Verifiable Homomorphic Tallying for the Schulze Vote Counting Scheme

No Author Given

No Institute Given

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 verifying how that voter had voted.

End to end verifiability and the related notation of software independence [16] have been claimed properties for many voting schemes. Kusters et al [9] gave a cryptographic formulation whose value is highlighted by the attacks it revealed in established voting schemes [10].

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 [7] 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 [6]. Informally, a ZKP is a probabilistic and interactive proof where one entity provides evidence about knowledge of a secret without providing no information other than the truth of the statement, with overwhelming probability. Later results [5][2] 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 voting scheme. We show how the Schulze Method 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.

Leftover Text?

In general, every election has purpose to elect candidates based on cast ballots while maintaining the integrity of election i.e. tallied correctly, maintained individual ballot privacy, and coercions free. Many of these properties we get for free in paper ballot election, but for electronic voting, not only our protocol should have these properties, but at the same time they should be evident. The inherent nature of tension (conflict) between privacy, keeping the votes private and anonymous (unidentifiable), and universal verifiablity, anyone can verify tallying is correct (consistent) according to election rules, makes electronic voting schemes very challenging. If we publish the ballots in plain text it certainly offers universal verifiability, but at the same time it opens the possibility of coercion [3], and if we don't publish then it certainly offers the more privacy, but hampers universal verifiablity.

The Schulze Method. The Schulze Method [18] is a preferential, single-winner vote counting scheme that is gaining popularity due to its relative simplicity while retaining near optimal fairness [17]. 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, for candidates x and y,

$m(x, y) =$ ballots that rank x higher than $y-$ballots that rank y higher than x.

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

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

is the lowest margin encountered on a path.

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

In previous work [14] 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 [14] 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 [13].

2 Verifiable Homomorphic Tallying

```
Bullet points of content for reference and later deletion

– general protocol, comparision with plaintext counting
– two-phase structure: margin matrix, then winners with winners as
  before: encryption commutes with margin computation
– homomorphic computation of margin matrix, ballot representation
– computation of winners (as for plaintext)
```

The realisation of verifiable of homomorphic tallying that we are about to describe follows the same two phases as the protocol: for each ballot, we decide whether it is formal, and if so, homomorphically update the margin function. We do this for every ballot, and record this in the certificate. We then record the decryption of the margin function in the certificate, and then publish the winner, together with evidence based on the margin matrix, as in the case of counting plaintext ballots (where evidence for winning also only depends on the margin matrix). We now describe the two phases 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

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

where $\oplus$ denotes homomorphic addition, eb is an encrypted ballot in matrix form (i.e. decrypting $\mathrm{eb}(x, y)$ 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 $\mathrm{eb}(x, y)$ 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 $\mathrm{eb} : C \times C \to \mathbb{CT}$ (where $\mathbb{CT}$ is a chosen set of ciphertexts), and it is valid if the pointwise encryption, i.e. the matrix ballot $b(x, y) = \mathsf{dec}(\mathsf{enc}(x, y))$ is valid (we use dec to denote 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 [1] to evidence the correctness of the shuffle, and zero-knowledge proofs of honest decryption [reference missing] to evidence correctness of decryption. This achieves ballot secrecy owing to the fact that the permutation is never revealed.

In summary, the homomorphic computation of the margin matrix starts with an encryption of the zero margin em, and for each ciphertext ballot eb

1. publishes a shuffle of eb together with a ZKP of correctness

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

Witnessing of Winners. Once all ballots are counted, the computed margin is decrypted, and winners (together with evidence of winning) are being computed as for plaintext counting. We discuss this this part only briefly to be self contained as it is as for plaintext counting [14]. 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

```
Bullet points of content for reference and later deletion
– Axiomatisation of the cryptographic primitives
– correctness proof modulo correct crypto
– describe certificates and how to check them
– show that ‘‘certificates certify’’
```

Now we explain the cryptographic primitives in realization of verifiable homomorphic tally. These primitives are treated as abstract entity inside theorem prover Coq, and it includes private key, public key, and functions for encryption, decryption, generating permutation, generating commitment [15], shuffling a list of ciphertext by a given permutation, generating zero knowledge proof, and verification of zero knowledge proof. Since these entities are abstract entity inside Coq, we have assumed axioms about them, hence axiomatisation of cryptographic primitives.

(* I can explain these primitives that what does encryption and decryption do, what does generate permutation do, but how to connect them with counting explained below ?. *)

We describe homomorphic Schulze counting as the dependent inductive data type, *ECount*, which is parametrised by *group*, a triplet which includes *Generator*, *Prime*, and *Publickey*, list of cast ballots *bs*, and depends on another inductive data type *EState*. The purpose of *EState* to capture the intermediate states of counting i.e. constructing homomorphic margin function, or decryption of fully constructed homomorphic margin function, or determination of final result.

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

Intuitively, we can construct element of type EState using three constructors *epartial*, *edecrypt*, and *ewinners*. The interpretation is

– *epartial* takes a pair, list of uncounted ballots and invalid ballots seen so far, and computed homomorphic margin so far
– *edecrypt* takes final decrypted margin after all the ballots are counted
– *ewinners* take a boolean function which returns true(false) for winners(losers)

The interpretation of *ECount* is that inhabitant of this type is correct execution of homomorphic counting method. Intuitively, the counting start with the first constructor, *ecax*, with list of all uncounted ballots *us*, encrypted zero margin *encm*, decrypted zero margin *decm*, and zero knowledge proof *zkpdec* of that *decm* is indeed honest decryption of *encm* with assertions that ensure that all the uncounted ballots are equal to list of cast ballot, every entry in *decm* is zero and zero knowledge proof of honest decryption verifies. This constructor bootstraps the vote counting process, and at very high level (with omitted assertions)

Ecax

$$\frac{}{\text{(grp, bs)} \vdash \text{ECount grp bs (epartial (us, [\,]) encm)}}$$

In next step, we take a ballot, *u*, from pile of uncounted ballots, (*u* :: *us*), and depending on its validity, we either update the margin function homomorphically, constructor *ecvalid*, or move it to the list of invalid ballots, constructor *ecinvalid*.

Ecvalid: u is valid so update the margin nm = m $\bigoplus$ u

$$\frac{\text{(grp, bs)} \vdash \text{ECount grp bs (epartial (u :: us, inbs) m)}}{\text{(grp, bs)} \vdash \text{ECount grp bs (epartial (us, inbs) nm)}}$$

Ecinvalid: u is not valid so move it list of invalid pile (u :: inbs)

$$\frac{\text{(grp, bs)} \vdash \text{ECount grp bs (epartial (u :: us, inbs) m)}}{\text{(grp, bs)} \vdash \text{ECount grp bs (epartial (us, u :: inbs) m)}}$$

Basically deciding the validity of ballot involves shuffling its rows and columns by some permutation, which we don't want to reveal to achieve ballot secrecy, and decryption, so we need to produce compelling evidence to make sure that the operation performed in each step of deciding validity is verifiable.

To hide the permutation, we use the well known protocol Pedersen's commitment [15]. Pedersen commitment scheme has the following properties.

- Hiding: A dishonest party can't discover honest party's value
- Binding: A dishonest party cannot open his commitment in more than one way i.e. he should not be able open to a value different from the one he committed to

We can hide any given permutation, π, by committing it using Pedersen's commitment scheme and publishing the commitment, c_π, but for binding step rather than opening the permutation π to show that it's the permutation that we committed to, we generate zero knowledge proof, zkp_π, using permutation π and its commitment c_π. This zero knowledge proof can be used to prove that c_π is indeed a commitment of some permutation, and this permutation is used in (commitment consistent)shuffling without revealing(opening) it [19].

Now that we have a way to hide permutation pi, we use it to permute(shuffle) each row of ballot, u, producing ballot, v, with zero knowledge proof, zkppermuv which can be used to verify that each row of ballot v is indeed the row permutation of ballot u, and permuting each column of v to produce ballot, w, with zero knowledge evidence zkppermvw. We decrypt ballot w to produce ballot, b, which in turn used to decide the validity of original ballot u, and this step is again accompanied by zero knowledge evidence zkpdecw.

The constructor, **ecdecrypt**, marks the finish to counting with fully constructed encrypted margin **encm**, decrypted margin **decm** with honest decryption zero knowledge proof, **zkp**.

The final constructor, **ecfin**, takes fully decrypted margin **dm**, boolean function **w** that determines election winner, and **d**, that delivers type level evidence of winning or losing, consistent with w.

```
Inductive ECount (group : Group)
 (bs : list eballot) : EState -> Type :=
 | 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
    grp (decm c d) (encm c d) (zkpdec c d)
    = true) ->
    ECount grp bs (epartial (us, []) encm)
 | ecvalid (cpi : Commitment) (zkpcpi : PermZkp)
   (u v w : eballot) (b : pballot)
   (zkppermuv : cand -> ShuffleZkp)
   (zkppermvw : cand -> ShuffleZkp)
   (zkpdecw : cand -> cand -> DecZkp)
   (us inbs : list eballot)
```

```
  (m nm : cand -> cand -> ciphertext) :
  ECount grp bs (epartial (u :: us, inbs) m) ->
  matrix_ballot_valid b ->
  (verify_permutation_commitment grp
   (length cand_all) cpi zkpcpi = true)->
  (forall c, verify_row_permutation_ballot
    grp u v cpi zkppermuv c = true) ->
  (forall c, verify_col_permutation_ballot
    grp v w cpi zkppermvw c = true) ->
  (forall c d,
   verify_zero_knowledge_decryption_proof
    grp (b c d) (w c d) (zkpdecw c d) = true) ->
  (forall c d, nm c d =
   homomorphic_addition grp (u c d) (m c d)) ->
  ECount grp bs (epartial (us, inbs) nm)
| ecinvalid (cpi : Commitment) (zkpcpi : PermZkp)
  (u v w : eballot) (b : pballot)
  (zkppermuv : cand -> ShuffleZkp)
  (zkppermvw : cand -> ShuffleZkp)
  (zkpdecw : cand -> cand -> DecZkp)
  (us inbs : list eballot)
  (m : cand -> cand -> ciphertext) :
  ECount grp bs (epartial (u :: us, inbs) m) ->
  ~matrix_ballot_valid b ->
  (verify_permutation_commitment grp
   (length cand_all) cpi zkpcpi = true) ->
  (forall c, verify_row_permutation_ballot
    grp u v cpi zkppermuv c = true) ->
  (forall c, verify_col_permutation_ballot
    grp v w cpi zkppermvw c = true) ->
  (forall c d,
   verify_zero_knowledge_decryption_proof
    grp (b c d) (w c d) (zkpdecw c d) = true) ->
  ECount grp bs (epartial (us, (u :: inbs)) m)
| ecdecrypt
  (inbs : list eballot)
  (encm : cand -> cand -> ciphertext)
  (decm : cand -> cand -> plaintext)
  (zkp : cand -> cand -> DecZkp) :
  ECount grp bs (epartial ([], inbs) encm) ->
  (forall c d,
   verify_zero_knowledge_decryption_proof
    grp (decm c d) (encm c d) (zkp c d) = true) ->
  ECount grp bs (edecrypt decm)
| ecfin
```

```
  dm (w : cand -> bool)
  (d : (forall c, (wins_type dm c) +
                  (loses_type dm c))) :
  ECount grp bs (edecrypt dm) ->
  (forall c, w c = true <-> exists x, d c = inl x)->
  (forall c, w c = false<-> exists x, d c = inr x)->
  ECount grp bs (ewinners w).
```

We formally states that for every list of ballots, we can find a boolean function that decides winners [14], but what more important is that the term of type **ECount grp bs (ewinners f)**, that witnesses the correct execution of count, can be produced as evidence, known as certificate, to scrutineers to audit the election.

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

Not only we provide the tally sheet (discussed in details in next section) to verify the computation, but we provide proof of correctness against fully verified implementation of Schulze method [14]. The basic intuition behind the proofs is very simple.

(* This one is bit difficult to find *) It basically hinges on the fact that if a ballot (represented as function $C \to N$) contributes to margin function then its encryption would also contribute to homomorphic margin, and vice-versa.

```
Lemma final_correctness :
  forall  (grp : Group) (bs : list ballot) (pbs : list pballot)
  (ebs : list eballot) (w : cand -> bool)
  (H : pbs = map (fun x =>
       (fun c d => decrypt_message grp privatekey (x c d))) ebs)
  (H2 : mapping_ballot_pballot bs pbs),
  Count bs (winners w) -> ECount grp ebs (ewinners w).

Lemma final_correctness_rev :
  forall  (grp : Group) (bs : list ballot) (pbs : list pballot)
  (ebs : list eballot) (w : cand -> bool)
  (H : pbs = map (fun x =>
       (fun c d => decrypt_message grp privatekey (x c d))) ebs)
  (H2 : mapping_ballot_pballot bs pbs),
  ECount grp ebs (ewinners w) -> Count bs (winners w).
```

4 Extraction and Experiments

```
– extension of Unicrypt with homomorphic addition
– coupling between library and extracted code
```

- `need for light weight wrappers`
- `experimental results: how long, how much memory, certicicate size`

We use Coq extraction mechanism[12] to extract its proofs into programs retaining all the terms which lives in Type universe, and erases every term from Prop universe. We use this mechanism to extract our formalized Coq proofs into OCaml[11] program, and instantiate the axiomatic cryptographic primitives with corresponding primitives in Unicrypt library (* reference for unicrypt ? *). For cryptographic purpose, we had choice to use Unicrypt, and Verificatum, because both libraries are open source and exhaustively tested; however, we found Verificatum is difficult to use, so we settled with Unicrypt. Since Unicrypt is written in Java and our extracted code is in OCaml, we can not call Unicrypt functions directly from OCaml. Besides, it does not have all the primitives, homomorphic addition, construction of honest decryption zero knowledge proof and its verification, we need, so we have written wrapper, using Java-Ocaml library (* github link for library? Robert Constable Has given links of github in his paper about bar induction, the good, the bad and the ugly *), with functionality missing in Unicrypt, and it works as middle man between OCaml and Java.

After instantiating the cryptographic primitives in extracted OCaml code to Unicrypt library via wrapper, we ran it on set of small set of ballots with 3 candidate A, B, and C participating in election which produces the scrutiny 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.

```
M: AA(13.., 10..) AB(90.., 14..) AC(11.., 23..) BA(16.., 13..) BB(79.., 46..)
BC(12.., 14..) CA(50.., 53..) CB(70.., 68..) CC(23.., 82..),
D: [AA: 0 AB: 0 AC: 0 BA: 0 BB: 0 BC: 0 CA: 0 CB: 0 CC: 0],
Zero-Knowledge-Proof-of-Honest-Decryption: [..]
----------------------------------------------------------------------------
V: [AA(42.., 15..) AB(63.., 32..) AC(70, 44..) BA(47.., 34..) BB(16.., 28..)
BC(39.., 16..) CA(19.., 13..) CB(57.., 12..) CC(19.., 89..),..], I:
[],
M: AA(12.., 11..) AB(13.., 66..) AC(16.., 14.) BA(48.., 31..) BB(15.., 52..)
BC(15.., 68..) CA(39.., 69..) CB(12.., 78..) CC(10.., 40..),
Row-Permuted-Ballot: AA(53.., 16..) AB(23.., 44..) AC(72.., 47..)
BA(10.., 19..) BB(74.., 16..) BC(20.., 60..) CA(44.., 10..) CB(12.., 16..)
CC(59.., 98..),
Column-Permuted-Ballot: AA(81.., 41..) AB(17.., 14..) AC(10.., 14..)
BA(37.., 12..) BB(14.., 66..) BC(10.., 13..) CA(12.., 13..)
CB(14.., 16..) CC(12.., 10..),
Decryption-of-Permuted Ballot: AA0 AB-1 AC1 BA1 BB0 BC1 CA-1 CB-1 CC0,
Zero-Knowledge-Proof-of-Row-Permutation: [Tuple[...]],
Zero-Knowledge-Proof-of-Column-Permutation: [Tuple[..]],
Zero-Knowledge-Proof-of-Decryption: [Triple[..]],
Permutation-Commitment: Triple[..]
Zero-Knowledge-Proof-of-Commitment: Tuple[..]
-------------------------------------------------------------------------------------
.
.
```

```
.
---------------------------------------------------------------------------------------
V: [AA(36.., 10..) AB(20.., 13..) AC(75.., 43..) BA(13.., 31..) BB(27.., 82..)
BC(31.., 50..) CA(16.., 11..) CB(74.., 15..) CC(26.., 36..)], I: [], M: AA(86.., 38..
AB(21.., 14..) AC(16.., 25..) BA(16.., 22..) BB(18.., 15..) BC(11.., 63..)
CA(15.., 34..) CB(76.., 18..) CC(11.., 10..),
Row-Permuted-Ballot: .., Column-Permuted-Ballot: ..,
Decryption-of-Permuted-Ballot: AA0 AB-10 AC1 BA10 BB0 BC1 CA-1 CB-1 CC0,
Zero-Knowledge-Proof-of-Row-Permutation: [..],
Zero-Knowledge-Proof-of-Column-Permutation: [..],
Zero-Knowledge-Proof-of-Decryption: [..],
Permutation-Commitment: Triple[..],
Zero-Knowledge-Proof-of-Commitment: Tuple[..]
---------------------------------------------------------------------------------------
V: [], I: [AA(36.., 10..) AB(20.., 13..) AC(75.., 43..) BA(13.., 31..) BB(27.., 82..)
BC(31.., 50..) CA(16.., 11..) CB(74.., 15..) CC(26.., 36..)], M: ..,
D: [AA: 0 AB: 4 AC: 4 BA: -4 BB: 0 BC: 4 CA: -4 CB: -4 CC: 0],
Zero-Knowledge-Proof-of-Decryption: [..]
---------------------------------------------------------------------------------------
D: [AA: 0 AB: 4 AC: 4 BA: -4 BB: 0 BC: 4 CA: -4 CB: -4 CC: 0]
winning: A
   for B: path A --> B of strength 4, 5-coclosed set: [(B,A),(C,A),(C,B)]
   for C: path A --> C of strength 4, 5-coclosed set: [(B,A),(C,A),(C,B)]
losing: B
   exists A: path A --> B of strength 4, 4-coclosed set: [(A,A),(B,A),(B,B),(C,A),
   (C,B),(C,C)]
losing: C
   exists A: path A --> C of strength 4, 4-coclosed set: [(A,A),(B,A),(B,B),
   (C,A),(C,B),(C,C)]
```

Now, we will wear the hat of scrutinizer to audit the election based on scrutiny sheet to gain more understanding. In the very first line, all we have to make sure that every entry is encrypted margin function M is encryption of zero. As we can see that every entry in decrypted margin D of encrypted margin M is zero, but we can verify this fact with given Zero-Knowledge-Proof-of-Honest-Decryption.

In next few lines, we need to make sure that if ballot from the top of pile (all cast ballots) V is valid then it is added homomorphically to running margin M, and if not then moved to pile of invalid ballots I. In order to certify the claim that ballot under consideration is valid, or not, all we have to do is certify the claim that Row-Permuted-Ballot is indeed row shuffle of ballot under consideration by some (secret)permutation using the information Zero-Knowledge-Proof-of-Row-Permutation, Permutation-Commitment, and Zero-Knowledge-Proof-of-Commitment. We need to follow the same methodology to verify the column permuted ballot Column-Permuted-Ballot. Finally, all we have to verify the decryption claim of Decryption-of-Permuted-Ballot is honest decryption of Column-Permuted-Ballot using Zero-Knowledge-Proof-of-Decryption. We keep doing this step until we have processed all the ballots from pile V, i.e. V = [].

Once we have exhausted all the ballots from pile V, i.e. V = [], we need to certify the that decrypted margin function D is indeed the honest decryption of encrypted margin function M, and we can verify this claim using Zero-Knowledge-Proof-of-Decryption.

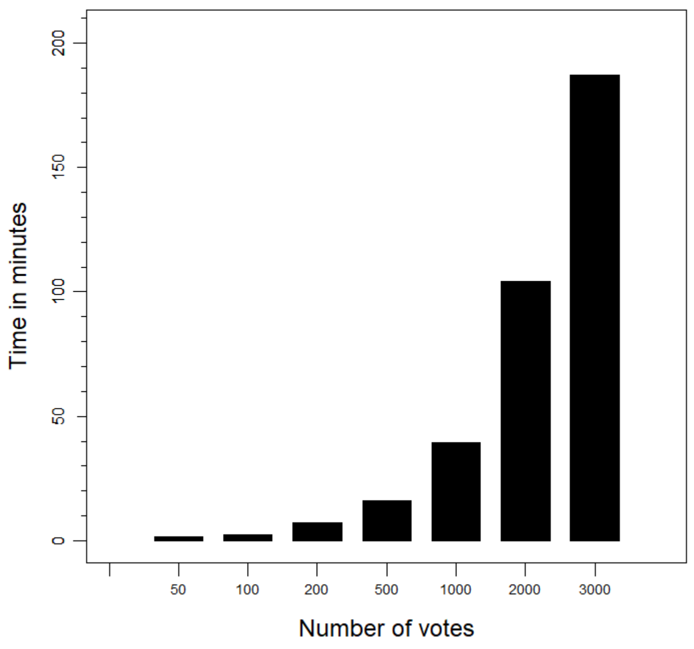
In final step, we need to check the claim of winner(s) and losers. For winner(s), we need to verify that the claimed path exists with the given strength, and claimed set is indeed the coclosed, and similar steps for losers.

In a nutshell, one can think of auditing the election involves verifying these three steps

– The correctness of steps computing homomorphic margin
– The decryption of fully constructed margin
– For winners

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.



5 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 expertiments 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 [8].

Modelling Assumptions. In our modelling of the cryptographic primitives, in particular the zero knowledge proofs, we have assumed that zero knowledge proofs can be verified, whereas in reality zero-knowledge proofs just confirm facts with very high probability which we have deliberately ignored. 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 [14]. 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. Shuffle-sum: 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. 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.
6. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *STOC*, pages 291–304. ACM, 1985.
7. 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.
8. 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.
9. 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.
10. 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.
11. X. Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rèmy, and Jérôme Vouillon. The ocaml reference manual, 2013.
12. 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.
13. 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.
14. 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.

15. 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.
16. 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.
17. 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.
18. 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.
19. 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.
20. Xun Yi, Russell Paulet, and Elisa Bertino. *Homomorphic Encryption and Applications*. Briefs in Computer Science. Springer, 2014.

Notes and Leftovers Start Here

A Format of Ballots

Each ballot, B, is a n×n matrix where n is the number of candidates participating in election. Each voter marks the entry (i,j) in ballot B as 1 if he prefers candidate i over j otherwise mark it 0. No candidate is preferred over itself, so all the diagonal entries are filled with 0. The reason for representing a ballot as a matrix because the precise

- motivate the choice of ballots and compare with plaintext version
- define encryption function that changes representation
- a ballot is valid if and only if there is evidence that proves its validity
- encrypting valid ballots leads to valid ballots, and the same for invalid ballots
- Each ballot, B, is a n×n matrix where n is the number of candidates participating in election. Each voter marks the entry (i, j) as 1 and (j, i) as -1 in ballot B if he prefers candidate i over j. If the voter has no preference between candidate i and j i.e. they are ranked same then voter would mark entry (i, j) and (j, i) as 0. No candidate is preferred over itself, so all the diagonal entries are filled with 0. The reason for representing a ballot as a matrix because the precise
  nature of our algorithm involving no decryption of individual ballot to protect ballot privacy, facilitating the scrutiny for validity, and ease of computing cumulative margin over encrypted data to compute the winners.
  Each entry in ballot is encrypted by additive ElGamal i.e. encrypting the message m as $(g^r, h^r g^m)$ where g is generator of cycle group G, and $h = g^x$ is private key. After performing the encryption, a plain text ballot i.e. matrix having entries of 0, and 1 changes to a matrix having pair for

– Each entry in ballot is encrypted by additive ElGamal i.e. encrypting the message m as $(g^r, h^r g^m)$ where g is generator of cycle group G of order q, private key x and randomness r are randomly chosen from $\{1, \ldots, q - 1\}$, and public key is computed as $h = g^x$. It is important to note that generator g, private key x and public key h are generated during key generation phase of Elgamal algorithm while randomness r is generated fresh for each message.

After performing the encryption, a plain text ballot i.e. matrix having entries of -1, 0, and 1 changes to a matrix having pair for each entry representing the encryption of corresponding plain text. A representation of plain text ballot

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \ldots & x_{1n} \\ x_{21} & x_{22} & x_{23} & \ldots & x_{2n} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ x_{n1} & x_{n2} & x_{n3} & \ldots & x_{nn} \end{bmatrix}$$

and data structure used to represent this ballot in Coq is

```
Definition plaintext := Z.
Definition pballot := cand -> cand -> plaintext.
```

A representation of encrypted ballot

$$\begin{bmatrix} (y_{11}, z_{11}) & (y_{12}, z_{12}) & (y_{13}, z_{13}) & \ldots & (y_{1n}, z_{1n}) \\ (y_{21}, z_{21}) & (y_{22}, z_{22}) & (y_{23}, z_{23}) & \ldots & (y_{2n}, z_{2n}) \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ (y_{n1}, z_{n1}) & (y_{n2}, z_{n2}) & (y_{n3}, z_{n3}) & \ldots & (y_{nn}, z_{nn}) \end{bmatrix}$$

and the corresponding data structure in Coq is

```
Definition ciphertext := (Z * Z)%type.
Definition eballot := cand -> cand -> ciphertext.
```

The reason for using additive ElGamal is that it's composability on ciphertext under same public key. In short, (Should I explain the algorithm here that why this scheme works or explain it encryption section ?)

$$enc_k(m_1) * enc_k(m_2) = enc_k(m_1 + m_2)$$

Our choice of representing ballot as matrix comes with price. A malicious voter could try to encode all sort of ballots including cyclic ones i.e. candidate A is preferred over candidate B, candidate B is preferred over candidate C, and candidate C is preferred over candidate A, and it is clear that voter has not expressed his choices clearly as it can not be expressed in linear preference order and should not be considered for counting, or he could inflate his favourite candidate by any other number higher that 1, and It makes the ballot invalid because any value greater that 1 in a ballot would be similar to voter has casted more than one vote. We define a ballot is valid if all the entries in matrix is encryption of -1, 0 and 1, and there is no cycle in it. Formally, We have

```
Definition matrix_ballot_valid (p : pballot) :=
      (forall c d : cand, In (p c d) [-1; 0; 1]) /\
       valid cand p.


 Definition valid (A : Type) (P : A -> A -> Z) :=
      exists f : A -> nat,
          forall c d : A,
          (P c d = 1 <-> (f c < f d)%nat) /\
          (P c d = 0 <-> f c = f d) /\
          (P c d = -1 <-> (f c > f d)%nat)

     (* Which one should keep ? Upper one is in Code while this one is
        more intutive after expanding the definition *)
    Definition matrix_ballot_valid (p : pballot) :=
      (forall c d : cand, In (p c d) [-1; 0; 1]) /\
      (exists f : A -> nat,
          forall c d : A,
          (p c d = 1 <-> (f c < f d)%nat) /\
          (p c d = 0 <-> f c = f d) /\
          (p c d = -1 <-> (f c > f d)%nat))
```

Intuitively, it says that a ballot p is valid if we can find a function which expresses the notion of linear preference ordering for all candidates, and we prove that we can decide for each ballot if it is valid or not

```
  Lemma matrix_ballot_valid_dec :
        forall p : pballot, {matrix_ballot_valid p} +
                            {~matrix_ballot_valid p}.
```

By now, a attentive reader would note that we have claimed that we don't decrypt the ballot to preserve the privacy, so how can we decide the validity without decrypting the original ballot. We will discuss our whole cryptographic scheme in next section, but to quench the thrust of reader for now, we have proved that if ballot p was valid before encryption, then encrypting it followed by permuting each row by secret permutation σ and permuting again the each column of resulting matrix from previous step by same permutation, σ, and decrypting it back would still lead to valid ballot. (* Write this same Lemma for $matrix_ballot_valid$ then any permutation of it is valid and if it is invalid then any permutation of it is invalid. The proof is already lying there in other proofs so it's basically copy paste. *)

```
 Lemma perm_presv_validity :
    forall (A : Type) (P : A -> A -> Z),
      (forall c d : A, {c = d} + {c <> d}) ->
      (forall c d : A, {P c d = 1} + {P c d = 0} + {P c d = -1}) ->
      forall sig : A -> A, Bijective sig -> valid A P <-> valid A (perm A P sig)
```

And similarly, for invalid ballot

```
Lemma not_perm_persv_validity :
 forall (A : Type) (P : A -> A -> Z),
      (forall c d : A, {c = d} + {c <> d}) ->
      (forall c d : A, {P c d = 1} + {P c d = 0} + {P c d = -1}) ->
      forall sig : A -> A, Bijective sig -> ~ valid A P <-> ~ valid A (perm A P sig).
```

B Homomorphic Computation of Margin

– motivate and introduce how the margin is computed homomorphcially
– prove that encrption commutes with computation of margin

C Homomorphic Counting

– prove that encruption commutes with computation of winners
– discuss the evidence: publish cleartext margin function
– describe certificates and how to check them

D Experimental Results

– how long does it take, and how much memory?
– how large are the certificates?

E Discussion and Further Work

One aspect that we have not considered here is encryption of ballots to safe-guard voter privacy which can be incorporated using protocols such as shuffle-sum [3] and homomorphic encryption [20]. 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.