Breaking verifiability and vote privacy in CHVote

Véronique Cortier, Alexandre Debant, and Pierrick Gaudry

Université de Lorraine, LORIA, INRIA, CNRS, Nancy, France
firstname.lastname@loria.fr

Abstract. CHVote is one of the two main electronic voting systems developed in the context of political elections in Switzerland, where the regulation requires a specific setting and specific trust assumptions.
We show that actually, CHVote fails to achieve vote secrecy and individual verifiability (here, recorded-as-intended), as soon as one of the online components is dishonest, contradicting the security claims of CHVote. In total, we found 9 attacks or variants against CHVote, 2 of them being based on a bug in the reference implementation. We confirmed our findings through a proof-of-concept implementation of our attacks.

1 Introduction

Several countries use Internet voting for politically binding elections, at least in trials. This often comes with the discovery of flaws, such as in Australia [21], Estonia [24], Switzerland [14], United States [28], or France [15]. One of the most demanding countries in terms of regulation is Switzerland. The requirements are provided in an ordinance of the Swiss Federal Chancellery [26] and include:

- *vote secrecy*: no one should know how a voter voted;
- *individual verifiability*: when a voter successfully completes their voting session, they must be guaranteed that their vote will be counted, as intended;
- *universal verifiability*: the result corresponds to the recorded ballots.

The ordinance comes with a demanding trust model: trust must be split between different components; some of them being online, others offline. Security goals must be met even if all but one online component are compromised. Moreover, for individual verifiability, the ordinance requires to leverage verification codes to protect against corrupted voting devices. Voters (securely) receive a voting sheet, with a verification code for each candidate. They confirm their ballot only if their voting device is able to display the right verification code, which guarantees that it has encrypted the right vote. This property is called cast-as-intended.

This work benefited from funding managed by the French National Research Agency under the France 2030 programme with the reference ANR-22-PECY-0006.

To improve confidence in the system, the Chancellery also asks for a public specification and a public code. Public scrutiny is encouraged through a bug bounty program and regular intrusion tests. A symbolic and a cryptographic proofs of the protocol are needed before its deployment.

CHVote. Since the last major flaw found in 2019 [14] against a protocol proposed by Scytl, the only system used in practice is the one developed by SwissPost, that is an evolution of the Scytl system. However, in the Swiss landscape, there is a second major proposed protocol, CHVote, the main competitor to the SwissPost protocol. The Canton of Geneva has been developing a voting system since 2003, and it has been used by 125 000 voters in 4 cantons [1]. A second generation of their system, much more secure, has been developed since 2016 [19]. This is the system we refer to when we talk about CHVote. While the funding from the Canton of Geneva has been discontinued in 2018 [2], CHVote is still under continuous development both in terms of specification (20 versions from 2017 to 2024) and reference code (last release of OpenCHVote is 2.3.1 on December 2024), with new funding from the Federal Chancellery in 2024 [20].

The CHVote protocol relies on Oblivious Transfer (OT) [17] in order to achieve cast-as-intended. The idea is natural: the online system as a whole must return the verification code corresponding to the candidate A when receiving an encryption of A, *without learning* A, which is exactly the principle of OT. CHVote then builds upon this idea and extends it in order to distribute the trust among the online components (only one is trusted) and to allow for k-out-of-n OT, in order to cover elections where voters may select up to k candidates among n. CHVote is claimed to fully satisfy the Chancellery requirements.

Our contributions. We conduct a systematic review of the CHVote protocol and we discovered that it satisfies neither vote secrecy nor individual verifiability, as soon as one online component is dishonest. On the first side, our attack on vote secrecy relies on an issue on the OT primitive used in CHVote. Indeed, it does not protect against *selective failure attack* [10]. Hence a dishonest component may selectively modify one of their (shared) verification codes and see whether the voter can still proceed. If yes, the voter did not vote for the candidate corresponding to the modified code. We show that this attack can be made completely silent (when the voter did not vote for the modified option) in the subsequent phases of the protocol, and undetected by the other (honest) components of the system, including the external auditor.

On the other side, our verifiability attack relies on the fact that CHVote misses an agreement procedure between the online components: they each answer independently, even if they do not share the same view. This can be exploited by a malicious component, collaborating with a dishonest voting device, to register a ballot for B while the voter has selected A and received the corresponding verification code for A. The dishonest component will be able to pretend that it has only seen a ballot for B, and the honest components will silently accept a confirmation phase made of mixed contributions (from ballots for A and B). The inconsistency between the views of the honest and dishonest components will be

caught only after the tally, by the external auditor, with no way of distinguishing between the honest and dishonest behaviors.

Moreover, we discovered that the reference implementation of CHVote [3] fails to perform some of the checks mentioned in the specification of the components of the system. We exploit this implementation bug to show that the (first) online component of the system can fully manipulate the received ballots, replacing them by any vote of its choice, without being detected by the other online components of the system. This attack will, again, be detected after the tally, by the external auditor. However, the Chancellery does not assume any honest auditor for the individual verifiability property.

We also considered the case where all online components are dishonest for vote secrecy, as prescribed by the Chancellery when all the online components are operated by the same private company (which is the case in practice). CHVote does not claim any security in this case and, actually, vote secrecy completely collapses in this case since we show that honest but curious online components fully learn who voted what, although they do not have the entirety of the decryption key. Since they do not need to behave maliciously, the attack is undetectable. It simply relies on the fact that the last decryption step, performed by an external offline authority, is missing a round of shuffling, probably due to organizational constraints.

We implemented our main attacks against the reference implementation and confirmed our findings. In the last part of this paper, we discuss possible directions on how to fix the CHVote protocol.

Related work. An attack against individual verifiability was found in 2017 [9], due to the fact that the oblivious transfer primitive did not achieve sender privacy. Hence a dishonest voting device could actually obtain the verification code corresponding to candidate A while encrypting a vote for B. This flaw was fixed in a later version of the specification, at the cost of a higher complexity ($O(nk)$ where n is the number of candidates and k is the number of selected candidates). Up to our knowledge, no other attack was found against CHVote since then. A symbolic and a cryptographic security proofs have been proposed [8], for verifiability only (no proof for vote secrecy). These proofs fail to catch our verifiability attacks due to the fact that they considered that the counted ballots are the ones registered by the honest online component. This issue however is that it is impossible to identify which component is honest.

2 Overview of CHVote and trust model

2.1 Presentation of the protocol

In the work, we refer to the latest version of the specification at the time of writing, namely the 20-th revision of [20], corresponding to version 4.3. The full protocol supports a wide variety of possibilities, including multiple questions, multiple answers, several counting circles, and even write-ins. To illustrate our attacks, we concentrate on a minimalistic situation, where there is only one question, and the voter must select exactly one voting option among n possibilities.

Cryptography and notations. We loosely follow the notation system of the specification. Let G be a cyclic group generated by g of prime order q. The protocol relies on ElGamal encryptions in G. If m is a group element, then the encryption of m with public key pk and randomness r is $\mathsf{Enc}^r_{\mathsf{pk}}(m) = (\mathsf{pk}^r\, m, g^r)$. A list of group elements $p_1, p_2, \ldots, p_n$ encode the n voting options: if the option number s is selected by the voter, the voting client encrypts $m = p_s$.

Various zero-knowledge proofs are used. They are classical proofs of knowledge based on Sigma protocols. In this paper, we denote by π_X a proof of knowledge of X. We use hash as a notation for a cryptographic hash function, and we write $\mathsf{short\_hash}$ when the output is small, due to usability considerations, and can be brute-forced. We always use the notation $X^{(j)}$ for some data X related to the j-th authority. We denote $[X^{(j)}]_j$ or $[X_j]_j$ a vector of data indexed by j.

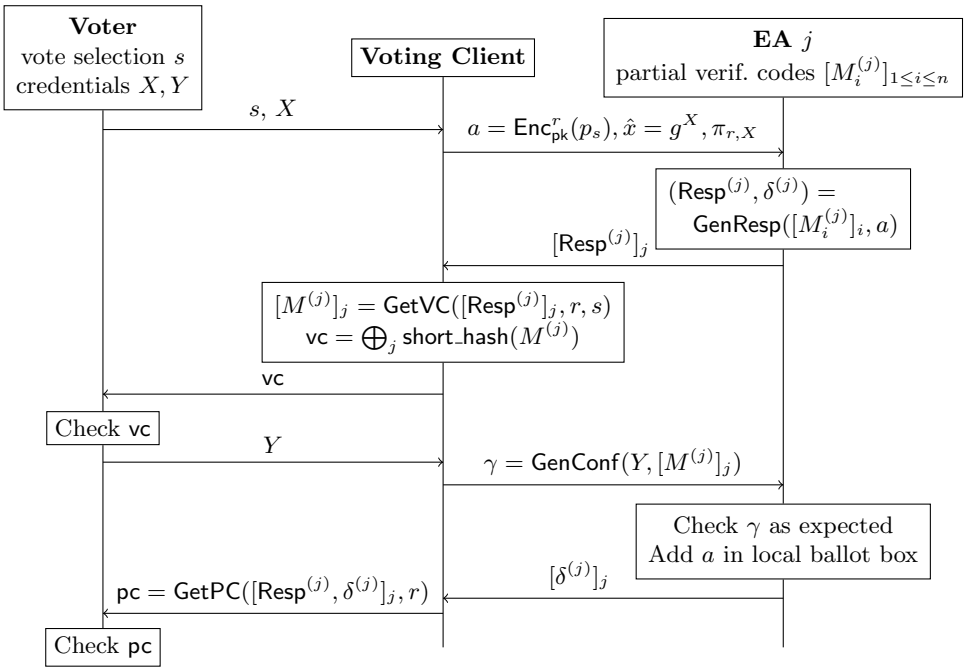
Participants. The protocol relies on several authorities:

- **Election administrator (Admin).** It defines the general setting (electoral roll, questions, etc), and holds a share of the decryption key.
- **Election authorities (EA).** They produce voting material to be sent to voters by postal mail; during the voting phase, they collect the encrypted ballots, and send the verification codes to the voters; each EA also holds a share of the decryption key.
- **Printing authority (Printer).** During the setup, it collects the data from the election authorities, prints them, and sends them to voters by post.
- **Verifier.** This group of third-parties checks that the data at the end of the election is consistent.

Finally, in order to study the cast-as-intended property, it is important to separate the **Voter** from their **Voting client**.

In a typical Swiss situation, the number of Election authorities is 4, but this is not at all fixed in the protocol.

Setup. First, Admin defines the setting, including the n possible voting options, and, together with the EAs, collectively generates a public key pk. Then, for each voter, the j-th EA picks a set of partial verification codes $[M_i^{(j)}]_{1 \le i \le n}$, one for each voting option. Upon reception of these, for all i, the Printer hashes each $M_i^{(j)}$ to a short bit string $V_i^{(j)}$, and aggregate them with a xor to produce a verification code vc_i printed on the voter's voting sheet. The EAs also collectively produce the eligibility data $\hat{x} = g^X$, where $X = \sum_j X^{(j)}$ (called voting code in the specification), the confirmation data $\hat{y} = g^Y$, where $Y = \sum_j Y^{(j)}$, and the vote validity data $\hat{z} = g^Z$, where Z can be deduced from the $[M_s^{(j)}]_j$, for any selection s (more details about Z will be given when describing the verifiability attack). The values X and Y are computed by the Printer by aggregating contributions of all the EAs, and printed on the voter's sheet. The EAs know $\hat{x}$, $\hat{y}$ and $\hat{z}$ for each voter.



$\mathsf{GenResp}([M_i^{(j)}]_{1\le i\le n},\ a = (a_1, a_2))$
(run by the j-th EA *)*
Pick $z_1^{(j)}, z_2^{(j)}, \beta \in_R \mathbb{Z}_q$
$b^{(j)} := a_1^{z_1^{(j)}}\ a_2^{z_2^{(j)}}\ \beta$; $\quad d^{(j)} := \mathsf{pk}^{z_1^{(j)}}\ g^{z_2^{(j)}}$
For $i \in [1, n]$:
$k_i := p_i^{z_1^{(j)}}\ \beta$
$C_i^{(j)} := M_i^{(j)} \oplus \mathsf{hash}(k_i)$
$\mathsf{Resp}^{(j)} := (b^{(j)}, d^{(j)}, [C_i^{(j)}]_{i\in[1,n]})$
Return $\mathsf{Resp}^{(j)}$ and $\delta^{(j)} = (z_1^{(j)}, z_2^{(j)})$

$\mathsf{GenConf}(Y, [M^{(j)}]_j)$
(run by Voting Client *)*
$\hat{y} := g^Y$
For all j:
See $M^{(j)}$ as point $(x^{(j)}, y^{(j)})$
$Z := \sum_j y^{(j)}$; $\quad \hat{z} := g^Z$
$\pi_{Y,Z} :=$ pok of Y and Z
Return $\gamma = (\hat{y}, \hat{z}, \pi_{Y,Z})$

$\mathsf{GetVC}([\mathsf{Resp}^{(j)}]_j, r, s)$
(run by Voting Client *)*
Expand $\mathsf{Resp}^{(j)}$ as $(b^{(j)}, d^{(j)}, [C_i^{(j)}]_i)$
For all j:
$k^{(j)} := b^{(j)}(d^{(j)})^{-r}$
$M^{(j)} := C_s^{(j)} \oplus \mathsf{hash}(k^{(j)})$
Return $[M^{(j)}]_j$

$\mathsf{GetPC}([\mathsf{Resp}^{(j)}, \delta^{(j)}]_j, r)$
(run by Voting Client *)*
Expand $\mathsf{Resp}^{(j)}$ as $(b^{(j)}, d^{(j)}, [C_i^{(j)}]_i)$
For all j:
Check $d^{(j)} = \mathsf{pk}^{z_1^{(j)}}\ g^{z_2^{(j)}}$
$\beta^{(j)} := b^{(j)}(d^{(j)})^{-r} p_s^{-z_1^{(j)}}$
For $i \in [1, n]$:
$k_i^{(j)} := p_i^{z_1^{(j)}}\ \beta^{(j)}$
$M_i^{(j)} := C_i^{(j)} \oplus \mathsf{hash}(k_i^{(j)})$
$P^{(j)} := \mathsf{short\_hash}(M_1^{(j)}, \ldots, M_n^{(j)})$
Return $\mathsf{pc} = \bigoplus_j P^{(j)}$

Fig. 1. Overview of the voting phase of the CHVote protocol. This is a simplified version of Protocols 7.6 and 7.7 in [20]. The four algorithms are simplified versions of Algorithms 8.27 to 8.38.

Voting phase. This is a 2-round protocol between the Voter and the EAs, via the Voting client, summarized in Figure 1. During this phase, the EAs do not communicate with each other. First, the voter gives their eligibility data X and their selected voting option to their Voting client, which encrypts it with pk and sends it to all the EAs, together with $\hat{x} = g^X$ and a ZKP. The j-th EA does its "sender" part of the OT protocol, and produces n values $[C_i^{(j)}]_{1 \leq i \leq n}$. The tricky part is that the ElGamal ciphertext sent by the Voting client is seen as the "query" part of the OT. From the $[C_i^{(j)}]_i$ and other opening data, the Voting client can compute $[M_s^{(j)}]_j$, and then follow the same procedure as the printer to produce the verification code vc, shown to the Voter.

The Voter checks that the code corresponds to the one printed on paper, and if this is the case, it initiates the second round by giving its confirmation code Y to the Voting client, which can deduce $\hat{y}$ and $\hat{z}$ from it and previously received information. Together with a ZKP, this form γ that is sent to the EAs. Each authority checks that γ corresponds to the values committed during the setup phase. If this is the case, they record the ballot in their (local) ballot box, and then send additional data, so that the Voting client can compute all the partial verification codes, and not only the one corresponding to s. From these, the Voting client deduces a participation code pc, that the Voter can compare to what is printed on their sheet.

Tally phase. The EAs perform one round of verifiable mixnets, and then one round of partial decryption. Finally, Admin completes its partial decryption, and publishes the result. The Verifier checks that all the data is consistent.

2.2 Trust model and security claims

Security properties. Like most e-voting protocols, CHVote aims at preserving vote secrecy and verifiability. Verifiability can be defined as three properties:

- *recorded-as-intended*: if a voter has successfully performed all their checks, then they are guaranteed that their ballot is correctly recorded, for the vote they intended.
- *universal verifiability*: the result of the election corresponds to the recorded ballots.
- *eligibility*: recorded ballots only come from voters who have effectively voted.

Recorded-as-intended combines cast-as-intended (the cast ballot corresponds to the voter's intent) and recorded-as-cast (the recorded ballot is the cast one). It is often called individual verifiability in the literature. However, in the Chancellery terminology [26], individual verifiability stands for the combination of recorded-as-intended and eligibility, hence we avoid here the terminology "individual verifiability" to avoid confusion.

Trust model. In the trust model considered by the Swiss Chancellery, the adversary controls all communications over the Internet. Moreover it controls several authorities depending on the security property.

– One-out-of-4 EAs is assumed to be honest for all properties (privacy and verifiability), except if all EAs are operated by a private company. In the second case, none are trusted for privacy but one-out-of-4 remains trusted for verifiability.
– The Voting client is trusted for vote privacy (since the vote is entered in clear) but untrusted for verifiability.
– The Election administrator is trusted (their share of the decryption key remains secret).
– The Verifier is always trusted as a whole, except for recorded-as-intended. It means that the Swiss Chancellery almost always assumes that at least one third-party will behave honestly and do the expected checks (others can be dishonest; it is of interest in particular for vote privacy). Regarding recorded-as-intended, the Swiss Chancellery considers that the property must hold relying on the voter's checks and the behaviour of the honest trusted authority only.

Discussion on the trust model. CHVote is claimed to be secure for the trust model of the Swiss Chancellery. However, this claim comes with the strong assumption that at least one EA is honest. While this made sense in the original version of the Ordinance, this no longer matches the current state of the Ordinance nor the recent deployments of e-voting in Switzerland, in which a single company (SwissPost) is operating the elections (with distinct servers administered by distinct teams for each EA). In this context, the Chancellery requires that none of the EAs is trusted for vote secrecy.

For verifiability, it is assumed that at least one EA is honest. There is no other choice for recorded-as-intended. Indeed, when all EAs are corrupted, they can collaborate with the Voting client to compute the verification code of any candidate and hence, they can register a ballot for a candidate B while the voter will be convinced to have voted for A. Similarly for universal verifiability, since there is no public bulletin board, colluding malicious EAs can easily drop any ballot they wish. The trust assumption is more questionable for eligibility. Indeed, the voter could have some private data generated during the Setup, used to "sign" a ballot, which would prevent ballot stuffing even if all EAs are dishonest. For example, the protocol developed by SwissPost [27] guarantees eligibility even when the 4 online authorities are compromised. We therefore consider additionally the scenario where all EAs are dishonest for eligibility.

Security claims and attacks We summarize on Figure 2 the security claims of the CHVote protocol, namely, both vote secrecy and all three verifiability properties should be satisfied as soon as one EA is honest.

In the remaining of the paper, we show that, actually, both vote secrecy and recorded-as-intended are broken as soon as one EA is dishonest. We provide several variants of our attack against vote secrecy and we propose a fix to make all our vote secrecy attacks detectable. However, our fix no longer works if all EAs are dishonest, which is the trust model considered in actual deployments. Moreover, we show that CHVote is subject to ballot stuffing when all EAs are

	CHVote claims			Our findings		
	VD	EAs		VD	EAs	
Vote secrecy						
• base case	H	≥ 1 H	✔	H	< 4 H	*1. weak Oblivious Transfer* ✘ attack by complaint (Section 3.1) ✘ undetectable attack (Section 3.2) ✘ variant using OT malleability (Section 3.3)
				H	EA_1 D	*2. missing check for EA_1* ✘^{I} secrecy breach by drop (Section 3.5)
• single priv. comp.	H	all D	–	H	all D	1. + 2. + ✘ full vote disclosure, undetectable (Section 3.6)
Recorded-as-intended	D	$\geq$1 H	✔	D H	< 4 H EA_1 D	✘ missing consensus algorithm (Section 4.1) ✘^{I} missing check by honest EAs (Section 4.2)
Universal verifiability	D	$\geq$1 H	✔	–	–	–
Eligibility						
• base case	H	$\geq$1 H	✔	–	–	–
• single priv. comp.	H	all D	–	H	all D	✘ ballot stuffing (Section 4.3) ✘ ballot stuffing, online only (Section 4.3)

Fig. 2. Trust model of the Chancellery and attacks on CHVote.
In gray: outside the trust model of the Chancellery.
D stands for dishonest, H for honest. ≥ 1 *means that 1 out 4 authorities is honest and* < 4 *means that there is at least one dishonest authority.*
✘^{I}*: attack on the implementation only, not the specification.*

compromised. While outside the trust model considered by CHVote and the Chancellery, we believe that this is flaw since it could be avoided.

3 Vote secrecy attacks

3.1 Attack by complaint

The idea of the attack against vote secrecy is very simple. We assume that one EA is dishonest (say EA_1 w.l.o.g.) and wishes to learn how Alice voted. During setup, EA_1 has generated partial verification codes $[M_i^{(1)}]_{1\leq i\leq n}$, for Alice, one for each voting option. EA_1 simply modifies one partial verification code, e.g., $M_1^{(1)}$, and observes the behaviour of the voter under attack:

- if Alice did not vote for candidate 1, her voting device will compute the expected verification code and she will be able to proceed, entering her confirmation code.
- if Alice did vote for candidate 1, then her voting device won't produce a valid verification code and Alice will stop voting and will report the incident.

Hence, EA_1 can deduce if Alice voted for candidate 1. EA_1 can also choose to modify all partial verification codes but $M_1^{(1)}$ if it prefers that Alice does not complain when her vote is guessed correctly. More generally, EA_1 can modify any number of its partial verification codes depending on the information it wants to learn.

A protection against this kind of attack is discussed in an early version of CHVote [19]. The goal is to make this attack detectable not only to voters (they complain) but also to the (honest) EAs. This idea relies on a centralized bulletin board and the fact that each EA reveals z_1, z_2 to the bulletin board when sending their contribution to the participation code. However, this approach has not been fully explored in CHVote, that has no central bulletin board.

3.2 Privacy attack without detection

A drawback of the attack explained in the previous section is that, even when the voter receives a correct verification code, they won't be able to complete the voting phase. Indeed, the participation code pc computed by their voting device is a function of the list of the partial verification codes $[M_i^{(1)}]_{1\leq i\leq n}$ received during the first phase. Hence, if a malicious EA tampered with their partial verification codes, pc will be modified. We show that if a malicious EA (say EA_1) correctly guessed Alice's vote so that she did not complain while EA_1 modified some of the $[M_i^{(1)}]$ then it is possible to improve the attack so that Alice successfully completes her voting session, while she has lost vote secrecy.

Pre-image. The attack relies on the fact that the contribution of each EA is hashed to a short element *before* being xor-ed together. Indeed, the participation code pc equals $\bigoplus_j P^{(j)}$ where each $P^{(j)}$ is a contribution of EA_j: $P^{(j)} := \mathsf{short\_hash}(M_1^{(j)}, \ldots, M_n^{(j)})$. Since $\mathsf{short\_hash}$ outputs a string with an entropy up to 24 bits according to the specification, it is very easy to perform a brute-force search and find a second pre-image with another value of $M_1^{(1)}$ giving the same code.

Hence a malicious EA_1 can modify $M_1^{(1)}$ into M_1', to conduct the attack explained in the previous section, in such a way that

$$\mathsf{short\_hash}(M_1^{(1)}, M_2^{(1)}, \ldots, M_n^{(1)}) = \mathsf{short\_hash}(M_1', M_2^{(1)}, \ldots, M_n^{(1)}) \quad (1)$$

This lets pc unchanged, hence neither Alice nor her voting device can detect the manipulation. None of the honest EAs will detect the manipulation either, which leads to a secrecy loss, without any detection. A more detailed explanation of the attack can be found in Appendix (Figure 4).

Collision. One advantage of the previous attack is that it is sufficient to corrupt the dishonest authority during the voting phase, hence when it is an online server, offering a large attack surface. However, if the dishonest authority is corrupted from the setup phase, then it is possible to speed-up the search for collisions using the Birthday paradox: EA_1 will search for two values $M_1^{(1)}$ and M_1', such that Equation (1) holds. This way, a collision will be found in an (asymptotic) expected number of trials $\sqrt{\pi\ell/2}$ instead of ℓ, where ℓ is the number of possible participation codes. This can be useful to attack many voters, and also shows that mitigations based on enlarging the length of the code are probably unrealistic.

3.3 Attack variants

We provide a variant of our attack that exploits the malleability of the OT primitive. This variant does not provide a more powerful attack but shows that the OT primitive would require an in-depth modification to prevent our attack.

Harmless malleability. We first notice that a malicious authority EA_j can apply a different algorithm for $\mathsf{GenResp}$, without any change in the protocol behavior. In particular, it can chose $z_1 = z_2$ and, for any m_{att}, replace a_1 by $a'_1 = a_1 m_{\mathsf{att}}^{-1}$ and a_2 by $a'_2 = a_2 m_{\mathsf{att}}$ without any noticeable change. Indeed, in the second line of $\mathsf{GenResp}$, the normal computation is $b^{(j)} := a_1^{z_1} a_2^{z_2} \beta$. If instead, the dishonest EA computes $b^{(j)}$ as $b^{(j)} := a_1'^{z_1} a_2'^{z_2} \beta = a_1^{z_1} a_2^{z_2} \beta (m_{\mathsf{att}})^{z_2 - z_1} = a_1^{z_1} a_2^{z_2} \beta$ if $z_1 = z_2$.

This does not lead to any attack but it highlights the fact that a security proof would require to characterize complex harmless adversarial behaviors.

Harmful malleability. This malleability can be turned into an effective attack against secrecy. A dishonest EA may use the following algorithm to generate its response, where we highlight the changes in violet:

$\mathsf{GenResp}'([M_i^{(j)}]_{1 \le i \le n},\ a = (a_1, a_2))$
  $\mathsf{Pick}\ z_1, z_2, \beta \in_R \mathbb{Z}_q$
  $b^{(j)} := (a_1 p_{\mathsf{att}}^{-1})^{z_1} (a_2 p_{\mathsf{att}})^{z_2} \beta$; $d^{(j)} := \mathsf{pk}^{z_1} g^{z_2}$ *(* p_{att} chosen by EA *)*
  $\mathsf{For}\ i \in [1, n]$:
    $k_i := p_i^{z_2} \beta$ *(* instead of $k_i := p_i^{z_1} \beta$ *)*
    $C_i^{(j)} := M_i^{(j)} \oplus \mathsf{hash}(k_i)$
  $\mathsf{Resp}^{(j)} := (b^{(j)}, d^{(j)}, [C_i^{(j)}]_{i \in [1,n]})$
  $\mathsf{Return}\ \mathsf{Resp}^{(j)}\ \mathsf{and}\ \delta^{(j)} = (z_1, z_2)$

Then, when the voting client computes the verification code corresponding to the sent ballot $(a_1, a_2) = (\mathsf{pk}^r p_s, g^r)$, it computes in particular:

$$\begin{aligned} k^{(j)} := b^{(j)} (d^{(j)})^{-r} &= (a_1 p_{\mathsf{att}}^{-1})^{z_1} (a_2 p_{\mathsf{att}})^{z_2} \beta \mathsf{pk}^{-r z_1} g^{-r z_2} \\ &= \mathsf{pk}^{-r z_1} p_s^{z_1} g^{r z_2} p_{\mathsf{att}}^{z_2 - z_1} \beta \mathsf{pk}^{-r z_1} g^{-r z_2} \\ &= p_s^{z_1} p_{\mathsf{att}}^{z_2 - z_1} \beta. \end{aligned}$$

- Either the voter voted for the candidate att guessed by the attacker, in which case $k^{(j)} = p_{\mathsf{att}}^{z_2} \beta$, as computed by the dishonest EA;
- or the voter voted for another candidate and $k^{(j)}$ computed by the voter won't correspond to the one used by EA and the vc will be an arbitrary random value.

We hence retrieve the same kind of attack than the one in Section 3.1.

3.4 Mitigation

An obvious mitigation is to compute the $\mathsf{short\_hash}$ only at the end, that is, $\mathsf{pc} = \mathsf{short\_hash}(\bigoplus_j Q^{(j)})$ where $Q^{(j)} := \mathsf{hash}(M_1^{(j)}, \ldots, M_n^{(j)})$. This does not fix

the vote secrecy attack explained in Section 3.1, but at least, all voters under attack will detect an issue during the confirmation phase.

We note however that this mitigation is insufficient when all EAs are dishonest since they could again find collisions that let `pc` unchanged. The only counter-measure to keep this mitigation is to increase the size of `pc` to 256 bits (to avoid the square root attacks relying on the Birthday paradox). This would however cause usability issues.

3.5 Attack by drop

In the reference implementation [3], we noticed an implementation bug that offers another type of vote secrecy attack, namely, the proof of shuffle performed by EA_1 (specifically) is never verified by the other EAs. The specification somehow assumes that an EA receives the shuffle information from the others in the correct order, but the implementation has to take care of the fact that they can be re-ordered. Therefore the code uses the following greedy strategy: when receiving a shuffle from EA number j, then it checks whether data number $j-1$ is available and if so, checks the ZKP, and it does the same with data number $j+1$. This typically leads to off-by-one issues that must be carefully addressed. In the present case, when checking data number $j-1$, it was also checking whether the ZKP number $j-1$ was available, instead of of the ZKP number j (i.e. the one that relates a shuffle between $j-1$ and j). Concretely, in file `S400.java`, in the first test `get_bold_pi_tilde().isPresent(k - 1)`, the value $k-1$ should be replaced by k. And actually, testing the presence of the ZKP is maybe useless.

Since the shuffle of EA_1 is not verified, EA_1 may actually drop some of the ballots, reducing the anonymity set. For example, in an extreme case, it could remove all ballots except Alice's ballot, wait for decryption, and learn Alice's vote. Even when EA_1 only remove some of the ballots, this forms a theoretical attack since it would learn another information than the expected tally. This attack would in particular break existing definitions of vote secrecy [6,16,7] and hence forbid a security proof. Note that this attack will be detected by the Verifier but too late, since the ballots are already decrypted when the Verifier checks the election data.

We describe a more dramatic exploit of this bug in Section 4.2.

Discussion. When the Chancellery has introduced the requirement that the system remains secure even when all EAs are dishonest, in case they are operated by a single private company, they have also excluded the attack by drop we just described (see 2.7.2 of [26]). Indeed, this attack is unavoidable when all EAs are dishonest, since they may always drop ballot. We see however no reason to tolerate such attacks when at least one EA is honest.

3.6 Full vote disclosure

When the EAs are operated by a single company, which corresponds to "real-world setting within the Swiss context" according to the CHVote specification

([27], page 59), then the Chancellery requires that vote secrecy still holds when the EAs are all dishonest.

Unfortunately, vote secrecy completely collapses in CHVote in this threat model. Indeed, each EA holds a share of the decryption key, while the last share is owned by the Administrator, operated by another entity (typically, a Canton). However, the Administrator decrypts the ballots *without shuffling them*. Hence, the EAs learn exactly who voted what, for all voters, in a honest but curious setting. They simply need to remember how they shuffled the ballots. Since they do not even need to conduct any active attack, the loss of vote secrecy (for the entire set of voters) is undetectable.

4 Verifiability attacks

4.1 Breaking recorded-as-intended

One dishonest EA collaborating with a dishonest voting client is actually sufficient to break recorded-as-intended. More precisely, a voter can successfully complete their voting procedure for a candidate s while their vote will not be counted. This contradicts the security claim that "the voter is given proofs [...] to confirm that no attacker has altered any partial vote before the vote has been registered as cast in conformity with the system" [26].

The attack relies on the fact that there is no consensus between the EAs: they respond independently, even if they do not receive the same encrypted vote. Moreover, we realized that the confirmation phase is completely independent of the vote of the voter.

The idea of the attack is simple and is depicted in Figure 3. When the voter selects candidate s, the (dishonest) voting client actually prepares two ballots: $a = \mathsf{Enc}^{r}_{\mathsf{pk}}(p_s)$ and $a' = \mathsf{Enc}^{r'}_{\mathsf{pk}}(p_{s'})$ where s' is the candidate chosen by the attacker. The honest EAs are given a, while the dishonest EA gets both a and a'. The honest EAs respond normally. The dishonest EA provides 2 responses: one computed from a, such that the voting client can compute the expected verification code vc. But it also computes the response from a'. This way, the dishonest authority will behave "honestly" (e.g., regarding its logs), as if it had only received the ballot a'. We then notice that γ computed by the voting client and the δ^j sent by the EAs do not depend on the ballot a, as explained in Appendix B. Hence, the rest of the confirmation procedure continues normally and the voter successfully receives their participation code.

At the end of the voting phase, the EAs have inconsistent views: the honest EAs have seen a while the dishonest EA will claim to have seen a'. But their transcript are made of purely honestly computed data so it is impossible to detect who misbehaved. Since CHVote does not provide any agreement procedure, the system simply crashes in this case, so our attack is detectable. Yet, it breaches the security claim since it results in a "manipulation [of the vote] that goes undetected by the voter but not by the system" (explanation of the rewarded attacks in the bug bounty program [4] organized for the Swiss Post protocol).

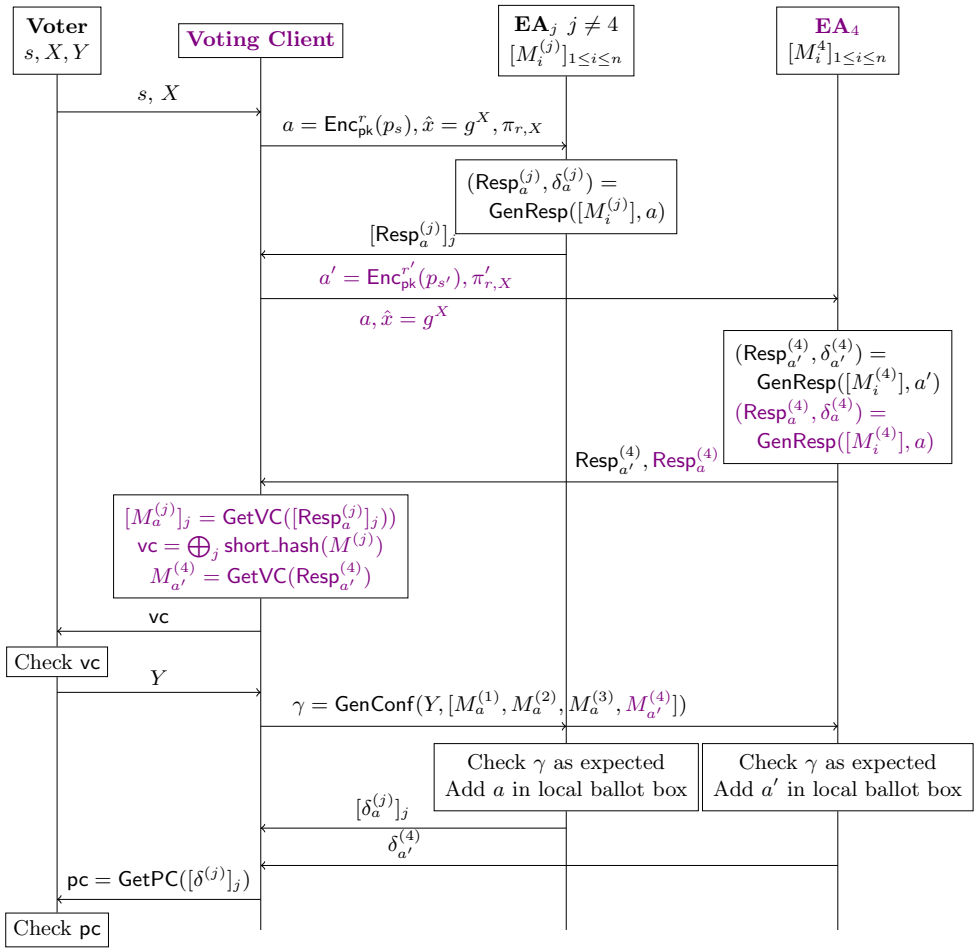


Fig. 3. Attack against individual verifiability (recorded-as-intended). We arbitrarily assign the dishonest authority to EA_4. Dishonest steps are written in violet.

4.2 Full vote manipulation, due to a missing check

There is actually a simpler attack, due to the implementation bug explained in Section 3.5. Indeed, since the shuffle performed by the first EA is never verified, EA_1 does not even need to collaborate with corrupted voting devices to change the votes. The attack works as follows:

- during the voting phase, EA_1 behaves normally,
- during the tally, instead of shuffling, EA_1 produces a novel list of ElGamal ciphertexts, for candidates of their choice, with an empty proof of shuffling,
- this manipulation is undetected by the other three (honest) EAs.

This attack will be detected, after the publication of the result, by the Verifier. However, the Chancellery does not assume an honest Verifier for the recorded-as-intended property. This attack is very easy to fix, by forcing the EAs to check all the proofs of shuffle.

4.3 Ballot stuffing

Each voter needs two secret credentials X and Y to cast a vote. The EAs have collectively computed these secret credentials and stored the corresponding public credentials $\hat{x} = g^X$ and $\hat{y} = g^Y$. Obviously, if they are all compromised, they can cast a vote on behalf of a voter using X and Y. Such an attack is outside the security model in Switzerland because eligibility is part of a larger set of properties (called universal verifiability) and hence they assume that at least one EA is honest. We believe that this assumption is too strong for eligibility and can be avoided, as it is done in the SwissPost protocol [27].

In the rest of this section, we show that actually, the online components can break eligibility, that is, can add ballots, even if the (offline) setup phase is run honestly and the secret shares of X and Y are erased. This leads to a more powerful attack since online components are more vulnerable to attacks. Even if outside the official security model, we believe that such a weakness should be avoided. Moreover, our attacks reveal some missing checks in the verification algorithm, that could be easily added.

Our attack assumes a dishonest voter and all dishonest EAs. Given the credentials X and Y of one dishonest voter, the EAs can forge several ballots $b_1, \ldots, b_k$ corresponding to this voter. When providing the audit data to the Verifier, the EA are supposed to give the list of public credentials $\hat{x_1}, \ldots, \hat{x_n}$ and $\hat{y_1}, \ldots, \hat{y_n}$ of the n voters. Instead, they can replace the credentials of absentee voters (say $\hat{x_1}, \ldots, \hat{x_k}$) by $\hat{x}, \ldots, \hat{x}$ replicated k times and similarly for the credentials $\hat{y}$. They can then add $b_1, \ldots, b_k$ as the first k ballots of the election.

This attack is not caught by the Verifier due to two shortcomings:

- The online EAs share their views of the protocol with the Verifier only once, at the very end of the protocol (see [20, Section 7.5]). Instead, authorities should commit to their data as soon as possible: once after the setup (with the public credentials), once after the voting phase, and once after tally.
- The Verifier does not check that the credentials are pairwise distinct (see [20, Algorithm 8.54]).

5 Experiments

The CHVote designers provide OpenCHVote [3], a reference implementation of the protocol. For our experiments, we used the latest release 2.3.1. This project also contains a simulator to easily set-up test elections and run experiments. We leveraged this implementation to demonstrate the validity of our attacks.

OpenCHVote. OpenCHVote is a Java project organised as a multimodule Maven project to ease its compilation. Even if it is a large project made of dozens of thousands of lines of code, it was easy to navigate between the different modules, packages, files, and identify the key classes that implement the algorithms of interest. Indeed, the code is perfectly aligned with the protocol specification [20]: there is one module per role, one class per task (i.e. atomic step of the protocol identified in the flow charts), and one class per algorithm or sub-algorithm.

Instrumentation. We instrumented the source code to test the validity of our main attacks, i.e. Sections 3.1, 3.2, 4.1, 4.2. The instrumented source code is available at [5] for reproducibility. Our changes are twofold: first, we implemented the Byzantine behaviours of the dishonest parties in the different attack scenarios as variants of the different algorithms and sub-algorithms. Second, we relaxed the checks done by the different parties to prevent early aborts in attack scenarios for which the attack is known to be detectable. Instead of aborting, we extended the output of the simulator to precisely identify which check fails. Finally, we provide a script that can be used to run our different attacks and observe the behaviour of the system. We also observed that fixing the off-by-one bug made the corresponding attacks disappear.

Results. We managed to reproduce the attacks and confirm the undetectability of the claimed ones. The compilation and the simulations run in a few minutes on a standard laptop (i.e. Apple M2 Pro with 32GB RAM). Computationally-wise, the most critical parts lie in the privacy attack described in Section 3.2. Indeed, this requires to compute a collision for the $\mathsf{short\_hash}(\cdot)$ function. In our experiments, the code performs about 10,000 iterations per second, where an iteration prepares a sequence of $[M_i]_i$ and computes the $\mathsf{short\_hash}$. The attack based on second preimage requires about 65500 iterations, hence 6.5s, (avg. on 10,000 tests) which remains reasonable to be computed on the fly. As expected, looking for a collision is more efficient and requires only 319 iterations, hence 0.032s, (avg. on 10,000 tests). These numbers are consistent with the theoretical estimates based on the security parameter: this last was set to `Level_1` during our experiments meaning that the participation code pc was a 5-digits long code (i.e. belongs to $\{0, \ldots, 9\}^5$). This would correspond to an entropy of 16.66 bits, but due to byte-level rounding in the implementation, this is just 16 bits, which is consistent with our experimental measurements.

6 Discussion

We have shown a variety of serious attacks against CHVote. According to the bug bounty program [4] organized for the Swiss Post protocol (which, of course, does not apply to CHVote), our two attacks against recorded-as-intended would have been rewarded between 50 to 70 kE each, while our family of attacks against vote secrecy would be valued beween 40 to 50kE. Some of our attacks are easily fixable, others require more in-depth modifications that we discuss here.

Missing agreement procedure. As shown in Section 4.1, the system may reach a state in which the different EAs do not agree on the list of the ballots to be included in the tally. Since the views of each EA look like honest ones, it seems impossible to define an agreement procedure to solve this inconsistency without changing the protocol. SwissPost encountered the exact same problem in 2022 and updated their protocol to solve it. A similar idea could be re-used in CHVote. Informally, the fix consists in adding a round of "commitments" during the voting phase to ensure that the EAs agree on the ballot that will be eventually counted during the tally for each voter. Then an EA can convince a third-party that a ballot must be included in the tally by proving the knowledge of a commitment from all EAs. Of course, such a change in the protocol deserves a formal analysis of its effectiveness and impact on the security of the protocol.

Fixing oblivious transfer. We already discussed some possible mitigations to make our attack detectable for vote privacy, although it does not suffice when all EAs are dishonest. However, a detectable attack against vote secrecy remains an attack: if voters complain when they receive an incorrect verification code, their vote may be leaked to a malicious EA, which is not acceptable in the current threat model. Note that CHVote does not provide a mechanism to (provably) identify the misbehaving entity and this is a tricky issue since a voter may also make mistakes or intentionally complain without any reasons.
The discovered vulnerability on CHVote is known as a *selective failure attack* in OT literature [10]. Fixing the protocol would require an in-depth modification of the protocol. The first idea is to use *committed oblivious transfer* [12,13] so that an EA cannot later modify their M_j. The idea of committed oblivious transfer is that the sender (each EA in our protocol) commits their secrets $\mathsf{commit}(M_1, r_1), \ldots, \mathsf{commit}(M_n, r_n)$, the receiver (the voter) commits to their choice i with $\mathsf{commit}(i, r)$ and at the end of the protocol, it obtains a commitment $\mathsf{commit}(M_i, u)$ with the opening u, where each party proves that they behaved as expected. The first proposals by Crépeau [12,13] were inefficient but several more efficient protocols have been proposed later on [18,23,22]. However, the setting in evoting is particular since the voter is not present during the setup. For example, [22,23] assume that the receiver (i.e. voter here) has a private key, which may be difficult to establish in the evoting context. Moreover, since the voters may choose k-out-of-n candidates, the 1-out-of-2 oblivious transfer protocol needs to be turned into k-out-of-n [25], as already done in CHVote but not for committed oblivious transfer. In addition, there is not a single sender but it needs to be distributed over the 4 Election authorities, among which 3 are fully dishonest (and not semi-dishonest as incorrectly assumed in CHVote [11]). To summarize, all the needed building blocks seem to be available in the literature but the design of an OT protocol suitable in the evoting context will require a careful design and may affect the overall efficiency of the protocol.

Security proof. The OT primitive used in CHVote has been modified after a first attack against cast-as-intended [9]. Our attack against recorded-as-cast shows that the verifiability proof published in 2018 [8] does not cover the agreement

phase that should tell which ballots are tallied. Indeed, the proof assumes that the honest EA is known *a priori* by the system. More importantly, no privacy proof has ever been conducted on CHVote, while this protocol uses an OT primitive in a crafted way (not blackbox). For example, it embeds an ElGamal encryption inside the OT. Moreover, it extends the OT with subsequent exchanges (confirmation code and participation code) that have an impact on the security. For example, our vote secrecy attack could be detectable with longer codes. Therefore, a proof of vote secrecy appears to be critical to assess the security of future versions of CHVote. A challenging aspect of the proof will be to identify if the usual security property of OT (here receiver privacy) is indeed sufficient for vote secrecy as a whole.

References

1. E-voting system - CHVote - Canton de Genève. `https://republique-et-canton-de-geneve.github.io/chvote-1-0/index-fr.html`. Last visited on Dec. 2024.
2. Point presse du conseil d'etat du 28 novembre 2018. `https://www.ge.ch/document/point-presse-du-conseil-etat-du-28-novembre-2018#extrait-12897`, 2018.
3. OpenCHVote. Version 2.3.1. `https://gitlab.com/openchvote`, 2024.
4. Policy for the Swiss Post E-Voting public bug bounty programme. `https://yeswehack.com/programs/swiss-post-evoting`, 2024.
5. Source code artefact for reproducibility. `https://homepages.loria.fr/PGaudry/openchvote-2.3.1-attacks.zip`, 2025. Instrumented version of OpenCHVote 2.3.1.
6. J. Benaloh. *Verifiable secret-ballot elections*. PhD thesis, Yale University, 1987.
7. David Bernhard, Veronique Cortier, David Galindo, Olivier Pereira, and Bogdan Warinschi. A comprehensive analysis of game-based ballot privacy definitions. In *36th IEEE Symposium on Security and Privacy (S&P'15)*, pages 499–516, May 2015.
8. David Bernhard, Véronique Cortier, Pierrick Gaudry, Mathieu Turuani, and Bogdan Warinschi. Verifiability analysis of CHVote. Cryptology ePrint Archive, Paper 2018/1052, 2018.
9. Achim Brelle and Tomasz Truderung. Cast-as-intended mechanism with return codes based on PETs. In *E-Vote-ID 2017*, Lecture Notes in Computer Science, 2017.
10. Jan Camenisch, Gregory Neven, and Abhi Shelat. Simulatable adaptive oblivious transfer. In Moni Naor, editor, *Advances in Cryptology - EUROCRYPT 2007*. Springer, 2007.
11. Cheng-Kang Chu and Wen-Guey Tzeng. Efficient k-out-of-n oblivious transfer schemes with adaptive and non-adaptive queries. In *Public Key Cryptography - PKC 2005*, 2005.
12. Claude Crépeau. Verifiable disclosure of secrets and applications (abstract). In *Advances in Cryptology - EUROCRYPT '89*, Lecture Notes in Computer Science, 1989.
13. Claude Crépeau, Jeroen van de Graaf, and Alain Tapp. Committed oblivious transfer and private multi-party computation. In *Advances in Cryptology - CRYPTO'95*, Lecture Notes in Computer Science, 1995.

14. Chris Culnane, Aleksander Essex, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. Knights and knaves run elections: Internet voting and undetectable electoral fraud. *IEEE Secur. Priv.*, 2019.
15. Alexandre Debant and Lucca Hirschi. Reversing, breaking, and fixing the French legislative election e-voting protocol. In *USENIX Security Symposium 2023*, 2023.
16. Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols. *Journal of Computer Security*, 17(4):435–487, 2009.
17. Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.
18. Juan A. Garay. Efficient and universally composable committed oblivious transfer and applications. In *First Theory of Cryptography Conference, TCC 2004*, Lecture Notes in Computer Science, 2004.
19. Rolf Haenni, Reto E. Koenig, and Eric Dubuis. Cast-as-intended verification in electronic elections based on oblivious transfer. In *Electronic Voting - First International Joint Conference, E-Vote-ID 2016*, Lecture Notes in Computer Science, 2016.
20. Rolf Haenni, Reto E. Koenig, Philipp Locher, and Eric Dubuis. CHVote protocol specification. Cryptology ePrint Archive, Paper 2017/325, 2017. Version 4.3, December 13, 2024.
21. J. Alex Halderman and Vanessa Teague. The New South Wales iVote system: Security failures and verification flaws in a live online election. In *E-Voting and Identity - 5th International Conference, VoteID 2015*, LNCS. Springer, 2015.
22. Stanislaw Jarecki and Vitaly Shmatikov. Efficient two-party secure computation on committed inputs. In *Advances in Cryptology - EUROCRYPT 2007*, Lecture Notes in Computer Science, 2007.
23. Mehmet S. Kiraz, Berry Schoenmakers, and José Villegas. Efficient committed oblivious transfer of bit strings. In *Information Security, 10th International Conference, ISC 2007*, Lecture Notes in Computer Science, 2007.
24. Johannes Mueller. Breaking and fixing vote privacy of the Estonian e-voting protocol IVXV. In *Workshop on Advances in Secure Electronic Voting*, 2022.
25. Moni Naor and Benny Pinkas. Oblivious transfer with adaptive queries. In *Advances in Cryptology - CRYPTO '99*, Lecture Notes in Computer Science, 1999.
26. Swiss Federal Chancellery. Federal Chancellery Ordinance on Electronic Voting (OEV). `https://www.fedlex.admin.ch/eli/cc/2022/336/en`, 2022.
27. Swiss Post. Swiss Post voting system: System specification. version 1.4.0. Technical report, Swiss Post, February 2024.
28. Scott Wolchok, Eric Wustrow, Dawn Isabel, and J. Alex Halderman. Attacking the Washington, D.C. internet voting system. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security FC 2012, Kralendijk*. Springer, 2012.

Conflict of interest

The authors are collaborating with SwissPost through research contracts between the company and their (academic) research laboratory. SwissPost is a Swiss company developing and selling the evoting system currently in use in Switzerland for local and federal elections. This work has been performed outside the time dedicated to these contracts; the results, views and opinions presented in the paper are those of the authors alone.

A Privacy attack

We provide a detailed description of our attack in Figure 4.

B Recorded-as-intended attack for an arbitrary number k of selections

For the sake of clarity, we have presented CHVote in the simple case where voters select $k = 1$ candidate among n possible choices. We now explain the general case, with an arbitrary number of selections $k \leq n$. The corresponding algorithms, in a simplified version, are displayed in Figure 5. This leaves our attack against recorded-as-intended, presented in Section 4.1, unchanged.

Setup. Each EA generates partial verification codes $[M_i^{(j)}]_{1 \leq i \leq n}$ as follows: for each voter they first pick some random polynomial $A^{(j)}$ of degree $k-1$ and then generate n random points $M_i^{(j)} = (x_i, y_i)$ where x_i is a random value and $y_i = A^{(j)}(x_i)$. This corresponds to Algorithm GenPoints (Algorithm 8.11 of [20]).

The EAs also collectively compute $Z_v = \prod_j g^{A^{(j)}(0)}$. It is used during the confirmation phase to check that it corresponds to the second component Z of γ.

Confirmation phase During the confirmation phase, the voting client obtains k partial verification codes, hence k points of each polynomial $A^{(j)}$. It then uses a Lagrange interpolation to compute $A^{(j)}(0)$ and can deduce $Z = g^{\sum_j A^{(j)}(0)}$. The resulting GenConf algorithm is displayed in Figure 5, it corresponds to Algorithms 8.32 and 8.33 of [20].

The interesting point for our attack is that $A^{(j)}(0)$ does not depend on the exact points received during the first step of the voting phase. In particular $A^{(j)}(0)$ remains identical whether it has been computed thanks to the ballot a or a' (using the notations of Section 4.1).

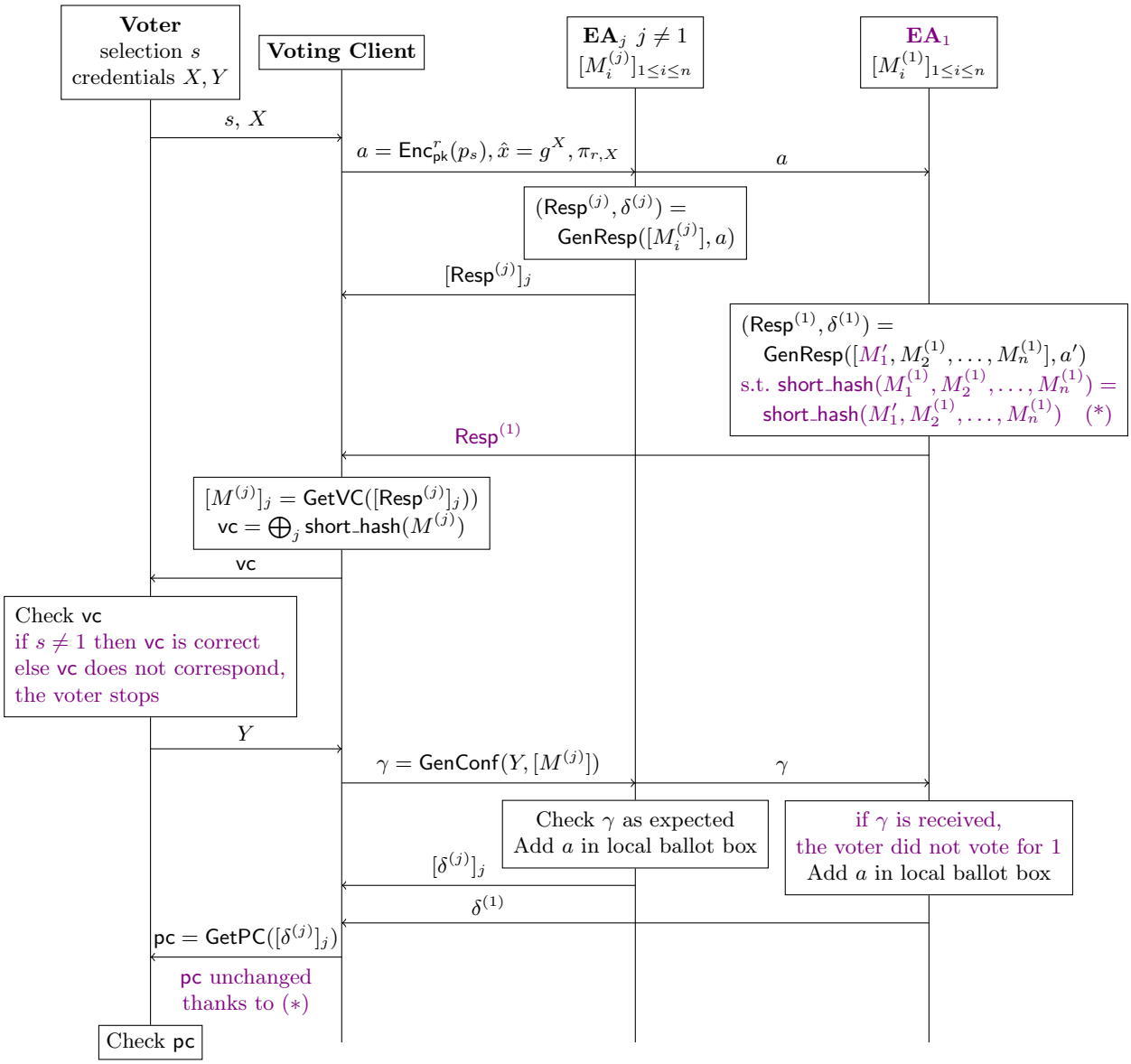


Fig. 4. Privacy attack by (non) complaint, undetectable.

GenPoints(n, k)

(run by the j-th EA *)*

Pick $A^{(j)} \in_R \mathbb{Z}_q[X]$ random polynomial of degree $k-1$

For $i \in [1, n]$:

$x_i \in_R \mathbb{Z}_q$

$y_i := A^{(j)}(x_i)$

$M_i^{(j)} := (x_i, y_i)$

Return $[M_i^{(j)}]_j$ and $y_0 := A^{(j)}(0)$

GenConf$(Y, [M^{(j)}]_j)$

(run by Voting Client *)*

$\hat{y} := g^Y$

For all j:

See the k codes $M_i^{(j)}$ as point (x_i, y_i)

$$z^{(j)} := \sum_{i=1}^{k} y_i \prod_{j=1, j\neq i}^{k} \frac{x_j}{x_j - x_i}$$

(ie $z^{(j)} := A^{(j)}(0)$ *)*

$Z := \sum_j z^{(j)}; \quad \hat{z} := g^Z$

$\pi_{Y,Z}$:= pok of Y and Z

Return $\gamma = (\hat{y}, \hat{z}, \pi_{Y,Z})$

Fig. 5. Generation of the partial verification codes and of the confirmation response for an arbitrary number k of selections.