Report on the SwissPost-Scytl e-voting system, trusted-server version

Olivier Pereira – Vanessa Teague*

July 17, 2019

Executive Summary

We were given a mandate by the Swiss Federal Chancellery to examine the SwissPost-Scytl e-voting system, version v1, which was certified for use by up to 50% of voters. This followed previous work (with Sarah Jamie Lewis) in which we discovered a cast-as-intended verifiability failure [LPT19a]. Our mandate was accomplished on a best-effort basis, within very limited time-constraints. The specification documents were the core of our mandate, not the code.

We did not find any new vulnerabilities in cast-as-intended verifiability that are exploitable in the code. We did find examples in which the specification can be interpreted in a way that causes cast-as-intended verifiability to fail, though implementation choices in the current version of the code prevent successful attacks. However, the design updates are so recent that it is impossible to rule out exploitable errors.

We did find significant errors and omissions in the proof of individual verifiability [Scy17b] (*i.e.* cast-as-intended verifiability), as well as significant deviations from the current protocol. There are also differences between the specification and the code.

As a result, we conclude that cast-as-intended verifiability may or may not be sound now, but it is not proven and has not had sufficient examination as far as we know.

Although we did not find new attacks within the minimum adversary model in which the server-side of the voting system is completely trusted, we did find specific attacks within the stronger model that is often informally claimed throughout the documents. A single malicious entity on the server side can read and undetectably alter votes. This is easy given the absence of use of a verifiable mixing protocol but, less intuitively, would remain possible even if a verifiable mixnet was used.

We would like to thank SwissPost for their prompt and comprehensive answers to our technical questions, and for their intelligent response to our earlier security disclosures.

*The University of Melbourne, Parkville, Australia. vjteague@unimelb.edu.au

Mandate and background

This report was prepared under a mandate from the Swiss Federal Chancellery. We examined the following documents:

[Scy17b] Scytl, April 2017, "Swiss Online Voting System Cryptographic proof of Individual Verifiability,"

[Bv17] Basin and Čapkun, May 2017, "Review of Electronic Voting Protocol Models and Proofs (Combined Final Report),"

[Scy19a] Scytl, 2019, "Scytl Online Voting Protocol Specifications – Differences between versions 5.2 and 5.0,"

[Scy19b] Scytl, 2019, "Scytl Online Voting Protocol Specifications – Document version 5.2,"

[Scy19c] Scytl, 2019, "Scytl Online Voting Protocol Specifications – Document version 5.3," (communicated on June 23, 2019)

[Sec19] Kudelski Security, May 2019, "Swiss Post Security Review of Key Cryptographic Elements of the E-Voting Solution (Version with individual verifiability at 50% of the electorate),"

[Scy19d] Scytl, May 2019, "Security analysis of key cryptographic elements for individual verifiability, v 1.1."

The first two are publicly available; the rest remain confidential at the time of writing. A symbolic proof is also publicly available [Scy17a], but we did not examine it in detail.

We were also provided with the source code for the system and extensive answers to technical and nontechnical questions about the system specification from experts at SwissPost.

Methodology

The core focus of our mandate was to analyze, on a best-effort basis, whether the cryptographic protocol specified in [Scy17b] is correctly instantiated in the system specification. Version 5.3 of the protocol specification [Scy19c], delivered at a late stage in this mandate, already included changes related to a significant number of observations that we had previously made.

Our methodology consisted of reading the documentation carefully and, when it was not clear, seeking clarification either from SwissPost or from the code directly. This means that we have conducted a thorough examination of the proof document and a fairly careful read of the parts of the specification that relate to the main privacy and verifiability properties, but we have only spot-checked the code to answer particular questions when the specification was not clear.

Contents

1. Introduction: defining the security goals **4**
- 1.1. The logic of assurance . . . 5
- 1.2. Thought experiment: super-simple voting system . . . 6

2. Foundations: examining the Cryptographic proof of Individual Verifiability **7**
- 2.1. The structure and importance of computational security proofs . . . 8
- 2.2. Gaps in the proof of individual verifiability . . . 8
  - 2.2.1. Assumptions . . . 9
  - 2.2.2. Theorem statement . . . 11
  - 2.2.3. Proof . . . 12
- 2.3. Differences between the spec and the proof of individual verifiability . . . 15

3. Corrections to the specification—within the trust model **16**
- 3.1. Returning choice codes only if all retrievals have succeeded . . . 17
  - 3.1.1. The setting . . . 19
  - 3.1.2. Substituting a choice but generating the correct return code . . . 19
  - 3.1.3. How does the proof of individual verifiability deal with this case? 20
  - 3.1.4. How to address the problem . . . 21
- 3.2. Correcting the specifications of the zero knowledge proofs . . . 22
  - 3.2.1. Checking the lengths of input values: exponentiation proof . . . 23
  - 3.2.2. Checking the lengths of input values: proof of plaintext equality . 23
  - 3.2.3. Using extra-long ciphertexts to submit inconsistent vote and partial choice codes . . . 24
  - 3.2.4. Recommendations . . . 26
- 3.3. Summary of attacks on the spec, but not the code, within the trust model 27

4. Examining sVote's trusted server side—outside the trust model **27**
- 4.1. The lack of verifiable mixing . . . 29
- 4.2. HMAC-based code recovery . . . 30
  - 4.2.1. Dropping a confirmation while returning the correct Vote Cast Code 30
  - 4.2.2. Brute-forcing a confirmation that the voter did not want . . . 31
  - 4.2.3. Discussion and possible mitigations . . . 32
- 4.3. The use of trapdoored parameter generation . . . 33
  - 4.3.1. Vote privacy - server-side only . . . 34
  - 4.3.2. False return-code generation - client-server collusion . . . 35

5. Receipt Freeness **36**

6. Discussion and Summary **37**

A. Trapdoored election parameters **40**

> There are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies.
>
> — C. A. R. Hoare [Hoa81].

1. Introduction: defining the security goals

There are numerous legal requirements for an Internet voting system certified for use by up to 50% of the Swiss electorate [Cha18b, Cha18a]. The ones most relevant to this report are summarised here.

- **Cast-as-intended verifiability.** This is intended to detect a cheating client, on the assumption of the trustworthiness of the server system.

  "If a system is to be authorised to cover more than 30 per cent of the cantonal electorate, the voters must be able to ascertain whether their vote has been manipulated or intercepted on the user platform or during transmission." ([Cha18a] individual verifiability, 4.1 and 4.2).

  "For the purpose of individual verification, voters must receive proof that the server system has registered the vote as it was entered by the voter on the user platform as being in conformity with the system. Proof of correct registration must be provided for each partial vote." ([Cha18b], Article 4, Paragraph 2.)

- **Vote privacy.** "It is guaranteed that neither employees nor externals obtain data that allow a connection to be made between the identity of voters and the votes they have cast." ([Cha18a], 2.8.1)

- **Coercion resistance.** "the risk of vote selling is not significantly greater than with postal voting." ([Cha18a], 4.2.2)

- **Evidence of protocol security.** "A cryptographic and a symbolic proof must be provided. The proofs relating to cryptographic basic components may be provided according to generally accepted security assumptions (for example, the 'random oracle model', 'decisional Diffie-Hellman assumption', 'Fiat- Shamir heuristic')." ([Cha18a], 5.1.1)

Annex Sections 4.2.3 and 4.2.4 also describe limitations on the opportunity for a malicious system to cast a vote without the voter's approval, and to limit the attacker's probability of forging an apparent (but false) verification to at most 0.1%. We will return to these provisions in Section 4.2

1.1. The logic of assurance

In order to be convinced that sVote v1 satisfied a certain cryptographic property, we would follow three steps:

A (computational) cryptographic proof document would define the property, specify a cryptographic protocol and a set of assumptions, and prove that the protocol satisfied the property given the assumptions. For example, one important property is cast-as-intended verifiability (as described above). In the accepted model, an attacker has access to many clients but no server-side components.

A precise specificiation would describe exactly how the cryptographic protocol was to be implemented. For example, where the cryptographic protocol describes zero knowledge proof systems with certain properties, the specification describes the exact parameters that are chosen for the proof system, the format of the corresponding messages, the input validation process with the corresponding error signals, ...

The code would implement the specification exactly as described.

At each transition, the assumptions made at a higher level of abstraction must be delivered by the lower level—there is nothing gained by proving something that does not match the real system. If this sequence of steps were followed, there would be some assurance that, given the assumptions in the proof, the software system had the relevant security property.

The first sections of this report follow the structure of that assurance. Although we did not find new attacks against Individual Verifiability that fall within the security model and are exploitable in the code, we identified many places where the logic of assurance falls short. In Section 2 we identify several gaps and errors in the proof, along with significant differences between the proof and the specification. In Section 3 we examine the specification, version 5.3 [Scy19c]. We explain some important details that need to be filled in and errors to be corrected—in each case, these issues could potentially lead to attacks within the trust model if the spec were interpreted in a certain way, though these attacks do not actually work on the code as it is now implemented. We identify several points in which the code differs from the specification. Even if these differences between the code and the specification appear to prevent some attacks, it is unclear whether these, or other differences, would actually introduce other weaknesses that do not appear in the specification.

In Section 4, we examine the trusted server model and undetectable verifiability failures that can arise with even one cheating server-side component. We also consider the implications of improper generation of the voting parameters. Although these attacks are outside the security model, it is important to understand that they are present, and to communicate clearly about the possibility. Finally, in Section 5 we examine the receipt freeness of the scheme.

We begin with a thought experiment: a super-simple voting system against which to compare sVote's security properties.

1.2. Thought experiment: super-simple voting system

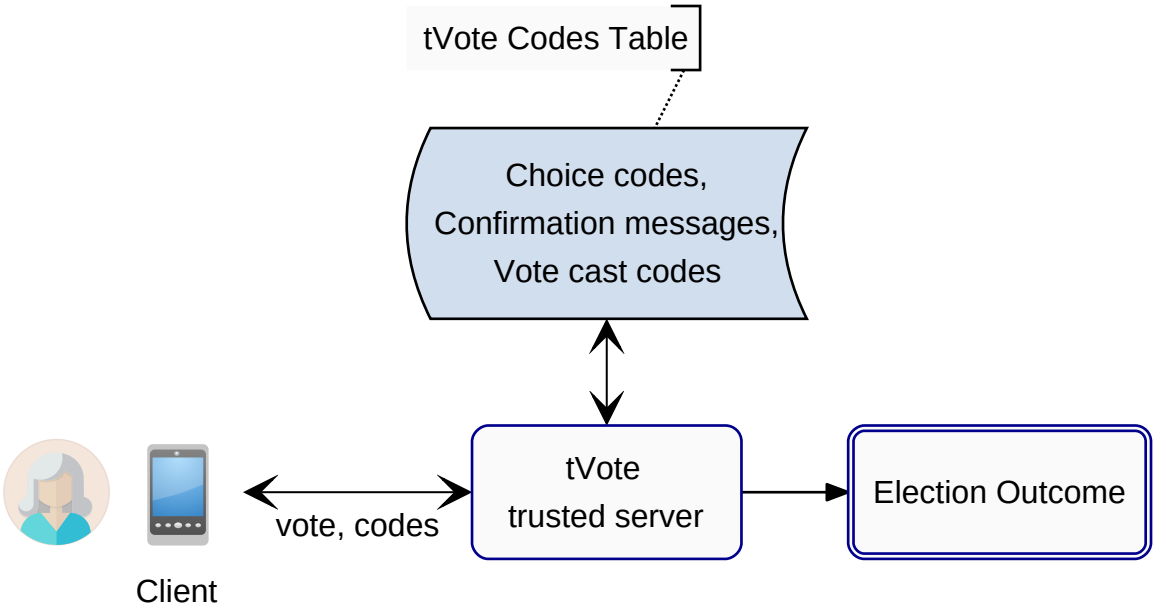

Figure 1: A super-simple voting system with vote choice codes, ballot casting and vote cast codes, and a trusted server side

Consider an imaginary super-simple trusted voting system tVote, depicted in Figure 1. tVote has a single server *tVote*, who also generates a code sheet for each voter. At the beginning of the election, *tVote* generates and publishes an encryption public key pk_{tVote}. At voting time, the voter uses their voting client to authenticate to *tVote*, then submits a vote encrypted with pk_{tVote}. *tVote* decrypts the vote, looks up the voter's choice codes in its code table (which it remembers because it generated them), and returns to the voter the choice code corresponding to the vote she cast. The system also has ballot casting and confirmation codes, also generated by *tVote* and printed on the voters' cards. When a voter receives the correct vote choice codes, she enters her ballot casting key. If *tVote* receives the expected ballot casting key from an authenticated voter, it confirms the vote and returns a Vote Cast Code to the voter. At the close of polls, *tVote* counts all confirmed votes and announces the results.

We haven't provided a formal proof but, apart from that, it seems this super-simple system is compliant with (a minimal interpretation of) the requirements given above. Assuming that *tVote* is trustworthy, a cheating client cannot generate proper choice codes without having submitted the voter's chosen vote. Likewise, the cheating client cannot derive the proper Vote Cast Code without going through the process in which *tVote* confirms the vote. Vote privacy is protected in the sense that third-party observers

cannot learn how people vote, so it satisfies the second requirement, assuming that we trust *tVote* not to divulge individual votes to employees or externals.

However, this voting system has some obvious limitations. Vote privacy is not protected from *tVote* itself, or anyone it chooses to leak to. Nor is integrity: *tVote* could simply lie about the election result. It could easily collude with a cheating client to return the codes that the voter expected, even if the sent or recorded votes were completely different.

We will examine the security properties of sVote v1, the SwissPost-Scytl e-voting system that was certified for use by up to 50% of voters, while comparing it with the (obvious) security properties of tVote. Our analysis considers the cryptographic protocol properties, not the procedural or administrative protections that might be applied to either system. We find that, despite its much greater complexity, sVote v1 actually doesn't have stronger security properties than tVote. Although, strictly speaking, it seems to meet the requirements for cast-as-intended verification under the assumption of a trusted server, neither privacy nor integrity are protected against server-side malicious behaviour, even by a single component. We did not find any specific places in which sVote v1's security properties are strictly less than tVote's, though we have in the past [LPT19a]. The more complex system is more, not less, likely to have unnoticed subtle bugs that undermine the intended security properties, especially given the numerous inconsistencies and gaps that are identified in our report.

2. Foundations: examining the Cryptographic proof of Individual Verifiability

A description of the sVote v1 protocol is available online [Scy17b] (dated April 2017) and, in the same document, a computational proof of individual verifiability is offered, proof that is a formal requirement for a system certified for use by more than 30% of the cantonal electorate.

The protocol description in this document is the basis of the analysis performed in this mandate.

We show that the protocol description is imprecise in various places, and also there are important gaps in the computational proof. Furthermore, we point to several significant discrepancies between the protocol description [Scy17b] and its instance in the protocol specification [Scy19b, Scy19c].

Many of the issues we raise were already noted by prior reports [Bv17, Sec19]. For example, Basin and Čapkun note several "remaining issues," even for the questions that they describe as being mostly resolved. They recommend that "a single, unambiguous, description of the e-voting protocols should be created, with an explicit control flow," in order to ensure that that abstract model that was proven secure would match the specification that was implemented. We find the same sorts of issues still unresolved.

But before we enter this discussion, we would like to explain the structure and importance of a computational proof for a cryptographic protocol for the reader who may be

unfamiliar with this specific field of computer science.

2.1. The structure and importance of computational security proofs

Computational security proofs (or just "proofs" for the rest of this section) are a fundamental tool of modern cryptography. As Katz and Lindell put it in their standard textbook on cryptography [KL15]:

> Without a proof that no adversary with the specified resources can break some scheme, we are left only with our intuition that this is the case. Experience has shown that intuition in cryptography and computer security is disastrous. There are countless examples of unproven schemes that were broken, sometimes immediately and sometimes years after being developed.

The requirement set by the Swiss Federal Chancellery to offer such a proof is therefore in full compliance with the standard best practices in the field.

In the context of cryptographic protocols (such as sVote), proofs require the presence of three components [KL15, Chap. 1].

1. *Formal definitions* state which security guarantees are desired for the protocol. Such definitions clearly express what threats are in scope, and what threats are excluded. They help making sure that the person deploying a protocol shares with the protocol designer a common understanding of what the protocol is supposed to achieve.

2. *Precisely-stated assumptions* express what the components used in the system are supposed to achieve. sVote, just like most cryptographic protocols, cannot be shown to be secure unconditionally. It is therefore of the utmost importance to clearly understand which assumptions are needed for a protocol to be secure, and to validate these assumptions to a maximum extent (*e.g.*, by having them publicly reviewed by a large expert community).

3. A *proof of security* shows that, if the assumptions are satisfied, then the proposed cryptographic protocol must be secure in the sense of the formal definitions expressed before.

In the following subsection, we identify various gaps in the proof of individual verifiability, focusing on the last two aspects: assumptions and proof of security.

2.2. Gaps in the proof of individual verifiability

Our discussion follows the order of the proof document [Scy17b].

2.2.1. Assumptions

The protocol description starts by listing the cryptographic building blocks used in sVote [Scy17b, Sec. 2]. We point out various shortcomings observed in their description, following their order of presentation.

Symmetric key encryption scheme The description of AES-GCM is confusing, as it does not mention the existence of an Initialization Vector (IV), which is a key element of this encryption scheme, and a part of its standardized API [NIS07]. We assume that, for the purpose of this document, the IV is considered to be part of the ciphertext (the scheme would not be functional otherwise). In itself, this is a minor observation, except that it leads to some further confusion in the proof, as we will see below.

Signature scheme RSA-PSS is briefly presented, but the notation is also confusing. In particular:

- We assume that `ME` represents the PSS transform. If this is correct, why is it applied to $\mathsf{H}_s(m)$ and not to m?
- Or is H_s the actual hash function (or random oracle) intended to be used to hash the message as part of the PSS transform? But that hash function is supposed to have an output much shorter than n, which appears to be in contradiction with the requirement of $\mathbb{Z}_n$ as a range.
- And why would the verification function work as it is presented, since `ME` has a random padding? (The PSS verification function would not use `ME` for this.)

These aspects create confusion when reading step B.4 of the computational security proof (see below).

Non-interactive zero-knowledge proofs of knowledge We faced several difficulties with the description of these NIZKPK:

1. A NIZKPK is defined as a combination of four algorithms, the first one being called GenCRS, which is expected to produce a common reference string (CRS). The purpose of a CRS is unclear for the ZK protocols that are described in this document, which rely on a random oracle and, as pointed in a footnote, this algorithm is simply omitted from the description of all four of the NIZKPKs that are presented in the rest of this section. What is its purpose here, if there is any?
2. The fourth algorithm in the definition of a NIZKPK is $\mathsf{NIZKSimulate}$, which is expected to take a (false) statement as input and output a simulated proof. As it is, the mere existence of such an algorithm contradicts the soundness of the NIZKPK, which states that it is not possible to produce a valid proof for a false statement. The description should be more precise (*e.g.*, explain that the simulator has the possibility to control the random oracle used by the cheating verifier, and under which conditions), in order to be able to explain

why this is not an issue. Note that this oversight is directly related to the attacks against sVote that have been previously described [LPT19c].

3. The proof of correct decryption DecP is presented as being EqDl with different notations, and EqDl is actually used as a proof of correct decryption in the CreateVote function. Even if they are both non-interactive versions of the Chaum-Pedersen protocol, they are proof systems for different families of statements, which is reflected in the different inputs given to the random oracle in the Fiat-Shamir transform: for instance, the message m is part of ProveDec but does not exist in ProveEq. So, these actually are two different protocols. This will lead to difficulties in the security proof – see discussion below.

Pseudo-random functions The exponentiation function is presented here as a pseudo-random function (PRF). This is obviously not true since it is homomorphic (this property is noted later in the document). It is known that this function is a *weak* pseudo-random function under the Decisional Diffie-Hellman assumption [NPR99], that is, it would only behave as a PRF when the adversary sees queries on random inputs. The document then states that this PRF is "computed over the small primes representing the voting options". These are not random group elements, even though they may, in some conditions, complicate the exploitation of the homomorphic property of this function. Still, this use of the exponentiation function remains incompatible with the assumptions of the proof of Naor et al. [NPR99], and its security would not seem to be reducible to any of the 35 variants of the discrete logarithm problem reported in the ECRYPT report on "Main Computational Assumptions in Crytpography" [ECR13]. Besides, this restriction to small primes is not strictly satisfied, as this PRF is also applied, for instance, to the square of the Ballot Casting Key in the ballot confirmation step, which is unlikely to be prime (it is a random integer) [Scy17b, p.17], and could actually be the product of the small primes used to encode the candidates. This will be the source of some of the difficulties pointed in Section 4.

Verifiable mixnet The description of the Mix algorithm omits that this algorithm needs to have at least the public key that is used to encrypt the ciphertexts, in order to be able to perform re-encryption, possibly a secret key if partial decryption is performed, and its own public key (typically for commitments), or a common reference string. Ignoring this CRS and/or public key is precisely the source of a previously published protocol flaw in sVote [LPT19b]. It is claimed that the mixnet used in sVote is the scheme of Bayer and Groth, and that it has been proven to be sound. We observe that the Bayer and Groth proof [BG12] focuses on the interactive setting, and only mentions that a non-interactive version could be obtained through the Fiat-Shamir transform, without saying how this should be done. Even though the Fiat-Shamir transform may be relatively standard for 3 rounds Σ-protocols (with some caveats [LPT19c]), it is much less standard for protocols that have more than 3 rounds, and the Bayer-Groth protocol is 9 rounds.

Indicating how the Fiat-Shamir transformation is applied for such a 9 round protocol, and why the proposed transformation is sound, would be important. (Note that, even though this verifiable mixnet is present in the 50% version of sVote, it is actually not needed. Nevertheless, it does not seem advisable to keep potentially insecure components in the system.)

2.2.2. Theorem statement

The theorem statement is puzzling in many ways, and just seems wrong as it is stated:

1. Although the protocol described in the proof document is quite general, the actual security proof models only the case in which there is one question on the ballot. The spec, obviously, incorporates multiple different choices within one vote. This does not seem to be an immediate extension, as the ZK proofs would need to take care of products – we will return to this observation in 3.1.

2. It is required that $(\mathsf{ProveEq}, \mathsf{VerifyEq}, \mathsf{SimEq})$ is a sound NIZKPK scheme. However (and we will come back to this in the corresponding proof step), this cannot be sufficient, as the scheme faces an adaptive adversary. As previous attacks have shown [LPT19a], adaptivity has already been used to break individual verifiability of this system. So, it appears that this assumption in the theorem statement should be modified in order to make it possible to build a valid proof.

3. The (adaptive) soundness of the $(\mathsf{ProveEq}, \mathsf{VerifyEq}, \mathsf{SimEq})$ proof system relates to a set of statements, but this proof system is actually used for two different sets of statements: to prove exponentiations (π_{exp}), and to prove correct decryption (π_{pleq}). It should be explained why this single proof system would offer the right form of soundness property for those two sets of statements and, in particular, why the product $\prod_{\ell=1}^{t}(\mathsf{pCC}_{\ell}^{\mathtt{id}})$ can safely be omitted from the inputs of the Fiat-Shamir transform in π_{pleq}, even though it is included in the decryption proof [Scy17b, Sec. 2.4].

4. It is required that the symmetric encryption scheme can be modeled as a pseudorandom function. This is a very strong requirement, which is not satisfied by the AES-GCM encryption scheme used in the system. Indeed, AES-GCM requires the use of an IV in order to be secure – a fact that is not discussed in the document, as pointed above. So, either one assumes that the IV is picked by the encryption algorithm, but the encryption algorithm then becomes nondeterministic/probabilistic/stateful, which is incompatible with being a PRF, or one assumes that the IV is provided as an input to the encryption algorithm, but the requirement that this IV should be unique, which is crucial for the security of the GCM mode, is incompatible with the requirement that the encryption function would be a PRF, since a PRF can be challenged on any input, and not only on (partially) unique ones.

5. It is required that the size of the group $\mathbb{G}$ should be much larger than the number of voters. This is very vague. Is there any security parameter that governs the difference? (For instance, something like: q should be at least 2^{80} times bigger than the square of the number of voters?)

6. It is required that H is collision resistant. But H is actually modeled as a random oracle, as stated at the end of the statement, which would make it trivially collision resistant. Or is this a way to state that the range of H should be large enough?

7. It is required that the hash function underlying the signature scheme should be collision resistant. Why is the assumption based on that hash function and not on the signature scheme? Does the security of that hash function matter if the signature scheme itself is broken? (The security proof of the RSA-PSS signature scheme models the hash function as a random oracle, which is certainly collision resistant.)

8. The theorem claims that the attacker has a negligible advantage when trying to defeat the cast-as-intended verifiability property. However, there is no rigorous definition of that advantage at this stage: it only comes in the next section. Why using an asymptotic notion (negligible advantage)? Such a notion would need to define a security parameter in order to make sense, and there is none. Concrete security bounds, rather than asymptotic ones, would be much more expressive.

2.2.3. Proof

The proof of sVote follows the standard game hopping approach [Sho04]. That approach starts by expressing an attack game between an *adversary* who tries to break the security of the protocol, as expressed in the definition, and a *challenger*, who interacts with the adversary on behalf of the protocol's honest components. This game is then progressively modified, step by step, until it becomes obvious that it cannot be won any more with any advantage. The key element of the proof is then to show that any single change made between two game steps is indistinguishable for the adversary. This is where the assumptions of the theorem come into play: the proof would show that any adversary able to spot a difference between two successive games could be efficiently used to invalidate one of the assumptions made for the security proof. Eventually, the proof concludes by observing that, since the last game in the proposed sequence of steps is obviously lost by the adversary, and since any pair of consecutive games in the sequence cannot be distinguished by the adversary, then the initial game in which the adversary was trying to break the system must also be lost by the adversary. One key element of this approach is that no assumption is made about the adversary's attack strategy: the proof simply shows that no strategy could succeed to win the game proposed by the challenger with any significant advantage.

We observed several issues in the use of this proof methodology.

Game A.1 A vague assumption on the size of the group is used here. It should be possible to derive from here how big the group must be to guarantee the expected

negligible advantage, so that the reader can be convinced that there is indeed something negligible here.

Game A.2 Contrary to what is stated (and needed for the proof), it is quite easy to distinguish Game A.2 from Game A.1. In Game A.1, ProcessVote verifies all three ZK proofs: $\pi_{\texttt{sch}}, \pi_{\texttt{exp}}$ and $\pi_{\texttt{pleq}}$. However, in Game A.2, PerfectProcessVote verifies the Schnorr proof, but not the other two. This means that, if the attacker submits invalid $\pi_{\texttt{exp}}$ and $\pi_{\texttt{pleq}}$, the challenger will stop these proofs in A.1, but not in A.2. We suspect that adding the verification of these two proofs in the process would solve this issue.

Game A.2 It is claimed that the adversary cannot distinguish Games A.1 and A.2 because of the soundness of the ZK proof. However (and this is a general remark for all proof steps), no reduction is made to actually prove that this assumption is sufficient to make these games indistinguishable. The soundness property only offers guarantees when the proof statement is given as an input to the cheating prover (and to the verifier). However, in the current game, the prover is actually free to chose (large parts of) the statement in an adaptive way, which means that, at a minimum, some form of adaptive soundness would be needed. It is also unclear that this would be sufficient: extra properties of the NIZKPK, *e.g.*, simulation soundness, may be needed. Relying on (basic) soundness (which is actually undefined in the paper) is actually the source of the issues we identified in a previous version of the system [LPT19c].

Game A.3 It is a bit surprising to replace the PRF $f_{\mathsf{c}_{\mathsf{sk}}}$ with a random oracle here. It appears to just be making random choices inside the challenger instead of applying a PRF. Would it be more clear to talk about replacing f with a random function?

Game A.4 This part of the proof uses the problematic assumption (as discussed in Sec. 2.2.2, item 4) that the encryption function (AES-GCM) behaves like a PRF. However, this assumption does not seem to be needed: standard CPA indistinguishabilty seems sufficient. Furthermore, this game may actually be completely superfluous for a computational proof of the protocol that is actually implemented, since the adversary does not see these ciphertexts at all in the absence of a public bulletin board. (See also discussion in Sec. 2.3.)

Games B.1, B.2, B.3 Same observations as in Games A.1., A.3, A.4 (respectively).

Game B.4 This proof step is quite different from the others. Its goal (even though not clearly stated) seems to be to prove that a signature (which is an authentication primitive) leaks nothing about the signed message. Unlike the other proof steps, it focuses on low-level properties of RSA-PSS rather than on properties of generic cryptographic components (PRF, ZK proofs, ...). We faced several difficulties:

1. Is there any need for the notations $\mathcal{O}_h$, $\mathsf{H'}_s$ and $\mathcal{T}_h$? They seem to just all relate to the definition of a basic random oracle, which could be $\mathsf{H'}_s$.

2. How could it be that Game B.3 and B.4 are indistinguishable if, in B.3, the attacker uses an actual function H_s (as suggested in the conclusion of the B.4 game) while, in B.4, it must query an oracle instead? Distinction seems trivial based on the different game interfaces.
3. If, on the contrary, H_s is modeled as a random oracle, why would there be any ϵ_h gap here?

Game C We did not review the "C" series of game in detail, but the same general remarks that will be given at the end of this section apply to them as well.

Based on this analysis, we feel that the computational security proof offered in this document [Scy17b] leaves too many questions open and does not offer convincing evidence of the soundness of the protocol design with respect to individual verifiability.

A symbolic proof of security is also offered for this protocol [Scy17a]. Even though we did not review it, that analysis does not fill any of the gaps identified here. Indeed, and as stated by the author of the symbolic analysis report: "symbolic models of cryptographic protocols deal with abstractions thereof. As a result, they omit numerous cryptographic and mathematical properties of the underlying primitives [...]". Those properties are precisely those analyzed in a computational security proof.

We would make the following recommendations.

Recommendation 1 (complex; proof)**.** *The relevant properties of the cryptographic primitives on which the voting scheme relies should be properly defined. In particular, we would expect to see:*

- *The relevant properties of the NIZKPK should be provided and defined. The algorithms that make a NIZKPK should also be defined in a meaningful way.*
- *A clarification on the form of pseudorandomness that is expected from the symmetric encryption scheme, and of why it is sufficient.*
- *The PSS transform should be clarified, as its internal properties are used in the proof. The distinction between hash functions and random oracles should also be clear.*
- *The presentation of the exponentiation function as a PRF should be corrected.*
- *The 9-round Fiat-Shamir transform needed for the mix-net should be defined and its security documented. Or, the whole discussion about the verifiable mix-net should be removed, since it is not used.*

Recommendation 2 (complex; proof)**.** *The theorem statement should be clarified. The assumptions that are needed should be expressed precisely and, based on the definition of the security properties coming from the previous recommendations, concrete security bounds should be provided.*

Recommendation 3 (complex; proof)**.** *The security proof needs to be clarified in various places, and the identified gaps need to be fixed. In particular, actual reductions would be needed at least for the most important game hops, in order to detect if there are no other damaging gaps (such as the one related to the soundness of the proofs). Such reduction, written in sufficient detail, will also allow fixing presentation issues in the protocol description, where some functions receive inputs that they make no use of, while others use variables that they never received.*

The proof could probably be considerably simplified if the bulletin board, which is not given to the adversary in the real system, were taken away from the adversary in the security model used for the proof as well. (Of course, that model would not be acceptable in a universal verifiability context.)

2.3. Differences between the spec and the proof of individual verifiability

There are significant differences between the spec [Scy19c] and the proof of individual verifiability [Scy17b]. Some examples:

- In the proof, there is only one public key for both votes and partial choice codes: the proof of plaintext equality (p.13) is really a decryption proof, showing that $\tilde{c}$ encrypts the product of the partial choice codes (which are sent unencrypted). The voter sends simply $\{pCC_l\}_{l=1}^{t}$ (bottom p.13). In the spec, it sends them encrypted with a public key - the partial choice code public key pk_{CC} - that doesn't exist in the proof. Rather than sending just pCC's, it sends $E_2 = Encr(pCC; pk_{CC})$.

  This means that more corrections are necessary to fill in the gaps in the model of zero knowledge proofs described in Section 2.2.2. The proof document says that $\pi_{\texttt{pleq}}$—the proof of consistency for the partial choice codes—is an application of `ProveEq`, the proof of equality of discrete logs. However, the presence of a different public key means that the proof needs different inputs in order to be valid.

  The protocol specification also relies on a proof of plaintext equality [Scy19c, Sec. 5.2], which is not defined in the proof of individual verifiability document, even though it is definitely important for the verifiability of the protocol described in the specification document.

- In `ProcessConfirm`, the proof (p.15) doesn't mention using VC_{id} in the computation of VCC^{id} - it simply takes $f_{C_{sk}}(\texttt{CM}^{id})$. The spec, however, (Step 5.3(9)a, p. 49), says $lVCC = HMAC(CM|VCID|EEID, CodesSK)$. The crucial difference is the inclusion of the VCID and the election ID. Similarly, in the proof, the long choice codes are $f_{\texttt{CSK}}(pCC_l^{id})$ (p. 14, `createCC`), but in the spec it's $HMAC(pCC|VCID|EEID|\{correctnessID\}, CodesSK)$. The latter seems better, but the proof should be updated to match the current version of the protocol.

- The proof has an explict function `VerifyTally`, which includes verifying the shuffle proof. The spec does not include details on either generating or verifying a shuffle proof.

- The proof has a bulletin board, accessible by voters and everyone else, which the voters can use in some circumstances to verify transactions, though this is not part of the standard workflow. The spec allows voters only to query data via their client. Voting data is stored on the Election Information Context, which the voter can query (in 5.2.1 to see her Choice Codes and in 5.3.1 to retrieve her Vote Cast Code) again. An extra layer of encryption is also used here, which is not part of the security proof.

- Basin and Čapkun [Bv17] write, "write-in votes have been eliminated from the system design, and hence need no longer be part of the model." Write-in votes have now reappeared in the system design, though they are still not part of the model. Although write-in votes are not covered by the individual verifiability requirement, there needs to be careful analysis to show that the existence of write-ins does not undermine individual verifiability for those voters who do not choose a write-in candidate.

The absence of a bulletin board fundamentally alters the assumptions for individual verifiability. And the absence of mixing verification completely alters the opportunities for server-side misbehaviour.

Similarly, to prove verifiability for only single-question elections excludes a whole class of potential attacks relating to manipulating different parts of the ballot. It is clear that the current version of the spec includes several measures for preventing attacks of this style, such as the inclusion of the voter's ID into the hash ([Scy19c] 5.2 10(c)). These measures are not modelled in the proof, and hence not proven secure.

Overall, we did not find any new ways that cast-as-intended verifiability fails for the protocol described in the proof document [Scy17b], but there is no convincing proof of cast-as-intended verifiability that matches the current version of the protocol.

Recommendation 4 (complex; proof)**.** *Update the proof of individual verifiability to match the current protocol.*

3. Corrections to the specification—within the trust model

In this section we describe two different attacks on cast-as-intended verifiability that are possible based on a reasonable interpretation of the spec and correspond to aspects that are not discussed in the security proof document: the first is related to the case of voters picking multiple candidates (the proof considers only a single candidate), and the second is related to input validation in the ZK proofs (which is not discussed in the proof document). In both cases, the code is not vulnerable because it contains extra

c) For each partial choice code pCC:

- Compute a (long) Choice Code (lCC) using the partial Choice Code concatenated with the Verification Card ID, the Election Event ID, the corresponding voting option attributes sent together with the partial choice code and the Codes Secret Key as:

$$lCC = HMAC(\text{"pCC|VCID|EEID|correctnessID"}, CodesSK)$$

- Compute the hash of the long Choice Code: $Hash(lCC)$ and a symmetric key $EKey = KDF(lCC, keylength)$.
- Retrieve the encrypted short choice code from the mapping table using $Hash(lCC)$ and decrypt it using $EKey$. The result is the short choice code to be sent to the voter. If it is not possible to find an entry in the mapping table an error is logged and the **Client Context** is informed.

d) Check that all the retrieved short choice codes are different.

11) If short choice codes are correctly retrieved, the **Election Information Context** generates the receipt and stores the vote:

Figure 2: Iteration over choice code retrievals in the spec. It is unclear whether a single failed attempt at code retrieval halts execution.

protections that are not explicit in, or not the only interpretation of, the spec. In each case, this implies that the spec needs to be updated to include the protections that are already present in the code.

In both cases, a voter can receive the vote confirmation code that she expects although the cheating client has substituted a different choice.

So these attacks are consistent with the threat model and the definition of individual verifiability defined by the Federal Chancellery, and are possible based on some interpretation of the spec, but do not actually work on the current version of the code, because it implements a different interpretation.

3.1. Returning choice codes only if all retrievals have succeeded

The spec [Scy19c], Section 5.2(10)c, describes the process in which the Vote Verification Context computes the choice codes that are returned to the voter—this is shown in Figure 2. It iterates through the partial Choice Codes it has received, looking them up in the Codes Table and returning the corresponding short Choice Code. The question is what should happen if a lookup fails. The spec says, "If some validation is not successful, an error is logged, and the Client Context is informed." It does not say that the loop breaks. Step (11) then says, "If short choice codes are correctly retrieved, the Election Information Context generates the receipt and stores the vote." It does not say, "If *all* short choice codes are correctly retrieved..." A literal interpretation would take this to

```
for (byte[] longCode : longCodes) {
    byte[] hashedLongCodeBytes = primitivesService.getHash(longCode);
    String hashedLongCode = new String(Base64.encodeBase64(hashedLongCodeBytes), StandardCharsets.UTF_8);
    JsonString encryptedShortCodeJson = mappingForCard.getJsonString(hashedLongCode);
    if (encryptedShortCodeJson == null) {
        String errorMessage =
            String
                .format(
                    s: "Encrypted short code not found for tenant: %s, election event id: %s and verification card id: %s",
                    tenantId, eeid, verificationCardId);
        throw new ResourceNotFoundException(errorMessage);
    }
    String encryptedShortCode = encryptedShortCodeJson.getString();
    byte[] encryptedShortCodeBytes = Base64.decodeBase64(encryptedShortCode);

    CryptoAPIDerivedKey derivedKey = kdfDeriver.deriveKey(longCode, longCode.length);
    SecretKey mappingKey = symmetricService.getSecretKeyForEncryptionFromDerivedKey(derivedKey);
    byte[] shortCodeBytes = symmetricService.decrypt(mappingKey, encryptedShortCodeBytes);
    String shortCodeString = new String(shortCodeBytes);
    if (shortCodes.contains(shortCodeString)) {
        throw new CryptographicOperationException("Duplicated short choice code retrieved");
    }
    shortCodes.add(shortCodeString);
}

return shortCodes;
```

Figure 3: Iteration over choice code retrievals in the code. It is clear that a single failed attempt at code retrieval throws an exception.

mean that the subsequent actions are taken if any short choice codes are successfully retrieved. The next two steps (12 & 13) specify that the vote's status is updated and the choice codes are returned to the voter, without conditioning on whether all the choice codes are correctly retrieved.

This leads us to a simple recommendation for a spec correction:

Recommendation 5 (simple; spec)**.** *Update Section 5.2 so that any retrieval failure should stop execution, none of steps 11-13 should be performed, and none of the choice codes should be returned to the voter.*

However, when we look at the code, it throws an exception as soon as a validation is not successful—this is illustrated in Figure 3. So none of the subsequent steps in 5.2 are performed, and in particular no short choice codes are returned to the voter unless *all* short choice codes are correctly retrieved. This interpretation is confirmed by answers from Swiss Post. Hence this does not lead to any real attack that we could identify. Nevertheless it provides an interesting study in the importance of precise specifications.

In the rest of this section we examine what might happen if an implementer mistakenly interpreted the spec to allow the return of some short choice codes, even if later retrieval attempts for the same voter failed. We explain what might go wrong, and how this relates to the formal proof, the precise specification, and the actual code.

We are interested in the setting in which there are multiple questions on the ballot. We examine exactly what happens when a dishonest client sends valid partial choice codes for some questions but not others. We assume

1. That the server-side returns the short choice codes that can be retrieved, even if other partial choice codes are invalid.

2. That if the voter receives correct short choice codes for some questions but not others, she infers that the ones for which she received valid ones were correctly sent.

The first assumption is false for the current implementation in the code—this is clear from both answers from SwissPost and reading the source code. We do not know whether the second assumption is valid or not. The following section describes what would happen if these assumptions were true, in order to demonstrate the importance of clarifying this point in the spec.

We believe that this attack (if it worked on the code) would be within the scope of the Federal Chancellery requirements of individual verifiability, because, "Proof of correct registration must be provided for each partial vote." ([Cha18b], Article 4, Paragraph 2.) In this case, the voter receives her expected return code for the question on which the answer has been substituted. However, she can observe that there is a problem with her other codes if she checks carefully.

3.1.1. The setting

Suppose that there are two questions 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.[1]

The cheating client can substitute its preferred vote for Question 1 while retrieving the voter's expected return code. The vote for Question 2 remains valid but no valid choice code is retrieved. The attacker transfers the code substitution to Question 2 and hopes the voter doesn't notice.

3.1.2. Substituting a choice but generating the correct return code

The main observation in this section is that the exponentiation proof 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.

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.

Vote construction is described in the spec [Scy19c], Section 5.2. The cheating client generates a random r and computes the vote as

$$E_1 = (g^r, pk_{EB_1}^r p_{no} p_3)$$

[1] 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.

Note that 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 $pCC_1 = p_{yes}^{vcsk}$ and $pCC_2 = (p_{no}p_3/p_{yes})^{vcsk}$. 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 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. Let $\tilde{E}_2$ be the encryption of the product of the partial choice return codes, with the public key corresponding to the product of the choice code public keys. That is,

$$\tilde{E}_2 = (g^{r'}, (pk_{CC_1}pk_{CC_2})^{r'}pCC_1pCC_2)$$

Let $E_{exp} = (g^{r.vcsk}, pk_{EB_1}^{r.vcsk}(p_{no}p_3)^{vcsk})$, the result of exponentiating each element of E_1 by $vcsk$.

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

The exponentiation proof proves that E_{exp} is generated by exponentiating each element of E_1 to the private exponent corresponding to the public key g^{vcsk}, which it is.

The proof of plaintext equality proves that E_{exp} encrypts the same value, under pk_{EB_1}, 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

$$pCC_1pCC_2 = p_{yes}^{vcsk}(p_{no}p_3/p_{yes})^{vcsk} = (p_{no}p_3)^{vcsk}$$

which is exactly the plaintext of E_{exp}.

In summary, based on a literal reading of the current version of the spec, it seems that it will be noted that an error occurred, but the choice codes that were correctly retrieved will be sent back to the voter. However, in the current implementation no codes would be returned, and hence a threat that seems exploitable from a literal interpretation of the spec actually does not work on the software.

3.1.3. How does the proof of individual verifiability deal with this case?

We read the cryptographic proof of individual verifiability [Scy17b] carefully to understand whether the zero knowledge proof of exponentiation is properly modelled as proving only that the *product* of partial choice codes matches the exponentiated compressed vote, or incorrectly modelled as proving that each exponent is separately properly exponentiated. The answer is that it models neither, because the proof deals only with the case of a voter submitting a single choice in a one-question election.

The protocol description in [Scy17b] does make it clear that short choice codes should be returned only if *all* of them are successfully retrieved. However, this rule is never used because the security proof game involves only one vote. It is possible that if this rule had been used in the proof, then its importance would have been better understood and it would have been more carefully described in the spec.

We did not thoroughly examine the symbolic proof [Scy17a]. However, we note that in Sec 2, p.4, it explicitly says that it does not attempt to model this issue:

> Roughly speaking, $\pi_{\texttt{sch}}$ proves knowledge of nonce r, while $\pi_{\texttt{exp}}$ and $\pi_{\texttt{pleq}}$ prove that the partial choice codes $pCC_1, \ldots, pCC_k$ are linked to the ciphertext $\texttt{Enc}(\texttt{EB}_{pk}, \phi(v_1, \ldots, v_k), r)$, that encrypts the voting options, and the voters' verification card secret key $\texttt{VC}^{id}_{sk}$. The most important limitation of this modelling of the zero-knowledge proofs is that the compacting function $\phi(\cdot)$ enjoys commutative properties that cannot be captured in Proverif as of yet.

Hence the reasons that this cheating attempt does not succeed have nothing to do with the assurances given in either proof.

3.1.4. How to address the problem

It is clear that the spec should be updated to ensure that choice codes are returned only if *all* of them have been successfully retrieved. However, this doesn't really solve the fundamental problem, which is that the exponentiation proof is used informally as a proof that all elements are equal, but actually proves only that the products are equal. There may be other attacks that exploit this, even if this particular one is disallowed.

There is an important theoretical question of whether it is sufficient to provide a zero knowledge proof that the product of pCCs is the proper exponentiation of the product of vote choices, when what we need to use is that the sets of components are equal. It is obviously not true in general that $ab = cd \implies \{a, b\} = \{c, d\}$. There should be a lemma saying roughly that, in this situation, the fact that $E_1, \tilde{E}_2$ and E_{exp} satisfy the relations defined by the exponentiation and plaintext equality ZKPs implies that $\{p_1^{vcsk}, \ldots, p_k^{vcsk}\} = \{pCC_1, \ldots, pCC_k\}$.

Recommendation 6 (complex; proof)**.** *Prove that the facts proven in the ZKPs imply that the partial choice codes correspond to the votes that will be retrieved at decryption time. There are two ways that proving this could be approached:*

- *Examine the mathematical group structure and argue that the equality of the products implies the equality of the elements in this case. We are not sure whether this is true in this setting—it needs careful thought.*
- *Alternatively, remove the optimization in which votes are multiplied together, and instead use the multi-element versions of both the election and the Choice Codes public keys. Now the exponentiation proof would apply directly to the tuple of values, one for each choice, and similarly the proof of plaintext equality would be applied in the multiple-element version.*

Either way, the security proof should be updated, either to make it clear that a proof of matching products is sufficient, or to use instead the proof of elementwise matches.

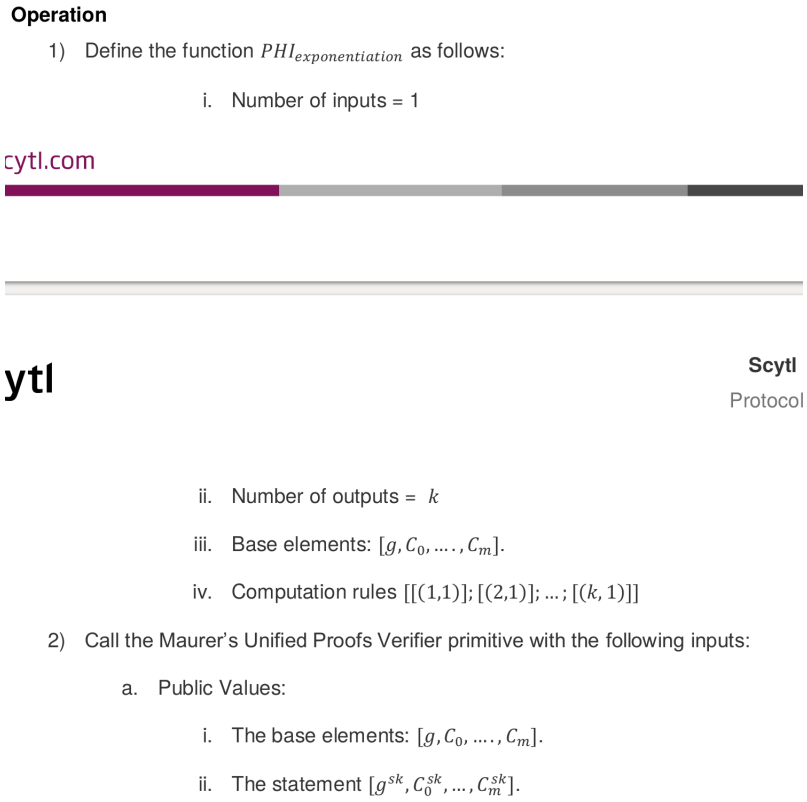

Figure 4: Operation of the Exponentiation proof verifier, in the spec. k is not defined but should be $m + 2$, which should be validated on input.

3.2. Correcting the specifications of the zero knowledge proofs

One of the main purposes of our mandate was to examine the zero knowledge proofs in the specification and the code. These have recently been updated following our discovery (with Sarah Jamie Lewis [LPT19a]) that weaknesses in their earlier version could lead to cast-as-intended verification failures. The newest version of the specification [Scy19c] has significant changes to address these problems. However, the current version is quite rushed. There are some typos and imprecisions, described below, that make it difficult to assess whether the corresponding verification code is secure.

The spec tends to assume that certain input elements have a certain form (such as a certain length), though it does not always make these checks explicitly. In this section we identify some of the omissions in the spec and explain why, if they were also omitted from the code, this would allow an attack against cast-as-intended verifiability.

Again, this attack does not actually work on the current version of the code, because the code contains input validation checks that are not explicit in the spec.

Here are some examples of corrections to be made to the spec:

3.2.1. Checking the lengths of input values: exponentiation proof

The Exponentiation proof verifier (7.2.4) contains a variable k which is used to generate the computation rules but isn't defined. An answer from SwissPost suggests this should be initialised as $k = m + 2$ and we agree. The code seems to initialize k to match the length of the input base elements (which seems correct).

However, nowhere in the spec does it check that the length of the statement (the exponentiated elements) is the same—it simply assumes, at the beginning of the Exponentiation proof verifier, that its two input lists are the same length – see Figure 4.

So, a literal reading of the spec (without adding explicit input validation) might allow an implementation that accepts a list of exponentiated elements that is longer than the list of base elements—this would pass verification (according to the spec) if the first k elements were correctly exponentiated, where k is length of the list of base elements.

Furthermore, the specification (see Sections 5.2.1.8-9 and 6.2.3 of [Scy19c] in particular), which does require various length validation steps, does not require any such step for the ZK proofs. So, it does not appear that such length differences would need to be verified outside the proof verifier.

Still, the code does check that its input lists are the same length, using an input-validation function called `validateExponentiationProofVerifierInput`.

3.2.2. Checking the lengths of input values: proof of plaintext equality

The proof of plaintext equality is still written in the spec (Sections 7.2.5 and 7.2.6) in the multi-element version, in order to prove equality of two $k + 1$-element ciphertexts for any $k \geq 1$. In Step 1 it takes the last k elements of the primary ciphertext and the last k elements of the secondary ciphertext and proves they encrypt the same values, relative to the base elements that are prepended. In particular, Section 7.2.5 reads:

> 2) Consider the last k elements of the Primary Ciphertext and the last k elements of the Secondary Ciphertext. Call them the Primary SubCiphertext $[C_1, \ldots, C_k]$ and Secondary SubCiphertext $[D_1, \ldots, D_k]$ respectively.

This seems correct *if the input ciphertexts are both of length* $k+1$. However, although it assumes that the primary and secondary ciphertexts have the same length, it never explicitly checks that this is so.

So, with a literal reading of the spec (without adding explicit input validation) it seems possible to append another ciphertext element (or several elements) to the end of either of the ciphertexts, and have only the last k elements used in the proof and the verification.

The code actually runs an input validation function, which does check that the lengths are as expected: `validatePlaintextEqualityProofVerifierInput`. It always sets $k = 1$, corresponding to the traditional setting in which an ElGamal ciphertext is a pair.

3.2.3. Using extra-long ciphertexts to submit inconsistent vote and partial choice codes

Those apparently benign issues could actually be quite problematic: we now show an attack that would be successful if the code followed the spec more closely and did not perform any length-checking on the ZK proof inputs.

The purpose of this section is to explain why explicit input validation should be included in the spec.

The idea is based on three observations, introduced above:

1. When the Election Information Context validates the received vote [Scy19c, Section 5.2(8)], it checks the number of elements in the vote ciphertext (Step (h)) and the number of encrypted partial choice codes (Step (i)). It does not check the length of the ciphertext exponentiations that are used to prove that the vote and the choice codes are consistent. ('Ciphertext exponentiations' is the term used in the spec; we use E_{exp} in this report.)

2. The exponentiation proof verifier (7.2.4 in [Scy19c]) assumes a number of outputs, k, that is not defined anywhere and, in particular, is not constrained to be equal to the length of either the base ciphertext or the exponentiated ciphertext. It assumes that the base and exponentiated ciphertexts are of equal length, but never checks that they are. It seems possible to add extra elements to the end of the properly-exponentiated ones.

3. The plaintext equality proof verifier (7.2.6 in [Scy19c]), in Operation Step 2, takes the *last* k elements of each ciphertext, without explicitly checking whether the two input ciphertexts have length $k+1$—this is simply assumed to be true of the input.

There are also vote validations described in Section 6.2.3 of the spec, though these seem to duplicate validations that are performed by the Election Information Context upon receipt of the vote.

Submitting a different vote with the expected partial choice codes

The idea of the attack is simple: we will make E_{exp} contain three elements, not two, and we will use the first two in the exponentiation proof and the first and last in the proof of plaintext equality.

Assume a simple one-question election with a yes/no answer. Suppose again that the voter wants to vote p_{yes} but the cheating client prefers p_{no}. The cheating client generates a random r and computes the vote as

$$E_1 = (g^r, pk_{EB_1}^r p_{no}).$$

It then generates the partial choice code $pCC_1 = p_{yes}^{vcsk}$. Note that the partial choice code reflects the voter's choice, not the actual vote.

Recall that $\tilde{E}_2 = (g^{r'}, pk_{CC_1}^{r'} pCC_1)$ is the encryption of the product of partial choice codes, with the choice codes public key. (In this example we have only one code and hence

only one key, but it works just as well for multiple voting options.) The exponentiated ciphertext E_{exp} is supposed to be $(g^{r.vcsk}, pk_{EB_1}^{r.vcsk} p_{no}^{vcsk})$, the result of exponentiating each element of E_1 by $vcsk$.

The cheating client instead creates a three-element tuple

$$E_{exp} = (g^{r.vcsk}, pk_{EB_1}^{r.vcsk} p_{no}^{vcsk}, pk_{EB_1}^{r.vcsk} p_{yes}).$$

The client will use the first 2 elements in the exponentiation proof, with g^{vcsk} prepended by the server side, to prove (truthfully) that they are a correct exponentiation of (g, E_1), assuming that the verifier does not notice the extra value that is appended. It will then use the first and last elements as a ciphertext in the plaintext equality proof to prove (truthfully) that it contains the same plaintext as $\tilde{E}_2$.

What would happen in the exponentiation proof

The inputs to the exponentiation proof verifier (based on [Scy19c] 5.2(9)a) will now be:

- Base elements: $[g, g^r, pk_{EB_1}^r p_{no}]$
- Exponentiated elements: $[(g^{vcsk}, g^{r.vcsk}, pk_{EB_1}^{r.vcsk} p_{no}^{vcsk}, pk_{EB_1}^{r.vcsk} p_{yes}^{vcsk})]$
- Exponentiation proof

In the exponentiation proof verifier (7.2.4 in [Scy19c]), as described in Section 3.2.1, if no input validation is performed then the proof is performed on the first k elements—this is how the proof rules are defined. If we assume k is defined as the length of the base elements (which it is in the code), then the exponentiated elements referred to in the proof rule are only the first k, in this case the first three. Hence it amounts to proving that $(g^{vcsk}, g^{r.vcsk}, pk_{EB_1}^{r.vcsk} p_{no}^{vcsk})$ is a proper exponentiation of $[g, g^r, pk_{EB_1}^r p_{no}]$, which is true.

What would happen in the proof of plaintext equality The exponentiated elements in the exponentiation proof are the same as the primary ciphertext in the proof of plaintext inequality. The inputs to the plaintext equality proof verifier (based on [Scy19c] 5.2(9)b) will now be:

- Primary ciphertext: $[g^{r.vcsk}, pk_{EB_1}^{r.vcsk} p_{no}^{vcsk}, pk_{EB_1}^{r.vcsk} p_{yes}^{vcsk}]$
- Primary public key: pk_{EB_1}
- Secondary ciphertext: $\tilde{E}_2 = [g^{r'}, pk_{CC_1}^{r'} pCC_1]$
- Secondary public key: pk_{CC_1}
- Plaintext equality proof

In the plaintext equality proof verifier (7.2.6 in [Scy19c]), as described in Section 3.2.2, it takes the last k elements from each input ciphertext, without checking that they are the same. We will assume that $k = 1$, which is the way the code initialises it.

Step 2 of its operation takes the last k elements (in our example, one element) from each ciphertext. Call them the Primary SubCiphertext $[C_1]$ and the Secondary SubCiphertext $[D_1]$. All subsequent operations involve only C_0, D_0 (corresponding to $g^{r.vcsk}$ and $g^{r'}$, the first elements of each ciphertext) and the last k that have just been selected. It seems that if either ciphertext's length is longer than $k + 1$, its other values (in particular, its second value $pk_{EB_1}^{r.vcsk} p_{no}^{vcsk}$ in this example) are not used.

The statement input to Maurer's Unified Proofs Verifier in step 5 (a)*ii* of 7.2.6 is

$$\begin{aligned} [C_0, D_0, C_1(D_1)^{-1}] &= [g^{r.vcsk}, g^{r'}, pk_{EB_1}^{r.vcsk} p_{yes}^{vcsk} / (pk_{CC_1}^{r'} pCC_1)] \\ &= [g^{r.vcsk}, g^{r'}, pk_{EB_1}^{r.vcsk} / pk_{CC_1}^{r'}] \text{ (because } p_{yes}^{vcsk} = pCC_1) \end{aligned}$$

which will be a correctly-verifying input given the plaintext equality proof rules.

In summary, if a cheating client can submit extra-long ciphertext exponentiations, then it can retrieve the choice codes for "yes", but submit a vote for "no."

3.2.4. Recommendations

The input validation in the code seems to be much more thorough than that of the spec. The part of the spec dealing with the input and validation of the vote should be written more carefully to make that input validation explicit.

Part of the problem is that there are often not explicit names given to the values in the vote data structure. The spec tends to describe the elements in the form they are supposed to be input, rather than with names that derive from the form in which they were received. This makes it hard to write careful checks on what they contain. For example, in the Send a Vote section (5.2), the ciphertext we have referred to as E_{exp} is referred to as "ciphertext exponentiations" in (6j). We know that this is the same value referred to as the 'Exponentiated elements' when it is input into the Exponentiation proof (4b) and as the 'primary ciphertext' in the proof of plaintext equality (4c), but consistent names are not used.

Similarly, the spec tends to describe the elements provided as input to the various verification steps in the form they are supposed to be input, rather than with names that derive from the form in which they were received. For example, the exponentiated elements input to the Exponentiation proof verifier in 5.2(9) are denoted as $[g^{vcsk}, g^{r.vcsk}, (pk_{EB_1}^{r}.\Pi_{i=1}^{k} p_i)^{vcsk})]$, but of course they have this complex form only if they are truthfully constructed. It would be much less error-prone to give the values in the submitted vote names, and then use those names when calling the proof verifiers. Then the input validation step that precedes proof verification in the code could be made explicit in the spec.

Recommendation 7 (medium difficulty; spec)**.** *Make the input validations that are performed in the code before proof verification explicit in the spec. This will mean giving names to the values received from voters, and using those names as inputs to the proof*

verifiers (and their input validations) rather than using expressions that assume the form that they are checking for.

Again, we find that a literal interpretation of the spec can result in a violation of cast-as-intended verifiability within the security model. In this case, the spec is simply not clear—it assumes that the input has a certain form (with arrays of a particular length) but does not check. If they were simply assumed but not checked in the code, then this would allow a cheating client to retrieve the right choice codes after submitting the wrong vote.

3.3. Summary of attacks on the spec, but not the code, within the trust model

We have found two different examples in which the spec could be interpreted to allow an attack on individual verifiability by a misbehaving client. In both cases, the current implementation in the code prevents the attack, so the actual system is not vulnerable. However, in both cases a literal interpretation of the spec allows an implementation that would be vulnerable. This might be particularly dangerous if there is a plan, at some point, to build a second implementation of the voting system based on the spec.

Given that our primary mandate was to examine the spec (one of us did not even have the code), the fact that divergences between the code and the spec make all the difference between exploitable and un-exploitable vulnerabilities in cast-as-intended verification should not be interpreted as comforting, even though in every example we found, the code is actually better than the spec. Although in this case the small details mean that attacks on the spec do not succeed on the code, it is just as likely that other unnoticed small differences have the opposite effect. Since we looked at the code only when the spec seemed to contain a problem, we would not have noticed things that seem to be correct in the spec but were not implemented correctly.

The current version of the spec is very new and contains some typos and imprecisions. There are significant changes based on our previous discoveries, and these changes need to be described more carefully in the spec before we can be confident they are correct.

4. Examining sVote's trusted server side—outside the trust model

In this section we examine attacks that fall outside the trust model because they involve misbehaviour by server-side entities. They are based on reading the spec—as far as we know, they would succeed on the code, though we have not tested them in practice.

In sVote v1, there is a tremendous degree of complexity on the server side, which does not seem appropriately matched to the assumption that the "server side is trusted" (proof, p.2). We guess that the server-side complexity is intended to mitigate at least some server-side misbehaviour, by trying to make sure that it is difficult for a single server-side component to breach privacy or, in collusion with corrupted clients, to commit

undetectable electoral fraud. However, it is difficult to understand what those intended security properties were.

It is important to emphasize that server-side logging, even available to diligent and honest administrators and auditors, does not necessarily expose all server-side attacks. Server-side logging might offer evidence for some of the attacks that we are describing here, but not all. Even for the detectable attacks, it would require well-developed log inspection systems, which are not described here.

The formal proofs, and the legal requirements, both allow the server side to be entirely trusted. Although this means that server entities are not required to produce any proof of correct behaviour, much of the documentation hints that the system defends against some server-side misbehaviour. For example, p.18 of the (very recent) "Security analysis of key cryptographic elements for individual verifiability (v 1.1)" [Scy19d] contains the following remark after a discussion of the importance of secure key generation:

> This would imply that the attacker controls the voting server, which goes beyond the trust assumptions for voting systems up to 50% of the electorate. However, it is a best practice that these attacks are not possible even if an attack would go beyond the adversary's capabilities.

It is not clear whether only this particular attack, or all attacks by an adversary who controls the voting server, or all attacks by an adversary who controls any of the servers, are asserted to be impossible.

There are many ways that the assumption "the server side is trusted" could have been put in to practice. There are crucial differences between different models. For example in the Norwegian Internet voting system [Gjø11], the two main server-side components are trusted not to collude to cheat. In Pretty Good Democracy [RT09], the assumption is that no more than a certain threshold of the server-side entities that share the key collude. Other systems, such as Helios [ADMP+09], trust that not all the server-side entities that share the decryption key collude to break privacy, and guarantee integrity even if all the server side components are corrupted.

In sVote v1, this assumption has been realised in the weakest possible way: there are situations in which a single component, acting alone or in collusion with a cheating client, can undetectably affect the election outcome. Although this is not strictly inconsistent with the precise security model, it is certainly different from the impression given in some parts of the documentation. For example, at the bottom of p.18 the "Proof of Individual Verifiability," [Scy17b] says,

> We consider that the election authorities and the registrars behave properly ... In order to enforce this property, both election authorities and registrars can be distributed among a set of trustees which compute the required information using multiparty computation algorithms.

This gives the impression that it is easy to extend the protocol to one in which the trusted-server-side assumption is to trust a threshold, rather than trusting that no single entity on the server side cheats. However, multiparty computation algorithms can

be extremely computationally intensive, particularly when they do not follow simple arithmetic or logical circuits. (See [ABF+18] for a good description of what multiparty computation solutions work best in which circumstances.) We do not agree that this would be feasible for all relevant steps. Paper printing, for example, cannot practically be distributed (though there have been some clever efforts to try [ECHA09]). The fact that some parts of the server-side functionality come with proofs of their correctness (such as decryption proofs and a mixnet) merely adds to the confusion: these entities are entirely trusted—proving some parts of some entities' computations gives a false impression of overall protocol verifiability.

We give some important examples of processes that allow cheating by a single misbehaving server-side entity, possibly in collusion with a cheating client.

4.1. The lack of verifiable mixing

The most significant gap in the specification is the absence of any specification of a verifiable mixing process, or a way of verifying its proofs. Section 6.3 of the most recent spec [Scy19c] instructs the mix server to "Generate the cryptographic proofs to demonstrate that the shuffled and re-encrypted votes in the Mixnet Ballot Box are the same that those in the Cleansed Ballot Box." Then at the end of the Mixing process it is expected to "Output a list of Signed Mixed ballot boxes and a list of sets of Proofs of the mixing process," (6.3, p.59), but there is no shuffle proof generation or verification mentioned in the document. We were advised by SwissPost that the shuffle proof is generated but not verified.

The presence in the document of a verification step that is actually not specified and not used gives an incorrect impression of the trust model. In fact, the mixing process can easily substitute votes. If it is implemented as a series of mix servers, any one of them can substitute votes. If it is implemented as a single trusted server, then this server alone can link a voter's identity to her decrypted vote.

Given the possibility of vote substitution in the mixing step, there is no logical gain from proving decryption (as described in the spec Section 6.4.1). There is also no gain in having the decryption key distributed between multiple entities in order to protect privacy: a cheating mixer could make a valid shuffle of the encrypted votes as they are submitted (hence, while they are not anonymous), wait until decryption is performed, and use the knowledge of the permutation applied in the mixing to determine who voted for whom.

Again, it seems only to give a naive reader the impression that certain kinds of server-side attacks are defended against, when they are not. We think it would be better to omit these proofs too.

Recommendation 8 (Complex or simple; spec)**.** *Either include shuffle verification in the spec and make the shuffle proofs available for independent verification, or remove references to verifiable shuffle proofs from the spec. There does not seem to be any gain from generating proofs that are not verified. Consider removing decryption proofs also, for the same reason.*

Even if a verifiable mixnet is incorporated in a later version of this software, there remain opportunities for a single malicious server-side entity to read and manipulate votes. We describe a few examples below. The attacks on integrity require collusion with a cheating client; the privacy attacks do not.

4.2. HMAC-based code recovery

The sVote system uses a vote-confirmation process to guard against the inclusion of votes that have not returned the correct choice codes to the voters. When a voter receives the correct choice codes, she is supposed to enter a Ballot Casting Key (BCK), which is then processed at the server-side to confirm the vote and issue her with her (short) Vote Cast Code. In this section we examine whether it is possible for an attacker to forge a confirmation that the voter did not want, or to undetectably block a confirmation that the voter did want. Since only confirmed votes are counted, this may have the effect of either counting a vote the voter rejected, or not counting a vote she wanted to confirm.

We did not find any way that a client alone can cheat in the confirmation process. However, the Vote Verification Context, which holds the Codes secret key (C_{SK} in the proof, and CSK in the spec) can cheat, as described below. This is not inconsisent with the complete trust in the server side. However, it is important because the code-recovery process uses both a cryptographic hash and a symmetric encryption function - although it would be possible in theory to distribute these functions using MPC and a secret-shared key, in practice it would be prohibitively computationally expensive. Neither mixing nor the decryption proofs would make any difference, because they work only on votes that have been confirmed. Hence this function is performed by a single entity, in a fashion that is not available to scrutiny by anyone who does not know C_{SK}.

We show two attacks in which a single corrupt server-side component colludes with a cheating client. In the first, the Vote Verification Context in collusion with a cheating client can silently fail to confirm votes despite returning the correct Vote Cast Code (sVCC) to the voter. In the second, they can brute-force the confirmation message (without learning the true BCK from the voter) and fabricate a confirmation that the voter did not want.

4.2.1. Dropping a confirmation while returning the correct Vote Cast Code

The Vote Verification Context is supposed to input a confirmation message CM^{id} and use it to generate a long Vote Cast Code ($lVCC$) as[2]

$$lVCC^{id} = HMAC(pCC|VCID|EEID|\{attributes\}, CSK).$$

It then looks in the Codes Mapping Table to find $H(lVCC^{id})$. If it finds it, it uses it to decrypt the corresponding signed short vote code ($sVCC$) which is returned to the voter.

[2]The proof and the spec differ slightly here—the spec incorporates VCID and EEID. This does not affect the attack described in this section.

The attack is simply to send $sVCC$ back to the client via some other channel, without changing the status of the vote to CAST (as instructed in step 10, p.51 of the spec). Thus the voter receives the $sVCC$, but the vote is not counted. The colluding client makes it appear as if the $sVCC$ came through the official channel.

This attack is detectable by a diligent voter who checks whether her client has received the signed vote and receipt that is supposed to be sent in Step 11 (it will not be) or who logs back in with a non-colluding client (as described in Section 5.3.1) to check whether the Election Information Context has confirmed her vote (it will not have). However, a "normal" voter who trusts her codes alone will be deceived.

4.2.2. Brute-forcing a confirmation that the voter did not want

In this section we show that, if there is collusion between the client and the cheating Vote Verification Context, they can undetectably either include votes that do not reflect the voter's intention, or exclude votes that the voter wanted to confirm.

Consider what information is required to decrypt the Codes Mapping Table and retrieve the $sVCC$. It seems to be intended that three pieces of information are required (apart from the Voting Card ID VC_{id}): the Codes Secret Key (C_{sk}, or $CodesSK$ in the spec), which is held by the Vote Verification Context, the Verification Card Private Key (VC_{sk}^{id}, or VCSK in the spec), which is held by the client, and the voter's Ballot Casting Key BCK, known only to the voter. However, BCK is only 8 decimal digits, and the Codes Mapping Table has sufficient information to test the correctness of a guess. Therefore an entity that knows C_{sk}, VC_{id} and VC_{sk}^{id} can brute-force it easily as follows:

Algorithm 1 Brute-forcing short vote cast code ($sVCC$): client-Vote Verification Context Collusion.

for $b = 0$ to $10^8 - 1$ **do**
  Compute the confirmation message $CM^{id} = b^{2VC_{sk}^{id}} \bmod p$.
  Set $lVCC = HMAC(pCC|VCID|EEID|\{attributes\}, CodesSK)$.
  Search for $H(lVCC)$ in the Codes Mapping Table for VC_{id}.
  if $H(lVCC)$ is present **then**
    set $BCK = b$
    set $EKey = KDF(lVCC, keylength)$
    Retrieve the encrypted $sVCC$ associated with $H(lVCC)$;
    Decrypt it with $EKey$.
    Return the short, signed $sVCC$.
  end if
end for

This requires computing a few hundred million hashes, which can be done in seconds.

The attack could be performed by a cheating Vote Verification Context to whom the Verification Card Private Key VC_{sk}^{id} had (deliberately or accidentally) leaked. Note that new side-channel key leakage attacks appear all the time (see [KGGY20] for example).

If the Vote Verification Context actively colludes with a cheating client, a vote could be counted even though it does not match the voter's intention. The cheating client could send a vote other than what the voter wanted, and also leak VC_{sk}^{id} (and VC_{id}) to the cheating Vote Verification Context. The voter will receive short Choice Codes she does not expect, and she will not confirm the vote, refusing to submit her BCK. The cheating client submits some fake value for BCK instead. However, the Vote Verification Context can brute-force $sVCC$ as above, and follow the protocol exactly as if the BCK value it received was the correct one. From the server side, even with complete information and perfect logs, this looks almost exactly like a vote that was sent and confirmed—the discrepancy is apparent only to an auditor who can double-check the HMAC computation knowing $CodesSK$. Thus the vote is counted although the short Choice Codes were wrong and the voter did not confirm the vote.

Attacks such as the ones described above could also be facilitated by the fact that the exponentiation function is actually homomorphic (and not a PRF, as we discussed in Section 2.2.1). The impact of the following strategy looks largely theoretical, but it shows that the non-PRF character of the exponentiation function may lead to some unexpected effects.

Suppose, that a malicious Vote Verification Context sees the sequence of $pCC_i = p_i^{VC_{sk}^{id}}$ of a voter. He may now, even ignoring the value of VC_{sk}^{id}, explore all sorts of products of pCC_i, derive the corresponding $lVCC$ values and test if $H(lVCC)$ appears in the Codes Mapping Table. This may succeed with non-negligible probability and make it possible to decrypt $sVCC$ even if the voter never started the vote confirmation phase.

Of course, this strategy would only work if the BCK happens to be a product of the primes corresponding to the candidates selected by the voter. We observed, for instance, that 10504 of the 8-digit integer values that BCK can take are a product of primes between 3 and 29, which is a proportion around 10^{-4}. The actual success probability will be smaller, since only specific combinations of primes can make valid votes, and not all of them will be QR's. It would still be much larger than what it would be if a real PRF were used.

4.2.3. Discussion and possible mitigations

This attack is detectable by a diligent voter who checks whether her client has received the signed vote and receipt that is supposed to be sent in Step 11 (it will not be) or who logs back in with a non-colluding client (as described in Section 5.3.1) to check whether the Election Information Context has confirmed her vote (it will not have). However, a voter who trusts her codes alone will be deceived.

We emphasise that this attack is not inconsistent with the proof of individual verifiability, nor with the requirement for individual verifiability in the context of an entirely-trusted server side. It is, however, inconsistent with the informal hints that server-side misbehaviour can be detected and prevented. It is also inconsistent with the informally-expressed idea that server-side manipulation can be detected by diligent logging or careful administration. Neither of these attacks would be evident in the logs—one is completely indistinguishable at the server side from a true refusal to confirm; the other

is distinguishable only given secret information ($CodesSK$).

These issues are not easy to fix—indeed, the whole purpose of the complete verifiability version (both of the Federal Chancellery requirements and of sVote) is to remove these sorts of problems. Hence we recommend simply communicating the security properties more accurately.

Recommendation 9 (simple; spec and other documents)**.** *Be more explicit about the trust in* every *server-side component to be honest. And be clear that undetectable attacks by single server-side components are possible.*

4.3. The use of trapdoored parameter generation

Section 4.1.2 of the spec ([Scy19b] and [Scy19c]) describes some conditions that the election prime parameters (p, q, g and the primes representing the voting options) must satisfy, but does not describe how they are generated or where the "list of prime numbers" comes from.

Returning to the "Security analysis of key cryptographic elements for individual verifiability," we will illustrate the importance of verifiable parameter generation. It says:

> Individual verifiability of sVote relies on the inability of an external attacker to express g as a combination of voting options. g has to be selected in such a way that finding the correlation between any two voting options and g is hard. Otherwise, an attacker who found a correlation for example between v_1 , v_2 and g will we able to break might individual verifiability by computing Choice Return Code for v_1 based on Choice Return Code for v_2 and public values (This would imply that the attacker controls the voting server, which goes beyond the trust assumptions for voting systems up to 50% of the electorate. However, it is a best practice that these attacks are not possible even if an attack would go beyond the adversary's capabilities). To avoid the problem of checking all possible combinations in case when p and q are generated in a verifiable way, the generator g of the cyclic group QR q is selected as the first small prime of the list of all small primes that qualifies.

We were unable to find any precise specification of how the group parameters (p, q, g) and the primes v_1, v_2 used to indicate voting choices are chosen, though the security analysis later mentions that "The values of p , q and g are configured at deploy-time. It is possible to generate a fresh pair p and q in a verifiable way according to FIPS 186-4 recommendations." (Section 4.2) This is not specified in the spec, however. Section 4.1.2 of the spec, and section 2.5 of the older proof [Scy17b] describe the conditions that the parameters must satisfy, not how they are generated. The spec simply adds "This activity may be performed in a pre-configuration step." In this section we explain why verifiable generation is important, and why verification of the generation process should be performed, especially if this generation does not happen within trusted components.

Note also there's a very important difference between private key material and parameter generation. These parameters are public and there is no problem if they are leaked.

This may suggest that this generation of public parameter generation, which may be a relatively expensive task from a computational point of view, could happen externally, in a less trusted and more convenient environment. The problem is that if these public parameters are maliciously generated with a trapdoor, this could allow attacks on both privacy and integrity.

It is not necessary for the attacker to be able to "express g as a combination of voting options." Suppose that the parameter generation is compromised so that someone knows w, v, a, b s.t. $w^a = v^b \bmod p$, where v and w are small primes used to represent voting options. This is probably hard to compute for a *given* p, but it is not hard to generate values of p for which such relationships are known—an example is included at the end of this report. We will use this as an example throughout this section, but knowledge of other nontrivial relationships among the parameters (such as knowing $\log_g v_i$) would permit similar attacks.[3]

We will examine what attacks are possible for clients and servers that have this knowledge. This section makes use of ideas from Matt Green, Nadia Heninger and Hovav Shacham (who also generated trapdoored parameters), which they expressed in the context of sVote 2.0, the complete verifiability version of the system.

4.3.1. Vote privacy - server-side only

Assume that there are two or more questions in the election, and that v, w are prime numbers used to represent answers to the first and second questions respectively.

Suppose that the Vote Verification Context VVC knows w, v, a, b s.t. $w^a = v^b \bmod p$. The VVC can decrypt the partial choice codes pCC_1^{id}, pCC_2^{id} sent by the client. (In the "proof of individual verifiability," they are sent in the clear so this attack could be performed by anyone; in the spec they are encrypted with the Choice Codes Public Key pk_{CC} for which the corresponding private key is held by the VVC (Spec 5.2 (3) and (11a), p.45–6)). The attacker (who observes but need not alter the VCC) would like to learn the contents of the vote.

Recall that $pCC_i = v_i^{VC_{sk}}$ (we omit the id index because we consider one voter at a time). If it happens that this voter answered v to the first question and w to the second question, then $pCC_1 = v^{VC_{sk}}$ and $pCC_2 = w^{VC_{sk}}$, so $pCC_1^b = pCC_2^a$. This is overwhelmingly unlikely to occur for different vote choices. Hence this simple test can be used to detect if the voter made exactly that pair of choices. Note that this does not require collusion from the client because it does not need knowledge of VC_{sk}. Nor is it significantly ameliorated by separating the pCC-decryption authority from the VCC—whichever entity derives the Partial Choice Codes can perform this attack, if it knows the trapdoor.

A very similar attack works when there is only one question on the ballot, but requires knowledge of C_{sk}. The Vote Verification Context, given one value of pCC, can exponentiate it by a/b. If $H(f_{CodesSK}(pCC^{a/b}))$ is in the codes table, it is overwhelmingly likely that pCC represents a vote for w. (Again the protocol update to incorporate the voter

[3]This is because the attacker knows $g^{VC_{sk}}$, which is public information, so it can calculate $v_i^{VC_{sk}}$.

and election ID into f makes no difference to this attack.)

4.3.2. False return-code generation - client-server collusion

The knowledge of a nontrivial relationship in the voting parameters allows for an attack very similar to that described in Section 4.2.2, but resulting in a forged Vote Choice Code. This attack requires collusion between the Vote Verification Context (VVC) and the cheating client. Again, it does not break the security model, but it is another example in which a single server-side component, in collusion with a cheating client, can subvert verification in a way that leaves no trace in logs and requires no out-of-band communication. It would not be detected by a voter who logged back in (as permitted in 5.2.1) to see her vote choice codes via a non-colluding client, because it stores the choice code that matches her intended vote, not the choice code for the vote that was actually cast.

Again suppose that the Vote Verification Context VVC knows w, v, a, b s.t. $w^a = v^b \bmod p$, but now assume that v and w are different answers to the same question, for example one means 'yes' and the other 'no.' For simplicty, assume there is only one question on the ballot and that the voter chooses v. The cheating client instead sends a perfectly consistent vote and partial choice codes for w. We will show how the VVC can generate and return the correct short choice codes for v, despite this being stored and counted as a vote for w.

The VVC decrypts the partial choice code $pCC_1 = w^{VC_{sk}}$ sent by the client. It sets

$$pCC'_1 = (pCC_1)^{a/b} = w^{VC_{sk}a/b} = v^{VC_{sk}}$$

which is the correct partial choice code for v. It then performs the rest of the protocol exactly as specified, but with pCC'_1 instead of pCC_1. This allows it to retrieve the long Choice Code corresponding to v from the table, and then decrypt the short choice code to be returned to the voter. This matches the vote that the voter requested, though not what has been stored on her behalf. We assume that the voter would respond with her BCK and that the vote would be confirmed. (Of course, if she didn't then the attack from Section 4.2.2 could be deployed.)

This attack would not be visible in the logs at all—it creates exactly the trail of a properly submitted and confirmed vote. Like the attack in Section 4.2.2, it is distinguishable only by an observer who knows the codes secret key (*CodesSK* in the spec, or C_{sk} in the proof).

It does require some way for the VVC to know what the voter asked for, in contrast to the value sent by the client, but this could be agreed in advance (assuming the VVC knows in advance which clients are corrupt) or communicated using some of the bits of randomness in the encryptions.

The main point here is to ensure that no trapdoors of this form are known.

Recommendation 10 (medium; spec and code)**.** *Generate the election parameters p, q and the primes used for vote choices in a transparent way.*

On a similar theme, we note that the protocol specification requires p to be chosen as a 2048 bit prime. While this is not expected to be a security issue today, it is below most current standard recommendations, which indicate to use primes of at least 3000 bits when new primes are selected.

For instance:

- The ECRYPT-CSA report of 2018 [ECR18] recommends 3072 bits for "near term protection".
- The BSI recommends, in a report of 2019 [BSI19], the use of 2000 bit for security until 2022, and 3000 bits for applications that will be in use beyond 2022. This second time horizon looks reasonable for the protection of votes.

Similar numbers can be found from recent reports of the NIST or the American NSA.[4]

It would be in line with these reports to take the opportunity of the changes in the generation process of p and q to adapt the length of these primes.

Recommendation 11 (simple; spec and code)**.** *Generate the election parameters so that p is at least 3072 bits long.*

5. Receipt Freeness

The spec [Scy19c, 5.3 Confirms a Vote], Step 11 on p. 51 says "The Election Information Context retrieves the vote and the receipt from the ballot box and the Voting Workflow Context sends this information together with the Vote Cast Code and its signature to the Client Context." The receipt includes a hash of the verification card public key and the voting card ID, making it hard to pretend that you have received someone else's. This introduces a privacy issue because it allows the client to *prove* that a particular vote was cast. Although it is reasonable that the client must be trusted for privacy, it is not inevitable that a cheating client can prove that a certain vote was cast: simply leaking the information would be open to some degree of doubt, so that a voter who wanted to deceive a coercer who hadn't been watching her vote, could simply fake screenshots or choice codes that pretended to do what the coercer wanted without actually casting that vote. The return of the signed vote makes this impossible.

To prove how a particular person voted (or more precisely, how a particular VCID and VC_{pk} voted), the client needs to remember the randomness r used to generate the encrypted vote initially. It can then exhibit this value, the claimed vote, and the other information described in 5.3 Step 12 (such as the signed receipt, signed encrypted vote, authentication token signature) and send it to the coercer. The coercer can verify the relevant server-side signatures and recompute the encryption to check that the vote was cast as claimed. These signed values could only be available if exactly that vote was confirmed.

[4] See `https://www.keylength.com`.

This aspect is not described in the public protocol description [Scy17b], in which only the vote cast code, not the vote, is returned to the voter.

Receipt-freeness [BT94]—meaning the impossibility of proving how you voted—is an important privacy-related property of elections. It is very hard to achieve for universally-verifiable e-voting systems. Nevertheless it seems easily achievable in a system that requires only individual verifiability in the context of a trusted server side.

It is debatable whether "the risk of vote selling is not significantly greater than with postal voting." (Annex [Cha18a] 4.2.2). This signed receipt certainly allows a person selling their vote to provide incontrovertible evidence of what vote was cast, even after the election, to a coercer who did not watch them vote. This does not seem possible with postal voting, though of course postal voting credentials can simply be handed over before election day.

Recommendation 12 (medium difficulty ; spec and code)**.** *Reconsider whether sending a signed vote back to the voter is a good point on the tradeoff between evidence and privacy.*

6. Discussion and Summary

- We did not find any new client-only attacks on individual verifiability that could be exploited on the latest version of the code.
- We found a series of gaps and errors in the proof of individual verifiability, which render it unconvincing. It also differs significantly from the current version of the protocol.
- We found client-only attacks on individual verifiability that succeed on a literal interpetation of the spec, though they do not succeed on the current version of the code. These imply edits are needed to the spec to make the defences explicit.
- We found instances in which a single misbehaving server-side entity can read or undetectably alter votes. Although not inconsistent with the formal requirements, this is inconsistent with informal hints that are given throughout the documentation that suggest a weaker trust model on the server side.
- We showed that voters can produce a completely convincing proof of how they voted, even to a third party who did not watch them vote.

In summary, compared to the super-simple trusted voting system described in the Introduction, it appears that sVote v1 achieves approximately the same security goals with vastly greater complexity. We did not find new attacks on cast-as-intended verifiability, but we also did not find convincing evidence that no such attacks exist. The informal claims to have better defence against a malicious server than *tVote* are not true.

We compared the crytographic protocols, without examining voter authentication or the administrative and procedural protections that are in place around sVote. However,

complexity affects these measures too: it is much easier to protect a simple system with known shortcomings, than a more complex system whose administrators may not be aware of all its limitations and vulnerabilities.

sVote v1 has at least the same shortcomings as *tVote*, but they are not obvious. Its complex design means that it is more, not less, likely than a simple system to have more undetected individual verifiability failures. Also there are (still) some functionally important differences between the source code and the specification—the system could be much more genuinely said to have had proper scrutiny if the source code and specification documents were made available online. We have only scratched the surface—it is likely that there are more issues in the parts of the specification and code we have not had time to examine.

References

[ABF+18] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Kazuma Ohara, and Hikaru Tsuchida. How to choose suitable secure multiparty computation using generalized spdz. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2198–2200. ACM, 2018.

[ADMP+09] Ben Adida, Olivier De Marneffe, Olivier Pereira, Jean-Jacques Quisquater, et al. Electing a university president using open-audit voting: Analysis of real-world use of helios. *EVT/WOTE*, 9(10), 2009.

[BG12] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *Advances in Cryptology - EUROCRYPT 2012*, pages 263–280. Springer, 2012.

[BSI19] BSI. Kryptographische verfahren:empfehlungen und schlussellangen. `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf`, January 2019. BSI TR-02102-1.

[BT94] Josh Benaloh and Dwight Tuinstra. Receipt-free secret-ballot elections. In *STOC*, volume 94, pages 544–553, 1994.

[Bv17] David Basin and Srdjan Čapkun. Review of electronic voting protocol models and proofs (combined final report), May 2017. `https://www.post.ch/-/media/post/evoting/dokumente/zertifikat-pruefung-des-kryptographischen-protokolls.pdf?la=en&vs=1`.

[Cha18a] Swiss Federal Chancellery. Annex to the FCh (OEV, SR 161.116) ordinance of 13 december 2013 on electronic voting - version 2.0, July 2018.

[Cha18b] Swiss Federal Chancellery. Federal chancellery ordinance 161.116 on electronic voting (veles) of 13 december 2013, July 2018.

[ECHA09] Aleks Essex, Jeremy Clark, Urs Hengartner, and Carlisle Adams. How to print a secret. In *Proceedings of the 4th USENIX conference on Hot topics in security, Hot-Sec*, volume 9, pages 3–3, 2009.

[ECR13] ECRYPT. Final report on main computational assumptionsin cryptography. `https://www.ecrypt.eu.org/ecrypt2/documents/D.MAYA.6.pdf`, January 2013.

[ECR18] ECRYPT – CSA. Algorithms, key size and protocols report (2018). `http://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf`, February 2018.

[Gjø11] Kristian Gjøsteen. The norwegian internet voting protocol. In *International Conference on E-Voting and Identity*, pages 1–18. Springer, 2011.

[Hoa81] Charles Antony Richard Hoare. The emperor's old clothes. *Communications of the ACM*, 24(2):75–83, 1981.

[KGGY20] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. Rambleed: Reading bits in memory without accessing them. In *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[KL15] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, 2nd edition, 2015.

[LPT19a] Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. Addendum to how not to prove your election outcome. 2019. `https://people.eng.unimelb.edu.au/vjteague/HowNotToProveElectionOutcomeAddendum.pdf`.

[LPT19b] Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. Ceci n'est pas une preuve: The use of trapdoor commitments in bayer-groth proofs and the implications for the verifiabilty of the scytl-swisspost internet voting system, 2019. `https://people.eng.unimelb.edu.au/vjteague/UniversalVerifiabilitySwissPost.pdf`.

[LPT19c] Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome: The use of non-adaptive zero knowledge proofs in the scytl-swisspost internet voting system, and its implications for decryption proof soundness, 2019. `https://people.eng.unimelb.edu.au/vjteague/HowNotToProveElectionOutcome.pdf`.

[NIS07] NIST. SP 800-38D Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. `https://csrc.nist.gov/publications/detail/sp/800-38d/final`, November 2007.

[NPR99] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and kdcs. In *Advances in Cryptology - EUROCRYPT '99*, volume 1592 of *Lecture Notes in Computer Science*, pages 327–346. Springer, 1999.

[RT09] Peter YA Ryan and Vanessa Teague. Pretty good democracy. In *International Workshop on Security Protocols*, pages 111–130. Springer, 2009.

[Scy17a] Scytl. Analysis of cast-as-intended verifiability and ballot privacy properties for scytl's swiss on-line voting protocol using proverif (version 2). `https://www.post.ch/-/media/post/evoting/dokumente/analysis-verifiability-and-privacy-properties-for-swiss-post-voting.pdf?la=en&vs=1`, 2017.

[Scy17b] Scytl. Swiss online voting system cryptographic proof of individual verifiability, April 2017. `https://www.post.ch/-/media/post/evoting/dokumente/swiss-post-cryptographic-proof-of-individual-verifiability.pdf?la=de&vs=2`.

[Scy19a] Scytl. Scytl online voting protocol specifications – document differences between versions 5.2 and 5.0, 2019.

[Scy19b] Scytl. Scytl online voting protocol specifications – document v. 5.2, 2019.

[Scy19c] Scytl. Scytl online voting protocol specifications – document v. 5.3, 2019.

[Scy19d] Scytl. Security analysis of key cryptographic elements for individual verifiability, v 1.1, May 2019. `https://www.post.ch/-/media/post/evoting/dokumente/swiss-post-cryptographic-proof-of-individual-verifiability.pdf?la=de&vs=2`.

[Sec19] Kudelski Security. Swiss post security review of key cryptographic elements of the e-voting solution (version with individual verifiability at 50% of the electorate), May 2019.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. `https://eprint.iacr.org/2004/332`.

A. Trapdoored election parameters

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 $|p| = 2046$.

$v = 11, w = 53, a = 592, b = 357,$
$p = 70661253006860938188288686008587306877924989809763017605234587520311617337105046449553576599718412087023157743527914173027880612549152925893965244558545474129308217060017773882336283820366471809571051189156176768816344699208105091538533363999412975733618190464709094803803163968319799200086181544516808280230172888032317476018477679086575899964746340368641784343728714991157449798990790914967361122128203357908982556730725948241307410998309683403135701834466166179508219320004771007201603990880213385798586078593775866801311055884555209942565902767953591074394931972664914027713315544580116256428902163022146337955527.$