue and the correction require further public scrutiny.

How can there be a trapdoor when the system has been formally proven secure? Any formal proof of correctness for any system makes some assumptions that become axioms in the formal proof. Scytl's formal proof of security [Scy18] simply models the mixnet as sound, based on an informal interpretation of Bayer and Groth's security proof. It does not model the proper generation of commitment parameters. We do not

[2]This technique is used, for example, in the Verificatum mixnet [Wik] (that is, for a different shuffling algorithm).

see any reason to believe there is an error in Scytl's proof, but when the axioms are mistaken the conclusions are not valid.

This does not mean that formal proofs are not valuable—at an absolute minimum, they clarify assumptions and explain the reasons for trust—but it does mean that they are not a substitute for broad and open public scrutiny. It is quite possible that there are errors in the implementations of other cryptographic primitives, that their details may not be modelled in the formal proofs, and that they may affect either privacy or verifiability.

Source of the problem Nothing in our analysis suggests that this problem was introduced deliberately. It is entirely consistent with a naive implementation of a complex cryptographic protocol by well-intentioned people who lacked a full understanding of its security assumptions and other important details. Of course, if someone did want to introduce an opportunity for manipulation, the best method would be one that could be explained away as an accident if it was found. We simply do not see any evidence either way.

4. Conclusion

This mixnet has a trapdoor—a malicious administrator or software provider for the mix could manipulate votes but produce a proof transcript that passes verification. Thus complete verifiability fails.

Even if this particular issue is corrected, we do not know whether there might be other ways of manipulating votes while still producing an apparently-verifiable election outcome, or other manipulations that would lead to vote privacy violations.

The issues reported here are the result of the analysis of an isolated, but critical, part of the code. This voting system is highly complex, there are many other critical parts, and we did not look at them. As a result, we have no reason to believe, based on this work, than there are no other crticial issues in this implementation.

5. Acknowledgements

Many thanks to Andrew Conway for tremendous help with the code, and to Aleks Essex, Matt Green and Hovav Shacham for many valuable discussions.

6. A note on code authenticity

We did not officially enrol for the Swiss Post researcher test. We downloaded this codebase from an unofficial repository and received confirmation of its authenticity from researchers with access to the official codebase. We are highly confident that this is a real trapdoor in the current implementation.

References

[BG12] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 263–280. Springer, 2012.

[Fis01] Marc Fischlin. *Trapdoor commitment schemes and their applications.* PhD thesis, Goethe-University of Frankfurt, 2001.

[Scy18] 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.

[Wik] Douglas Wikström. Verificatum mixnet. `https://www.verificatum.com/`.

A. Technical detail on how to generate a fake proof transcript with known randomness

A.1. Calculating ρ

This section shows why we get the expression for ρ that we use above.

We needed to find ρ s.t.

$$\vec{C}^{\vec{x}} = \mathcal{E}_{pk}(1;\rho)\vec{C'}^{\vec{b}}$$

where $\vec{C}$ are the input ciphertexts and $\vec{C'}$ are the output ciphertexts. (Bayer-Groth p.8)

$$\begin{array}{rcl}
LHS & = & \vec{C}^{\vec{x}} \\
 & = & \Pi_{j=1}^{4} C_j^{x^j} \\
 & = & \mathcal{E}_{pk}(q_1^{x+x^2} q_2^{x^3} q_4^{x^4}; \sum_{i=1}^{4} x^i \rho'_i) \\
 & & \\
RHS & = & \mathcal{E}_{pk}(1;\rho)\vec{C'}^{\vec{b}} \\
 & = & \mathcal{E}_{pk}(q_1^{x+x^2} q_2^{x^3} q_4^{x^4}; \rho + (\rho_1+\rho'_1)(x+x^2) + (\rho_3+\rho'_3)x^3 + (\rho_4+\rho'_4)x^4). \\
\text{So } \rho & = & -\rho_1 x - (\rho_1+\rho'_1)x^2 + x^2\rho'_2 - \rho_3 x^3 - \rho_4 x^4.
\end{array}$$

Note ρ'_4 is unknown but $\rho'_4 x^4$ cancels out.