No More Excuses: Automated Synthesis of Practical and Verifiable Vote-Counting Programs for Complex Voting Schemes

Lyria Bennett Moses[1], Rajeev Goré[2], Ron Levy[3], Dirk Pattinson[2(✉)], and Mukesh Tiwari[2]

[1] Faculty of Law, UNSW, Sydney, Australia
[2] Research School of Computer Science, ANU, Canberra, Australia
dirk.pattinson@anu.edu.au
[3] College of Law, ANU, Canberra, Australia

Abstract. We argue that electronic vote-counting software can engender broad-based public trust in elections to public office only if they are formally verified against their legal definition and only if they can produce an easily verifiable certificate for the correctness of the count. We then show that both are achievable for the Schulze method of vote-counting, even when the election involves millions of ballots. We argue that our methodology is applicable to any vote-counting scheme that is rigorously specified. Consequently, the current practice of using unverified and unverifiable vote counting software for elections to public office is untenable. In particular, proprietary closed source vote-counting software is simply inexcusable.

1 Introduction

The integrity of electronic elections depends on many factors and spans the entire process from vote-casting to vote-counting, and the determination of winners. The notion of *universal verifiability* of vote counting (any voter can check that the announced result is correct on the basis of the published ballots [15]) has long been recognised as being central, both for guaranteeing correctness, and building trust, in electronic elections. For vote-counting (on which we focus in this paper), verifiability means that every stakeholder, or indeed any member of the general public, has the means to check that the computation of election winners is correct.

In practice, however, the computer software used to determine winners from the set of ballots cast, offers no such assurance. This applies for example to the software used in the Australian state of New South Wales (where the vote-counting software is closed-source and proprietary) and to the eVACS system used in the Australian Capital territory (where the vote-counting software has been open-sourced), see e.g. [2,9,10].

In this paper, we argue that both *verification* of the computer software that counts votes, and *verifiability* of individual counts are critical for building trust in

R. Krimmer et al. (Eds.): E-Vote-ID 2017, LNCS 10615, pp. 66–83, 2017.
DOI: 10.1007/978-3-319-68687-5_5

an election process where ballots are being counted by computer. We moreover demonstrate by means of a case study that both are achievable for elections of realistic sizes. Given the mission-critical importance of correctness of vote-counting, both for the legal integrity of the process and for building public trust, together with the fact that both can be achieved technologically, we argue that it is imperative to replace the currently used, black-box software for vote-counting with a counterpart that is both verified, and produces verifiable certificates that guarantee correctness.

The leading analogy that informs our notion of verifiability of ballot counting is that of counting by hand. We argue that the result of a count is correct, if we have evidence that every action performed by a counting official is consistent with the (legal) description of the voting protocol. In a setting where votes are counted by hand, this is precisely the duty (and purpose) of election scrutineers. In the absence of scrutineers, and as a thought experiment that is evidently impractical, one can envisage one, or several, cameras that record the entire vote-counting process to a level of detail that allows us to ascertain the validity of every step that has been undertaken to determine the election result.

Correctness can then be verified independently by an analysis of the recording, and potential errors can be identified by exhibiting precisely that part of the recording where an incorrect action has been taken. The notion of certificate for the correctness of an electronic count implements this metaphor: instead of producing a recording of a hand-count, we record every individual step that has been undertaken by the software to determine the outcome electronically. We understand this data as a *certificate* that can then subsequently be either machine-checked in its entirety, or spot-checked for validity by humans.

This de-couples the process of *counting* the votes by computer from the process of *verifying* that ballots have been correctly tallied. We argue this notion of externally certifying electronic vote counting, together with transparency of the entire process, is imperative to building public trust in the integrity of electronic vote counting.

Our goal is therefore to closely integrate three pieces of data: the set of ballots cast, the winner(s) of the election, and the certificate data that links both. This leads to the following key requirements:

1. the ability to verify that a certificate is correctly constructed
2. the ability to verify that the certificate is indeed based on the ballots cast
3. the ability to verify that a correctly constructed certificate indeed provides evidence for the claimed set of winners.

As long as all three requirements are met, we accept *any* valid certificate as evidence of the correctness of the count, irrespective of the means by which it was constructed. In particular, this completely eliminates the need for trust in computer hardware or individuals operating the computing machinery.

We demonstrate, by means of a case study, that all three requirements can be met simultaneously, and that the software that produces these results scales to the size of real-world elections.

For the case study, we choose the Schulze method [26]. Despite the fact that the Schulze Method is not used for elections to public office, it provides an interesting case study for our purposes, as there is a non-trivial gap to bridge between certificates and the winning conditions of the vote counting scheme (item 3 above). We close this gap by giving formal proofs that connect certificates with the actual winning conditions.

One important aspect of our work is that it adds value to the integrity of the electoral process along several dimensions. First, the formal specification enforces a rigid analysis of the voting protocol that eliminates all disambiguities inherent in a textual specification of the protocol. While an independent re-implementation of the protcol (assuming that all votes are published) may give assurance of the officially announced result, the question of correctness remains open if both programs diverge. Checking the validity of a certificate, on the other hand, allows us to precisely pinpoint any discrepancies. Finally, it is much simpler to implement a program that *validates* a certificate compared to a fully-blown re-implementation of the entire voting protocol which increases the number of potential (electronic) scrutineers.

Related Work. This paper discusses the notions of verification, and verifiability, from the perspective of law and trust, and we use the Schulze method as an example to show that both can be achieved in realistic settings. We do not describe the formalisation in detail which is the subject of the companion paper [24]. Apart from the analysis of verifiably correct vote-counting from the perspective of law and trust, the main technical differences between this paper and its companion is scalability: we refine both proofs and code given in [24] to effortlessly scale to millions of ballots.

Formal specification and verification of vote-counting schemes have been discussed in [3,8] but none of these methods produce verifiable results, and as such rely on trust in the tool chain that has been used in their production. The idea of evidence for the correctness of a count has been put forward in [23] as a technical possibility. This paper complements the picture by (a) establishing verification and verifiability also as legal desiderata, and (b) showing, by means of a case study, that both can be achieved for real-world size elections.

2 Verification and Verifiability

Verification is the process of proving that a computer program implements a specification. Here, we focus on *formal* verification [12], where the specification consists of formulae in a formal logic, and the correctness proof of a program consists of applying logical deduction rules. This takes place inside a (formal) theorem prover that then validates the correctness of each and every proof step. One crucial aspect of this is that every correctness proof itself is machine-checked which gives the highest possible level of correctness, as the proof-checking functionality of a theorem prover is a comparatively small and heavily scrutinised part of the entire system.

As a consequence, once we are satisfied with the fact that the *specification* of the program indeed expresses the intended notion of correctness, we have *very* high assurance that the results of the computation are indeed correct.

In order to ascertain that the results of a verified program are indeed correct, one therefore needs to

1. read, understand and validate the formal specification: is it error free, and does it indeed reflect the intended functionality?
2. scrutinize the formal correctness proof: has the verification been carried out with due diligence, is the proof complete or does it rely on other assumptions?
3. ensure that the computing equipment on which the (verified) program is executed has not been tampered with or is otherwise compromised, and finally
4. ascertain that it was indeed the verified program that was executed in order to obtain the claimed results.

The trust in correctness of any result rests on all items above. The last two items are more problematic as they require trust in the integrity of equipment, and individuals, both of which can be hard to ascertain once the computation has completed. The first two trust requirements can be met by publishing both the specification and the correctness proof so that the specification can be analysed, and the proof can be replayed. Both need a considerable amount of expertise but can be carried out by (ideally more than one group of) domain experts. Trust in the correctness of the result can still be achieved if a large enough number of domain experts manage to replicate the computation, using equipment they know is not compromised, and running the program they know has been verified. As such, trust in *verified* computation mainly rests on a relatively small number of domain experts.

The argument for correctness via *verification* is that we have guarantees that *all* executions of a program are correct, and we therefore argue that this in particular applies to any one given instance.

Verifiability, on the other hand, refers to the ability to independently ascertain that a *particular* execution of a program did deliver a correct result. This is usually achieved by augmenting the computation so that it additionally produces a certificate that can be independently checked, and attests to the correctness of the computation, see e.g. [1]. Attesting to the correctness of the computation therefore requires to

1. ensure that the certificate is valid (usually by means of machine-checking it)
2. ensure that the certificate is indeed associated to the computation being scrutinized, i.e. it matches both input and output of the computation
3. establish that a valid certificate indeed guarantees the correctness of the computation.

Here, the first two items are mechanical and can be accomplished by relatively simple and short, independently developed computer programs for which little expert knowledge, other than basic programming skills, are necessary. The difficulty lies in establishing the third requirement: to verify that a correct certificate indeed implies the correctness of the result.

To maximise trust, reliability and auditability of electronic vote counting, we argue that both approaches need to be combined. To ensure (universal) verifiability, we advocate that vote-counting programs do not only compute a final result, but additionally produce an independently verifiable certificate that attests to the correctness of the computation, *together* with a formal verification that valid certificates indeed imply the correct determination of winners. In other words, we solve the problem outlined under (3) above by giving a *formal*, machine-checkable proof of the fact that validity of certificates indeed implies correct determination of winners. In contrast to scrutiny sheet published by electoral authorities, a certificate of this type contains all the data needed to reconstruct the count.

Given a certificate-producing vote-counting program, external parties or stakeholders can then satisfy themselves to the correctness of the count by checking the certificate (and whether the certificate matches the election data), and validate, by means of machine-checking the formal proof given for item (3) that validity of certificates indeed entails the correctness of the count. In particular, once it has been established (and vetted by a large enough number of domain experts) that valid certificates do indeed imply correctness, this step does not have to be repeated for each individual election. For every particular election, trust in the correctness of the count can be established solely by machine-checking the generated certificates. As argued above, this task can be accomplished by a large class of individuals with basic programming skills.

In fact, we go one step further: we demonstrate that fully verified programs can be employed to count real-size elections that involve millions of ballots and produce both independently verifiable and provably correct certificates. While the use of verified programs is not essential for building trust in the correctness of the count (as long as certificates are validated), it gives us formal assurance that the certificates produced will *always* be valid.

3 Legal Aspects of Verification and Verifiability

Any system for counting votes in democratic elections needs to satisfy at least three conditions: (1) each person's vote must be counted accurately, according to a mandated procedure, (2) the system and process should be subjectively trusted by the electorate, (3) there should be an objective basis for such trust, or in other words the system must be trustworthy. While subjective trust cannot be guaranteed through greater transparency [21], transparency about both the voting system and the actual counting of the vote in a particular election are important in reducing errors and ensuring an accurate count, promoting public trust and providing the evidential basis for demonstrated trustworthiness. In particular, it is a lack of transparency that has been the primary source of criticism of existing systems, both in the literature [7,9] and among civil society organisations [27] (for example, blackboxvoting.org and trustvote.org). International commitment to transparency is also demonstrated through initiatives such as the Open Government Partnership. Another important concept referred to both

in the literature and by civil society organisations is public accountability, which requires both giving an "account" or explanation to the public and when called on (for example, in court) as well as being held publicly responsible for failures. Transparency is thus a crucial component of accountability, although the latter will involve other features (such as enforcement mechanisms) that are beyond the scope of this paper.

There are two contexts in which transparency is important in the running of elections. First, there should be transparency in Hood's sense [14] as to the process used in elections generally. This is generally done through legislation with detailed provisions specifying such matters as the voting method to be used as well as the requirements for a vote to count as valid. In a semi-automated process, this requires a combination of legislation (instructions to humans) and computer code (instructions to machines). The second kind of transparency, corresponding to Meijer's use of the term [20], is required in relation to the performance of these procedures in a specific election. In a manual process, procedural transparency is generally only to intermediaries, the scrutineers, who are able to observe the handling and tallying of ballot papers in order to monitor officials in the performance of their tasks. While the use of a limited number of intermediaries is not ideal, measures such as allowing scrutineers to be selected by candidates (e.g. Commonwealth Electoral Act 1918 (Australia) s 264) promote public confidence that the procedure as a whole is unbiased. However imperfect, procedural transparency reduces the risk of error and fraud in execution of the mandated procedure and enhances trust.

Electronic vote counting ought to achieve at least a similar level of transparency along both dimensions as manual systems in order to promote equivalent levels of trust. Ideally, it would go further given physical limitations (such as the number of scrutineers able to fit in a room) apply to a smaller part of the process. The use of a verified, and fully verifiable system is transparent in both senses, with members of the public able to monitor both the rules that are followed and the workings and performance of the system in a particular instance.

First, the vote counting procedure needs to be transparent. For electronic vote counting, the procedure is specified in both legislation (which authorises the electronic vote counting procedure) and in the software employed. The use of open source code ensures that the public has the same level of access to instructions given to the computer as it has to legislative commands given to election officials. The use of open source code is crucial as is demonstrated through a comparison of different jurisdictions of Australia. In Australia, for example, the Federal Senate and NSW state election vote counting are based on proprietary black box systems while the Australian Capital Territory uses open source eVACS software [2,9,10]. This has significant impact on the ability of researchers to detect errors both in advance of elections and in time to correct results [9]. Private verification systems have been less successful, in both Australia and the US, in providing equivalent protection against error to open source software [7,9]. Further, private verification provides a lower level of public transparency than the use of manual

systems which rely on public legislation (instructions to humans) as the primary source of vote counting procedures [7]. It should also be noted that there are few public advantages in secrecy since security is usually enhanced by adopting an open approach (unless high quality open source vote counting software were unavailable), and private profit advantages are outweighed by the importance of trust in democratic elections.

Second, verifiability provides a method of ascertaining the correctness of results of a specific election. External parties are able to check a certificate to confirm that the counting process has operated according to the rules of the voting procedure. Under a manual process, tallying and counting can only be confirmed by a small number of scrutineers directly observing human officials. The certification process allows greater transparency not limited to the number of people able to fit within a physical space, although we recognise that physical scrutiny is still required for earlier elements of the voting and vote counting process (up to verification of optical scanning of ballots). Certification reduces the risk of error and fraud that would compromise accuracy and provides an evidence-base for trustworthiness. It is also likely to increase subjective public trust, although this will require engagement with the public as to the nature of verification involved. While it is likely that in practice checking will be limited to a small group with the technical expertise, infrastructure and political interest to pursue it, knowledge as to the openness of the model is likely to increase public trust. Currently in Australia, neither open source nor proprietary vote counting systems provide an equivalent level of procedural transparency for monitoring the count in a particular election (for example, compare Commonwealth Electoral Act 1918 (Australia) s 273A).

Ultimately, legislation, computer code (where relevant) and electoral procedures need to combine to safeguard an accurate count in which the public has justified confidence. The verification and verifiability measures suggested here go further to ensure this than current methods used in Australia and, as far as we are aware, public office elections around the world.

In the remainder of the paper, we describe a particular voting method (the Schulze method) to demonstrate that we can achieve both verification and verifiability for real-world size elections.

4 The Schulze Method

The Schulze Method [26] is a preferential voting scheme that elects a single winner. While not used for elections to public office, it provides us with an example that show-cases all aspects of verifiability discussed in Sect. 2, as the correspondence between valid certificates and election winners is not trivial, i.e. a valid certificate cannot immediately be matched to the winning condition. We bridge this gap by a formal proof that we outline in Sect. 5.

In Schulze counting, each ballot expresses a preference ordering over the set of candidates where all candidates need to be ranked, but candidates may be given equal preference. The requirement of ranking all candidates can be relaxed by assuming that non-ranked candidates tie for last position.

From a social choice perspective, the Schulze voting scheme has been shown to satisfy a large number of desirable properties, such as monotonicity, independence of clones, and reversal symmetry, established in the original paper [26].

From a game theoretic perspective, it has also been experimentally established that the Schulze Method is better than other, more established voting schemes such as plurality and instant-runoff voting and Borda count [25]. Despite the fact that the Schulze method isn't used in elections for public office, there is rapid uptake in a large number of organisations, including e.g. various national branches of the Pirate Party and numerous open software initiatives.

Academically, the Schulze method has been investigated further, and it has been established that Schulze voting is resistant to bribery and control [22] in the sense that both problems are computationally infeasible, but have been found to be fixed-parameter tractable with respect to the number of candidates [13].

The Schulze Method is guaranteed to always elect a Condorcet winner, that is, a candidate that a majority prefers to every other candidate in a pairwise comparison.

The distinguishing feature of Schulze counting is the resolving of cycles in collective preferences. These situations appear to arise in real elections [16] and it has been demonstrated that different choices of resolving cycles indeed lead to different outcomes. Consider for example the following scenario taken from [6] where we have three candidates A, B, and C and the following distribution of votes:

$$4 : A > B > C \qquad 3 : B > C > A \qquad 2 : B > A > C \qquad 4 : C > A > B$$

where the number before the colon indicates the multiplicity of the vote, and $>$ indicates the order of preference so that e.g. $3 : B > C > A$ denotes three votes where B is preferred over C who is in turn preferred over A. In this example, a majority of candidates prefer A over B as eight ballots prefer A over B compared to five ballots preferring B over A. Similarly, a majority of candidates prefer B over C, and a majority prefer C over A, leading to a cyclic collective preference relation.

The main idea of the method is to resolve cycles by considering *transitive preferences* or a generalised notion of margin. That is, if $m(c, d)$ is the margin between candidates c and d (the difference between the number of votes that rank c higher than d and the number of votes that rank d higher than c), Schulze voting considers *paths* of the form

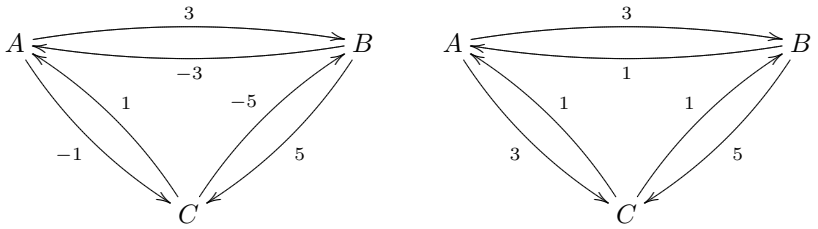
$$c_1 \xrightarrow{m(c_1,c_2)} c_2 \xrightarrow{m(c_2,c_3)} c_3 \qquad \cdots \qquad c_{n-1} \xrightarrow{m(c_{n-1},c_n)} c_n$$

i.e. sequences of candidates annoted with the margin between successive candidates. A path like the above induces the path-based margin $\min\{m(c_i, c_{i+1} \mid 1 \leq i < n\}$ between c_1 and c_n given as the minimum of the margins between successive candidates, and the *generalised margin* between two candidates c and d is the largest path-based margin considering all possible paths between c and d.

This allows us to replace the *margin* used to determine Condorcet winners by the *generalised margin* introduced above. The key result of Schulze's paper [26] (Sect. 4.1) is that the induced ordering is now indeed transitive.

A *Schulze Winner* can then be taken to be a candidate that is not defeated by any other candidate in a pairwise comparision, using generalised margins. In symbols, candidate c is a winner if $g(c,d) \geq g(d,c)$ for all other candidates d, where $g(\cdot,\cdot)$ denotes generalised margins.

In the above example, we have the following margins (on the left) and generalised margins (on the right):



Note that the margins on the left are necessarily symmetric in the sense that $m(x,y) = -m(y,x)$ as margins are computed as the difference between the number of ballots that rank x higher than y and the number of ballots that rank y higher than x. This property is no longer present for generalised margins, and A is the (only) winner as A wins every pairwise comparison based on generalised margins. In summary, vote counting according to the Schulze method can be described as follows:

1. compute the *margin function* $m(c,d)$ as the number of ballots that strictly prefer c over d, minus the number of ballots that strictly prefer d over c
2. compute the *generalised margin function* $g(c,d)$ as the maximal path-based margin between c and d
3. Compute winning candidate, i.e. candidates for which $g(c,d) \geq g(c,d)$, for all other candidates d, and apply tie-breaking if more than one winning candidate has been elected.

It has been shown in Schulze's original paper that at least one winner in this sense always exists, and that this winner is unique in most cases, i.e. Schulze counting satisfies the resolvability criterion.

5 Provably Correct and Verifiable Schulze Counting

Our implementation of the Schulze method consists of three parts:

Formal Specification. First, we provide a formal specification of the winning condition for elections counted according to Schulze. This takes the form of a logical formula that directly reflects the voting scheme.

Certificate. Second, we establish what counts as a *certificate* for the winning condition to hold, and give a formal proof that existence of a certificate for winning is equivalent to winning in the sense of the initial specification. The main difference between both notions of winning is that the former is a mere logical assertion, whereas the latter is formulated in terms of verifiable data.

Proofs. Third, we provide a full proof of the fact that the existence of a certificate is logically equivalent to the specification being met. Moreover, we give a full proof of the fact that winners can always be computed correctly, and certificates can be produced, for any set of ballots.

We exemplify the relationship of these components with a simple example. Consider the notion of being a list of integers sorted in ascending order. The *formal specification* of this operation consists of two sentences:

- the elements of the resulting list should be in ascending order
- the elements of the resulting list should be a permutation of the elements of the input list.

In this case, we don't need to certify that a list is sorted: this can be checked easily (in linear time). Ascertaining that the result list is a permutation of the input list is (slightly) less trivial. Here, a *certificate* can be a recipe that permutes the input list to the resulting list: to verify that the resulting list is indeed a permutation, the only thing the verifier needs to do is to alter the input list according to the given recipe, and then checking whether the two lists are equal.

In this case, the computation produces *two* pieces of data: we not only get to see the sorted list, but also a permutation that witnesses that the resulting list is a permutation of the input list. A *proof* then amounts to establishing that given

- the input list, and the (sorted) resulting list, and
- a recipe that permutes the input list to its sorted version

we can conclude that the sorting operation indeed meets the formal specification.

The main (and important) difference between the specification and the certificate is that the former is merely a proposition, i.e. a statement that can either be true or false. The certificate, on the other hand, gives us concrete means to *verify* or ascertain the truth of the specification. The proofs provide the glue between both: if a certificate-producing computation delivers a result together with a certificate, we in fact know that the specification holds. On the other hand, we need to establish that every correct computation can in fact be accompanied by a valid certificate.

For vote counting, our development takes place inside the Coq theorem prover [5] that is based on the Calculus of Inductive Constructions. Technically, Coq distinguishes logical formulae or propositions (that are of type `Prop`) from data (that is of type `Set` or `Type`). The former are correctness assertions and are erased when programs are generated from proofs, whereas the latter are computed values that are preserved: our certificates therefore need to be `Type`s. To give a simple example, a function that sorts a list will take a list (say, of integers)

and produce a sorted list, where the fact that the list is sorted is expressed as a proposition, so that sorted lists are pairs where the second component is a proof that the first component is ordered, and is deleted by extraction. To make sorting of lists *verifiable*, we would need to *additionally* output a certificate, i.e. data from which we can infer that the result list is really a permutation of the input list.

The logical specification of the winning condition is based on an integer-valued *margin function* and a *path* between candidates:

```
Variable marg : cand -> cand -> Z.

Inductive Path (k: Z) : cand -> cand -> Prop :=
| unit c d : marg c d >= k -> Path k c d
| cons c d e : marg c d >= k -> Path k d e -> Path k c e.
```

Paths are additionally parameterised by integers that give a lower bound on the path-based margin (the *strength* of the path in the terminology of [26]). We interpret an assertion `Path k c d` as the existence of a path between `c` and `d` that induces a path-based margin of at least `k`. Such a path can be constructed if the margin between `c` and `d` is $\geq k$ (the `unit` constructor). Alternatively, a path between `c` and `e` of strength $\geq$ `k` can be obtained if there is there is candidate `d` for which the margin between `c` and `d` is $\geq$ `k` and `d` and `e` are already connected by a path of strength $\geq$ `k` (via the `cons` constructor). This gives the following formula that expresses that a candidate `c` wins a Schulze vote:

```
Definition wins_prop (c: cand) :=
 forall d : cand, exists k : Z,
  Path k c d /\ (forall l, Path l d c -> l <= k).
```

Simply put, it says that for each candidate `d`, there exists an integer `k` and a path from `c` to `d` of strength `k`, and any other path going the reverse direction induces at most the same path-based margin. In terms of the generalized margin function, candidate `c` wins, if for every candidate `d`, the generalized margin between `c` and `d` is greater than or equal to the generalised margin between `d` and `c`. We reflect the fact that the above is a logical proposition in the name of the formula. The certificate for winning then needs to consist of data that evidences precisely this.

One crucial component of a certificate that evidences that a particular candidate `c` is a Schulze-winner therefore consists of displaying a sufficiently strong path between `c` and any other candidate. We achieve this by pairing the propositional notion of path with a type-level notion `PathT` that can be displayed as part of a certificate for winning, and will not be erased by extraction.

```
Inductive PathT (k: Z) : cand -> cand -> Type :=
| unitT c d : marg c d >= k -> PathT k c d
| consT c d e : marg c d >= k -> PathT k d e -> PathT k c e.
```

The second part of the winning condition, i.e. the non-existence of a stronger path going the other way, is more difficult to evidence. Rather than listing all

possible paths going the other way, we use *co-closed sets* of pairs of candidates which leads to smaller certificates. Given an integer `k`, a set $S \subseteq$ `cand` $\times$ `cand` of candidate pairs is k-coclosed if none of its elements `(c, d)` can be connected by a path of strength `k` or greater. This means that

- for any element `(c, d)` $\in S$, the margin between `c` and `d` is $<$ `k`, and
- if `(c, d)` is in the co-closed set and `m` is a candidate (a "midpoint"), then either the margin between `c` and `m` is $<$ `k`, or `m` and `d` cannot be connected by a path of strength $\geq$ `k`.

The second condition says that `c` and `d` cannot be connected by a path of the form `c, m, ..., d` whose overall strength is $\geq$ `k`.

We represent co-closed sets by boolean functions of type `cand -> cand -> bool` and obtain the following formal definitions:

```
Definition coclosed (k : Z) (f : (cand * cand) -> bool) :=
 forall x, f x = true -> W k f x = true.
```

where `W: (cand -> cand -> bool) -> (cand -> cand -> bool)` is an operator on sets of pairs of candidates that is given by

```
Definition W (k: Z) (p: cand * cand -> bool) (x: cand * cand) :=
 andb (marg_lt k x)
 (forallb (fun m => orb (marg_lt k (fst x, m)) (p (m, snd x)))
          cand_all).
```

and `marg_lt` is a boolean function that decides whether the margin between two candidates is less than a given integer, and `cand_all` is a list containing all candidates that stand for election.

The *certificate* for a candidate `c` to be winning can then be represented by a table where for every other (competing) candidate `d`, we have

- an integer `k` and a path from `c` to `d` of strength `k`, and
- a `k+1`-coclosed set that evidences that no path of strength $>$ `k` exists between `d` and `c`.

This leads to the following definition and equivalence proof where f plays the role of coclosed set:

```
    Definition wins_type c := forall d : cand,
    existsT (k : Z), ((PathT k c d) *
      (existsT (f : (cand * cand) -> bool),
        f (d, c) = true /\ coclosed (k + 1) f))%type.

Lemma wins_type_prop : forall c, wins_type c -> wins_prop c.
Lemma wins_prop_type : forall c, wins_prop c -> wins_type c.
```

and `existsT` is a type-level existential quantifier (technically, a Σ-type).

Going back to the trichotomy of specification, certificate and proof outlined at the beginning of the section, the first lemma (`wins_type_prop`) says that the existence of a certificate indeed implies the validity of the specification. The second lemma (`wins_prop_type`) tells us that the notion of the certificate is so that any correct computation can indeed be certified. That is, the notion of certificate is general enough to certify *all* correct computations.

It is precisely the formal proof of equivalence of both notions of winning that formally justifies our notion of certificate, as it ensures that a valid certificate *indeed* witnesses the winning condition. This implements the third requirement discussed on page 2.

The considerations so far rely on a previously computed margin function. To obtain a formal specification and ensuing notion of certificates, all we need to do is to provide a way of constructing the margin function step-by-step. We do this by exhibiting two stages of the count:

1. in the first state, we process all ballots and iteratively update the margin function until all ballots have been processed. This gives us the margin function on which subsequent computations are based.
2. in the second step, we compute winners, and evidence for winning, on the basis of the margin function we have constructed in the first step.

The complete specification then takes the form of an inductive type that *only* allows us to construct valid stages of the count. In more detail, we have four constructors:

- `ax` where we construe all ballots as uncounted, and start with the zero margin
- `cvalid` where we update the margin function based on a formal ballot
- `cinvalid` where we discard an informal ballot and do not change the margin
- `fin`, where we assume that all ballots have been processed, and we finalise the count by providing winners, and evidence for winning.

As a consequence, *every* element of this type represents a valid state of the computation, and a count in state `fin` describes the result of the process.

```
Inductive Count (bs : list ballot) : State -> Type :=
  | ax us m : us = bs -> (forall c d, m c d = 0) ->
    Count bs (partial (us, []) m)          (* zero margin      *)
  | cvalid u us m nm inbs : Count bs (partial (u :: us, inbs) m) ->
    (forall c, (u c > 0)%nat) ->           (* u is valid       *)
    (forall c d : cand,
      ((u c < u d) -> nm c d = m c d + 1) (* c preferred to d *) /\
      ((u c = u d) -> nm c d = m c d)     (* c, d rank equal  *) /\
      ((u c > u d) -> nm c d = m c d - 1))(* d preferred to c *) ->
    Count bs (partial (us, inbs) nm)
  | cinvalid u us m inbs : Count bs (partial (u :: us, inbs) m)  ->
    (exists c, (u c = 0)%nat)              (* u is invalid     *) ->
    Count bs (partial (us, u :: inbs) m)
  | fin m inbs w (d: (forall c, (wins_type m c)+(loses_type m c))):
```

```
    Count bs (partial ([], inbs) m)       (* no ballots left  *) ->
    (forall c, w c = true <-> (exists x, d c = inl x)) ->
    (forall c, w c = false <-> (exists x, d c = inr x)) ->
    Count bs (winners w).
```

The formulation above relies on the following assumptions. First, ballots are represented as functions from candidates into natural numbers that represent the ranking. We assume that preferences start with 1 and interpret 0 as the failure to denote a preference for a given candidate which renders the vote invalid. A `State` is either a partial count that consists of a list of unprocessed ballots, a list of informal ballots, and a partially constructed margin function, or of a boolean function that determines the election winners. We have elided the definition of *losing* that is dual to that of winning.

The task of computing the winners of a Schulze election given a list `bs` of ballots is then reduced to exhibiting a boolean function `w: cand -> bool` that determines the winners, and an element of the type `Count bs (winners w)`. While the first part (the boolean function) is the result of the computation, the second part (the element of the `Count`-type) consists of the verifiable certificate for the correctness of the count.

We exemplify the nature of certificates by returning to the example presented in Sect. 4. We construe e.g. the ballot $A > B > C$ as the function $A \mapsto 1$, $B \mapsto 2$ and $C \mapsto 3$. Running a Schulze-election then corresponds to executing the function that computes winners, which produces the following certificate (we have added some pretty-printing):

```
V: [A1 B2 C3,..], I:  [], M: [AB:0 AC:0 BC:0]
-----------------------------------------
V: [A1 B2 C3,..], I:  [], M: [AB:1 AC:1 BC:1]
-----------------------------------------
             . . .
-----------------------------------------
V: [A2 B3 C1], I:  [], M: [AB:2 AC:0 BC:6]
---------------------------------------
V: [], I: [], M: [AB:3 AC:-1 BC:5]
---------------------------------
winning: A
   for B: path A --> B of strenght 3, 4-coclosed set:
     [(A,A),(B,A),(B,B),(C,A),(C,B),(C,C)]
   for C: path A --> B --> C of strenght 3, 4-coclosed set:
     [(A,A),(B,A),(B,B),(C,A),(C,B),(C,C)]
losing: B
   exists A: path A --> B of strength 3, 3-coclosed set:
     [(A,A),(B,A),(B,B),(C,A),(C,B),(C,C)]
losing: C
   exists A: path A --> B --> C of strength 3, 3-coclosed set:
     [(A,A),(B,A),(B,B),(C,A),(C,B),(C,C)]
```

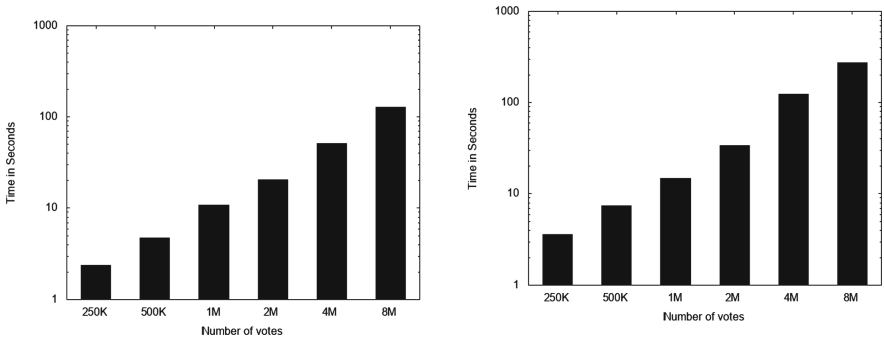
The initial stages are the construction of the margin function, where the first component are the ballots to be processed. Here, a ballot of the form `A2, B3, C1` represents a first preference for `C`, a second preference for `A` and a third preference for `B`. The partial margin function is displayed in the rightmost column, and lists pairwise margins, for example AB:1 encodes $m(A, B) = 1$. Note that the margin function is symmetric, i.e. $m(x, y) = -m(y, x)$ so that the above is a complete representation. We do not have any invalid votes so that the `I`-component always remains empty. The ellipsis (`. . .`) indicates the omission of some steps of constructing the margin which we have elided to save space. Once the margin is fully constructed, we present evidence, in this case, for `A` winning the election (and everybody else losing). As described above, this evidence consists of a path, and a coclosed set, for each candidate distinct from `A`. The subsequent entries (that we haven't discussed in this paper) show that every candidate except `A` is not winning. A losing candidate (in this example, e.g. `B`) is a candidate for which there exists a competitor (here: `A`) so that the generalised margin of `A` over `B` is strictly larger than the generalised margin of `B` over `A`. This is evidenced similarly to winning candidates, by giving a path and a co-closed set.

6 Experimental Results

We report on the results of implementing the Schulze method in the Coq theorem prover [5] that automatically extracts into the OCaml programming language [18]. Coq comes with an extraction mechanism [19] that allows us to extract both functions and proofs into executable code via the Haskell, Ocaml and Scheme programming languages. As Coq is written in OCaml itself, the OCaml extraction mechanism is the best developed, and OCaml produces faster executables than the other two languages. As Coq is based on constructive logic, we can turn both functions written in Coq, as well as proofs into executable code. Given that the correctness of the count is witnessed by an inductive data type, counting itself amounts to populating this type, and we use a mix of proofs (showing that a count exists amounts to a function that produces a count) and verified functional programs (that compute data directly), using the latter for performance-critical tasks.

The most performance critical aspect of our code is the margin function. Recall that the margin function is of type `cand -> cand -> Z` and that it depends on the *entire* set of ballots. Internally, it is represented by a closure [17] so that margins are re-computed with every call. The single largest efficiency improvement in our code was achieved by memorization, i.e. representing the margin function (in Coq) via list lookup. With this (and several smaller) optimisation, we can count millions of votes using verified code. Below, we include our timing graphs, based on randomly generated ballots while keeping number of candidates constant i.e. 4.

On the left, we report timings (in seconds) for the computation of winners, whereas on the right, we include the time to additionally compute a universally verifiable certificate that attests to the correctness of the count. This is consistent



(a) Computation of Winners

(b) Computation of Winners and Certificate

Fig. 1. Experimental results

with known computational complexity of Schulze counting i.e. linear in number of ballots and cubic in candidates. The experiments were carried out on a system equipped with Intel core i7 processor and 16 GB of RAM. We notice that the computation of the certificate adds comparatively little in computational cost (Fig. 1).

At this moment, our implementation requires that we store *all* ballots in main memory as we need to parse the entire list of ballots before making it available to our verified implementation so that the total number of ballots we can count is limited by main memory in practise. We can count real-world size elections (8 million ballot papers) on a standard, commodity desktop computer with 16 GB of main memory.

7 Discussion and Further Work

This paper argues that there is no excuse to use vote counting software in elections to public office (or otherwise) that is neither *verified* (i.e. the correctness of the software has been established using formal methods) nor *verifiable* (i.e. stakeholders can independently ascertain the correctness of individual executions of the software). We have argued that both verification and verifiability are desiderata from the perspective of law and trust. Finally, our experimental results show that both verification and verifiability can be achieved in realistic settings.

Our case study (Schulze voting) was chosen as it showcases how we can bridge a non-trivial gap between certificates and the winning conditions of the voting scheme under consideration. Despite the fact that the Schulze method is not used for elections to public office, we are convinced that the same programme can (and should!) be carried out for other preferential voting schemes.

As the precise notion of certificate depends on the exact description of the voting protocol, it is clear that this paper merely provides a case study. For other voting systems, in particular the notion of certificate needs to be adapted

to in fact witness the correctness of the determination of winners. As a toy example, this has been carried out for first-past-the-post (plurality) voting and a simple version of single transferable vote [23]. The more realistic scenario of single transferable vote with fractional transfer values is being considered in [11] where real-word size case studies are being reported. Given that the nature of certificates is crucially dependent on the voting protocol under scrutiny, the complexity and size of certificates necessarily differs from case to case. While our general experience seems to indicate that computing certificates incurs little overhead, this remains to be investigated more formally.

One aspect that we have not considered here is encryption of ballots to safeguard voter privacy which can be incorporated using protocols such as shuffle-sum [4] and homomorphic encryption [28]. The key idea here is to formalise a given voting scheme based on encrypted ballots, and then to establish a homomorphic property: the decryption of the result obtained from encrypted ballots is the same as the result obtained from the decrypted ballots. We leave this to further work.

References

1. Arkoudas, K., Bringsjord, S.: Computers, justification, and mathematical knowledge. Minds Mach. **17**(2), 185–202 (2007)
2. Australian Electoral Commission. Letter to Mr Michael Cordover, LSS4883 Outcome of Internal Review of the Decision to Refuse your FOI Request no. LS4849 (2013). http://www.aec.gov.au/information-access/foi/2014/files/ls4912-1.pdf. Accessed 14 May 2017
3. Beckert, B., Goré, R., Schürmann, C., Bormer, T., Wang, J.: Verifying voting schemes. J. Inf. Sec. Appl. **19**(2), 115–129 (2014)
4. Benaloh, J., Moran, T., Naish, L., Ramchen, K., Teague, V.: Shuffle-sum: coercion-resistant verifiable tallying for STV voting. IEEE Trans. Inf. Forensics Secur. **4**(4), 685–698 (2009)
5. Bertot, Y., Castéran, P., Huet, G., Paulin-Mohring, C.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004). doi:10.1007/978-3-662-07964-5
6. Brandt, F., Conitzer, V., Endriss, U., Lang, J., Procaccia, A.D.: Introduction to computational social choice. In: Brandt, F., Conitzer, V., Endriss, U., Lang, J., Procaccia, A.D. (eds.) Handbook of Computational Social Choice. Cambridge University Press, Cambridge (2016)
7. Carrier, M.A.: Vote counting, technology, and unintended consequences. St Johns Law Rev. **79**, 645–685 (2012)
8. Cochran, D., Kiniry, J.: Votail: a formally specified and verified ballot counting system for Irish PR-STV elections. In: Pre-proceedings of the 1st International Conference on Formal Verification of Object-Oriented Software (FoVeOOS) (2010)
9. Conway, A., Blom, M., Naish, L., Teague, V.: An analysis of new south wales electronic vote counting. In: Proceedings of ACSW 2017, pp. 24:1–24:5 (2017)
10. Elections ACT. Electronic voting and counting (2016). http://www.elections.act.gov.au/elections_and_voting/electronic_voting_and_counting. Accessed 14 May 2017

11. Ghale, M.K., Goré, R., Pattinson, D.: A formally verified single transferable vote scheme with fractional values. In: Krimmer, R., Volkamer, M., Binder, N.B., Kersting, N., Schürmann, C. (eds.) E-Vote-ID 2017. LNCS, vol. 10615, pp. 163–182. Springer, Cham (2017)
12. Hales, T.: Formal proof. Not. AMS **55**, 1370–1380 (2008)
13. Hemaspaandra, L.A., Lavaee, R., Menton, C.: Schulze and ranked-pairs voting are fixed-parameter tractable to bribe, manipulate, and control. Ann. Math. Artif. Intell. **77**(3–4), 191–223 (2016)
14. Hood, C.: Transparency. In: Clarke, P.B., Foweraker, J. (eds.) Encyclopedia of Democratic Thought, pp. 700–704. Routledge, London (2001)
15. Kremer, S., Ryan, M., Smyth, B.: Election verifiability in electronic voting protocols. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 389–404. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15497-3_24
16. Kurrild-Klitgaard, P.: An empirical example of the condorcet paradox of voting in a large electorate. Publ. Choice **107**(1/2), 135–145 (2001)
17. Landin, P.J.: The mechanical evaluation of expressions. Comput. J. **6**(4), 308 (1964)
18. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system release 4.04 documentation and user's manual. Technical report, Institut National de Recherche en Informatique et en Automatique (INRIA) (2016)
19. Letouzey, P.: Extraction in Coq: an overview. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 359–369. Springer, Heidelberg (2008). doi:10.1007/978-3-540-69407-6_39
20. Meijer, A.: Transparency. In: Bovens, M., Goodin, R.E., Schillemans, T. (eds.) The Oxford Handbook of Public Accountability, pp. 507–524. Oxford University Press, Oxford (2014)
21. O'Neill, O.: A Question of Trust. Cambridge University Press, Cambridge (2002)
22. Parkes, D., Xia, L.: A complexity-of-strategic-behavior comparison between Schulze's rule and ranked pairs. In: Hoffmann, J., Selman, B. (eds.) Proceedings of AAAI 26, pp. 1429–1435. AAAI Press (2012)
23. Pattinson, D., Schürmann, C.: Vote counting as mathematical proof. In: Pfahringer, B., Renz, J. (eds.) AI 2015. LNCS, vol. 9457, pp. 464–475. Springer, Cham (2015). doi:10.1007/978-3-319-26350-2_41
24. Pattinson, D., Tiwari, M.: Schulze voting as evidence carrying computation. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499. Springer, Cham (2017). doi:10.1007/978-3-319-66107-0_26
25. Rivest, R.L., Shen, E.: An optimal single-winner preferential voting system based on game theory. In: Conitzer, V., Rothe, J. (eds.) Proceedings of COMSOC 2010. Duesseldorf University Press (2010)
26. Schulze, M.: A new monotonic, clone-independent, reversal symmetric, and Condorcet-consistent single-winner election method. Soc. Choice Welfare **36**(2), 267–303 (2011)
27. Vogl, F.: Waging War on Corruption: Inside the Movement Fighting the Abuse of Power. Rowman & Littlefield, Lanham (2012)
28. Yi, X., Paulet, R., Bertino, E.: Homomorphic Encryption and Applications. SpringerBriefs in Computer Science. Springer, Heidelberg (2014). doi:10.1007/978-3-319-12229-8