One vote is enough for analysing privacy*

Stéphanie Delaune and Joseph Lallemand

Univ Rennes, CNRS, IRISA, France

Abstract. Electronic voting promises the possibility of convenient and efficient systems for recording and tallying votes in an election. To be widely adopted, ensuring the security of the cryptographic protocols used in e-voting is of paramount importance. However, the security analysis of this type of protocols raises a number of challenges, and they are often out of reach of existing verification tools.
In this paper, we study *vote privacy*, a central security property that should be satisfied by any e-voting system. More precisely, we propose the first formalisation of the recent BPRIV notion in the symbolic setting. To ease the formal security analysis of this notion, we propose a reduction result allowing one to bound the number of voters and ballots needed to mount an attack. Our result applies on a number of case studies including several versions of Helios, Belenios, JCJ/Civitas, and Prêt-à-Voter. For some of these protocols, thanks to our result, we are able to conduct the analysis relying on the automatic tool Proverif.

1 Introduction

Remote electronic voting systems aim at allowing the organisation of elections over the Internet, while providing the same guarantees as traditional paper voting. Although relying on e-voting for large-scale elections is controversial, it is already in use in many lower-stakes elections today (*e.g.* the Helios [3] voting system has been used to elect the IACR board of directors since 2010), and is likely to be used even more in the future, for better or for worse. These elections may involve a large number of voters and may have an important impact on democracy when it comes to elect political leaders. It is therefore of paramount importance to ensure the security of these systems.

As for security protocols in general, formal methods provide powerful techniques to analyse e-voting systems, and prove their security. Identifying what makes a good, secure e-voting system is a complex problem that has not yet been completely solved, and is actively being researched. It is however rather universally acknowledged that a central security guarantee e-voting systems should provide is *vote privacy*. Intuitively, this property states that votes must remain secret, so that no one can learn who voted for which candidate.

* This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement n° 714955-POPSTAR), as well as from the French National Research Agency (ANR), under project TECAP, and under the France 2030 programme with reference ANR-22-PECY-0006.

One common way of formalising vote privacy, which we will call SWAP, is to require that an attacker is not able to distinguish between the situation where Alice is voting *yes* and Bob is voting *no* from the situation where the two voters swapped their vote. That formalisation was first proposed by Benaloh [9], originally in a computational model. It has since been adapted to the symbolic setting [26], and applied to many voting schemes, *e.g.* [5,23,4,24,20,7]. The SWAP notion was originally written considering the specific case of a referendum, where the result is the number of *yes* and *no* votes. It has then been generalised to cover other kinds of elections [8], but remains limited w.r.t. the way of counting votes – essentially, it only makes sense when the result of the election is the number of votes for each candidate, excluding more complex counting procedures such as Single Transferable Vote (STV).

More recently, a new definition, called BPRIV for "ballot privacy", has been proposed to overcome such limitations [10]. Essentially, BPRIV lets the attacker interact with the system, and see either real ballots, or fake ones containing fake votes. Using oracles, he can choose the values of real and fake votes, and cast any ballot he can construct (in the name of corrupted voters). In the end, the tally of real ballots is published. To be BPRIV, the attacker should be unable to distinguish the two scenarios, *i.e.* no information is leaked on the ballots' content.

Privacy-type properties, and in particular vote privacy, are often expressed using a notion of behavioural equivalence [25]. A notable exception is the definition of (α, β)-privacy [31] which nevertheless relies on some notion of static equivalence. Proving equivalences is cumbersome, and is difficult to do in details by hand, as witnessed by the manual analysis of the SWAP property done for *e.g.* the Helios protocol [23] and the Norwegian one [24]. Regarding mechanisation, several mature tools are available for analysing trace properties such as secrecy or authentication in the symbolic setting: most notably, Proverif [11,12] and Tamarin [30]. These tools support equivalence properties [13,6], although they remain limited to a restricted form of equivalence, called *diff-equivalence*. Some e-voting schemes have been analysed with these automated tools in the symbolic model, *e.g.* the Neuchâtel [20] or BeleniosVS [19] protocols. Proverif even has an extension called ProSwapper [14], that specifically handles swapped branches that typically occur in the SWAP definition. These tools have proved very helpful for the study of e-voting systems. However, they still suffer from limitations that restrict their applicability, as they *e.g.* cannot handle homomorphic encryption, or manipulate lists of arbitrary size to encode the bulletin board, and tend to quickly run into performance issues when the number of agents in parallel increases.

An interesting option to ease the security analysis is to rely on reduction results. This approach has been used to bound the number of agents involved in an attack for both reachability [17], and equivalence properties [18]. Reduction results bounding the number of sessions [28,27] have also been proposed in more restricted settings. All these results do not apply in the context of e-voting protocols. Here, we would like to bound the number of voters (agents) participating in the election. However, since only one vote is counted for each voter, we can *not* replace a session played by A by one played by B, as was

done *e.g.* in [18]. The only existing result in that context is the result proposed in [4], where the authors give bounds on the number of voters and ballots – respectively 3 and 10 – needed for an attack on the SWAP notion This allows them to carry out several case studies using Proverif. No such results, however, exist for the newer and more general BPRIV definition.

Contributions. Our contributions are threefold. First, we propose a definition of BPRIV adapted for the symbolic model. BPRIV has been first introduced in the computational setting where some subtleties regarding the communication model have been overlooked. Second, we identify some conditions under which BPRIV can be analysed considering only *one* honest voter and k dishonest ones. Actually, in most usual cases, we have $k = 1$, and the number of ballots being tallied is reduced to 1. These reduction results are generic, in particular we do not assume anything regarding the equational theory,and our result applies for different counting functions. Revoting is also allowed. Finally, we apply our result on several e-voting protocols from the literature relying on the tool Proverif. Our bounds for BPRIV, better than those obtained in [4] when considering SWAP, allow us to analyse many protocols in a reasonable time (whereas several hours were needed in some cases in [4]). We also identify an issue in the security analysis performed in [4] where a protocol has been declared secure while it is not.

2 Modelling security protocols

In this section, we introduce background notions on protocol modelling. We model security protocols in the symbolic model with a process algebra inspired from the applied pi-calculus [2]. Our model is mostly standard, except that in order to model the stateful nature of e-voting protocols, we consider memory cells, that can store a persistent state across processes. We need to avoid concurrent accesses to memory cells while updating them: to that end, we use a specific instruction that atomically appends a message to the content of a memory cell.

2.1 Messages

We assume an infinite set $\mathcal{N}$ of *names* used to model keys, nonces, *etc.* We consider two infinite and disjoint sets of *variables* $\mathcal{X}$ and $\mathcal{W}$. Variables in $\mathcal{X}$ are used to refer *e.g.* to input messages, and variables in $\mathcal{W}$, called *handles*, are used as pointers to messages learned by the attacker. Lastly, we consider two disjoint sets of constant symbols, denoted Σ_0 and Σ_{err}. Constants in Σ_0 represent public values, *e.g.* identities, nonces or keys drawn by the attacker. This set is assumed to be infinite. Constants in Σ_{err} will typically refer to error messages. We fix a *signature* Σ consisting of a finite set of function symbols together with their arity. We denote $\Sigma^+ = \Sigma \uplus \Sigma_0 \uplus \Sigma_{\mathsf{err}}$. We note $\mathcal{T}(\mathcal{F}, \mathcal{D})$ the set of terms built from elements in $\mathcal{D}$ by applying function symbols in the signature $\mathcal{F}$. The set of names (resp. variables) occurring in a term t is denoted $\mathsf{names}(t)$ (resp. $\mathsf{var}(t)$). A term t is *ground* if $\mathsf{var}(t) = \emptyset$. We refer to elements of $\mathcal{T}(\Sigma^+, \mathcal{N})$ as *messages*.

Example 1. We consider the signature $\Sigma_{\mathsf{err}} = \{\mathsf{err}_{\mathsf{vote}}, \mathsf{err}_{\mathsf{invalid}}\}$ to model error messages. The signature $\Sigma_{\mathsf{list}} = \{\mathsf{nil}, \mathsf{hd}, \mathsf{tl}, ::\}$ allows us to model lists of arbitrary size. We often write $[t_1, \ldots, t_n]$ for $t_1 :: \cdots :: t_n :: \mathsf{nil}$. The operators hd and tl are used to retrieve the head and the tail of a list. Lastly, we consider $\Sigma_{\mathsf{ex}} = \{\mathsf{aenc}, \mathsf{adec}, \mathsf{pk}, \mathsf{zkp}, \mathsf{check}_{\mathsf{zkp}}, \mathsf{true}, \langle\,\rangle_3, \mathsf{proj}_1^3, \mathsf{proj}_2^3, \mathsf{proj}_3^3, \mathsf{yes}, \mathsf{no}\}$ to model asymmetric encryption, zero-knowledge proofs, and pairing operators. As a running example, we will consider a model of the Helios protocol (in its original version, as seen in [23]) and $\Sigma_{\mathsf{Helios}} = \Sigma_{\mathsf{ex}} \cup \Sigma_{\mathsf{list}}$.

Let $\mathsf{id}_{\mathsf{H}} \in \Sigma_0$, $r, sk \in \mathcal{N}$, and $pk = \mathsf{pk}(sk)$. Intuitively, id_{H} represents the identity of a honest voter, and yes her vote (these data are known to the attacker), whereas r and sk are private names, modelling respectively the randomness used in the encryption and the private key of the authority. Let $e_{\mathsf{yes}} = \mathsf{aenc}(\mathsf{yes}, pk, r)$, and $b_{\mathsf{yes}}^{\mathsf{id}_{\mathsf{H}}} = \langle \mathsf{id}_{\mathsf{H}}, e_{\mathsf{yes}}, \mathsf{zkp}(e_{\mathsf{yes}}, \mathsf{yes}, r, pk)\rangle_3$. The first term encrypts the vote, and the second one is the ballot sent by the voter in the voting phase of Helios.

An element of $\mathcal{T}(\Sigma^+, \mathcal{W})$ is called a *recipe* and models a computation performed by the attacker using his knowledge. A *substitution* σ is a mapping from variables to messages, and $t\sigma$ is the application of σ to term t, which consists in replacing each variable x in t with $\sigma(x)$. A *frame* ϕ is a substitution that maps variables from $\mathcal{W}$ to messages, and is used to store an attacker's knowledge.

In order to give a meaning to function symbols, we equip terms with an *equational theory*. We assume a set E of equations over $\mathcal{T}(\Sigma, \mathcal{X})$, and define $=_{\mathsf{E}}$ as the smallest congruence containing E that is closed under substitutions.

Example 2. Continuing Example 1, we consider the equational theory E_{ex} given below and $\mathsf{E}_{\mathsf{list}} := \{\mathsf{hd}(x :: y) = x,\ \mathsf{tl}(x :: y) = y\}$.

$$\mathsf{E}_{\mathsf{ex}} = \left\{ \begin{array}{c} \mathsf{adec}(\mathsf{aenc}(x, \mathsf{pk}(y), z), y) = x \quad \mathsf{proj}_i^3(\langle x_1, x_2, x_3\rangle_3) = x_i \quad \text{with } i \in \{1,2,3\} \\ \mathsf{check}_{\mathsf{zkp}}(\mathsf{zkp}(\mathsf{aenc}(x, y, z), x, z, y), \mathsf{aenc}(x, y, z), y) \ = \ \mathsf{true} \end{array} \right\}$$

We have $\mathsf{adec}(e_{\mathsf{yes}}, sk) =_{\mathsf{E}_{\mathsf{ex}}} v$, and $\mathsf{check}_{\mathsf{zkp}}(\mathsf{proj}_3^3(b_{\mathsf{yes}}^{\mathsf{id}_{\mathsf{H}}}), v, r, pk) =_{\mathsf{E}_{\mathsf{ex}}} \mathsf{true}$.

In the following, we consider an arbitrary signature $\Sigma^+ = \Sigma \uplus \Sigma_0 \cup \Sigma_{\mathsf{err}}$ together with its equational theory E (equations built over Σ only), and we assume it contains at least the formalisation of lists given in Example 1 and Example 2, *i.e.* $\Sigma_{\mathsf{list}} \subseteq \Sigma$ and $\mathsf{E}_{\mathsf{list}} \subseteq \mathsf{E}$.

2.2 Processes

We model protocols using a process calculus. We consider an infinite set of channel names $\mathcal{C}\mathsf{h} = \mathcal{C}\mathsf{h}_{\mathsf{pub}} \uplus \mathcal{C}\mathsf{h}_{\mathsf{pri}}$, partitioned into infinite sets of public and private channel names. We also assume an infinite set $\mathcal{M}$ of names to represent memory cells (used to store states). The syntax of processes is:

$$\begin{array}{lll} P, Q ::= 0 & \mid \mathsf{out}(c, u).\ P & \mid m := u.\ P \\ \quad \mid P \mid Q & \mid \mathsf{in}(c, x).\ P & \mid \mathsf{read}\ m\ \mathsf{as}\ x.\ P \\ \quad \mid !\,P & \mid !\,\mathsf{new}\ d.\ \mathsf{out}(c, d).\ P & \mid \mathsf{append}(c, u, m).\ P \\ \quad \mid \mathsf{new}\ n.\ P & \mid \mathsf{let}\ x = u\ \mathsf{in}\ P & \mid \mathsf{phase}\ i.\ P \\ \quad \mid \mathsf{new}\ d.\ P & \mid \mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q & \end{array}$$

where $n \in \mathcal{N}$, $x \in \mathcal{X}$, $m \in \mathcal{M}$, $u \in \mathcal{T}(\Sigma^+, \mathcal{X} \cup \mathcal{N})$, $d \in \mathcal{C}\mathsf{h}_{\mathsf{pri}}$, $c \in \mathcal{C}\mathsf{h}$, $i \in \mathbb{N}$.

This syntax is rather standard, except for the memory cell operations. Intuitively, $\mathsf{read}\ m\ \mathsf{as}\ x$ stores the content of m in the variable x, whereas $\mathsf{append}(c, u, m)$ represents the agent with channel c appending u to memory m. In addition, we use a special construct $!\,\mathsf{new}\ d.\ \mathsf{out}(c, d).\ P$, to generate as many times as needed a new public channel d and link it to channel c, in a single step. This could be encoded using the other instructions, but having a separate construction lets us mark it in the execution traces, which is convenient for the proofs. The constructs $\mathsf{in}(c, x).P$, $\mathsf{let}\ x = u\ \mathsf{in}; P$, and $\mathsf{read}\ m\ \mathsf{as}\ x.\ P$ bind x in P. Given a process P, $fv(P)$ denotes its free variables, and we say that it is *ground* when $fv(P) = \emptyset$. Moreover, we usually omit the final $\mathsf{0}$ in processes.

Example 3. Continuing our running example, we consider the process P:

$$P = \mathsf{in}(c, b).\ \mathsf{if}\ \langle \mathsf{check}_{\mathsf{zkp}}(\mathsf{proj}_3^3(b), \mathsf{proj}_2^3(b), \mathsf{pk}(sk)), \mathsf{proj}_1^3(b)\rangle = \langle \mathsf{true}, \mathsf{id}_\mathsf{D}\rangle \\ \mathsf{then}\ \mathsf{out}(c, b).\ \mathsf{append}(c, b, m_{\mathsf{bb}})\ \mathsf{else}\ \mathsf{out}(b, \mathsf{err}_{\mathsf{invalid}}).$$

where $b \in \mathcal{X}$, $sk \in \mathcal{N}$, and $\mathsf{id}_\mathsf{D} \in \Sigma_0$. This represents an agent that receives a ballot b as input, and then checks the validity of the zero knowledge proof contained in b, as well as the identity of the voter. Depending on the outcome of this test, it either outputs the ballot and appends it in the cell m_{bb} modelling the ballot box, or simply outputs an error message.

Definition 1. *A* configuration *is a tuple $(i; \mathcal{P}; \phi; M)$, composed of an integer i, a multiset $\mathcal{P}$ of ground processes, a frame ϕ, and a mapping M from a subset of memory names $\mathcal{M}$ to messages. We write $\mathcal{P}$ instead of $(0; \mathcal{P}; \emptyset; \emptyset)$.*

The semantics of our calculus is defined as a transition relation $\overset{a}{\Longrightarrow}$ on configurations. Each transition step is labelled with an action a representing what the attacker can observe when performing it (it can be an input, an output, an append action, or a silent action ϵ). This relation is defined in a standard manner. As a sample, depicted below are the rules for input, errors, and append.

$$(i; \{\!|\mathsf{in}(c, u).\ P|\!\} \cup \mathcal{P}; \phi; M) \xrightarrow{\mathsf{in}(c,R)} (i; \{\!|P\sigma|\!\} \cup \mathcal{P}; \phi; M) \\ \text{if } c \in \mathcal{C}\mathsf{h}_{\mathsf{pub}}, \text{ and } R \text{ is a recipe such that } \mathsf{var}(R) \subseteq \mathsf{dom}\phi \\ \text{and } R\phi =_\mathsf{E} u\sigma \text{ for some } \sigma \text{ with } \mathsf{dom}(\sigma) = \mathsf{var}(u)$$

$$(i; \{\!|\mathsf{out}(c, \mathsf{c}_{\mathsf{err}}).\ P|\!\} \cup \mathcal{P}; \phi; M) \xrightarrow{\mathsf{out}(c,\mathsf{c}_{\mathsf{err}})} (i; \{\!|P|\!\} \cup \mathcal{P}; \phi; M) \ \text{ if } c \in \mathcal{C}\mathsf{h}_{\mathsf{pub}},\ \mathsf{c}_{\mathsf{err}} \in \Sigma_{\mathsf{err}}$$

$$(i; \{\!|\mathsf{append}(c, u, m)|\!\}.\ P \cup \mathcal{P}; \phi; M) \xrightarrow{\mathsf{append}(c)} (i; \{\!|P|\!\} \cup \mathcal{P}; \phi; M\{m \mapsto u :: M(m)\}) \\ \text{if } m \in \mathsf{dom}(M)$$

For instance, considering an input on a public channel, the attacker can inject any message he is able to build using his current knowledge. The outputs performed on a public channel are made available to the attacker either directly through the label (when it corresponds to an error message), or indirectly through the frame (this rule is not shown). Lastly, we present the rule corresponding to our new append action $\mathsf{append}(c, u, m)$ which simply consists in appending a term u to the memory cell m. The full formal semantics is given in [1].

Definition 2. *The set of* traces *of a configuration K is defined as*

$$\mathsf{traces}(K) = \{(\mathsf{tr}, \phi) \mid \exists i, \mathcal{P}, M \text{ such that } K \overset{\mathsf{tr}}{\Longrightarrow}^{\star} (i; \mathcal{P}; \phi; M)\}$$

where $\overset{\cdot}{\Longrightarrow}^{\star}$ is the reflexive transitive closure of $\overset{\cdot}{\Longrightarrow}$, concatenating all (non-silent) actions into the sequence tr.

Example 4. Continuing Example 3 with $\phi_{\mathsf{yes}} = \{\mathsf{w}_0 \mapsto \mathsf{pk}(sk), \mathsf{w}_1 \mapsto b^{\mathsf{id_H}}_{\mathsf{yes}}\}$, and $K_0^{\mathsf{yes}} = (2; \{P\}; \phi_{\mathsf{yes}}; \{m_{\mathsf{bb}} \mapsto \mathsf{nil}\})$. We have that:

$$K_0^{\mathsf{yes}} \xrightarrow{\mathsf{in}(c,\mathsf{w}_1).\mathsf{out}(c,\mathsf{err}_{\mathsf{invalid}})} (2; \emptyset; \{\mathsf{w}_0 \mapsto \mathsf{pk}(sk), \mathsf{w}_1 \mapsto b^{\mathsf{id_H}}_{\mathsf{yes}}\}; \{m_{\mathsf{bb}} \mapsto \mathsf{nil}\})$$

$$K_0^{\mathsf{yes}} \xrightarrow{\mathsf{in}(c,R_0).\mathsf{out}(c,\mathsf{w}_2).\mathsf{append}(c)} (2; \emptyset; \{\mathsf{w}_0 \mapsto \mathsf{pk}(sk), \mathsf{w}_1 \mapsto b^{\mathsf{id_H}}_{\mathsf{yes}}, \mathsf{w}_2 \mapsto b^{\mathsf{id_D}}_{\mathsf{yes}}\}; \{m_{\mathsf{bb}} \mapsto b\})$$

with $R_0 = \langle \mathsf{id_D}, \mathsf{proj}^3_2(\mathsf{w}_1), \mathsf{proj}^3_3(\mathsf{w}_1)\rangle_3$, and $b^{\mathsf{id_D}}_{\mathsf{yes}} = R_0\phi^{\mathsf{id_H}}_{\mathsf{yes}} =_{\mathsf{E}_{\mathsf{ex}}} \langle \mathsf{id_D}, e_{\mathsf{yes}}, zkp\rangle_3$. The term zkp here denotes the zero-knowledge proof from $b^{\mathsf{id_H}}_{\mathsf{yes}}$. It does not contain the identity of the voter who computes it, and can therefore be reused by a dishonest voter to cast the ballot in her own name.

2.3 Equivalences

Our definition of the BPRIV property relies on two usual notions of equivalence in the symbolic model: *static equivalence*, for the indistinguishability of sequences of messages, and *trace equivalence*, for the indistinguishability of processes.

Definition 3. *Two frames ϕ and ϕ' are* statically equivalent*, denoted by $\phi \sim \phi'$, if $\mathsf{dom}(\phi) = \mathsf{dom}(\phi')$ and for any recipes $R_1, R_2 \in \mathcal{T}(\Sigma^+, \mathsf{dom}(\phi))$, we have that $R_1\phi =_{\mathsf{E}} R_2\phi \Leftrightarrow R_1\phi' =_{\mathsf{E}} R_2\phi'$.*

Definition 4. *Two ground processes P, Q are in* trace inclusion*, denoted by $P \sqsubseteq_t Q$, if for all $(\mathsf{tr}, \phi) \in \mathsf{traces}(P)$, there exists ϕ' such that $(\mathsf{tr}, \phi') \in \mathsf{traces}(Q)$ and $\phi \sim \phi'$. We say that P and Q are* trace equivalent*, denoted by $P \approx_t Q$, if $P \sqsubseteq_t Q$ and $Q \sqsubseteq_t P$.*

Example 5. We can consider a configuration K_0^{no} similar to K_0^{yes} but with no instead of yes in the initial frame. We can establish that $K_0^{\mathsf{no}} \approx_t K_0^{\mathsf{yes}}$. This is a non trivial equivalence. Now, we replace P by P^+ in both configurations, adding a simple process modelling the tally (for one vote), *e.g.*

$$P^+ = P \mid \mathsf{phase}\ 3.\ \mathsf{read}\ m_{\mathsf{bb}}\ \mathsf{as}\ bb.\ \mathsf{let}\ res = \mathsf{adec}(\mathsf{proj}^3_2(bb), sk)\ \mathsf{in}\ \mathsf{out}(c_r, res).$$

We have that the resulting equivalence does *not* hold. This is simply due to the fact that $\mathsf{tr} = \mathsf{in}(c, R_0).\mathsf{out}(c, \mathsf{w}_2).\mathsf{append}(c).\mathsf{phase}\,3.\mathsf{out}(c_r, \mathsf{w}_3)$ can be executed starting from both configurations, and the resulting frames contains $\mathsf{w}_3 \mapsto \mathsf{no}$ on the left, and $\mathsf{w}_3 \mapsto \mathsf{yes}$ on the right. This breach of equivalence is not, strictly speaking, an attack, as the processes do not formalise the BPRIV property. However it follows the same idea as the ballot copy attack against Helios from [23]: a dishonest voter copies a honest voter's ballot, introducing an observable difference in the result. This attack can be prevented by patching Helios, either by weeding out duplicate ballots from the ballot box, or by adding the voter's *id* to the ZKP, which then becomes invalid for any other voter.

3 Modelling the general BPRIV notion

In this section, we present our formal model of e-voting protocols, and our BPRIV privacy notion. While BPRIV itself is not novel, our symbolic formalisation is.

3.1 Modelling e-voting protocols

When modelling voting systems, we often need to encode some computations (*e.g.* performed by the ballot box) that cannot be represented by recipes (*e.g.* iterating through an arbitrary-sized list). We encode these computations as processes, that do not share any names, channels, or memory cells with the rest of the process, except for a channel to return the result of the computation.

Definition 5. *A* computation *is a process $C_d(\vec{p})$ without free names, channels, or variables (not counting those in d, $\vec{p}$), without memory cell operations, and without phases. It is parametrised by a channel d, and terms $\vec{p}$, meant to be the channel where the result is output, and the terms given as input parameters.*

This process must be such that for all inputs $\vec{p}$, there exists a ground term t_0 such that for all channel name d, we have

$$\mathsf{traces}(C_d(\vec{p})) = \{(\epsilon, \emptyset)\} \cup \{(\mathsf{out}(d, \mathsf{w}), \{\mathsf{w} \mapsto t_0\}) \mid \mathsf{w} \in \mathcal{W}\}.$$

We then call t_0 the result *of the computation. As it does not depend on the channel, we will often omit it and let $C(\vec{p})$ denote the result.*

To use such a process to compute a term inside a process P, we will typically run it in parallel with an input waiting to retrieve the result on d, followed by the continuation process. We will write as a shortcut $\mathsf{let}\ x = C(\vec{p})\ \mathsf{in}\ P$ for $\mathsf{new}\ d.\ (C_d(\vec{p}) \mid \mathsf{in}(d, x).\ P)$, where d is a fresh private channel name (*i.e.* that does not appear anywhere else in the ambiant process).

We assume a set $\mathsf{Votes} \subseteq \mathcal{T}(\Sigma, \Sigma_0)$ of public ground terms representing the possible values of the votes. A voting system is modelled by a collection of processes that model the behaviour of voters, and a process modelling the tallying authority. The election process is composed of several phases.

Phases 0 and 1: Setup. The election material is generated and published.

Phase 2: Casting. The voters send their ballots to the ballot box. In our model, a memory m_{bb} will play the role of the ballot box, recording all ballots received by the voting server. The voters' processes will first publish their ballot on a dedicated public channel, and then append it to the memory cell m_{bb}. This models the fact that voters are authenticated when they submit their ballot, and the ballot cannot be modified on its way to the ballot box. However, the attacker is able to block a ballot before it reaches the ballot box.

Each voter has a private credential $cr \in \mathcal{N}$, with an associated public credential computed by a recipe $\mathsf{Pub}(cr, u)$, that may use a random value u. Some protocols, such as Civitas, use this value to randomise the public credential, while

others, such as Belenios, do not use it – in such cases we can omit it. To model the construction of ballots, we assume a recipe Vote with 5 variables: the term $\mathsf{Vote}(pk, id, cr, v, r)$ represents a ballot generated for voter id with credential cr, public election key pk, randomness r, and containing a vote v.

When modelling vote privacy, the attacker chooses the vote v he wants the voter to use to construct the ballot. Hence, we will need to check that v is indeed a possible value for a vote, *i.e.* $v \in \mathsf{Votes}$. If the set of candidates is finite, this can be tested exhaustively. In other cases, such as write-in votes, it can be done *e.g.* if legal votes have a specific format (start with a tag, *etc.*), or trivially if any value is legal. In a voting scheme, once a ballot is received by the voting server, another check is performed to ensure the ballot is *valid*, *i.e.* correctly constructed. Typically, it can consist in verifying signatures or zero-knowledge proofs included in the ballot. To keep our model generic, we simply assume a recipe Valid with four variables: $\mathsf{Valid}(id, pcr, b, pk)$ represents the validity test performed for the agent id, whose public credential is pcr, who submits a ballot b. The term it computes is meant to be equal to true if, and only if, ballot b cast by id is valid w.r.t. her public credential pcr and the election public key pk. We incorporate this validity check directly in the process modelling the voter, before publishing and adding the ballot to m_{bb}. In reality, it is performed by the ballot box, but this modelling choice is both simpler (no need for an extra process) and closer to the cryptographic game (where the voting oracle performs the test).

The formal definition of the voter's process is given in Section 3.2 as it incorporates elements specific to the modelling of the property.

Example 6. Continuing Example 2, for Helios, we use the following recipes:

$$\mathsf{Vote}_{\mathsf{Helios}}(pk, id, v, r) = \langle id, \mathsf{aenc}(v, pk, r), \mathsf{zkp}(\mathsf{aenc}(v, pk, r), v, r, pk)\rangle_3$$
$$\mathsf{Valid}_{\mathsf{Helios}}(id, b, pk) = \mathsf{check}_{\mathsf{zkp}}(\mathsf{proj}_3^3(b), \mathsf{proj}_2^3(b), pk).$$

Phase 3: Tallying. In the final phase, the $\mathsf{Tally}(sk)$ process is in charge of reading the contents of the ballot box, and using the key sk to compute and publish the result on a dedicated channel c_r. To leave it as generic as possible, we simply assume a computation $C_{\mathsf{Tally}}(bb, sk)$, that takes as parameters a list bb of ballots, and sk, and computes the result as specified by the protocol. We then assume the following form for Tally:

$$\mathsf{Tally}(sk) = \mathsf{read}\ m_{\mathsf{bb}}\ \mathsf{as}\ bb.\ \mathsf{let}\ res = C_{\mathsf{Tally}}(bb, sk)\ \mathsf{in}\ \mathsf{out}(c_r, res).$$

Example 7. We continue Example 6 and we consider for simplicity the case of a referendum with two possible votes yes and no. We assume function symbols $\mathsf{zero}/0$ and $\mathsf{incr}/1$, without any associated equations, that we use to count in unary. Slightly abusing notations with the use of pattern-matching in input, the tallying computation can be written as follows:

$$\begin{array}{l} C_{\mathsf{Tally}}(bb, sk) = \\ \quad \mathsf{new}\ c.\ \big(\ \mathsf{out}(c, \langle \mathsf{zero}, \mathsf{zero}, bb\rangle_3) \\ \qquad \mid\ \mathsf{in}(c, \langle x, y, \mathsf{nil}\rangle_3).\ \mathsf{out}(c_r, \langle x, y\rangle) \\ \qquad \mid\ !\ \mathsf{in}(c, \langle x, y, \langle id, b, p\rangle_3 :: l\rangle_3).\ \mathsf{let}\ v = \mathsf{adec}(b, sk)\ \mathsf{in} \\ \qquad\quad \mathsf{if}\ v = \mathsf{yes}\ \mathsf{then}\ \mathsf{out}(c, \langle \mathsf{incr}(x), y, l\rangle_3)\ \mathsf{else}\ \mathsf{out}(c, \langle x, \mathsf{incr}(y), l\rangle_3).\ \big) \end{array}$$

3.2 A symbolic definition of BPRIV

We model vote privacy by adapting the BPRIV notion, originally formulated as a cryptographic game [10], to our symbolic setting. The idea remains the same as for the original notion: an attacker should not learn any information on the votes contained in the ballots, other than the final result of the election. This is modelled by letting the attacker suggest two possible values for the vote of each honest voter: a "real" one and a "fake" one. The attacker then sees the honest voters' ballots, containing either the real or fake votes, and then in the end the real result of the election, computed on the real votes. We model the behaviour of honest voter id, who uses channel c, private and public credentials cr, pcr, and election public key pk in these two scenarios by the two following processes.

$$
\begin{array}{l}
\mathsf{HVoter}^{\mathsf{L}}(c, id, cr, pcr, pk) = \\
\quad \mathsf{in}(c, z). \\
\quad \mathsf{let}\ (v^0, v^1) = (\mathsf{proj}_1^2(z), \mathsf{proj}_2^2(z))\ \mathsf{in} \\
\quad \mathsf{if}\ v^0, v^1 \in \mathsf{Votes}\ \mathsf{then} \\
\qquad \mathsf{new}\ r^0.\ \mathsf{new}\ r^1. \\
\qquad \mathsf{let}\ b^0 = \mathsf{Vote}(pk, id, cr, v^0, r^0)\ \mathsf{in} \\
\qquad \mathsf{let}\ b^1 = \mathsf{Vote}(pk, id, cr, v^1, r^1)\ \mathsf{in} \\
\qquad \mathsf{if}\ \mathsf{Valid}(id, pcr, b^0, pk) = \mathsf{true} \\
\qquad\quad \mathsf{then}\ \mathsf{out}(c, b^0).\ \mathsf{append}(c, b^0, m_{\mathsf{bb}}) \\
\qquad\quad \mathsf{else}\ \mathsf{out}(c, \mathsf{err}_{\mathsf{invalid}}) \\
\quad \mathsf{else}\ \mathsf{out}(c, \mathsf{err}_{\mathsf{vote}})
\end{array}
\qquad
\begin{array}{l}
\mathsf{HVoter}^{\mathsf{R}}(c, id, cr, pcr, pk) = \\
\quad \mathsf{in}(c, z). \\
\quad \mathsf{let}\ (v^0, v^1) = (\mathsf{proj}_1^2(z), \mathsf{proj}_2^2(z))\ \mathsf{in} \\
\quad \mathsf{if}\ v^0, v^1 \in \mathsf{Votes}\ \mathsf{then} \\
\qquad \mathsf{new}\ r^0.\ \mathsf{new}\ r^1. \\
\qquad \mathsf{let}\ b^0 = \mathsf{Vote}(pk, id, cr, v^0, r^0)\ \mathsf{in} \\
\qquad \mathsf{let}\ b^1 = \mathsf{Vote}(pk, id, cr, v^1, r^1)\ \mathsf{in} \\
\qquad \mathsf{if}\ \mathsf{Valid}(id, pcr, b^1, pk) = \mathsf{true} \\
\qquad\quad \mathsf{then}\ \mathsf{out}(c, b^1).\ \mathsf{append}(c, b^0, m_{\mathsf{bb}}) \\
\qquad\quad \mathsf{else}\ \mathsf{out}(c, \mathsf{err}_{\mathsf{invalid}}) \\
\quad \mathsf{else}\ \mathsf{out}(c, \mathsf{err}_{\mathsf{vote}})
\end{array}
$$

In both cases, the process receives the two possible vote instructions (v^0, v^1) from the attacker, and constructs two corresponding ballots b^0, b^1. It then tests for validity, and publishes, either the real b^0 (on the left), or the fake b^1 (on the right). However, since the result is always computed on the real votes, the ballot secretly added to the ballot box m_{bb} is always b^0. If any of the tests fail, we return error messages $\mathsf{err}_{\mathsf{invalid}}, \mathsf{err}_{\mathsf{vote}} \in \Sigma_{\mathsf{err}}$.

The attacker has complete control over the ballots submitted by dishonest voters. Hence, we model them by a process that receives an arbitrary ballot from the attacker, and adds it to the ballot box m_{bb} after checking its validity:

$$
\begin{array}{l}
\mathsf{DVoter}(c, id, cr, pcr, pk) = \mathsf{in}(c, b).\ \mathsf{if}\ \mathsf{Valid}(id, pcr, b, pk) = \mathsf{true} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{then}\ \mathsf{out}(c, b).\ \mathsf{append}(c, b, m_{\mathsf{bb}})\ \mathsf{else}\ \mathsf{out}(c, \mathsf{err}_{\mathsf{invalid}}).
\end{array}
$$

To a reader used to symbolic modelling of protocols, it may seem strange that dishonest voters are modelled by a process, rather than being left completely under the control of the attacker. It may similarly be surprising that the voters' processes include the validity checks and write directly to the ballot box, while these operations are not actually performed by the voter but by an independent entity (typically the server storing the ballot box). While not essential for our results, we decided to adopt this style of modelling to follow more closely the original formulation as a cryptographic game. In that formalism, the protocol and the scenario considered are modelled as oracles. Our symbolic processes are written in the same spirit: they should be seen as models of what happens when a voter votes, rather than directly models of the voter's behaviour.

We then consider n voters: for each $i \in [\![1, n]\!]$, we let $\overrightarrow{v_i} = (c_i, id_i, cr_i, pcr_i)$, where $c_i \in \mathcal{C}\mathsf{h}_{\mathsf{pub}}$ is a dedicated public channel, $id_i \in \Sigma_0$ is the voter's identity, $cr_i \in \mathcal{N}$ her private credential, and $pcr_i = \mathsf{Pub}(cr_i, u_i)$ her public credential randomised with $u_i \in \mathcal{N}$. We will say that for $i \neq j$, $\overrightarrow{v_i}$ and $\overrightarrow{v_j}$ are *distinct voters*, to signify that they have different identities, credentials, and channels, *i.e.* $c_i \neq c_j \ \wedge \ id_i \neq id_j \ \wedge \ cr_i \neq cr_j \ \wedge \ u_i \neq u_j \ \wedge \ u_i \neq cr_j \ \wedge \ cr_i \neq u_j$.

We then define the BPRIV property as follows.

Definition 6. *A voting scheme is* BPRIV *for p honest voters and $n-p$ dishonest voters, written* $\mathsf{BPRIV}(p, n-p)$*, if*

$$\mathsf{Election}^{\mathsf{L}}_{p,n-p}(\overrightarrow{v_1}, \ldots, \overrightarrow{v_n}) \approx_t \mathsf{Election}^{\mathsf{R}}_{p,n-p}(\overrightarrow{v_1}, \ldots, \overrightarrow{v_n})$$

where $\mathsf{Election}^{\mathsf{X}}_{p,n-p}(\overrightarrow{v_1}, \ldots, \overrightarrow{v_n}) =$

$$\begin{array}{l}
\quad \mathsf{new}\ sk.\ m_{\mathsf{bb}} := \mathsf{nil}.\ \mathsf{out}(ch, \mathsf{pk}(sk)). \\
\quad\quad \big(\ \mathsf{phase}\ 1.\ \mathsf{out}(c_1, pcr_1).\ \mathsf{phase}\ 2.\ \mathsf{HVoter}^{\mathsf{X}}(\overrightarrow{v_1}, \mathsf{pk}(sk)) \\
\quad\quad\quad \mid \ldots \\
\quad\quad\quad \mid \mathsf{phase}\ 1.\ \mathsf{out}(c_p, pcr_p).\ \mathsf{phase}\ 2.\ \mathsf{HVoter}^{\mathsf{X}}(\overrightarrow{v_p}, \mathsf{pk}(sk)) \\
\quad\quad\quad \mid \mathsf{phase}\ 1.\ \mathsf{out}(c_{p+1}, \langle cr_{p+1}, pcr_{p+1} \rangle).\ \mathsf{phase}\ 2.\ \mathsf{DVoter}(\overrightarrow{v}_{p+1}, \mathsf{pk}(sk)) \\
\quad\quad\quad \mid \ldots \\
\quad\quad\quad \mid \mathsf{phase}\ 1.\ \mathsf{out}(c_n, \langle cr_n, pcr_n \rangle).\ \mathsf{phase}\ 2.\ \mathsf{DVoter}(\overrightarrow{v_n}, \mathsf{pk}(sk)) \\
\quad\quad\quad \mid \mathsf{phase}\ 3.\ \mathsf{Tally}(sk) \ \big)
\end{array}$$

with $ch \in \mathcal{C}\mathsf{h}_{\mathsf{pub}}$, $\mathsf{X} \in \{\mathsf{L}, \mathsf{R}\}$.

While we designed our symbolic definition to follow as closely as possible the original computational formulation of the property, there are two notable differences.

First, in the original notion, the oracle modelling honest voters was executed atomically: once the adversary submits his vote instructions, the generated ballot is immediately placed in the ballot box. That is not the case here. This difference is an important one, and is fully intentional: we wanted to model a scenario where the attacker can intercept and block ballots on their way to the ballot box. This gives him more power, and thus makes for a stronger privacy property. A consequence of that choice however, is that our definition is not suited to studying protocols that rely on weeding out duplicate ballots from the ballot box (*e.g.* some fixed versions of Helios). Indeed, the weeding operation only makes sense when assuming that all generated ballots have reached the ballot box.

Second, many voting schemes include mechanisms allowing everyone to check that the tallying authority computed the result correctly. Typically, the talliers publish, alongside the result itself, zero-knowledge proofs showing that they *e.g.* correctly decrypted the ballots in the ballot box. In BPRIV however, having them output this proof would immediately break the property. The proof only holds for the actual ballots being tallied, so the attacker could just check it against the ballots he saw, which would succeed on the left but fail on the right. The original formalisation handles this by using a simulator for the proof on the right. This sort of operation does not really have a counterpart in the symbolic model, and we decided (for now) to simply abstract this proof away and not model it.

3.3 Auxiliary properties

In [10], the authors propose two companion properties to BPRIV, called *strong correctness* and *strong consistency*. Together with BPRIV, they imply a strong simulation-based notion of vote privacy. Although we do not prove such a simulation – these are not really used in the symbolic model – we still define symbolic counterparts to the original computational side-conditions. They are useful when establishing our reduction result, and we will from now on assume they hold.

Strong correctness. Honest voters should always be able to cast their vote, *i.e.* their ballots are always valid. Formally, for any $id, cr, r, u, sk \in \Sigma_0 \cup \mathcal{N}$, $v \in \mathsf{Votes}$, we must have: $\mathsf{Valid}(id, \mathsf{Pub}(cr, u), \mathsf{Vote}(\mathsf{pk}(sk), id, cr, v, r), \mathsf{pk}(sk)) =_{\mathsf{E}} \mathsf{true}$.

Strong consistency. The tally itself should only compute the result of the election, and nothing else – it cannot accept hidden commands from the attacker coded as special ballots, *etc.* Formally we assume two functions extract and count:

- $\mathsf{extract}(b, sk)$ is meant to extract the vote, and the voter's id and credential from b, using key sk, or return $\perp$ if b is not readable (ill-formed, *etc.*).
- count is the counting function, meant to compute the result from the list of votes. It is assumed to always return a public term in $\mathcal{T}(\Sigma, \Sigma_0)$.

We assume that: if $\mathsf{Valid}(id, \mathsf{Pub}(cr, u), b, \mathsf{pk}(sk)) =_{\mathsf{E}} \mathsf{true}$ then $\mathsf{extract}(b, sk) = (id, cr, v)$ for some $v \in \mathsf{Votes}$. In other words, extraction always succeeds on valid ballots. Moreover, extract must behave as expected on honestly generated ballots, *i.e.* $v = v_0$ when $b = \mathsf{Vote}(\mathsf{pk}(sk), cr, v_0, r)$. We let $\mathsf{extract}([b_1, \ldots, b_n], sk)$ be the list of non-$\perp$ values in $[\mathsf{extract}(b_1, sk), \ldots, \mathsf{extract}(b_n, sk)]$.

Lastly, we assume that these functions characterise the behaviour of the C_{Tally} computation, *i.e.* for all list bb of messages, for all $sk \in \mathcal{N}$, we have:

$$C_{\mathsf{Tally}}(bb, sk) = \mathsf{count}(\mathsf{lst}(\mathsf{extract}(bb, sk)))$$

where lst is a function that only keeps the vote in each tuple returned by extract. Later on, when considering the case of revote, lst will be replaced with a function applying a revoting policy to determine which vote to keep for each voter.

Example 8. The Valid recipe and C_{tally} computation from Examples 6 and 7 satisfy these assumptions, where extract simply decrypts the ciphertext in the ballot, and count returns the pair of the numbers of votes for yes and no.

4 Reduction

We first establish our reduction in the case where voters vote only once. Some systems allow voters to vote again by submitting a new ballot that will *e.g.* replace their previous one, in the interest of coercion-resistance. We extend our result to that setting in Section 5. Our BPRIV definition stated in Section 3 is parametrized by the number n of voters among which p are assumed to be honest. We prove our reduction result in two main steps. We first establish that it is enough to consider the case where $p = 1$, *i.e.* one honest voter is enough, and then we establish the conditions under which the number of dishonest voters can be bounded as well.

4.1 Reduction to one honest voter

In order to remain faithful to the original computational BPRIV notion, and to define a strong privacy property, we decided to write our symbolic BPRIV property in a general way, *i.e.* considering an arbitrary number of honest voters. Each voter receives two vote instructions (v_0, v_1) from the attacker, and shows him the ballot for one or the other. Reducing the number of honest voters by replacing them by dishonest ones is non trivial. This comes from the fact the behaviour of an honest voter is *not* exactly the same on both sides of the equivalence, as it is the case for a dishonest voter. Nevertheless, we establish the following result: one honest voter is enough.

Proposition 1. *Consider a voting scheme $\mathcal{V}$, and p, n such that $1 \leq p \leq n$. If $\mathcal{V}$ does not satisfy* $\mathsf{BPRIV}(p, n-p)$*, then it does not satisfy* $\mathsf{BPRIV}(1, n-1)$*.*

Proof (Sketch). The general idea of this proof is to show we can isolate one specific honest voter whose ballot is the one causing $\mathsf{BPRIV}(p, n-p)$ to break. We then leave that voter as the only honest one, and use dishonest voters to simulate the $p-1$ others, and obtain an attack against $\mathsf{BPRIV}(1, n-1)$.

The difficulties are *(i)* how to find this particular voter, and *(ii)* how to simulate the honest voters with dishonest ones. The simulation would be easy for a honest voter *id* voting for the same candidate v on both sides: simply use the dishonest voter to submit a ballot $\mathsf{Vote}(pk, id, cr, v, r)$ for some random r, and the correct credential cr. However, in the Election processes, *id* uses different values v_0, v_1 on the left and on the right, so that we cannot easily construct a single dishonest ballot simulating *id*'s on both sides at the same time.

To solve both issues, the main idea is to go gradually from the $\mathsf{Election}^{\mathsf{L}}$ process, where all HVoters are $\mathsf{HVoter}^{\mathsf{L}}$ and use the real vote (their v_0), to the $\mathsf{Election}^{\mathsf{R}}$ process, where they are $\mathsf{HVoter}^{\mathsf{R}}$ and use the fake one (their v_1). We consider intermediate processes $P_0, \ldots, P_p$: in P_i, the first i HVoters are $\mathsf{HVoter}^{\mathsf{R}}$, and the others are $\mathsf{HVoter}^{\mathsf{L}}$. Since $\mathsf{BPRIV}(p, n-p)$ does not hold, $P_0 = \mathsf{Election}^{\mathsf{L}}$ and $P_p = \mathsf{Election}^{\mathsf{R}}$ are not equivalent. Hence, there must exist some i_0 such that P_{i_0+1} and P_{i_0} are not equivalent. These two processes differ only by the $i_0 + 1^{\text{th}}$ HVoter, who is $\mathsf{HVoter}^{\mathsf{L}}$ in P_{i_0}, and $\mathsf{HVoter}^{\mathsf{R}}$ in P_{i_0+1}. This voter will be our particular voter, who will remain honest, solving issue *(i)*. All other HVoters behave the same in P_{i_0} and P_{i_0+1}: they vote with their right vote for the first i_0, and their left for the last $p - i_0 - 1$. For them, issue *(ii)* is thus solved, and we can simulate them with dishonest voters. This way, we recover an attack with only one honest voter, and $(n-p) + (p-1) = n-1$ dishonest voters. □

Note that, in the case of the earlier reduction result from [4] for the SWAP definition, a simple version of vote privacy is used from the start. They consider only two honest voters who swap their votes, and not the general definition (as stated *e.g.* in [8,10]) involving an arbitrary permutation between an arbitrary number of honest voters. Due to this, in [4], this first step was trivial. The argument in our case is more involved, as we start from the general notion.

4.2 Bounding the number of dishonest voters

This second reduction result allows one to bound the number of dishonest voters when considering BPRIV. More precisely, we consider a unique honest voter, and we show that k dishonest voters are sufficient to mount an attack against vote privacy (if such an attack exists). Here, we reduce the number of voters from n to $k+1$ (k dishonest voters plus one honest voter), and the resulting bound depends on the counting function. Roughly, as formally stated below, we have to ensure that when there is a difference in the result when considering n votes, then a difference still exists when considering at most k votes.

Definition 7. *A counting function* count *is* k-bounded *if for all* n*, for all lists* $l_{\mathsf{tally}} = [v_1, \ldots, v_n]$ *and* $l'_{\mathsf{tally}} = [v'_1, \ldots, v'_n]$ *of size* $n > k$ *of elements in* Votes*, such that* $\mathsf{count}(l_{\mathsf{tally}}) \neq_{\mathsf{E}} \mathsf{count}(l'_{\mathsf{tally}})$*, there exist* $k' \leq k$*, and* $i_1 < \ldots < i_{k'}$*, such that* $\mathsf{count}([v_{i_1}, \ldots, v_{i_{k'}}]) \neq_{\mathsf{E}} \mathsf{count}([v'_{i_1}, \ldots, v'_{i_{k'}}])$.

This assumption needed to establish our reduction results captures the most common counting functions such as multiset, sum, majority (see Appendix A).

Lemma 1. *The functions* $\mathsf{count}_{\#}$*,* count_{Σ}*, and* $\mathsf{count}_{\mathsf{Maj}}$ *are* 1*-bounded.*

This can be easily established by noticing that, when considering $\mathsf{count}_{\#}$ (resp. count_{Σ} or $\mathsf{count}_{\mathsf{Maj}}$), as soon as two lists $[v_1, \ldots, v_n]$ and $[v'_1, \ldots, v'_n]$ of votes give different results, it means that there exists at least an indice i_0 such that $v_{i_0} \neq v'_{i_0}$. Hence, keeping this vote is enough to keep a difference. We can also consider more involved counting functions, such as Single Transferable Vote (STV), used *e.g.* in Australian legislative elections, for which we have established that it is 5-bounded when considering 3 candidates. Under this k-boundedness assumption, we are then able to bound the number of dishonest voters.

Proposition 2. *Let* $\mathcal{V}$ *be a voting scheme whose associated counting function is* k*-bounded for* $k \geq 1$*. If* $\mathcal{V}$ *does not satisfy* $\mathsf{BPRIV}(1, n)$ *for some* $n \geq 1$*, then* $\mathcal{V}$ *does not satisfy* $\mathsf{BPRIV}(1, k')$ *for some* $k' \leq k$*. Moreover, in that case there exists a witness of this attack where no more than* k' *ballots reached the ballot box.*

Proof (Sketch). If $\mathsf{BPRIV}(1, n-1)$ does not hold, the difference appears either (i) when the honest voter outputs her ballot, or (ii) when outputting the result. Indeed, the behaviour of a dishonest voter who simply outputs the message he received does not help to mount an attack. Moreover, the only test that a dishonest voter performs is a public test from which the attacker will not infer anything. In case (i), no dishonest voters are even needed, and the claim holds.

In case (ii), we know that that the public terms representing the final result are different on both sides. We apply our k-boundedness hypothesis, and we know that a difference is still there when considering k voters (or even less). Removing the corresponding actions performed by dishonest voters, the trace still corresponds to an execution assuming that the validity tests do not depend on the the other ballots on the bulletin board. Hence, we have a witness of non-equivalence with at most k ballots, and thus at most $k-1$ dishonest voters. □

4.3 Main result

Combining Propositions 1 and 2, we get our main reduction theorem establishing that it suffices to consider one honest voter, and at most k dishonest ones.

Theorem 1. *Let $\mathcal{V}$ be a voting scheme whose associated counting function is k-bounded for some $k \geq 1$, and p, n be two integers such that $1 \leq p \leq n$. If $\mathcal{V}$ does not satisfy $\mathsf{BPRIV}(p, n-p)$, then $\mathcal{V}$ does not satisfies $\mathsf{BPRIV}(1, k')$ for some $k' \leq k$. Moreover, in that case there exists a witness of this attack where no more than k' ballots reached the ballot box.*

Example 9. The ballot copy attack on Helios (with the 1-bounded multiset count) from [23], mentioned in Example 5, can be performed against $\mathsf{BPRIV}(p, n-p)$: a honest voter is told to vote yes or no, her ballot is copied by a dishonest voter but remains valid, and the result is then $\{\!|\mathsf{yes}, \mathsf{yes}|\!\}$ on the left (as the "yes" ballot was seen and copied), and $\{\!|\mathsf{yes}, \mathsf{no}|\!\}$ on the right (as the "no" ballot was seen).

In accordance with Theorem 1, one honest voter, one dishonest, and one accepted ballot are actually sufficient: the attacker can simply block the honest ballot, so that only the copy is counted leading to $\{\!|\mathsf{yes}|\!\}$ on the left and $\{\!|\mathsf{no}|\!\}$ on the right, which suffices for the attack.

5 Dealing with revoting

We now consider the case where re-voting is allowed. We first adapt the BPRIV definition to this setting. The processes HVoter, DVoter, and Tally are left unchanged. Only the main $\mathsf{Election}$ processes, and the consistency assumption change. The tallying now takes into account a revote policy, indicating how to proceed when a voter casts multiple votes. A revote policy is a function:

$$\mathsf{policy} : (\Sigma_0 \times \mathcal{N}_{\mathsf{priv}} \times \mathsf{Votes})\ \mathsf{list} \to \mathsf{Votes}\ \mathsf{list}.$$

This policy function replaces lst in the strong consistency assumption (Section 3.3). We consider here the two most common revote policies. The last and first policies, that select resp. the last or the first vote from each voter.

We reuse the notations from Section 3.2, and we introduce in addition $\overrightarrow{w_i} = (d_i, id_i, cr_i, pcr_i)$ for each $i \in \{1, \ldots, n\}$ where d_i are different private channel names. The privacy property $\mathsf{BPRIVR}(p, n-p)$ is written as follows:

$$\mathsf{ElectionRevote}^{\mathsf{L}}_{p,n-p}(\overrightarrow{v_1}, \ldots, \overrightarrow{v_n}) \approx_t \mathsf{ElectionRevote}^{\mathsf{R}}_{p,n-p}(\overrightarrow{v_1}, \ldots, \overrightarrow{v_n})$$

where $\mathsf{ElectionRevote}^{\mathsf{X}}_{p,n-p}(\overrightarrow{v_1}, \ldots, \overrightarrow{v_n}) =$

$\mathsf{new}\ sk.\ m_{\mathsf{bb}} := \mathsf{nil}.\ \mathsf{out}(ch, \mathsf{pk}(sk)).$

$($ $\mathsf{phase}\ 1.\ \mathsf{out}(c_1, pcr_1).\ \mathsf{phase}\ 2.\ !\ \mathsf{new}\ d_1.\ \mathsf{out}(c_1, d_1).\ \mathsf{HVoter}^{\mathsf{X}}(\overrightarrow{v_1}, \mathsf{pk}(sk))$

$\mid \ldots$

$\mid \mathsf{phase}\ 1.\ \mathsf{out}(c_p, pcr_p).\ \mathsf{phase}\ 2.\ !\ \mathsf{new}\ d_p.\ \mathsf{out}(c_p, d_p).\ \mathsf{HVoter}^{\mathsf{X}}(\overrightarrow{w_p}, \mathsf{pk}(sk))$

$\mid \mathsf{phase}\ 1.\ \mathsf{out}(c_{p+1}, pcr_{p+1}).\ \mathsf{phase}\ 2.\ !\ \mathsf{new}\ d_{p+1}.\ \mathsf{out}(c_{p+1}, d_{p+1}).\mathsf{DVoter}(\overrightarrow{w_{p+1}}, \mathsf{pk}(sk))$

$\mid \ldots$

$\mid \mathsf{phase}\ 1.\ \mathsf{out}(c_n, pcr_n).\ \mathsf{phase}\ 2.\ !\ \mathsf{new}\ d_n.\ \mathsf{out}(c_n, d_n).\ \mathsf{DVoter}(\overrightarrow{w_n}, \mathsf{pk}(sk))$

$\mid \mathsf{phase}\ 3.\ \mathsf{Tally}(sk)\ )$

with $ch \in \mathcal{C}\mathsf{h}_{\mathsf{pub}}$, $\mathsf{X} \in \{\mathsf{L}, \mathsf{R}\}$.

Note that a replication operator has been added in front of the voter processes to model the fact that revote is now possible.

Theorem 2. *Let $\mathcal{V}$ be a voting scheme whose associated counting function is k-bounded for some $k \geq 1$, and p, n be two integers such that $1 \leq p \leq n$. If $\mathcal{V}$ does not satisfy $\mathsf{BPRIVR}(p, n-p)$, then $\mathcal{V}$ does not satisfy $\mathsf{BPRIVR}(1, k')$ for some $k' \leq k$. Moreover, in that case there exists a witness of this attack where no more than k' ballots reached the ballot box (each from a different voter).*

The proof of this Theorem follows the same lines as the one when revote is not allowed and is given in the full version [1]. We may note that replication operators are still there, and thus establishing such an equivalence property (even when $p = 1$, and $k = 1$) is not trivial. Traces of unbounded length still must be considered. However, as we are able to establish that, in a minimal attack trace, at most k ballots reached the ballot box (each by a different voter), we can easily remove the replication operator in front of a dishonest voter. This reasoning does not apply for the honest voter, as the output she performed may be useful to mount an attack (contrary to the output of a dishonest voter who outputs a term known by the attacker). This has been overlooked in the reduction result presented in [4]. The security analysis of Helios with revote has been done without considering this replication operator, leading to erroneous security analysis.

6 Applications and case studies

To illustrate the generality of our result, and to showcase how useful it can be in practice, we apply it to several well-known voting protocols from the literature. For our case study, we chose the following protocols: two variants of Helios [3], corresponding to its original version, subject to the attack discussed earlier, and a fixed version that includes identities in the ZKP; Belenios [21], and the related BeleniosRF [15] and BeleniosVS [19]; Civitas [29]; and Prêt-à-Voter [16,32].

We modelled these protocols as processes satisfying our assumptions, and analysed them using Proverif. We only prove BPRIV itself with Proverif. Strong correctness only involves terms, and can easily be proved by hand. Strong consistency requires to show that the tallying process rightly computes the tally, which Proverif is not well-suited to do, as it requires 1) modelling the tally in the general case, *i.e.* with no bounds on the lengths of lists, and 2) comparing it to the abstract definition of the counting function, which Proverif cannot really manipulate. The property clearly holds though, and could be proved by hand.

All model files for our case study are available at [1]. The results are presented in Table 1. As expected, we find the attack on Helios from [23].

We conduct the analysis for different counting functions, using our result to bound the number of agents and ballots. We considered majority, multiset, sum, and STV (restricted to 3 candidates). In fact, in the case of 1-bounded functions, since only one ballot needs to be accepted by the ballot box, the tallying is trivial, and ends up being the same for different functions (majority, multiset, *etc.*). Thus, a single Proverif file is enough to model several counting functions as once.

	Protocols \ Counting	Multiset/Maj/Sum (2 voters/1 ballot)	Single Transferable Vote (6 voters/5 ballots)
without revote	Helios (*id* in ZKP)	✓ ≤ 1s	✓ ∼ 24 min
	Helios (ZKP without *id*)	✗ ≤ 1s	✗ ∼ 27 min
	Belenios	✓ ≤ 1s	✓ ∼ 27 min
	BeleniosRF	✓ ∼ 3s	⏱
	BeleniosVS	✓ ∼ 3s	⏱
	Civitas	✓ ≤ 1s	✓ ∼ 39 min
	Prêt-à-Voter	✓ ≤ 1s	⏱
revote	Helios (*id* in ZKP)	✓ ≤ 1s	✓ ∼ 23 min
	Helios (ZKP without *id*)	✗ ≤ 1s	✗ ∼ 42 min
	Belenios	✓ ≤ 1s	✓ ∼ 23 min

Table 1. Summary of our results. ✓: Proverif proves the property. ✗: Proverif finds an attack trace. ⏱: timeout (≥ 24h). Execution times are on an Intel i7-1068NG7 CPU.

We considered both the cases without and with revote, for protocols that support revoting (except Civitas, which in that case uses rather complex mechanisms that do not fit our setting). As mentioned earlier, when revote is allowed, our result does not get rid of the replication operator. Bounding the number of voters is still useful in that case, as it simplifies our models. More importantly, bounding the number of ballots means we can encode the ballot box as a fixed-length list, which is very helpful as Proverif does not support arbitrary length lists.

In some cases, we made slight adjustments to the protocols, so that they fit our framework. Detailed explanations on these modelling choices can be found in the files. Notably, many protocols use homomorphic encryption: talliers add all encrypted votes before decryption. While our result still applies in principle to such primitives, Proverif cannot handle the associated equations. Hence, we instead verify versions of the protocols that use a mixnet, *i.e.* mix ballots in a random order before decryption.

Overall, as can be seen in the table, our result allows for efficient verification of all protocols we considered. Thanks to the small bounds we establish, we get even better performance than previous work [4] in scenarios where that result applies – *i.e.* the first column, for multiset counting. In that case, some analyses took several hours/days in [4], due to the higher bounds. Our result is more general and can handle *e.g.* STV counting. On most tested protocols, performance remains acceptable in that case. However Proverif did not terminate on three files after 24h: this is likely due to the combination of the complex equational theories used by these protocols, and the theory for STV, which is itself large.

7 Conclusion

We have proposed a symbolic version of the BPRIV vote privacy notion, and established reduction results that help us efficiently verify it on several voting protocols, with different counting functions, using automated tools.

As mentioned earlier, a limitation of our definition is the modelling of the correct tallying proofs, which we abstracted away. In the computational definition,

they are handled using simulators. It remains to be seen whether such techniques can be adapted to the symbolic setting, and how.

Our attacker already controls the channel between voters and the ballot box. A natural further step is to consider an even stronger attacker, that can modify the content of the ballot box (altering already cast ballots, *etc.*). BPRIV has recently been extended to such a scenario in the computational model [22], at the cost of a much more complex definition – adapting that work to the symbolic setting constitutes exciting future work.

References

1. Delaune, S., Lallemand, J.: One vote is enough for analysing privacy (2022), https://hal.inria.fr/hal-03669664
2. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Hankin, C., Schmidt, D. (eds.) Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001. pp. 104–115. ACM (2001)
3. Adida, B.: Helios: Web-based open-audit voting. In: van Oorschot, P.C. (ed.) Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA. pp. 335–348. USENIX Association (2008)
4. Arapinis, M., Cortier, V., Kremer, S.: When are three voters enough for privacy properties? In: Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS'16). LNCS, Springer (2016)
5. Backes, M., Hritcu, C., Maffei, M.: Automated verification of remote electronic voting protocols in the applied pi-calculus. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008. pp. 195–209. IEEE Computer Society (2008)
6. Basin, D.A., Dreier, J., Sasse, R.: Automated symbolic proofs of observational equivalence. In: Ray, I., Li, N., Kruegel, C. (eds.) Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 1144–1155. ACM (2015)
7. Basin, D.A., Radomirovic, S., Schmid, L.: Alethea: A provably secure random sample voting protocol. In: 31st IEEE Computer Security Foundations Symposium, (CSF'18). IEEE Computer Society (2018)
8. Benaloh, J.: Verifiable secret-ballot elections. Ph.D. thesis, Yale University (1987)
9. Benaloh, J.C., Yung, M.: Distributing the power of a government to enhance the privacy of voters (extended abstract). In: Halpern, J.Y. (ed.) Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing, Calgary, Alberta, Canada, August 11-13, 1986. pp. 52–62. ACM (1986)
10. Bernhard, D., Cortier, V., Galindo, D., Pereira, O., Warinschi, B.: A comprehensive analysis of game-based ballot privacy definitions. In: Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15). IEEE Computer Society Press, San Jose, CA, USA (2015)
11. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14). pp. 82–96. IEEE Computer Society, Cape Breton, Nova Scotia, Canada (2001)
12. Blanchet, B.: Modeling and verifying security protocols with the applied pi calculus and ProVerif. Foundations and Trends in Privacy and Security **1**(1–2), 1–135 (2016)

13. Blanchet, B., Abadi, M., Fournet, C.: Automated Verification of Selected Equivalences for Security Protocols. In: 20th IEEE Symposium on Logic in Computer Science (LICS 2005). pp. 331–340. IEEE Computer Society, Chicago, IL (2005)
14. Blanchet, B., Smyth, B.: Automated reasoning for equivalences in the applied pi calculus with barriers. Journal of Computer Security **26**(3), 367–422 (2018)
15. Chaidos, P., Cortier, V., Fuchsbauer, G., Galindo, D.: BeleniosRF: A non-interactive receipt-free electronic voting scheme. In: 23rd ACM Conference on Computer and Communications Security (CCS'16). pp. 1614–1625. ACM, Vienna, Austria (2016)
16. Chaum, D., Ryan, P.Y.A., Schneider, S.A.: A practical voter-verifiable election scheme. In: di Vimercati, S.D.C., Syverson, P.F., Gollmann, D. (eds.) Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings. LNCS, vol. 3679, pp. 118–139. Springer (2005)
17. Comon-Lundh, H., Cortier, V.: Security properties: two agents are sufficient. In: Proc. 12th European Symposium on Programming (ESOP'03). LNCS, vol. 2618, pp. 99–113. Springer, Warsaw, Poland (2003)
18. Cortier, V., Dallon, A., Delaune, S.: Bounding the number of agents, for equivalence too. In: Proc. 5th International Conference on Principles of Security and Trust (POST'16). pp. 211–232. LNCS, Springer (2016)
19. Cortier, V., Filipiak, A., Lallemand, J.: BeleniosVS: Secrecy and verifiability against a corrupted voting device. In: 32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019. IEEE (2019)
20. Cortier, V., Galindo, D., Turuani, M.: A formal analysis of the Neuchâtel e-voting protocol. In: 3rd IEEE European Symposium on Security and Privacy (Euro S&P'18). pp. 430–442. London, UK (2018)
21. Cortier, V., Gaudry, P., Glondu, S.: Belenios: A simple private and verifiable electronic voting system. In: Foundations of Security, Protocols, and Equational Reasoning - Essays Dedicated to Catherine A. Meadows. LNCS, vol. 11565, pp. 214–238. Springer (2019)
22. Cortier, V., Lallemand, J., Warinschi, B.: Fifty shades of ballot privacy: Privacy against a malicious board. In: 33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020. pp. 17–32. IEEE (2020)
23. Cortier, V., Smyth, B.: Attacking and fixing Helios: An analysis of ballot secrecy. Journal of Computer Security **21**(1), 89–148 (2013)
24. Cortier, V., Wiedling, C.: A formal analysis of the Norwegian e-voting protocol. Journal of Computer Security **25**(15777), 21–57 (2017)
25. Delaune, S., Hirschi, L.: A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols. Journal of Logical and Algebraic Methods in Programming **87**, 127–144 (2017)
26. Delaune, S., Kremer, S., Ryan, M.D.: Verifying privacy-type properties of electronic voting protocols. Journal of Computer Security **17**(4), 435–487 (2009)
27. D'Osualdo, E., Ong, L., Tiu, A.: Deciding secrecy of security protocols for an unbounded number of sessions: The case of depth-bounded processes. In: Proc. 30th Computer Security Foundations Symposium, (CSF'17). pp. 464–480. IEEE Computer Society (2017)
28. Fröschle, S.: Leakiness is decidable for well-founded protocols? In: Proc. 4th Conference on Principles of Security and Trust (POST'15). LNCS, Springer (2015)
29. Juels, A., Catalano, D., Jakobsson, M.: Coercion-resistant electronic elections. In: Chaum, D., Jakobsson, M., Rivest, R.L., Ryan, P.Y.A., Benaloh, J., Kutylowski, M., Adida, B. (eds.) Towards Trustworthy Elections, New Directions in Electronic Voting. LNCS, vol. 6000, pp. 37–63. Springer (2010)

30. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In: Computer Aided Verification, 25th International Conference, CAV 2013, Princeton, USA, Proc. LNCS, vol. 8044, pp. 696–701. Springer (2013)
31. Mödersheim, S., Viganò, L.: Alpha-beta privacy. ACM Trans. Priv. Secur. **22**(1), 7:1–7:35 (2019)
32. Ryan, P.Y.A., Schneider, S.A.: Prêt à voter with re-encryption mixes. In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Hamburg, Germany, September 18-20, 2006, Proceedings. LNCS, vol. 4189, pp. 313–326. Springer (2006)

Appendix A Some counting functions

A.1 Some 1-bounded counting functions

Multiset. The result is the multiset of all votes. Formally, in our setting, a term representing that multiset is computed: for all n, $\mathsf{count}_{\#}([v_1,\ldots,v_n]) = f(\{\!|v_1,\ldots,v_n|\!\})$ where f is a function such that $f(M_1) =_{\mathsf{E}} f(M_2)$ (equality on terms) iff $M_1 =_{\#} M_2$ (equality on multisets). For instance, if we just output the list of all votes, the order cannot matter, *i.e.* $\mathsf{count}_{\#}([a,b]) =_{\mathsf{E}} \mathsf{count}_{\#}([b,a])$.

Sum. A total of points total is given to each voter who decides to distribute them among the candidates of his choice. The result is a vector of integers representing the total of points obtained by each candidate. Assuming c candidates, for all n, we have: $\mathsf{count}_{\Sigma}([v_1,\ldots,v_n]) = f(\sum_{i=1}^{n} v_i)$ where $v_i = (p_1,\ldots,p_c)$ with $1 \leq i \leq n$, and $p_1,\ldots,p_c \in \mathbb{N}$ with $p_1+\ldots+p_c \leq \mathsf{total}$, and f is a function from vectors of c integers to terms such that $f(\overrightarrow{u_1}) =_{\mathsf{E}} f(\overrightarrow{u_2})$ (equality on terms) iff $\overrightarrow{u_1} = \overrightarrow{u_2}$ (equality on vectors of integers).

Majority. The majority function between two choices yes and no simply outputs yes if $\#\mathsf{yes} > n/2$ where n is the number of votes, and no otherwise. For all n, $\mathsf{count}_{\mathsf{Maj}}([v_1,\ldots,v_n]) = \mathsf{yes}$ if $\#\{i \mid v_i = \mathsf{yes}\} > n/2$; and $\mathsf{count}_{\mathsf{Maj}}([v_1,\ldots,v_n]) = \mathsf{no}$ otherwise. Here, yes and no are two public constants ($\mathsf{yes} \neq_{\mathsf{E}} \mathsf{no}$).

Lemma 1. *The functions* $\mathsf{count}_{\#}$, count_{Σ}, *and* $\mathsf{count}_{\mathsf{Maj}}$ *are* 1*-bounded.*

Proof. Let $[v_1,\ldots,v_n]$ and $[v'_1,\ldots,v'_n]$ be two lists of votes with $n > 1$, such that $\mathsf{count}_{\#}([v_1,\ldots,v_n]) \neq \mathsf{count}_{\#}([v'_1,\ldots v'_n])$. Since $\mathsf{count}_{\#}$ is a function, we have $\{\!|v_1,\ldots,v_n|\!\} \neq \{\!|v'_1,\ldots,v'_n|\!\}$, and thus there exists i_0 such that $v_{i_0} \neq v'_{i_0}$. Hence, $\mathsf{count}([v_{i_0}]) \neq \mathsf{count}([v'_{i_0}])$, which concludes the proof for $\mathsf{count}_{\#}$. A similar reasoning applies for count_{Σ}, and $\mathsf{count}_{\mathsf{Maj}}$. □

A.2 Single Transferable Vote

Single Transferable Vote (STV) is a system where each voter casts a single ballot containing a total ordering of all candidates. A vote goes to the voter's first choice. If that choice is later eliminated, instead of being thrown away, the vote is transferred to her second choice, and so on. In each round, the least popular

candidate is eliminated. His votes are transferred based on voters' subsequent choices. The process is repeated until one candidate remains, who is declared the winner. We assume a total order $\prec$ on candidates is picked beforehand, and is used to break ties. The STV counting function outputs a term representing the winning candidate; it is parametrised by the set of candidates and the order $\prec$. Let $\mathsf{Count}^3_{\mathsf{STV}}$ the STV function for candidates $\{a, b, c\}$ with $a \prec b \prec c$. Votes are 3-tuples: $(c_1; c_2; c_3)$ where $\{c_1, c_2, c_3\} = \{a, b, c\}$ and c_i denotes the i^{th} choice.

Example 10. Let $v = (a; b; c)$ and $v' = (a; c; b)$. We have $v \neq v'$, however $\mathsf{Count}^3_{\mathsf{STV}}([v]) = \mathsf{Count}^3_{\mathsf{STV}}([v']) = a$. Thus, the previous reasoning to establish 1-boundedness does not apply here.

Lemma 2. $\mathsf{Count}^3_{\mathsf{STV}}$ *is* 5*-bounded.*

Proof. We assume that $a \prec b \prec c$. Let $\ell = [v_1, \ldots, v_n]$ and $\ell' = [v'_1, \ldots, v'_n]$ be two lists of Votes such that $\mathsf{Count}^3_{\mathsf{STV}}(\ell) \neq \mathsf{Count}^3_{\mathsf{STV}}(\ell')$. For each $1 \leq i \leq n$, we denote $(c_{i,1}; c_{i,2}; c_{i,3})$ the vote v_i and $(c'_{i,1}; c'_{i,2}; c'_{i,3})$ the vote v'_i.

Case 1: There exists $1 \leq i_0 \leq n$ such that $v_{i_0} = (c_{i_0,1}; c_{i_0,2}; c_{i_0,3})$ and $v'_{i_0} = (c'_{i_0,1}; c'_{i_0,2}; c'_{i_0,3})$ with $c_{i_0,1} \neq c'_{i_0,1}$. In such a case, we keep this vote, and we have

$$c_{i_0,1} = \mathsf{Count}^3_{\mathsf{STV}}([v_{i_0}]) \neq \mathsf{Count}^3_{\mathsf{STV}}([v'_{i_0}]) = c'_{i_0,1}.$$

Case 2: Otherwise, for $1 \leq i \leq n$, we have $c_{i,1} = c'_{i,1}$. Thus, at the first round, the eliminated candidate is the same on both sides. Call it c_0. If c_0 does not occur as the first choice on a vote, *i.e.* $c_0 \neq c_{i,1}$ for all i (and thus $c_0 \neq c'_{i,1}$, as $c_{i,1} = c'_{i,1}$), then the eliminated candidate at the second round will be the same on both sides, and the winner as well, contradicting our hypothesis.

Hence, c_0 occurs as the first choice in some votes. Let $i_0, \ldots, i_k$ the indices of all such votes. We have $c_{i_j,1} = c'_{i_j,1} = c_0$ for any $j \in \{0, \ldots, k\}$. If the second choice is the same in all these votes, *i.e.* for $j \in \{0, \ldots, k\}$, we have $c_{i_j,2} = c'_{i_j,2}$, then the eliminated candidate at the second round, and thus the winner, would be the same on both sides, which contradicts our hypothesis.

Therefore, there exists $j \in \{i_0, \ldots, i_k\}$ such that $v_j = (c_0, c_1, c_2)$, $v'_j = (c_0, c_2, c_1)$ where $\{c_0, c_1, c_2\} = \{a, b, c\}$. We keep v_j, but we need more, as $\mathsf{Count}^3_{\mathsf{STV}}([v_j]) = \mathsf{Count}^3_{\mathsf{STV}}([v'_j]) = c_0$. Since c_0 is eliminated at the first round:

1. Either $c_0 = a$ and there exist j_1, j_2 such that $c_{j_1,1} = c'_{j_1,1} = b$, and $c_{j_2,1} = c'_{j_2,1} = c$. Keeping these two votes in addition to v_j/v'_j, we have that $\mathsf{Count}^3_{\mathsf{STV}}([v_j, v_{j_1}, v_{j_2}]) \neq \mathsf{Count}^3_{\mathsf{STV}}([v'_j, v'_{j_1}, v'_{j_2}])$.
2. Or $c_0 = b$ and there exist j_1, j_2, j_3 (all distinct) such that $c_{j_1,1} = c'_{j_1,1} = a$, $c_{j_2,1} = c'_{j_2,1} = a$, and $c_{j_3,1} = c'_{j_3,1} = c$. Keeping these three votes in addition to v_j/v'_j, we have that $\mathsf{Count}^3_{\mathsf{STV}}([v_j, v_{j_1}, v_{j_2}, v_{j_3}]) \neq \mathsf{Count}^3_{\mathsf{STV}}([v'_j, v'_{j_1}, v'_{j_2}, v'_{j_3}])$.
3. Or $c_0 = c$ and there exist distinct j_1, j_2, j_3, j_4 such that $c_{j_1,1} = c'_{j_1,1} = a$, $c_{j_2,1} = c'_{j_2,1} = a$, $c_{j_3,1} = c'_{j_3,1} = b$, and $c_{j_4,1} = c'_{j_4,1} = b$. Keeping these four votes in addition to v_j/v'_j, we get $\mathsf{Count}^3_{\mathsf{STV}}([v_j, v_{j_1}, v_{j_2}, v_{j_3}, v_{j_4}]) \neq \mathsf{Count}^3_{\mathsf{STV}}([v'_j, v'_{j_1}, v'_{j_2}, v'_{j_3}, v'_{j_4}])$.

We conclude that at most 5 votes are needed to ensure the result will be different. □