Verifiable Homomorphic Tallying for the Schulze Vote Counting Scheme

Thomas Haines[1], Dirk Pattinson[2], and Mukesh Tiwari[2]

[1] NTNU, Norway
[2] Research School of Computer Science, ANU, Canberra

Abstract. The encryption of ballots is crucial to maintaining integrity and anonymity in electronic voting schemes. It enables, amongst other things, each voter to verify that their encrypted ballot has been recorded as cast, by checking their ballot against a bulletin board.
We present a verifiable homomorphic tallying scheme for the Schulze method that allows verification of the correctness of the count—on the basis of encrypted ballots—that only reveals the final tally. We achieve verifiability by using zero knowledge proofs for ballot validity and honest decryption of the final tally. Our formalisation takes places inside the Coq theorem prover and is based on an axiomatisation of cryptogtaphic primitives, and our main result is the correctness of homomorphic tallying. We then instantiate these primitives using an external library and show the feasibility of our approach by means of case studies.

1 Introduction

Secure elections are a balancing act between integrity and privacy: achieving either is trivial but their combination is notoriously hard. One of the key challenges faced by both paper based and electronic elections is that results must substantiated with verifiable evidence of their correctness while retaining the secrecy of the individual ballot [4]. Technically, the notion of "verifiable evidence" is captured by the term *end-to-end (E2E) verifiability*, that is

- every voter can verify that their ballot was cast as intended
- every voter can verify that their ballot was collected as cast
- everyone can verify final result on the basis of the collected ballots.

While end-to-end verifiability addresses the basic assumption that no entity (software, hardware and participants) are inherently trustworthy, ballot secrecy addresses the privacy problem. Unfortunately, it appears as if coercion resistance is not achievable in the remote setting without relying on overly optimistic—to say the least—assumptions. A weaker property called receipt-freeness captures the idea that an honest voter—while able to verify that their ballot was counted—is required to keep no information that a possible coercer could use to verify how that voter had voted.

End to end verifiability and the related notation of software independence [19] have been claimed properties for many voting schemes. Küsters, Truderung and

Vogt [12] gave a cryptographic formulation whose value is highlighted by the attacks it revealed against established voting schemes [13].

The combination of privacy and integrity can be realised using cryptographic techniques, where encrypted ballots (that the voters themselves cannot decrypt) are published on a bulletin board, and the votes are then processed, and the correctness of the final tally is substantiated, using homomorphic encryption [10] and verifiable shuffling [1]. (Separate techniques exist to prevent ballot box stuffing and to guarantee cast-as-intended.) Integrity can then be guaranteed by means of Zero Knowledge Proofs (ZKP), first studied by Goldwasser, Micali, and Rackoff [9]. Informally, a ZKP is a probabilistic and interactive proof where one entity interacts with another such that the interaction provides no information other than that the statement being proved is true with overwhelming probability. Later results [2, 8] showed that all problems for which solutions can be efficiently verified have zero knowledge proofs.

This paper addresses the problem of verifiable homomorphic tallying for a preferential voting scheme, the Schulze Method. We show how it can be implemented in a theorem prover to guarantee both provably correct and verifiable counting on the basis of encrypted ballots, relative to an axiomatisation of the cryptographic primitives. We then obtain, via program extraction, a provably correct implementation of the vote counting, that we turn into executable code by providing implementations of the primitives based on a standard cryptographic library. We conclude by presenting experimental results, and discuss trust the trust base, security and privacy as well as the applicability of our work to real-world scenarios.

The Schulze Method. The Schulze Method [21] is a preferential, single-winner vote counting scheme that is gaining popularity due to its relative simplicity while retaining near optimal fairness [20]. A *ballot* is a rank-ordered list of candidates where different candidates may be given the same rank. The protocol proceeds in two steps, and first computes the *margin matrix* m, where $m(x, y)$ is the relative margin of x over y, that is, the number of voters that prefer x over y, minus the number of voters that prefer y over x. In symbols, given a collection B of ballots,

$$m(x, y) = \sharp\{b \in B \mid x <_b y\} - \sharp\{b \in B \mid y <_b x\}$$

where $\sharp$ denotes cardinality, and $<_b$ is the preference relation encoded by ballot b. We note that $m(x, y) = -m(y, x)$, i.e. the margin matrix is symmetric. In a second step, a *generalised margin* g is computed as the strongest path between two candidates

$$g(x, y) = \max\{\mathsf{str}(p) \mid p \text{ path from } x \text{ to } y\}$$

where a path from x to y is simply a sequence $x = x_0, \ldots, x_n = y$ of candidates, and the strength

$$\mathsf{str}(x_0, \ldots, x_n) = \min\{m(x_i, x_{i+1}) \mid 0 \leq i < n\}$$

is the lowest margin encountered on a path. Informally, one may think of the generalised margin $g(x, y)$ as transitive accumulated support for x over y. We say that x beats y if $g(x, y) \geq g(y, x)$ and a winner is a candidate that cannot be beaten by anyone. That is, w is a *winner* if $g(w, x) \geq g(x, w)$ for all other candidates x. Note that winners may not be uniquely determined (e.g. in the case where no ballots have been cast).

In previous work [17] we have demonstrated how to achieve verifiability of counting plaintext ballots by producing a verifiable *certificate* of the count. The certificate has two parts: The first part witnesses the computation of the margin function where each line of the certificate amounts to updating the margin function by a single ballot. The second part witnesses the determination of winners based on the margin function. In the first phase, i.e. the computation of the margin function, we perform the following operations for every ballot:

1. if the ballot is informal it will be discarded
2. if the ballot is formal, the margin function will be updated

The certificate then contains one line for each ballot and thus allows to independently verify the computation of the margin function. Based on the final margin function, the second part of the certificate presents verifiable evidence for the computation of winners. Specifically, if a candidate w is a winner, it includes:

1. an integer k and a path of strength k from w to any other candidate
2. evidence, in the form of a co-closed set, of the fact that there cannot be a path of strength $> k$ from any other candidate to w.

Crucially, the evidence of w winning the election *only* depends on the margin matrix. We refer to [17] for details of the second part of the certificate as this will remain unchanged in the work we are reporting here.

Related Work. The paper that is closest to our work is an algorithm for homomorphic counting for Single Transferable Vote [3]. While single transferable vote is arguably more complex that the Schulze Method, we have demonstrated the viability of our approach by implementing it in a theorem prover, and have extracted, and evaluated, an executable based on the formal proof development. The idea of formalising evidence for winning elections has been put forward (for plaintext ballots) in [16]. For non-preferential (plurality) voting, homomorphic tallying is now standard, and implemented e.g. in the Helios electronic voting system [**?**] from Version 2.0 onwards, and is used e.g. in public elections in Estonia [**?**].

2 Verifiable Homomorphic Tallying

The realisation of verifiable of homomorphic tallying that we are about to describe follows the same two phases as the protocol: We first homomorphically compute the margin matrix, and then compute winners on the basis of the (decrypted) margin. The computation also produces a verifiable certificate that

leaks no information about individual ballots other than the (final) margin matrix, which in turn leaks no information about individual ballots if the number of voters is large enough. As for counting of plaintext ballots, we disregard informal ballots in the computation of the margin. In accord with the two phases of computation, the certificate consists of two parts: the first part evidences the correct (homomorphic) computation of the margin, and the second part the correct determination of winners. We describe both in detail.

Format of Ballots. In preferential voting schemes, ballots are rank-ordered lists of candidates. For the Schulze Method, we require that all candidates are ranked, and two candidates may be given the same rank. That is, a ballot is most naturally represented as a function $b : C \to \mathbb{N}$ that assigns a numerical rank to each candidate, and the computation of the margin amounts to computing the sum

$$m(x,y) = \sum_{b \in B} \begin{cases} +1 & b(x) > b(y) \\ 0 & b(x) = b(y) \\ -1 & b(x) < b(y) \end{cases}$$

where B is the multi-set of ballots, and each $b \in B$ is a ranking function $b : C \to \mathbb{N}$ over a (finite) set C of candidates.

We note that this representation of ballots is not well suited for homomorphic computation of the margin matrix as practically feasible homomorphic encryption schemes do not support comparison operators and case distinctions as used in the formula above.

We instead represent ballots as matrices $b(x,y)$ where $b(x,y) = +1$ if x is preferred over y, $b(x,y) = -1$ if y is preferred over x and $b(x,y) = 0$ if x and y are equally preferred.

While the advantage of the first representation is that each ranking function is necessarily a valid ranking, the advantage of the matrix representation is that the computation of the margin matrix is simple, that is

$$m(c,d) = \sum_{b \in B} b(x,y)$$

where B is the multi-set of ballots (in matrix form), and can moreover be transferred to the encrypted setting in a straight forward way: if ballots are matrices $e(x,y)$ where $e(x,y)$ is the encryption of an integer, then

$$em = \bigoplus_{eb \in EM} eb(x,y) \tag{1}$$

where $\oplus$ denotes homomorphic addition, eb is an encrypted ballot in matrix form (i.e. decrypting $eb(x,y)$ indicates whether x is preferred over y), and EM is the multi-set of encrypted ballots. The disadvantage is that we need to verify that a matrix ballot is indeed valid, that is

- that the decryption of $eb(x,y)$ is indeed one of $1, 0$ or -1
- that eb indeed corresponds to a ranking function.

Indeed, to achieve verifiability, we not only need *verify* that a ballot is valid, we also need to *evidence* its validity (or otherwise) in the certificate.

Validity of Ballots. By a plaintext (matrix) ballot we simply mean a function $b : C \times C \to \mathbb{Z}$, where C is the (finite) set of candidates. A plaintext ballot $b(x, y)$ is *valid* if it is induced by a ranking function, i.e. there exists a function $f : C \to \mathbb{N}$ such that $b(x, y) = 1$ if $f(x) < f(y)$, $b(x, y) = 0$ if $f(x) = f(y)$ and $b(x, y) = -1$ if $f(x) > f(y)$. A *ciphertext (matrix) ballot* is a function $eb : C \times C \to \mathbb{CT}$ (where $\mathbb{CT}$ is a chosen set of ciphertexts), and it is valid if its decryption, i.e. the plaintext ballot $b(x, y) = \mathsf{dec}(eb(x, y))$ is valid (where dec denotes decryption).

For a plaintext ballot, it is easy to decide whether it is valid (and should be counted) or not (and should be discarded). We use shuffles (ballot permutations) to evidence the validity of encrypted ballots. One observes that a matrix ballot is valid if and only if it is valid after permuting both rows and columns with the same permutation. That is, $b(x, y)$ is valid if and only if $b'(x, y)$ is valid, where

$$b'(x, y) = b(\pi(x), \pi(y))$$

and $\pi : C \to C$ is a permutation of candidates. (Indeed, if f is a ranking function for b, then $f \circ \pi$ is a ranking function for b'). As a consequence, we can evidence the validity of a ciphertext ballot eb by

- publishing a shuffled version eb' of eb, that is shuffled by a secret permutation, together with evidence that eb' is indeed a shuffle of eb
- publishing the decryption b' of eb' together with evidence that b' is indeed the decryption of eb'.

We use zero-knowledge proofs in the style of [22] to evidence the correctness of the shuffle, and zero-knowledge proofs of honest decryption [6] to evidence correctness of decryption. This achieves ballot secrecy as the (secret) permutation is never revealed.

In summary, the evidence of correct (homomorphic) counting starts with an encryption of the zero margin em, and for each ciphertext ballot eb contains

1. a shuffle of eb together with a ZKP of correctness
2. decryption of the shuffle, together with a ZKP of correctness
3. the updated margin function, if the decrypted ballot was valid, and
4. the unchanged margin function, if the decrypted ballot is not valid.

Once all ballots have been processed in this way, the certificate determines winners and contains winners by

5. the fully constructed margin, together with its decryption and ZKP of honest decryption after counting all the ballots
6. publishes the winner(s), together with evidence to substantiate the claim

Cryptographic primitives. We require an additively homomorphic cryptosystem to compute the (encrypted) margin matrix according to Equation 1 (this implements Item 3 above). All other primitives fall into one of three categories.

Verification primitives are used to syntactically define the type of valid certificates. For example, when publishing the decrypted margin function in Item 5 above, we require that the zero knowledge proof in fact evidences correct decryption. To guarantee this, we need a verification primitive that – given ciphertext, plaintext and zero knowledge proof – verifies whether the supplied proof indeed evidences that the given ciphertext corresponds to the given plaintext. In particular, verification primitives are always boolean valued functions. While verification primitives *define* valid certificates, *generation primitives* are used to *produce* valid certificates. In the example above, we need a decryption primitive (to decrypt the homomorphically computed margin) and a primitive to generate a zero knowledge proof (that witnesses correct decryption). Clearly verification and generation primitives have a close correlation, and we need to require, for example, that zero knowledge proofs obtained via a generation primitive has to pass muster using the corresponding verification primitive.

The three primitives described above (decryption, generation of a zero knowledge proof, and verification of this proof) already allow us to implement the entire protocol with exception of ballot shuffling (Item 1 above). Here, the situation is more complex. While existing mixing schemes (e.g. [**?**]) permute an array of ciphertexts and produce a zero knolwedge proof that evidences the correctness of the shuffle, our requirement dictates that every row and colum of the (matrix) ballot is shuffled with the *same* (secret) permutation. In other words, we need to retain the identity of the permutation over multiple shuffles. We achieve this by committing to a permutation using Pedersen's commitment scheme [18]. In a nutshell, the Pedersen commitment scheme has the following properties.

- Hiding: the commitment reveals no information about the permutation
- Binding: no party can open the commitment in more than one way, i.e. the commitment is to one permutation only.

A combination of Pedersen's commitment scheme with a zero knowledge proof leads to a similar two step protocol (also known as commitment-consistent proof of shuffle)[23].

- Commit to a secret permutation and publish the commitment (hiding).
- Use a zero knowledge proof to show that shuffling has used the same permutation which we committed to in previous step (binding).

This allows us to witness the validity (or otherwise) of a ballot by generating a permutation π which is used to shuffle every row and column of the ballot. We hide π by committing it using Pedersen's commitment scheme and record the commitment c_π in the certificate. However, for the binding step, rather than opening π we generate a zero knowledge proof, zkp_π, using π and c_π, which can be used to prove that c_π is indeed the commitment to some permutation used in the (commitment consistent) shuffling without being opened [23]. We can now use the permutation that we have committed to for shuffling each row and column of a ballot, and evidence the correctness of the shuffle via a zero knowledge proof. To evidence validity (or otherwise) of a (single) ballot, we therefore:

1. generate a (secret) permutation and publish a commitment to this permutation, together with a zero knowledge proof that evidences commitment to a permutation
2. for each row of the ballot, publish a shuffle the row with the committed to permutation, together with a zero knowledge proof that witnesses shuffle correctness
3. for each column of the row shuffled ballot, publish a shuffle the column, also together with a zero knowledge proof of correctness
4. publish the decryption the ballot shuffled in this way, together with a zero knowledge proof that witnesses honest decryption
5. decide the validity of the ballot based on the decrypted shuffle.

The cryptographic primitives needed to implement this again fall into the same classes. To define validity of certificates, we need verification primitives

- to decide whether a zero knowledge proof evidences that a given commitment indeed commits to a permutation
- to decide whether a zero knowledge proof evidences the correctness of a shuffle relative to a given permutation commitment.

Dual to the above, to generate (valid) certificates, we need the ability to

- generate permutation commitments and accompanying zero knowledge proofs that evidence commitment to a permutation
- generate shuffles relative to a commitment, and zero knowledge proofs that evidence the correctness of shuffles.

Again, both need to be coherent in the sense that the zero knowledge proofs produced by the generation primitives need to pass validation. In summary, we require an additively homomorphic cryptosystem that implements the following:

Decryption Primitives. decryption of a ciphertext, creation and verification of honest decryption zero knowledge proofs.

Commitment Primitives. generating permutations, creation and verification of commitment zero knowledge proofs

Shuffling Primitives. commitment consistent shuffling, creation and verification of commitment consistent zero knowledge shuffle proofs

Witnessing of Winners. Once all ballots are counted, the computed margin is decrypted, and winners (together with evidence of winning) are computed using plaintext counting. We discuss this part only briefly, for completness, as it is identicial to the existing work on plaintext counting [17]. For each of the winners w and each candidate c we publish

- a natural number $k(w, x)$
- a path $w = x_0, \ldots, x_n = x$ of strength k
- a set $C(w, x)$ of pairs of candidates that is k-coclosed and contains (x, w)

where a set S is k-coclosed if for all $(x, z) \in C$ we have that $m(x, z) < k$ and either $m(x, y) < k$ or $(y, z) \in S$ for all candidates y. Informally, the first requirement ensures that there is no direct path (of length one) between a pair $(x, z) \in S$,and the second requirement ensures that for an element $(x, z) \in S$, there cannot be a path that connects x to an intermediate node y and then (transitively) to z that is of strength $\geq k$. We refer to *op.cit.* for the (formal) proofs of the fact that existence of co-closed sets witnesses the winning conditions.

3 Realisation in a Theorem Prover

We formalise homomorphic tallying for the Schulze Method inside the Coq theorem prover [5]. Apart from supporting an expressive logic and (crucial for us) dependent inductive types, Coq has a well developed extraction facility that we use to extract proofs into OCaml programs. Indeed, our basic approach is to first formally define the notion of a valid certificate, and then prove that a valid certificate can be obtained from any set of (encrypted) ballots. Extracting this proof as a programme, we obtain an executable that is correct by construction.

The purpose of this paper is not to verify cryptographic primitives, but use them as a tool to construct evidence which can be used to audit and verify the outcome during different phase of election. Here, we treat them as abstract entities and assume axioms about them inside Coq. In particular, we assume the existence of functions that implement each of the primitives described in the previous section, and postulate natural axioms that describe how the different primitives interact. As a by-product, we obtain an axiomatisation of a cryptographic library that we could, in a later step, verify the implementation of a cryptosystem against. In particular, this allows us to not commit to any particular cryptosystem in particular (although our development, and later instantiation, is geared towards El Gamal [7]).

The first part of our formailsation concerns the cryptographic primitives that we collect in a separate module. Below is an example of the generation / verification primitives for decryption, together with coherence axioms.

```
Variable decrypt_message :
    Group -> Prikey -> ciphertext ->  plaintext.

Variable construct_zero_knowledge_decryption_proof :
    Group -> Prikey -> ciphertext -> DecZkp.

Axiom verify_zero_knowledge_decryption_proof :
    Group -> plaintext -> ciphertext -> DecZkp -> bool.

Axiom verify_honest_decryption_zkp (group: Group):
  forall (pt : plaintext) (ct : ciphertext) (pk : Prikey),
    (pt = decrypt_message group pk ct) ->
    verify_zero_knowledge_decryption_proof group pt ct
    (construct_zero_knowledge_decryption_proof group pk ct) = true.
```

DP: add coherence axiom! Explain Variable / Axiom ??

The use of the keyword `Variable` indicates that we postulate the existence of a function, whereas `Axiom` specifies a property (here of the functions whose existence is postulated). In the above, the first two functions, `decrypt_message` and `construct_zero_knowledge_decryption_proof` are *generation* primitives, whereas the third, `verify_zero_knowledge_decryption_proof` is a *verification* primitive. We have two coherence axioms. The first says that if the verification of a zero knowledge proof of honest decryption suceeds, then the ciphertext indeed decrypts to the given plaintext. The second stipulates that generated zero knowledge proofs indeed verify.

For ballots, we assume a type `cand` of candidates, and represent plaintext and encrypted ballots as two-argument functions that take plaintext, and ciphertexts, as values.

```
Definition pballot := cand -> cand -> plaintext.
Definition eballot := cand -> cand -> ciphertext.
```

We now turn to the representation of certificates, and indeed to the definition of what it means to (a) count encrypted votes correctly according to the Schulze Method, and (b) produce a verifiable certificate of this fact. At a high level, we split the counting (and accordingly the certificate) into *states*. This gives rise to a (dependent, inductive) type `ECount`, parameterised by the ballots being counted.

```
Inductive ECount (group : Group) (bs : list eballot) :
  EState -> Type
```

Given a list `bs` of ballots, `ECount bs` is a dependent inductive type. In this case, given a state of counting (i.e. an inhabitant `estate` of `EState`), the type level application `ECount bs estate` is the *type of evidence that proves that `estate` is a state of counting that has been reached according to the method.* The states itself are represented by the type `EState` where

- `epartial` represents a partial state of counting, consisting of the homomorphically computed margin so far, the ist of uncounted ballots and the list of invalid ballots encountered so far
- `edecrypt` represents the final decrypted margin matrix, and
- `ewinners` is the final determination of winners.

This is readily translated to the following Coq code:

```
Inductive EState : Type :=
 | epartial : (list eballot * list eballot) ->
              (cand -> cand -> ciphertext) -> EState
 | edecrypt : (cand -> cand -> plaintext) -> EState
 | ewinners : (cand -> bool) -> EState.
```

The constructors of `EState` then allow us to move from one state to the next, under appropriate conditions that guaratnee correctness of the count.

The first constructor, `Ecax` kick-starts the count, and ensures that

– all ballots are initially uncounted
– margin matrix is an encryption of the zero matrix

The first constructor, as well as all the others, require

state data here, the list of uncounted and invalid ballots, and the encrypted homomorphic margin

verification data a zero knowledge proof that the encrypted homomorphic margin is indeed an encryption of the zero margin

correctness constraints here, the constuctor may only be applied if the list of uncounted ballots is equal to the list of ballots cast

The main difference between the correctness condition, and the verification data is that the former can be simply be inspected (here by comparing lists) whereas the latter requires additional data (here in the form of a zero knowledge proof). [3]

Schematically, we have the following representation with data in first column, and assertions about the data in second column.

Ecax

* cast ballots *bs*, and uncounted ballots *us*	* uncounted ballots are equal to cast ballots, i.e. $us = bs$
* encrypted zero margin matrix *encm*, its decryption (decrypted zero margin matrix) *decm*, and zero-knowledge proof *zkpdec* of honest decryption	* honest decryption verification primitive verifies the fact that *decm* is honest decryption of *encm* using *zkpdec*, i.e. it returns true
	* all the entries in *decm* are zero

$$\textsc{Ecax}\ \frac{}{\text{ECount group bs (epartial (us, []) encm)}}$$

The translation of high level representation into Coq representation is now easy, and we arrive at the following Coq code.

```
ecax (us : list eballot) (encm : cand -> cand -> ciphertext)
 (decm : cand -> cand -> plaintext)
 (zkpdec : cand -> cand -> DecZkp) :
 us = bs -> (forall c d : cand, decm c d = 0) ->
 (forall c d, verify_zero_knowledge_decryption_proof
   group (decm c d) (encm c d) (zkpdec c d) = true) ->
 ECount group bs (epartial (us, []) encm)
```

[3] DP: make this clearer

The constructor `ecvalid` represents the effect of counting a valid ballot. Here the crucial aspect is that validity needs to be evidenced. As before, we have:

state data as before, the list of uncounted and invalid ballots, the homomorphic margin, but additionally evidence that the previous state has been obtained correctly

verification data a commitment to a (secret) permutation, a row permutation of the ballot being counted, and a column permutation of this, and a decryption of the row- and column permuted ballot (all with accompanying zero knowledge proofs)

correctness constraints all the zero knowledge proofs verify, the new margin is the homomorphic addition of the previous margin and the counted ballot, and the decrypted (shuffled) ballot is indeed valid.

Schematically, we have the following, again with the data on the left, and the correctness conditions on the right.

Ecvalid

* commitment of a (secret) permutation *cpi*, and zero-knowledge-proof *zkpcpi* of this commitment
* ballot v whose each row is permutation of corresponding row of u, and zero-knowledge-proof of this fact *zkppermuv*. Similarly, ballot w whose each column is permutation of corresponding column of v by same (secret) permutation, and zero-knowledge-proof of this fact *zkppermvw*
* ballot b which is decryption of ballot w, and zero-knowledge-proof of this fact *zkpdecw*
* old margin matrix m, and updated margin matrix nm
* permutation commitment primitive verifies the claim (returns true) that *cpi* is commitment of a (secret) permutation whose zero-knowlege-proof is *zkpcpi*
* shuffle verification primitive verifies the claim that each row of ballot v is permutation of corresponding row of ballot u by same permutation whose commitment is *cpi* by using the zero-knowledge-proof, *zkppermuv*. Similarly, it verifies the claim for ballot v and w, but for columns.
* honest decryption verification primitive verifies the claim that ballot b is honest decryption of ballot w
* ballot validity primitive verifies that ballot b is valid
* update margin nm is equal to homomorphic addition of old margin m and ballot u

$$\textsc{Ecvalid}\ \frac{\text{ECount group bs (epartial (u :: us, inbs) m)}}{\text{ECount group bs (epartial (us, inbs) nm)}}$$

We elide the description of the third constructor that is applied when an invalid ballot is being encountered (here, the only difference is that the margin matrix is not being updated). Counting finishes when there are no more uncounted ballots, in which case the next step is to publish the decrypted margin matrix. Also here, we have

state data the decrypted margin function, plus evidence that a state with no more uncounted ballots has been obtained correctly

verification data a zero knowledge proof that demonstrates honest decryption of the final margin matrix

correctness constraints the given zero knowledge proof verifies, i.e. the given decrypted margin is indeed the decryption of the (last) homomorphically computed margin matrix.

Schematically:

Ecdecrypt

* fully constructed encrypted zero margin matrix *encm*, its decryption (decrypted zero margin matrix) *decm*, and zero-knowledge proof *zkpdec* of honest decryption
* honest decryption verification primitive verifies the fact that *decm* is honest decryption of *encm* using zero-knowledge-proof *zkpdec*, i.e. it returns true

$$\textsc{Ecdecrypt}\ \frac{\text{ECount group bs (epartial ([], inbs) encm)}}{\text{ECount group bs (edecrypt decm)}}$$

The last constructor finally declares the winners of the election, and we have:

state data a function `cand -> bool` that determines winners, plus evidence of the fact that the decrypted final margin matrix has been obtained correctly

verification data paths and co-closed sets that evidence the correctness of the function above

correctness constraints that ensure that the verification data verifies the winners given by the state data.

This last part is identical to our previous formalisation of the Schulze Method (for plaintext ballots), and we refer to [17] for more details.

Ecfin

$$\textsc{Ecfin}\ \frac{\text{ECount group bs (edecrypt dm)}}{\text{ECount group bs (ewinners w)}}$$

4 Correctness by Construction and Verification

In the previous section, we have presented a data type that *defines* the notion of a verifiably correct count of the Schulze Method, on the basis of encrypted ballots. To obtain an executable that in fact *produces* a verifiable (and provably correct) count, we can proceed in either of two ways:

1. implement a function that – give a list `bs` of ballots – produces a boolean function `w` (for winners) and an element of the type `ECount bs (winners w)`. This gives both the eletion winners (`w`) as well as evidence (the element of the `ECount` data type).
2. to prove that for every set `bs` of encrypted ballots, we have a boolean function `w` and an inhabitant of the type `ECount bs (winners w)`.

Under the proofs-as-programs interpretation of constructive type theory, both amount to the same. We chose the latter approach, and our first main theorem formally states that all elections can be counted according to the Schulze Method (with encrypted ballots), i.e. a winner can always be found. Formally, our main theorem takes the following form:

```
Lemma pschulze_winners (group : Group)
 (bs : list eballot) : existsT (f : cand -> bool),
 ECount group bs (ewinners f).
```

The proof proceeds by successively building an inhabitant of `EState` by homomorphically computing the margin matrix, then decrypting and determining the winners. Within the proof, we use both generation primitives (e.g. to construct zero knowledge proofs) and coherence axioms (to ensure that the zero knowledge proofs indeed verify).

The correctness of our entire approach stands or falls with the correct formalisation of the inductive data type `ECount` that is used to determine the winners of an election counted according to the Schulze Method. While one can argue that the data type itself is transparent enough to be its own specification, the cryptographic aspect makes things slightly more complex. For example, it appears to be credible that our mechanism for determining validity of a ballot is correct – however we have not given proof of this. Rather than scrutinising the details of the construction of this data type, we follow a different approach: we demonstrate that homomorphic counting always yields the same results as plaintext counting, where plaintext counting is already verified against its specification. Plaintext counting has been formalised, and verified, in the precursor paper [17]. This correspondence has two directions, and both assume that we are given two lists of ballots that are the encryption (resp. decryption) of one another.

The first theorem, `plaintext_schulze_to_homomorphic`, reproduced below shows that every winner that can be determined using plaintext counting can also be evidenced on the basis of encrypted ballots, whereas Theorem `homomorphic_schulze_to_plaintext` establishes the opposite.

```
Lemma plaintext_schulze_to_homomorphic
 (group : Group) (bs : list ballot):
 forall (pbs : list pballot) (ebs : list eballot)
 (w : cand -> bool), (pbs = map (fun x => (fun c d =>
 decrypt_message group privatekey (x c d))) ebs) ->
 (mapping_ballot_pballot bs pbs) ->
 Count bs (winners w) -> ECount group ebs (ewinners w).

Lemma homomorphic_schulze_to_plaintext
 (group : Group) (bs : list ballot):
 forall (pbs : list pballot) (ebs : list eballot)
 (w : cand -> bool) (pbs = map (fun x => (fun c d =>
 decrypt_message group privatekey (x c d))) ebs) ->
 (mapping_ballot_pballot bs pbs) ->
```

```
ECount grp ebs (ewinners w) -> Count bs (winners w).
```

The theorems above feature a third type of ballot that is the basis of plaintext counting, and is a simple ranking function of type `cand -> Nat`, and the two hypotheses on the three types of ballots ensure that the encrypted ballots (`ebs`) are in fact in alignment with the rank-ordered ballots (`bs`) that are used in plaintext counting. The proof, and indeed the formulation, relies on an inductive data type `Count` that can best be thought of as a plaintext version of the inductive type `ECount` given here. Crucially, `Count` is verified against a formal specification of the Schulze Method. Both theorems are proven by induction on the definition of the respective data types, where the key step is to show that the (decrypted) final margins agree. The key ingredient here are the coherence axioms that stipulate that zero knowledge proofs that verify indeed evidence shuffle and/or honest decryption.

5 Extraction and Experiments

As already mentioned, we are using the Coq extraction mechanism[15] to extract programs from existence proofs. In particular, we extract the proof of the Theorem `pschulze_winners`, given in Section 4 to a program that delivers not only provably correct counts, but also verifiable evidence. Give a set of encrypted ballots and a `Group` that forms the basis of operations, we obtain a program that delivers not only a set of winners, but additionally independently verifiable evidence of the correctness of the count.

Indeed, the entire formulation of our data type, and the split into state data, verification data, and correctness constraints, has been geared towards extraction as a goal. Technically, the verification conditions are *propositions*, i.e. inhabitants of Type *Prop* in the terminology of Coq, and hence erased at extraction time. This corresponds to the fact that the assertions embodied in the correctness constraints can be verified with minimal computational overhead, given the state and the verification data. For example, it can simply be verified whether or not a zero knowledge proof indeed verifies honest decryption by running it through a verifier. On the other hand, the zero knowledge proof itself (which is part of the verification data) is crucially needed to be able to verify that a plaintext is the honest decryption of a ciphertext, and hence cannot be erased during extraction. Technically, this is realised by formulating both state and verification data at type level (rather than as propositions).

As we have explained in Section 3, the formal development does not presuppose any specific implementation of the cryptographic primitives, and we assume the existence of cryptographic infrastructure. From the perspective of extraction, this produces an executable with "holes", i.e. the cryptographic primitives need to be supplied to fill the holes and indeed be able to compile and execute the extracted program.

To fill this hole, we implement the cryptoraphic primitives with help of the Unicrypt library[**?**]. [4] [5] Unicrypt is a freely available library, written in Java, that provides nearly all of the required functionality, with the exception of honest decryption zero knowledge proofs. We extract our proof development into OCaml and use Java/OCaml bindings OCaml-Java library[6] [7] to make the UniCrypt functionality available to our OCaml executive. Due to differences in the type structure between Java and OCaml, mainly in the context of sub-typing, this was done in the form of an OCaml wrapper around Java data structures. After instantiating the cryptographic primitives in the extracted OCaml code with wrapper code that calls UniCrypt we tested the executable on a three candidate elections between candidates `A`, `B` and `C`. which produces the tally sheet given below. In other words, it is trace of computation which can be used as a checkable record to verify the outcome of election. We present a schematic version of this trace where we elide cryptographic detail. A typical certificate would be obtained from the type `ECount` where the head of the certificate corresponds to the base case of the inductive type, here `ecax`. Below, `M` is encrypted margin matrix, `D` is its decrypted equivalent, required to be identically zero, and `Z` represents a matrix of zero knwoledge proofs, each establishing that the `XY`-component of `M` is in fact an encryption of zero. All these matrices are indexed by candidates and we display these matrices by listing their entries prefixed by a pair of candidates, e.g. the ellipsis in `AB(...)` denotes the matrix entry at row `A` and column `B`.

```
M: AB(rel-marg-of-A-over-B-enc), AC(rel-marg-of-A-over-C-enc), ...
D: AB(0)                       , AC(0)                       , ...
Z: AB(zkp-for-rel-marg-A-B)    , AC(zkp-for-rel-marg-A-B)    , ...
```

Note that one can verify the fact that the initial encrypted margin is in fact the zero margin by just verifying the zero knowledge proofs. Successive entries in the certificate will generally be obtained by counting valid, and discarding invalid ballots. If a valid ballot is counted after the counting commences, the certificate would continue by exhibiting the state and verification data contained in the `ecvalid` constructor which can be displayed schematically as follows:

```
V: AB(ballot-entry-A-B) , AC(ballot-entry-A-C), ...
C: permutation-commitment
P: zkp-of-valid-permutation-commitment
R: AB(row-perm-A-B)     , AC(row-perm-A-C)    , ...
RP: A(zkp-of-perm-row-A), B(zkp-of-perm-row-B), ...
C: AB(col-perm-A-B),      AC(col-perm-A-C)    , ...
CP: A(zkp-of-perm-col-A), B(zkp-of-perm-col-C), ...
D: AB(dec-perm-bal-A-B) , AC(dec-perm-bal-A-C), ...
Z: AB(zkp-for-dec-A-B)  , AC(zkp-for-dec-A-C) , ...
M: AB(new-marg-A-B)     , AC(new-marg-A-C)    , ...
```

Here `V` is the list of ballots to be counted, where we only diplay the first element. We commit to a permutation and validate this commitment with a zero

[4] https://github.com/bfh-evg/unicrypt
[5] DP: turn into a proper reference
[6] https://github.com/Julow/ocaml-java
[7] DP: turn into proper reference

knowledge proof, here given in the second and third line, prefixed with `C` and `P`. The following two lines are a row permutation of the ballot `V`, together with a zero knowledge proof of correctness of shuffling (of each row) with respect to the permutation committed to by `C` above. The following two lines achieve the same for subsequently permuting the columns of the (row permuted) ballot. Finally, `D` is the decrypted permuted ballot, and `Z` a zero knowledge proof of honest decryption. We end with an updated homomorphic margin matrix `M`. Again, we note that the validity of the decrypted ballot can be checked easily, and validating zero knowledge proofs substantiate that the decrypted ballot is indeed a shuffle of the original one. Homomorphic addition can simply be re-computed.

The steps where invalid ballots are being detected is similar, with the exception of not updating the margin matrix. Once all ballots are counted, the only applicable constructor is `ecdecrypt`, the data content of which would continue a certificate schematically as follows:

```
V: []
M: AB(fin-marg-A-B), AC(fin-marg-A-C), ...
D: AB(dec-marg-A-B), AC(dec-marg-A-C), ...
Z: AB(zkp-dec-A-B) , AC(zkp-dec-A-C) , ...
```

Here the first line indicates that there are no more ballots to be counted, `M` is the final encrupted margin matrix, `D` is its decryption and `Z` is a matrix of zero knowledge proofs verifying the correctness of decryption.

The certificate would end with the determination of winners based on the encrypted margin, and would end with the content of the `ecfin` constructor

```
winning: A, <evidence that A wins against B and C>
losing:  B, <evidence that B loses against A and C>
losing:  C, <evidence that C loses against A and B>
```

where the notion of evidence for winning and losing is as in the plaintext version of the protocol [**?**].

We note that the schematic persentation of the certificate above is nothing but a representation of the data contained in the extracted type `ECount` that we have chosen to present schematically. Concrete certificates can be inspected with the accompanying proof development, and are obtained by simply implementing datatype to string conversion on the type `ECount`.

Even though we produce the tally by formally verified code, election auditor has no information about how it was produced. This adds extra layer of confidence in election, because the tally can be audited by pool of scrutineers to certify the outcome of election regardless of how the tally was produced.

We have run our experiment on an Intel i7 2.6 GHz Linux desktop computer with 8GB of RAM for three candidates and randomly generated ballots. The highest amount of ballot we counted was 10,000 (not included in graph), and it ran for 25 hours. Clearly, it is not very efficient, and the reason we suspect is a lot of communication between OCaml runtime system and Java run time system.

6 Analysis

Summary. The main contribution of our formalisation is that of independently verifiable *evidence* for a set of candidates to be the winners of an election counted according to the Schulze method. Our main claim is that that our notion of evidence is both safeguarding the privacy of the individual ballot (as the count is based on encrypted ballots) and is verifiable at the same time (by means of zero knowledge proofs). To do this, we have axiomatised a set of cryptographic primitives to deal with encryption, decryption, correctness of shuffles and correctness of decryption. From formal and constructive proof of the fact that such evidence can always be obtained, we have then extracted executable code that is provably correct by construction and produces election winners together with evidence once implementations for the cryptographic primitives are supplied.

In a second step, we have supplied an implementation of these primitives, largely based on the Unicrypt Library. Our experiments have demonstrated that this approach is feasible, but quite clearly much work is still needed to improve efficiency.

Assumptions for Provable Correctness. While we claim that the end product embodies a high level of reliability, our approach necessarily leaves some gaps between the executable and the formal proofs. First and foremost, this is of course the implementation of the cryptographic primitives in an external (and unverified) library. We have minimised this gap by basing our implementation on a purpose-specific existing library (Unicrypt) to which we relegate most of the functionality. Another gap is the extraction mechanism of the Coq theorem prover which does not come with formal correctness guarantees that reach down to the machine code level such as for example CakeML [11].

Modelling Assumptions. In our modelling of the cryptographic primitives, in particular the zero knowledge proofs, we assumed certain things which in reality only hold with very high probability. As a consequence our correctness assertions only hold to the level of probability that is guaranteed by zero knowledge proofs.

Scalability. We have analysed the feasibility of the extracted code by counting an increasing number of ballots. While this demonstrates a proof of concept, our results show that the cryptographic layer adds significant overhead compared to plaintext tallying [17]. Our present hypothesis is that this overhead is created by the various layers of bindings between the executable (extracted into OCaml)

and the Unicrypt library (implemented in Java) as each appears to be very efficient individually.

Future Work. Our axiomatisation of the needed cryptographic primitives lays the foundation of creating a verified library. For scalability, a more detailed analysis (and profiling) of the software artefact are necessary. Orthogonal to what we have presented here, it would also be of interest to develop a provably correct verifier for the notion of certificate presented here.

References

1. Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In David Pointcheval and Thomas Johansson, editors, *Proc. EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 263–280. Springer, 2012.
2. Michael Ben-Or, Oded Goldreich, Shafi Goldwasser, Johan Håstad, Joe Kilian, Silvio Micali, and Phillip Rogaway. Everything provable is provable in zero-knowledge. In *CRYPTO*, volume 403 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 1988.
3. Josh Benaloh, Tal Moran, Lee Naish, Kim Ramchen, and Vanessa Teague. Shufflesum: coercion-resistant verifiable tallying for STV voting. *IEEE Trans. Information Forensics and Security*, 4(4):685–698, 2009.
4. Matthew Bernhard, Josh Benaloh, J. Alex Halderman, Ronald L. Rivest, Peter Y. A. Ryan, Philip B. Stark, Vanessa Teague, Poorvi L. Vora, and Dan S. Wallach. Public evidence from secret ballots. In Robert Krimmer, Melanie Volkamer, Nadja Braun Binder, Norbert Kersting, Olivier Pereira, and Carsten Schürmann, editors, *Proc. E-Vote-ID 2017*, volume 10615 of *Lecture Notes in Computer Science*, pages 84–109. Springer, 2017.
5. Yves Bertot, Pierre Castéran, Gérard Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, 2004.
6. David Chaum and Torben P. Pedersen. Wallet databases with observers. In *CRYPTO*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 1992.
7. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1984.
8. Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity for all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):691–729, 1991.
9. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *STOC*, pages 291–304. ACM, 1985.
10. Martin Hirt and Kazue Sako. Efficient receipt-free voting based on homomorphic encryption. In Bart Preneel, editor, *Proc. EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 539–556. Springer, 2000.
11. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *Proc. POPL 2014*, pages 179–192. ACM, 2014.

12. Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Accountability: definition and relationship to verifiability. In *ACM Conference on Computer and Communications Security*, pages 526–535. ACM, 2010.
13. Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Clash attacks on the verifiability of e-voting systems. In *IEEE Symposium on Security and Privacy*, pages 395–409. IEEE Computer Society, 2012.
14. X. Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rèmy, and Jérôme Vouillon. The ocaml reference manual, 2013.
15. Pierre Letouzey. A new extraction for coq. In Herman Geuvers and Freek Wiedijk, editors, *Proc. TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2003.
16. Dirk Pattinson and Carsten Schürmann. Vote counting as mathematical proof. In Bernhard Pfahringer and Jochen Renz, editors, *Proc. AI 2015*, volume 9457 of *Lecture Notes in Computer Science*, pages 464–475. Springer, 2015.
17. Dirk Pattinson and Mukesh Tiwari. Schulze voting as evidence carrying computation. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Proc. ITP 2017*, volume 10499 of *Lecture Notes in Computer Science*, pages 410–426. Springer, 2017.
18. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 129–140, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.
19. Ronald L Rivest. On the notion of software independencein voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3759–3767, 2008.
20. Ronald L. Rivest and Emily Shen. An optimal single-winner preferential voting system based on game theory. In Vincent Conitzer and Jrg Rothe, editors, *Proc. COMSOC 2010*. Duesseldorf University Press, 2010.
21. Markus Schulze. A new monotonic, clone-independent, reversal symmetric, and Condorcet-consistent single-winner election method. *Social Choice and Welfare*, 36(2):267–303, 2011.
22. Björn Terelius and Douglas Wikström. Proofs of restricted shuffles. In *AFRICACRYPT*, volume 6055 of *Lecture Notes in Computer Science*, pages 100–113. Springer, 2010.
23. Douglas Wikström. A commitment-consistent proof of a shuffle. In *Proceedings of the 14th Australasian Conference on Information Security and Privacy*, ACISP '09, pages 407–421, Berlin, Heidelberg, 2009. Springer-Verlag.