How not to prove your election outcome

Thomas Haines*, Sarah Jamie Lewis†, Olivier Pereira‡, and Vanessa Teague§
*Norwegian University of Science and Technology
†Open Privacy Research Society, Canada
‡UCLouvain – ICTEAM – B-1348 Louvain-la-Neuve, Belgium
§The University of Melbourne – School of Computing and Information Systems, Melbourne, Australia

***Abstract*—The Scytl/SwissPost e-voting solution was intended to provide complete verifiability for Swiss government elections. We show failures in both individual verifiability and universal verifiability (as defined in Swiss Federal Ordinance 161.116), based on mistaken implementations of cryptographic components. These failures allow for the construction of "proofs" of an accurate election outcome that pass verification though the votes have been manipulated. Using sophisticated cryptographic protocols without a proper consideration of what properties they offer, and under which conditions, can introduce opportunities for undetectable fraud even though the system appears to allow verification of the outcome.**

Our findings are immediately relevant to systems in use in Switzerland and Australia, and probably also elsewhere.

I. Introduction

Verifiability is a must-have for elections: if the outcome doesn't come with evidence of its correctness that can be verified by third parties, then the results can be manipulated. But designing verifiable systems is challenging: if a cryptographic protocol is designed or implemented in secret, if it comes with no convincing proof of the soundness of its verification process, and if no opportunity for independent scrutiny is given, then it is unlikely that it offers the security properties it advertises. It might seem to offer a chance to check the results, but those checks might not really prove that the election outcome is right.

We show multiple independent ways that cryptographic errors in Scytl's e-voting protocol sVote, proposed by SwissPost for Swiss government elections, can be used to fake a proof of an accurate election outcome that passes verification even though the votes have been manipulated. Our analysis focuses on sVote version 2.1, which purported to provide complete verifiability and was applying for certification for use by up to 100% of the electorate at the time we began our examination.

Our results have immediate implications for elections in Switzerland and New South Wales. One of the identified weaknesses was also present in version 1.0 of the sVote protocol, which has been deployed in binding elections by several Swiss cantons, and another was present in a system used in a state election in New South Wales. We do not know whether other countries using software from the same provider are affected by any of the weaknesses that we found: many other countries use the same vendor but do not publish their code.

Because of the findings discussed in this paper, no version of sVote was used during the May 2019 Swiss referendum or the Federal elections of October 2019. Both the Swiss and New South Wales systems have been updated for most (though not all) of the issues we describe here, though we have not thoroughly examined the remediation attempts.

This is unlikely to be the last proprietary verifiable multiparty computation protocol that doesn't meet its advertised security goals. The incorrect use of sophisticated protocols such as zero knowledge proofs (ZKPs) and multiparty computation (MPC) demonstrates pitfalls that potentially affect systems for applications other than voting. Many of the components have been proven secure elsewhere, but under assumptions that are not realised in this system. We provide a short summary of the generalisable failures we observed:

1) **Statically secure primitives are used in an adaptive setting.** The sVote system uses Maurer's unified proofs framework [1] for many of its ZKPs. These are proven to offer security in an interactive and static model (*i.e.* when the statement is given to the prover), but are used in sVote in a setting where the prover can choose the statement afterwards, thus making it vulnerable to the pitfall described in [2].
2) **The facts proven in the ZKPs are not sufficient.** A ZKP proves a particular statement (or more precisely, membership of a specific language). But in sVote, the conjunction of the facts proven by the voting client is not sufficient to imply the desired properties about the votes.
3) **The multiparty protocol isn't secure against collusion.** sVote uses an original multiparty protocol to compute the codes to be returned to voters. The protocol has five authorities and is intended to tolerate some dishonest participants. We show that one of them can misbehave alone (with a cheating client) and break verifiability.
4) **Hardness assumptions are not guaranteed.** Most of the protocols used in sVote rely on the hardness of some computational problems. For instance, in various places, it is expected that discrete logs of various group elements in various bases are unknown. However, sVote offers no evidence of how the group parameters and group elements are generated, which makes it impossible to verify whether a prover may know a trapdoor that would violate the specific instances of the computational problems used in the system. This can be used to violate both individual and universal verifiability in sVote.

Every one of these errors allows a successful vote manipulation that passes verification, though in some cases it is informally clear that something has gone wrong.

We describe the code and specifications as released for review in March 2019. We illustrate the ZKP unsoundness from Sections III and VI with cheating examples that pass verification in that code. These are available at https://git.openprivacy.ca/swiss-post-scytl-disclosure/cheating-proof-transcripts along with the rest of the codebase, so others can check that our proofs pass verification. Some of these issues have now been at least partially remediated [3], [4].

We also provide a list of other cryptographic errors in OR proofs, hashing, *etc.*, though they do not seem to lead to attacks.

These problems have been found thanks to Swiss Federal Chancellery Ordinance 161.116 on Electronic Voting, which mandates a public review of electronic voting systems used at a certain scale in Switzerland. It also requires that "[the source code] must be easily obtainable, free of charge, on the internet" and that "anyone is entitled to examine, modify, compile and execute the source code for ideational purposes, and to write and publish studies thereon." It is unlikely that the protocols deployed in other countries, which do not mandate such public review, would offer any better security guarantees. And in the absence of a legal framework supporting the open analysis and discussion of the properties of these protocols, the only people who have an incentive to find such issues are those who would seek to exploit them silently.

A. History and security goals of sVote in Switzerland

sVote 1.0 has been used since 2016 in four Swiss cantons[1] and was certified for use by up to 50% of voters.

In order to obtain this level of certification, *individual verifiability* is required. Before the election, each voter receives a paper code sheet by postal mail, which contains one secret choice code for each possible voting option. Voters can then enter a vote on an untrusted web client and receive assurance that their vote intent has been properly recorded by receiving choice codes back electronically from the voting system and matching these codes with those written on their code sheet. A malicious voting client that alters a vote should be detected when the voter doesn't receive the right codes.

sVote 2.1 was proposed in 2018 for certification for up to 100% of the voters of the cantons willing to use it. To obtain this certification, ordinance 161.116 requires that individual verifiability holds as long as at least one of the server-side components is honest. It also requires that the system offers a form of universal verifiability, which guarantees (approximately) that the result of an election is correct as long as at least one server-side component is honest. (The traditional universal verifiability notion requires that this property holds even if all the server-side components of the system are dishonest [5].) When combined with individual verifiability, the property is called "complete verifiability." In this setting no public bulletin board is needed, which contrasts with traditional universally verifiable voting schemes.

Any voting system that decrypts individual votes (even paper ones) must mix the votes before decryption, which introduces a potential point for cheating by substituting rather than truthfully shuffling. Like many other systems, sVote 2.1 attempts to achieve universal verifiability (while protecting vote privacy) with ZKPs of both honest mixing and correct decryption. It should not be feasible to produce a passing proof transcript for either mixing or decryption unless the set of encrypted votes entered in to the system matches the output plaintext votes.

The specification and code examined in this paper were made available by SwissPost under a non-disclosure agreement (NDA). This NDA required that "No Vulnerability shall be published within a period of forty five (45) days since the last communication exchanged with the Owners with regards to such potential Vulnerability". However, the specification and code also circulated fairly freely online, and this is how we accessed them. Based on our findings, we contacted the Federal Chancellery and Swiss Post and agreed on a synchronized communication agenda supporting public discussion.

B. Structure of this paper

We give a technical overview of the system, then demonstrate various attacks against complete verifiability.

The first three attacks concern individual verifiability. In Section III a malicious client uses the unsoundness of the ZKPs to retrieve the expected choice code values despite submitting a nonsense vote. The second attack, described in Section IV, allows a cheating client to submit a mix of valid and invalid vote choices while retrieving the expected vote codes for all but one of them. It is dependent on either a certain interpretation of the (ambiguous) specification, or collusion from one server-side component. This is possible even if the ZKPs are sound, because they are not sufficient to prove that the vote is well formed. In Section V we show that even if the vote is well-formed, a single cheating server-side component can substitute the choice code and still pass audit if the voting parameters are maliciously generated.

We then turn to universal verifiability and show three more attacks. In Section VI, we examine the proof of proper mixing and show that the non-verifiable generation of group parameters allows a cheating mix server to alter votes but pass verification. We give two examples of how this could be used. The first example allows the first mix to substitute votes for which it knows the randomness used to generate the encrypted vote (which could result from leakage from the client). The second example does not require knowledge of the random factors used to generate the votes, and could be used by the last mix in the sequence to alter votes with no client collusion.

Then, in Section VII, we observe that the use of non-adaptive ZKPs for ballot decryption also makes it possible to substitute nonsense votes for validly-submitted ones, and still pass verification.

Finally, motivated by the impact of the non-verifiable group parameter generation specified in sVote, we review various other mix-net implementations in Section VIII and observe that this issue is fairly widespread.

Some other cryptographic issues are described in Appendix E.

[1] See https://www.evoting.ch/en.

II. Technical Overview

We now describe the main cryptographic aspects of sVote 2.1. The following description omits or presents in a simplified way various components that are irrelevant for our discussions (*e.g.,* the handling of write-ins).

A. System components and Cryptographic setting

The main components of the system are:

- A print office, which generates and prints the code sheets, and is trusted.
- A voting client, which is trusted for privacy only.
- A voting server, which coordinates the server-side components of the election and is not trusted.
- Return codes control components (CCRs), among which one is assumed to be honest.
- Mixing control components (CCMs), among which one is assumed to be honest.
- Auditors, among which one is assumed to be honest.

These components have signing keys, which they use to authenticate (and log) their behavior, but we ignore them here.

The code sheet printing service needs to be trusted: it sees all return codes corresponding to all voter choices. If it colluded with a malicious code return service or with a malicious voting client, it would be possible to return any choice codes that a voter expects to see, whatever vote was recorded.

B. Cryptographic setup

The core of the cryptographic protocol happdens in a group $\mathbb{G}$ of prime order q made of the quadratic residues modulo a 2048-bits prime $p = 2q+1$. The group parameters are chosen so that $g = 2$ generates $\mathbb{G}$.

Before the election, each possible answer offered on a ballot is assigned a small prime v_i in $\mathbb{G}$. Each question on a ballot has at least three possible answers: yes, no, and blank, though it may instead have several different candidates to choose from.

sVote 2.1 uses ElGamal encryption [6] over $\mathbb{G}$. Each message is a small prime in $\mathbb{G}$, or the product of such primes. All choices for a ballot are multiplied together, encrypted as a single ciphertext, and sent to the server.

So, whatever vote is cast, the number of small primes needed to represent the vote is fixed. The system expects that the product of all the v_i primes selected for any ballot will be smaller than p.

Sometimes El Gamal is used in its multi-element version. A public key is a tuple $(K^{(1)}, K^{(2)} \ldots, K^{(\psi)})$ (where ψ is the maximum number of questions on the ballot). The corresponding private key is the tuple of discrete logs of the public key, mod p. A message m, which is a tuple $(m_1, m_2, \ldots, m_\psi)$ of quadratic residues mod p, is encrypted by generating a random $r \in [1, q]$ and setting the ciphertext to

$$E(m,r) = (g^r, m_1 K^{(1)r}, m_2 K^{(2)r} \ldots, m_\psi K^{(\psi)r}).$$

(This is not our chosen notation: to be clear, $K^{(i)r}$ represents raising the i-th element of the public key to r.)

The client sends both a vote ciphertext and a set of *partial Choice Codes* that are used to compute the choice codes, along with a non-interactive ZKP that they are consistent—this is detailed below. The server-side has two roles: it first computes the choice codes for each voter and then, when all the votes have been received, it mixes and decrypts the votes.

This means there are three uses of non-interactive ZKPs:

1) the client uses three ZKPs to prove that the partial choice codes match the vote;
2) the server-side uses ZKPs to prove that the codes it returns match the partial choice codes submitted by the client;
3) the server-side uses ZKPs to prove that the complete list of votes is properly mixed and decrypted.

We follow the notations of the sVote protocol specification [7, Section 5.4], with some simplifications. The ballot preparation and individual verifiability processes rely on the following keys.

1) h_{CCM} is the election public key, a standard single-element El Gamal key.
2) x_{CCM}, the corresponding private key, is secret-shared among the *CCM*'s.
3) $pk = pk^{(1)}, \ldots, pk^{(\psi)}$ is the choice return code public key, which is a multi-element El Gamal public key. The parameter ψ is the number of options to be expressed by each voter.
4) $sk_{CCR} = sk_{CCR}^{(1)}, \ldots, sk_{CCR}^{(\psi)}$, the corresponding private key, is secret-shared among the *CCR*'s.
5) K_{id} is the verification card public key, one for each verification card id (*i.e.* for each voter).
6) k_{id} is the corresponding private key, held by the voting client.
7) $\hat{k}$, which is different for each voter,[2] is secret-shared among the *CCR*'s.
8) $k = k_{id} \cdot \hat{k}$ is used to retrieve the choice code.

1) Ballot preparation and validity proof: The client prepares and proves validity of a vote as follows. Suppose the voter selected ψ options $v_1, v_2, \ldots, v_\psi$, each corresponding to an answer to a question in the election. The voting client has a codes card ID *id* and a card-specific private key k_{id}.

The vote is $\Pi_{i=1}^{\psi} v_i$. For each choice v_i, the client sends a *partial choice code* $pCC_i = v_i^{k_{id}}$ which is used later to compute the choice code that is returned to the voter. Obviously, the partial choice codes must match the vote, or the client can cheat by sending the vote that it wants with the partial choice codes that will please the voter. Since all these values are sent encrypted, the proof that they are consistent is quite involved. Below, E_1 is the encrypted vote and E_2 is the encrypted partial choice codes. The ciphertext F_1 is created only to prove that E_2 contains the matching partial choice codes for the vote in E_1. In Step 6, π_e proves that F_1 matches the vote, E_1; in Step 8, π_p proves that F_1 matches the choice codes, E_2. Hence E_1 and E_2 are meant to match each other. The voting client

[2] This is what the spec says, but the notation for these values is very strange: the sum of four shared values indexed by *id* is not itself indexed by *id*—we assume that all three of these values $(\hat{k}, k_{id}, k)$ are meant to vary among voters/cards.

1) computes the vote ciphertext

$$E_1 = (g^r, \Pi_{i=1}^{\psi} v_i \cdot EL^r).$$

2) For each choice v_i $(i = 1, \ldots, \psi)$, it computes a partial choice code $pCC_i = v_i^{k_{id}}$.
3) It encrypts each pCC_i with a separate element of the multi-element key pk, as

$$E_2 = (g^{r'}, pCC_1 \cdot (pk^{(1)})^{r'}, \ldots, pCC_\psi \cdot (pk^{(\psi)})^{r'}).$$

4) It uses the Schnorr protocol [8] to produce a proof of knowledge π_s of r used in E_1.
5) It computes F_1 as $E_1^{k_{id}}$, that is,

$$F_1 = (g^{rk_{id}}, (\Pi_{i=1}^{\psi} v_i \cdot EL^r)^{k_{id}}).$$

6) It generates a proof of exponentiation π_e to prove that (K_{id}, F_1) is indeed equal to $(g^{k_{id}}, E_1^{k_{id}})$ for a secret k_{id}.
7) It multiplies all but the first element of E_2 together to form a standard El Gamal encryption.

$$\tilde{E}_2 = (g^{r'}, \Pi_{i=1}^{\psi} pk(i)^{r'} pCC_i).$$

8) It generates a plaintext equality proof π_p to show that F_1 and $\tilde{E}_2$ encrypt the same value (that is, $\Pi_{i=1}^{\psi} p_i^{k_{id}}$), w.r.t. EL in the first case and w.r.t. $\Pi_{i=1}^{\psi} pk(i)$ in the second case.
9) The vote, defined as $E_1, E_2, F_1, \pi_s, \pi_e, \pi_p$, is submitted to the server.

2) Ballot Processing and code return: Now, for each vote id, the server-side needs to verify the proofs, compute the choice codes and send them back to the voter, who can then check them on her codes card.

Section 5.4.3 of the sVote2.1 spec [7] describes an original verifiable multiparty protocol in which the Control Components ($CCR_1, CCR_2, CCR_3, CCR_4$) and the Vote Verification Context (VVC) decrypt the partial choice codes, retrieve the appropriate choice code from a code table and prove that they have computed it correctly. This is summarised below. The intended property is that an incorrect generation/retrieval can be detected by the auditors if not all the *CCR*s collude. The VVC is not meant to be trusted.

1) Each Control Component (*CCR*) verifies the vote ZKPs and, if they pass, uses its share of $sk_{CCR}^{(i)}$ and $\hat{k}$ to compute a partial decryption of E_2 exponentiated by $\hat{k}$. (Details are omitted.)
2) This allows the VVC to generate the pre-Choice return code for each i $(i = 1, \ldots, \psi)$ as

$$\begin{aligned} pC_i^{id} &= g^{-r' \cdot sk_{CCR}^{(i)} \cdot \hat{k}} \cdot (pk_{CCR}^{(i)})^{r' \cdot \hat{k}} \cdot v_i^{k_{id} \cdot \hat{k}} \\ &= v_i^k \end{aligned}$$

3) The VVC then computes each *long Choice Code*

$$lCC_i^{id} = SHA256(v_i^k || id || \text{public data})$$

4) For $i = 1, \ldots, \psi$, the VVC uses $SHA256(lCC_i)$ as an index into a precomputed table of encrypted choice codes. It derives the decryption key for the i-th choice code from lCC_i and a secret known only to the VVC. The decrypted choice codes are returned to the voter.
5) Finally, the encrypted vote E_1 is passed to the mix servers.

Note that the VVC is not meant to be able to decrypt any choice codes for which it has not received a corresponding vote: because it does not know k, there is no obvious way for it to generate v_i^k and hence lCC_i.

There is a second code-based phase in which the voter can confirm that she received the choice code she expected, and thus finalise the casting of her vote. The details are similar to the choice-code process.

A later process for mixing and decrypting the votes is described in Section VI. Each vote is factorized at the end to recover the individual primes.

III. Pitfalls of the Fiat-Shamir transform: why individual verifiability fails (1)

In this section we show that the vote validity NIZKPs are not sound, which can be used by a malicious client to submit a nonsense vote, prove it is valid, and retrieve the right choice codes. The voter would then consider that her vote intent was correctly captured. However, when this vote was decrypted after being mixed, it would be invalid.

The Fiat-Shamir transform [9] 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.

sVote uses Maurer's unified proofs framework [1], which is proven secure in the non-adaptive setting, in which the statement is given to the prover. However, sVote applies it in an adaptive setting, in which the prover can choose the statement about which it wants to make a proof. It thus becomes crucial to also include that statement, in full, into the inputs of the hash function—soundness collapses otherwise [2].

The requirement of adaptive security is quite common in voting systems and, as we demonstrate here, it is needed for the sVote protocol. We want to stress that this issue is not present in Maurer's framework [1]—the problem lies in the misalignment of assumptions in Maurer's security proof with the setting of sVote.

Interestingly, the decryption proof described in the Verifiability Security Proof report [10] is different from the one that appears in the sVote protocol specification [7], which is the one implemented in the system. The ZKP described in that Security Proof report appears to be correct.

A. Producing a false ballot validity proof

Suppose for simplicity that the voter wants to submit a single vote v encoded as a prime quadratic residue mod p. (The extension to multiple (prime) vote choices is immediate.)

Write F_1 as (F_{10}, F_{11}) and similarly, $E_1 = (E_{10}, E_{11})$.

1) The proof of exponentiation: sVote's exponentiation proof is used in step 6 of ballot generation (Section II) to prove that F_1 is properly computed as $E_1^{k_{id}}$. The method is a slight generalisation of a well-known proof method due to Chaum and Pedersen [11]. It proceeds as follows:

1) Pick a random a and compute $(B_1, B_2, B_3) = (g^a, E_{10}^a, E_{11}^a)$.
2) **Compute $\mathbf{c} = \mathbf{H}(\mathbf{K}, \mathbf{F_1}, \mathbf{B_1}, \mathbf{B_2}, \mathbf{B_3})$.**
3) Compute $z = a + ck_{id}$.

The proof consists of (c, z). It is verified by computing $B'_1 = g^z/K^c$, $B'_2 = E_{10}^z/F_{10}^c$, $B'_3 = E_{11}^z/F_{11}^c$, and checking that c equals $H(K, F_1, B'_1, B'_2, B'_3)$.

2) Lack of adaptive soundness of the proof of exponentiation: As discussed above, this proof lacks adaptive soundness. The computation of the challenge c (in bold) shows that g and E_1 are not hashed, so there is no guarantee that they are chosen before the proof is computed. This can be used to generate a proof of a statement that is not true.

In what follows, we assume that g was generated honestly, in a verifiable way (but this is not required by the sVote specification) and focus on E_1. A malicious adaptive prover could then proceed as follows.

Start by picking F_1 as a random encryption of pCC_1 for the vote the voter intended, that is, as $(g^r, pCC_1 \cdot EL^r)$. Then:

1) Pick random (a_1, a_2, a_3) and compute $(B_1, B_2, B_3) = (g^{a_1}, g^{a_2}, g^{a_3})$.
2) Compute $c = H(K, F_1, B_1, B_2, B_3)$.
3) Compute $z = a_1 + ck_{id}$.

The proof is (c, z). Then, set $E_1 = ((B_2 \cdot F_{10}^c)^{1/z}, (B_3 \cdot F_{11}^c)^{1/z})$, which guarantees that the verification equation passes. But with overwhelming probability, $F_1 \neq E_1^{k_{id}}$, so E_1 will not be an encryption of the voter's choice.

3) Why individual verifiability fails: We now show how this can be used to construct a complete ballot that passes verification and returns the right choice codes, though it does not convey the voter's chosen vote.

Running the previous steps provides (E_1, F_1, π_e) that pass verification. F_1 is a valid encryption of the right partial return code, but E_1 is not an encryption of the right vote.

In order to complete the ballot, we can compute E_2 in a perfectly honest way, using whatever vote the voter asked for (which will not be cast) and the true $pCC_1 = v^{k_{id}}$. We then compute π_s in a completely honest way by observing that $E_{10} = (B_2 \cdot F_{10}^c)^{1/z} = g^{(a_2+rc)/z}$, so $(a_2 + rc)/z$ is the random value used to produce E_{10}. Finally, we compute π_p in a perfectly honest way as well, since it corresponds to a true statement for which we have a witness: F_1 and E_2 do encrypt the same value, which is the correct function of the intended vote.

All the proofs are valid, so E_2 will be used to derive the return codes corresponding to the vote intent v, which will then be accepted by an honest voter, who will have her vote confirmed. However, when E_1 is decrypted (after being processed through the mixnet), it will be declared invalid.

Summary of 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.

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. This includes all the base elements in the proof of exponentiation.

Current status of the problem: An effort is being made to remediate the problem in future versions.

IV. The client-side proofs are not sufficient: why individual verifiability fails (2)

Even if the ZKPs were sound, individual verifiability would still fail because the ZKPs used to prove consistency between the vote codes and the vote are not sufficient.

In this section we show how this allows a cheating client to submit a ballot that substitutes a voter's choice but states a corresponding *pCC* that matches the voter's request. This passes verification even if the NIZKPs are sound. Some other options will have invalid *pCC*s, so it is not clear whether this would result in a practical attack—the spec is ambiguous on whether the valid codes would be retrieved in this case. Our report to the Swiss Federal Chancellery on sVote 1.0 [4] includes a very similar issue.

The main problem is that the exponentiation proof π_e in the vote generation proves that the *product* of the partial choice codes has been correctly exponentiated—it does not prove that each individual element has been properly exponentiated.

A. Generating a proof of ballot validity when the choice codes do not match the vote

Suppose that there are two questions on the ballot, each with two options, but one question is much more important than the other. A cheating client will fabricate correct vote choice codes for the important question, despite sending the incorrect vote. The voter (if she checks carefully) will see that there are invalid vote choice codes for the unimportant question, but will receive the correct codes for the one she cares about.

Suppose p_{yes} and p_{no} are primes representing 'yes' and 'no' answers respectively to the important question (Question 1). The ballot also contains a second question of less importance (Question 2), with answers represented by p_3 and p_4.[3]

The cheating client can substitute its preferred vote for Question 1 while building the correct partial choice code (pCC_1) for the *voter's* expected code. The other partial choice code—pCC_1—is invalid. This construction, though invalid, passes verificaion. The attacker transfers the code substitution to Question 2 and hopes the voter doesn't notice.

Suppose the voter wants to vote 'yes' for Question 1, but the cheating client wants to vote 'no' and retrieve the voter's expected choice codes. Suppose the voter wants choice p_3 for Question 2.

The cheating client generates a random r and computes the vote as

$$E_1 = (g^r, EL_{pk}^r p_{no} p_3)$$

[3]In real Swiss referenda, there are also explicit "blank" codes, but they are omitted here for simplicity. The attack works exactly the same if they are included.

where EL^r_{pk} is the election public key and p_{no} and p_3 are primes used to represent vote choices. This correctly reflects the voter's choice for Question 2, but the client's substitute for Question 1.

It then generates the partial choice codes $pCC_1 = p_{yes}^{k_{id}}$ and $pCC_2 = (p_{no}p_3/p_{yes})^{k_{id}}$ where k_{id} is a secret specific to that card/voter (spec 5.4.1 (2)). Note that the partial choice code for Question 1 reflects the voter's choice, not the actual vote. The partial choice code for Question 2 is not valid.

It generates E_2 honestly from pCC_1 and pCC_2.

$$E_2 = (g^{r'}, (pk^{(1)}_{CCR})^{r'} \cdot pCC_1, (pk^{(2)}_{CCR})^{r'} \cdot pCC_2)$$

It generates the zero knowledge proofs entirely honestly, because all the facts they are claiming are true. To see why, observe that it needs to prove that the product of partial choice codes is properly exponentiated, not the individual codes.

Recall that $\tilde{E}_2$ is $\tilde{E}_2 = (g^{r'}, (pk_{CC_1}pk_{CC_2})^{r'} pCC_1 pCC_2)$, which it generates honestly from E_2.

Similarly, the exponentiation $F_1 = (g^{r.k_{id}}, EL^{r.k_{id}}(p_{no}p_3)^{k_{id}})$, the result of exponentiating each element of E_1 by k_{id}.

The Schnorr proof proves that the client knows the encryption used to generate E_1, which it does.

The exponentiation proof proves that F_1 is generated by exponentiating each element of E_1 to the private exponent corresponding to the public key $K_{id} = g^{k_{id}}$, which it is.

The proof of plaintext equality proves that F_1 encrypts the same value, under EL, as $\tilde{E}_2$, under the product public key $\Pi_{i=1}^{k} pk_{CC_i}$, which in our example is simply $pk_{CC_1}pk_{CC_2}$. This is true. The message represented by $\tilde{E}_2$ is

$$pCC_1 pCC_2 = p_{yes}^{vcsk}(p_{no}p_3/p_{yes})^{vcsk} = (p_{no}p_3)^{vcsk}$$

which is exactly the plaintext of E_{exp}.

So this vote will pass verification even though the pCC_1 does not match the vote. Now consider whether the first choice code will be returned from the server side.

B. How will this be treated at the server side?

Vote validity will pass because all the ZKPs are valid. Now consider what happens when the VVC attempts to retrieve the choice code. Choice codes are stored in a codes mapping table and retrieved in sequence. The first code will be successfully retrieved because it is an entirely valid code (for a different option from the one the client sent). The second code is not valid. The specification does not say explicitly what should happen when some, but not all, of the codes can be successfully retrieved. That part of the spec is shown in Figure 1.

In practice this would probably fail at the server side without collusion—the code for version 1.0 throws an exception and refuses to return any of the codes. However, the spec should be updated to ensure that a voter's codes are returned only if *all* choice codes are correctly retrieved.

Even if this last step defeats the attack, it doesn't really solve the fundamental problem, which is that the logical conjunction of all the facts proved by the ZKPs does not imply that the ballot is correctly formed. In particular, it does not imply that the pre-choice-codes are consistent with the vote. This can be corrected (at a significant cost to efficiency) by providing separate proofs for each element (rather than the products).

More importantly, even if the spec is corrected, a cheating VVC may simply ignore it and return the codes that can be retrieved even if others cannot. This would require no misbehaviour from the CCRs—see Step 4 of the code return in Section II-B2. This attack is within the security model because the server side is not supposed to be trusted.

Although a misleadingly comforting code can be returned with VVC collusion, it is much less clear that the manipulated ('no') vote can be inserted into the tally and pass audit. We have not detailed the audit specification for this part of the protocol in this paper, but we think it would detect (and refuse to pass) if a vote was accepted when one of its partial choice codes was invalid. However, there is nothing to stop the vote being dropped—this leaves us with a voter who receives their

Once the pre-Choice Return Codes are obtained, the **Vote Verification Context** does the following actions per Verification Card ID (vc_{id}):

1) Looks for the Codes Mapping Table corresponding to the Verification Card ID (vc_{id}) and Election Event ID ($eeid$) and for each pre-Choice Return Code $pC_i^{id} = v_i^k$:
   a) Concatenates it with the Verification Card ID (vc_{id}), the Election Event ID ($eeid$) and the corresponding voting option attributes with the flag "correctness = true". Call the Hash generation primitive with input the concatenated value. The result is the long Choice Return Code:

$$lCC_i^{id} = Hash(v_i^k || vc_{id} || EEID || \{attributes\})$$

   - For each long Choice return Code, call the Hash generation primitive to compute the hash of the long Choice Return Code (lCC_i^{id}) concantenated with the Codes Secret Key (C_{sk}) and call the Key Derivation Function: KDF1 specification to generate the Choice Return Code encryption symmetric key $skcc_i^{id} = KDF(Hash(lCC_i^{id} || C_{sk}), 256\ bits)$.
   - Retrieves the encrypted short Choice Return Code from the mapping table using $Hash(lCC_i^{id})$ and calls the Symmetric decryption primitive to decrypt it using $skcc_i^{id}$. The result is the short Choice Return Code to be sent to the voter.

   b) Checks that all the retrieved short Choice Return Codes are different. If not, interrupt the process and send a validation error.

2) The short Choice Return Codes and the computations done by the Control Components are sent to the **Voting Workflow Context.**
3) If the short Choice Return Codes are correctly retrieved, the **Voting Workflow Context** updates the status of the voting card to SENT BUT NOT CAST.
4) The **Election Information Context** stores the vote, the Control Components computations and generates the receipt (see next section).
5) The Choice Return Codes are sent back to the Voting Client, which displays them in the screen.

Fig. 1. Choice code return in the spec. It's not clear whether the first code is returned if subsequent lookups fail. Nor is it clear in Step 3 whether the status is updated if *any* codes are correctly retrieved, or only if *all* are.

expected choice code, when the vote has not been included. We also do not claim that the confirmation step (which we have also not detailed here) could be successfully subverted. In short, this seems to be a problem, though it is not clear that it leads to a real attack.

a) Summary of the problem: The vote ZKPs are not sufficient for vote validity. In particular, they rely on the product of the codes not the individual codes. This results in a practical attack if some codes are returned to the voter despite others being irretrievable, which might happen accidentally (the spec is vague) or deliberately (server-side collusion).

b) Fixing the problem: One possible fix is to produce separate ZKPs for each element. Alternatively, it might be possible to prove that the product proof is sufficient if the server-side can provably only return codes if *all* the codes are correctly retrieved. However, this latter approach seems difficult in the presence of possibly misbehaving servers.

c) Current status of the problem: It is not entirely agreed that this is a problem, since the voter does receive a wrong code (or no code at all) for at least some choices, though she receives the expected code for the choice that was manipulated. We believe the spec will be updated to specify that the VVC should return codes only if they are *all* retrieved, though this does not address the problem of a colluding VVC.

V. Individual Verifiability: returning the right choice codes for a manipulated vote with server collusion

In this section we show how a cheating client, in collusion with a cheating Vote Verification Context (VVC), can manipulate the vote but retrieve all the voter's expected choice codes.

This will pass verification and show the voter her expected choice codes even though the vote is chosen by the client.

The attack relies on maliciously generated parameters for representing the voting options, a reasonable assumption given the weaknesses that were identified in practice in version 1.0 [3]. Although patching that gap thwarts the attack described here, it does not really solve the fundamental problem, which is that the code-return process (Section II-B2) is meant to be a verifiable distributed computation among 5 parties, with detectable misbehaviour if any subset of the *CCR*s collude, but there is no proof that this property holds and, indeed, it does not hold in its current form. There may be other exploitable failures even if the system is patched to thwart this particular attack. For example, although the specification document requires the VVC to verify some zero knowledge proofs that the CCRs generate, the audit document [12] does not require the auditor to verify them again. There may be other attacks in which a cheating VVC colludes with a CCR to allow the wrong computation even though the ZKPs do not verify.

We require only a cheating VVC colluding with a client—all the *CCR*s can be honest. However, the VVC must know something about maliciously-generated voting parameters. In this example, assume that it knows a, b such that $p_{yes}^a = p_{no}^b \bmod p$. Easily-computed example parameters are in Appendix B1.

Assume that a voter would like to submit a *yes* vote. The voting client cheats and computes a *no* vote, that is, $E_1 = (g^r, EL^r p_{no})$, $E_2 = (g^{r'}, pk_{CCR}^{(1)})^{r'} \cdot p_{no}^{k_{id}})$, together with all the expected ZKPs, which can be honestly computed based on E_1 and E_2 since they are consistent.

A. Faking correct choice return code generation

1) The cheat: The cheating VVC follows the choice-code generation protocol (Section II) almost perfectly, until it needs to compute the pre-Choice Return Code. It is supposed to be generated as

$$pC_1^{id} = p_{no}^{k_{id}}.$$

If *VVC* uses this code, it will obtain the return code for the *no* choice, and an honest and diligent voter will notice that this code is wrong. However, *VVC* can also compute $(p_{no}^k)^{b/a} = (p_{no}^{b/a})^k = p_{yes}^k$, which is the pre-Choice return code from which it can derive the return code for the *yes* choice.

From this observation, we see that, if the group parameters are selected in a malicious way, then a cheating voting client and a cheating voting server (*VVC*) can collude to modify a voter's choice in a way that is completely undetectable. Indeed, the voter receives the codes that she expects to see, the votes that are transmitted and tallied are perfectly valid, all the ZK proofs are valid, and all the decryption operations lead to plausible values.

B. Does it pass verification?

We have not detailed the audit specification for this part of the election, but it is available in [12], Section 8.5. The audit process will pass because the multiplication that the auditor does compute in Steps 4 and 5 will work perfectly to retrieve p_{yes}^k from the codes table. The audit specifies that the auditor must check that the value is "a valid entry of the mapping table," but it has no way of knowing which value was sent back to the voter.

Summary of the problem: The underlying problem is that the MPC protocol to retrieve the choice codes *uses* five parties but is not secure against misbehaviour by one. This example shows that maliciously-generated voting parameters can be used by a malicious VVC, colluding only with a client, to return a correct choice code for a vote that was manipulated. The voter would receive exactly the expected codes, and verification would pass.

Fixing the problem: This specific example can be solved by provably-correct voting parameter generation. However, is much more difficult to guaranteeing security of the multiparty ballot processing protocol, in the presence of a dishonest VVC and up to three dishonest *CCR*s.

Current status of the problem: We believe that future versions will prove that the voting parameters have been properly generated, thus preventing the specific attack described here. We are not aware of any changes to the multiparty vote processing protocol. There may be other attacks that exploit it in a different way.

VI. Picking Election Public Parameters: The use of trapdoor commitments in Bayer-Groth proofs and why the shuffle proof can be faked

We now turn our attention to what happens when votes are received by the mix servers, after they have been processed as described in Section II-B2. At this point in the sVote process, a list of accepted encrypted votes is available. Auditors have to check whether those votes are properly shuffled and decrypted to produce the announced election outcome.

There are four servers called CCM_j for $j = 1, \ldots 4$. In order to achieve universal verifiability, each one is supposed to prove that the set of input votes it received correspond exactly to the differently-encrypted votes it output—this is called a proof of shuffle. (They also perform partial decryptions which they must prove correct—see Section VII.)

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.

A. Overview of this section: faking the shuffle proof

We show that the SwissPost-Scytl mixnet specification and code does not meet the assumptions of a sound shuffle proof.

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. The same point was made independently by Haenni [13].

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 [10], 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 data leakage from the client, but has some constraints on the candidates for which votes could be added or removed and some extra assumptions about corrupt parameter generation. If, for some reason, these constraints are not satisfied, then the same strategy can still be used to render some chosen votes invalid.

B. The soundness of the shuffle proof

The Scytl-Swisspost mixnet uses a provable shuffle due to Bayer and Groth [14]. 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 a misalignment of its assumptions with the context that sVote uses it in.

The issue concerns the soundness of the commitments. A core security requirement of commitment schemes is that they be *binding:* 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 $H, G_1, G_2, \ldots G_n$. 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 [15] 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 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 $G_i w.r.t G_j$, they can create a commitment that they can open in multiple ways.

The system should prove, and the verifiers should check, that the generators are selected properly, *i.e.,* with no way 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 shown in Figure 2.

The implementation of `getVectorRandomElement` gathers random group elements without proving where they came from. Even more worryingly, `getVectorRandomElement` calls `getRandomElement`, which proceeds as shown in Figure 3—it simply generates a random exponent and raises g to that value. This `randomExponent` 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 `randomExponent` is properly erased from the memory. These commitment parameters are eventually used 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 gives it a trapdoor that falsifies an assumption that is central to the security of the Bayer-Groth mixnet construction.

```
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);
}
```

Fig. 2. Code for generating commitment parameters.

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

Fig. 3. Code for generating a random group element. Note that the exponent is explicitly used in its computation, so the discrete log of the output is known.

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

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.

The binding property 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

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

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

Details of how to leverage this into a complete false shuffle proof are contained in Appendix A.

C. 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 e-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. An easily-computed example set of trapdoored parameters is in Appendix B.

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 proof of security [10] 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 see any reason to believe there is an error in Scytl's proof, but when the axioms are mistaken the conclusions are not valid.

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.

Summary of the problem: 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.

Fixing 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—concrete suggestions are contained in Section VIII. Every verifier then needs to check the generation of the commitment parameters as well as the rest of the proof transcript.

Current status of the problem: 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 and g. This issue and the correction require further public scrutiny.

VII. Universal Verifiability Failures from the weak Fiat-Shamir transform

In this section, we show that the error in the implementation of the Fiat-Shamir heuristic (already described in Section III) allows a cheating authority to produce a proof of proper decryption, which passes verification, but declares something other than the true plaintext.

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.

Just like the previous section, this voids the arguments that the 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.

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.[4] This exploit could then be used to political advantage to nullify only those votes with which the cheater disagreed.

We have provided two examples of decryption proofs that pass verification but change the plaintext. We have not implemented the inclusion of the fake decryption proof into the sVote mixing and decryption sequence (Section VII-B3).

A. 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 [12, 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 [7, 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 [7, 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 [7, 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.

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.

B. 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

$$\mathsf{dlog}_g pk = \mathsf{dlog}_{C_0}(C_1'), \text{ where } C_1' = C_1/m. \tag{2}$$

1) The Chaum-Pedersen proof: sVote's Decryption proof [7, Section 9.1.8] is very similar to the exponentiation proof described in Section III-A3. It uses a well-known proof method due to Chaum and Pedersen [11] to prove Equation 2.

Like the exponentiation proof, the Fiat Shamir heuristic is implemented incorrectly, with hash inputs that do not include C_0 (or g). Thus a cheating prover can choose C_0 after generating the rest of the proof. This allows it to produce a proof (c, z) that passes verification as a decryption proof that (g, pk, C_0, C_1') satisfy Equation 2 although (C_0, C_1') actually decrypts to a

[4] 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.

random value (not 1). The details are very similar to the exponentiation proof forgery, and are contained in Appendix C.

2) Transforming to a set of fake decryption proofs that pass verification: A decryption proof [7, 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) 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 Appendix C.

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 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.

C. 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.

Summary of the problem: 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.

Fixing the problem: Same as Section III. Hash all relevant data when using the Fiat-Shamir heuristic.

Current status of the problem: An effort is being made to remediate the problem in future versions.

VIII. How do Others Pick their Public Parameters? How should they?

In this section, we examine some of the ways independent generators—on which the commitment schemes rely—are generated in practice. We discuss the common approaches taken and highlight critical problems than can, and have, occurred.

One key issue that seems to have resulted in the problems we describe is the definition of the word "independent." When independent generators are required for the commitment scheme, the term means that the discrete log relation between them is unknown. In contrast, when discussing probability, the word independent means that probability does not change depending on the outcome of the others.

A. Common methods

There are two methods which are commonly used to generate independent generators. The first is contained in Algorithm A.2.3 Verifiable Canonical Generation of the Generator g from NIST fips186-4 [16] and is reproduced here as Algortihm 1. The second is Handbook of Applied Cryptography, Algorithm 4.80 and Note 4.81 [17] and as Algorithm 2 below.

The first method was specifically designed to find independent generators in Schnorr groups. The second is designed to find a single generator for a cyclic group. Both algorithms have commonly been adapted to other kinds of groups. In many libraries neither method is explicitly cited but the method used is nevertheless nearly identical to one of them.

The two algorithms put different demands on their source of randomness. In the NIST standard (Algorithm 1) a sufficiently controlled *domain_parameter_seed*, for instance the name of the election, combined with a hash function with sufficient domain would seem acceptable. In Algorithm 2, the randomness

Algorithm 1: Algorithm A.2.3 Verifiable Canonical Generation of the Generator g

Data: p, q for a Schnorr group of order q in the field $\mathbb{F}_p$ of order $p = kq+1$
domain_parameter_seed
index
Result: *status*, g

1 if (*index* is incorrect), then return **INVALID**
2 N = **len**(q)
3 $e = (p-1)/q$
4 count = 0
5 count = count +1
6 if (count = 0), then return **INVALID**
7 U = *domain_parameter_seed* $\|$ "*ggen*" $\|$ *index* $\|$ *count*
8 W = **Hash**(U)
9 $g = W^e \bmod p$
10 if ($g < 2$), then go to step 5
11 Return **VALID** and the value of g

Algorithm 2: Handbook of Applied Cryptography, Algorithm 4.80

Data: a cyclic group G of order n, and the prime factorisation $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$.
Result: a generator α of G.

1 Choose a random element α in G.
2 **for** *i from 1 to k do the following:* **do**
3 Compute $b \leftarrow \alpha^{n/p_i}$.
4 If $b = 1$ then go to step 1.
5 Return(α).

used in Step 1 must be chosen in a verifiable manner. Failing to check this risks a trapdoor like that described in Section VI.

In both these cases the key point is the way in which the candidate, for generator, element is mapped into the group. It does not suffice for the candidates to be chosen randomly (uniformly and independently) but rather they are chosen so that no additional information is revealed about the discrete log of the element.

B. Potential problems with these methods

In the first method, knowledge of process reveals no more information than just seeing the random generators, at least up to some assumptions on the hash function. This is true because the Tonelli-Shanks algorithm can easily be used to compute w from the generator g. In the second method, the way in which the random element is chosen is undefined; therefore, so is the security of the algorithm when being used to find independent generators.

C. In Practice

We now detail several real cases where this has and has not occurred. We again stress that in no case did the developer of the library not understand the issues involved. However, direct use of some of the libraries could, and likely would, result in an insecure e-voting scheme.

1) UniCrypt: The UniCrypt cryptographical framework[5] by the Bern University of Applied Sciences (BFH), Research Institute for Security in the Information Society (RISIS), E-Voting Group (EVG) is a framework for implementing e-voting that is or was used/trialed in a number of companies and solutions.[6]

It implements both of the methods described above but primarily the second. The exact way in which the random element is generated differs based on the group in question and this makes all the difference.

Group	Method	Security
Sub-group of order m of a cyclic group of integers $\mathbb{Z}_n^*$	$k \leftarrow^{\$} \mathbb{Z}_n^*, k^{n/m}$	Secure
Elliptic curve of order q	$k \leftarrow^{\$} (0, q-1), g^k$	Broken

In discussion with the authors of UniCrypt we discovered that the EC extension was a student project and not supported in the same way as the Schnorr groups; nevertheless, this is not obvious to us from looking at library and we suspect would not be obvious to others. To the best of our knowledge, all of the systems built on top of UniCrypt (which have been used in real elections) were instantiated using the secure version. However, in most cases a simple modification of about three lines would switch the system to the insecure variant.

2) Stadium: The Stadium project, and its descendants, have two implementations of Bayer-Groth mixnets in C++, one for Schnorr groups[7] and one for elliptic curves[8]. Both of these use the naive method described above and hence would invalidate the verifiability claim of the e-voting system in which they were used. This in no way undermines the value of the prototype but highlights the danger of using them without due care.

3) GRNET-Fauzi et al: The Zeus project [18] has implemented a prototype of the Fauzi et al mixnet [19].[9][10] Unlike the other mixnets mentioned here, the secure generation of the common reference string for Fauzi et al. is non-trivial and multiparty computation is probably required. These libraries do not securely implement the common reference string generation.

4) CHVote prototype (2.0): The CHVote prototype 2.0 correctly implements and uses a method to create independent generators.

5) Verificatum: Verificatum provides a good example of how to securely generate independent generators for elliptic curves.[11] We present a slightly simplified version here.

[5]https://github.com/bfh-evg/unicrypt
[6]We gathered this information on company usage by searching GitHub.
[7]https://github.com/derbear/verifiable-shuffle
[8]https://github.com/nirvantyagi/stadium/tree/master/groth
[9]https://github.com/grnet/ac16
[10]https://github.com/StefanosChaliasos/gsoc17module-zeus
[11]https://github.com/verificatum/verificatum-vcr

Algorithm 3: Verificatum Core Routines - randomElement (simplified)

Data: a cyclic elliptic curve group G of order q, over a finite field $\mathbb{Z}_p$ of order p. The curve is defined by the equation $y^2 = x^3 + ax + b$

Result: an element α of G.

for *i from 0 to* ∞ *do the following:* **do**
  Choose a random element β in $\mathbb{Z}_p$ in a verifiable way.
  if $(\beta^3 + a\beta + b)^{\frac{p-1}{2}} = 1$ **then**
    return $(\beta, \sqrt{\beta^3 + a\beta + b})$

D. Conclusion and Recommendations

Given that for most cases in e-voting it is possible to securely—and verifiably—generate independent generators without relying on multiparty computation, we recommend that this and only this should be done. We are impressed by the understanding and care taken by those researchers implementing e-voting libraries but concerned by the dangers still present by others using their libraries without understanding.

IX. Discussion/Conclusion

We have shown numerous serious issues with the complete election verifiability process of the sVote 2.1 protocol, which open the way for undetectable electoral fraud in the Scytl-SwissPost system. Fake proofs are possible at almost every step, from the client proving that the vote is valid, to the return of choice codes, to the mixing and decryption of the votes.

In all cases, formally, verification would pass, though in some cases it would probably be observed that something unusual occurred (such as the presence of invalid votes).

We are a small team of researchers investigating this code base for the first time. We inspected only a small fraction of it. The code is very complex and difficult to understand. There is no reason to think that correcting all the known flaws will be easy, or that it will produce a secure system with no further opportunities for undetectable electoral fraud.

We are now told[12] that code for generating commitment parameters properly was already present. But the code is so convoluted that (as far as we know) none of the official audits noticed that it was not actually being used to generate the parameters (Section VI). Besides, the specification describes the insecure process, and the verification specification does not mention verifying proper generation.

As well as the direct impact of the the attacks, our analysis of the ZKPs shows that they do not offer the security guarantees that are assumed in the "complete verifiability security proof" [10]. Therefore, successfully passing the current sVote audit process [12] cannot be used to draw any conclusion regarding the correctness of the election outcome.

This does not mean that formal proofs are useless—they are important for clarifying assumptions and security claims, and they provide an argument (which can be checked) for why the system should be trusted. However, proofs themselves can be mistaken or insufficient. They should be required, but they are a part of the scrutiny process not a substitute for open scrutiny.

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 "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." This enforces a process that increases the likelihood that you hear the truth about its security properties before you vote over the Internet.

It is unfortunate to make a code base (like sVote 2.1) available for public examination and then have to withdraw it because its failings have become evident, but it is much worse to run a flawed system (like sVote 1.0 and iVote) in a binding election and learn afterwards that it did not meet its security goals. The failures that have led to decreased confidence in Swiss e-voting should substantially increase doubt in other e-voting systems, not only those by the same vendor, but any that have not had an extensive period of open public scrutiny.

Thanks to the Swiss process, flaws that we publicly described were found to be present in other Scytl code being used for a running election in New South Wales (NSW), Australia. That code was not available for open review and discussion (though it now is). Without Switzerland's mandated transparency, opportunities for undetectable electoral fraud in NSW would probably have gone unnoticed through a running election.

Scytl claims to sell Internet voting systems in numerous other democracies,[13] including Canada, Brazil, Mexico, India and the UK, so quite possibly these same errors are present in other systems whose administrators have been even less forthcoming than those in NSW.

As applications of zero knowledge and MPC go mainstream, we can expect to see more proprietary systems in which non-experts attempt to guarantee sophisticated properties without detailed security proofs or open review. We hope this work can help customers assess what does, and does not, constitute genuine verifiable computation. There is nothing gained by using a proven-secure component if its assumptions are not met in the context in which it is used. Nor is there any advantage to sound ZKPs if they do not actually prove what is needed in the rest of the protocol.

The aim of verifiable election software is verifiable election outcomes, not proofs that pass. If the system itself does not come with meaningful evidence that its verification procedure is sound, then an apparently-successful verification implies nothing about the integrity of the election result.

[12] https://www.scytl.com/en/scytl-responds-misinterpretations-related-swiss-\posts-media-release/

[13] https://www.scytl.com/en/customers/

References

[1] U. M. Maurer, "Unifying zero-knowledge proofs of knowledge," in *Progress in Cryptology - AFRICACRYPT 2009*, ser. Lecture Notes in Computer Science, vol. 5580. Springer, 2009, pp. 272–286.

[2] D. Bernhard, O. Pereira, and B. Warinschi, "How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios," in *ASIACRYPT 2012*, ser. Lecture Notes in Computer Science, X. Wang and K. Sako, Eds., vol. 7658. Springer, 12 2012, pp. 626–643.

[3] P. Locher, R. Haenni, and R. E. Koenig, "Analysis of the cryptographic implementation of the swiss post voting protocol," https://www.bk.admin.ch/bk/en/home/politische-rechte/e-voting.html, Jul. 2019.

[4] O. Pereira and V. Teague, "Report on the swisspost-scytl e-voting system, trusted-server version," https://www.bk.admin.ch/bk/en/home/politische-rechte/e-voting.html, Jul. 2019.

[5] J. Benaloh and M. J. Fischer, "A robust and verifiable cryptographically secure election scheme (extended abstract)," in *26th Annual Symposium on Foundations of Computer Science (FOCS 1985)*. IEEE, 1985, pp. 372–382.

[6] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. IT-31, no. 4, pp. 469–472, July 1985.

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

[8] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

[9] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 186–194.

[10] 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.

[11] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *Advances in Cryptology - CRYPTO '92*. Springer, 1992, pp. 89–105.

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

[13] R. 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, Mar. 2019.

[14] S. Bayer and J. Groth, "Efficient zero-knowledge argument for correctness of a shuffle," in *Advances in Cryptology - EUROCRYPT 2012*. Springer, 2012, pp. 263–280.

[15] M. Fischlin, "Trapdoor commitment schemes and their applications." Ph.D. dissertation, Goethe-University of Frankfurt, 2001.

[16] P. FIPS, "186-4: Federal information processing standards publication. digital signature standard (dss)," *Information Technology Laboratory, National Institute of Standards and Technology (NIST), Gaithersburg, MD*, pp. 20 899–8900, 2013.

[17] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[18] G. Tsoukalas, K. Papadimitriou, P. Louridas, and P. Tsanakas, "From helios to zeus," in *2013 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '13, Washington, D.C., USA, August 12-13, 2013*. USENIX Association, 2013. [Online]. Available: https://www.usenix.org/conference/evtwote13/workshop-program/presentation/tsoukalas

[19] P. Fauzi, H. Lipmaa, and M. Zajac, "A shuffle argument secure in the generic model," in *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, J. H. Cheon and T. Takagi, Eds., vol. 10032, 2016, pp. 841–872. [Online]. Available: https://doi.org/10.1007/978-3-662-53890-6_28

[20] R. Cramer, I. Damgård, and B. Schoenmakers, "Proofs of partial knowledge and simplified design of witness hiding protocols," in *Advances in Cryptology - CRYPTO '94*, ser. Lecture Notes in Computer Science, vol. 839. Springer, 1994, pp. 174–187.

X. Acknowledgements

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

Olivier Pereira is also grateful to the Belgian Fund for Scientific Research (F.R.S.- FNRS) for its financial support provided through the the SeVoTe project, and to the European Union (EU) and the Walloon Region through the FEDER project USERMedia (convention number 501907-379156).

Thomas Haines acknowledges the support of the Luxembourg National Research Fund (FNR) and the Research Council of Norway for the joint project SURCVS.

We would like to thank Swiss Post for the civilised way they have received our analysis, and for running a public test of the code. Although we were not willing to sign on to the participation conditions, in practice the code did circulate quite freely on the web. This is a good thing for Swiss democracy.

XI. 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.

Appendix

A. Using trapdoored commitments to fake the shuffle proof

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

1) Faking a proof for 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.

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 $C_1 = \mathscr{E}_{pk}(M_1, \rho'_1), C_2 = \mathscr{E}_{pk}(M_2, \rho'_2)$, $C_3 = \mathscr{E}_{pk}(M_3, \rho'_3)$ 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

$$\begin{array}{rcccl} C'_1 & = & \mathscr{E}_{pk}(1;\rho_1)C_1 & = & \mathscr{E}_{pk}(M_1,\rho_1+\rho'_1) \\ \hline C'_2 & = & \mathscr{E}_{pk}(\mathbf{1};\boldsymbol{\rho_2})\mathbf{C_3} & = & \mathscr{E}_{pk}(\mathbf{M_3},\boldsymbol{\rho_2}+\boldsymbol{\rho'_3}) \\ \hline C'_3 & = & \mathscr{E}_{pk}(1;\rho_3)C_3 & = & \mathscr{E}_{pk}(M_3,\rho_3+\rho'_3) \\ \text{and } C'_4 & = & \mathscr{E}_{pk}(1;\rho_4)C_4 & = & \mathscr{E}_{pk}(M_4,\rho_4+\rho'_4) \end{array}$$

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 VI-B1. The cheating shuffler's initial message $\vec{c}_A$ is a (truthful) commitment to π. That is,

$$\vec{c}_A = \mathsf{com}_{ck}(\vec{A}_1; r_1), \mathsf{com}_{ck}(\vec{A}_2, r_2),$$

where $\vec{A}_1 = (\pi(1), \pi(2))$ and $\vec{A}_2 = (\pi(3), \pi(4))$.

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

$$\vec{c}_B = \mathsf{com}_{ck}(\vec{B}_1; s_1), \mathsf{com}_{ck}(\vec{B}_2, s_2),$$

where $\vec{B}_1 = (x^{\pi(1)}, x^{\pi(2)})$ 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.

a) 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:

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

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

- It sets

$$\rho = -\rho_1 x - (\rho_1 + \rho'_1)x^2 + x^2\rho'_2 - \rho_3 x^3 - \rho_4 x^4. \quad (3)$$

  (See Appendix A2a 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) Faking a proof for 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):

Suppose that the voting parameters are, again, maliciously generated. In this case the mixer knows values a and d so that vote options p_{yes} and p_{no} satisfy

$$2^a = (p_{no}/p_{yes})^d \bmod p$$

.

This is probably hard to generate for a *given* p, but it is not hard to generate values of p, p_{yes} and p_{no} for which such a and d are known. Several sets of complying parameters are contained in Section B below.

This allows allows a cheating mixer to change a vote for p_{yes} into a vote for p_{no} by multiplying by $2^{a/d}$.

For concreteness, suppose that we have a single-choice election and that the last mixer receives input ciphertexts $C_1 = \mathscr{E}_{pk}(M_1, \rho'_1), C_2 = \mathscr{E}_{pk}(M_2, \rho'_2)$, $C_3 = \mathscr{E}_{pk}(M_3, \rho'_3)$, $C_4 = \mathscr{E}_{pk}(M_4, \rho'_4)$ such that the cheater's preferred candidate, represented by p_{no}, does not win the election.

The last mixer can now perform the final decryption step in order to identify which of these ciphertexts contain a vote for p_{yes}. It does not learn the randomness $\rho'_1, \rho'_2, \rho'_3, \rho'_4$. Again, for simplicity, let us assume that the true result is unanimous: the mixer finds out that everyone voted for p_{yes}.

In order to manipulate the outcome, the mixer defines the output ciphertexts as $C'_i = \mathscr{E}_{pk}(2^{a/d}, \rho_i)C_i$. By the homomorphic property of ElGamal, we have multiplied each encrypted vote vote by p_{no}/p_{yes}. (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 $\vec{C}^{\vec{x}} = \mathscr{E}_{pk}(1;\rho)\vec{C'}^{\vec{b}}$ does not hold. Instead, the equation $\vec{C}^{\vec{x}} = \mathscr{E}_{pk}(2^{-(x+x^2+x^3+x^4)a/d};\rho)\vec{C'}^{\vec{b}}$ 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 $b_m = -(x + x^2 + x^3 + x^4)a/d$ 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 truthfulness of any of the underlying proofs.

a) Calculating ρ: This section shows why we get the expression for ρ that we use above.

We needed to find ρ s.t.

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

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

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

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

B. Trapdoored voting parameters

The following parameters are consistent with all the specified rules of generation (except that the first is a little short), and also satisfy $2^a = (c/b)^d \bmod p$, as required by the second attack in Section VI, in which we show that we can switch a valid vote for one candidate into a valid vote for another.

$a = 653$, $b = 107$, $c = 1097$, $d = 55$,
$p = 154416939733293841511253509950176540080235658179142828432034537739002300487264870649972196943240205309469342226350935416403841627526252460636822182648190876213685901769892542773697006229704670632244497722914530452418434027431462292187931277293070499453123834777026998428423476982337655176255426398664922523463.$

This one is the expected length (2048 bits):
$a = 1939$, $b = 149$, $c = 5297$, $d = 15$,
$p = 19722211808861961998510473803189009728961510664629545970129506316650165879602843455530588640041642263567488806264681220105302704546328968082270494358116782188791058782334971234980982393280439569631976455980647437891569674971597914519724800588842248812978910374796242903712426859854804327310464272426209417044887320406964517516088674160658753919846532769835002917041296630094712424310390226660330163500145364872846224264776993414544017768191588140498688094713424617499173689382179303046730867743281539925332972297626321785335694054401669188490647358257366842517594682494401585422982790377702210094769635988172380985519.$

1) Trapdoored election parameters for the attack in Section V: The following parameters were generated in a few hours on a standard laptop (along with many other similar parameter choices). They are such that v, w, p and $q = (p-1)/2$ are prime, 2, v, and w are quadratic residues modulo p, $v^a = w^b \bmod p$, and $|p| = 2046$.

$v = 11, w = 53, a = 592, b = 357$,
$p = 70661253006860938188288686008587306877924989809763017605234587520311617337105046449553576599718412087023157743527914173027880612549152925893965244558545474129308217060017773882336283820366471809571051189156176768816344699208105091538533363999412975733618190464709094803803163968319799200086181544516808280230172888032317476018477679086575899964746340368641784343728714991157449798907909149673611221282033579089825567307259482413074109983096834031357018344661661795082193200047710072016039908802133857985860785937758668013110558845552099425659027679535910743949319726649140277133155445801162564289021630221463379552 7.$

C. Details of decryption proof forgery

This section shows how to generate a fake proof of decryption that passes verification, as required in Section VII.

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 2. 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 $c = H(pk, C'_1, B_0, B_1)$.

D. 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 2. 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.

E. Other cryptographic issues

Here is a short list of other problems we have noticed. They do not seem to lead to attacks, but they do undermine the assumptions of some components.

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 above 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, proofs of plaintext equality, *etc.*) appear to not be adaptively secure. It is plausible that this weakness could be exploited in other ways.

To illustrate our concern, we show a second brief example with the Schnorr proof. The sVote Audit document [12, 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.

2) Missing verification steps in OR Proof: The code base defines a 1-out-of-n zero knowledge proof construction (ORProof)[14] 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 [20] 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 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.

3) Non-collision-resistant hash function: The hash function inputs numbers in a sequence of characters without describing the lengths or the types. 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.

[14] 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