Verifiable Homomorphic Tallying for the Schulze Vote Counting Scheme

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

[1] Polyas GmbH, Germany
[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. Or 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 the combination is notoriously hard. Cryptography is enormously helpful in resolving this tension. Of particular use are Zero Knowledge Proofs (ZKP). ZKP was first studied by Goldwasser, Micali, and Rackoff [7]. Informally, they are a kind of probabilistic and interactive proof which provide no additional information then that statement is true with overwhelming probability. Later results [6][2] showed that all problems for which solutions can be efficiently verified have zero knowledge proofs.

- general need for crypt in vote counting
- gernal need for verifiablility in vote counting: certificates!

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 that 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 [14] 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].

`[Thomas: is there a standard reference for E2E verifiability?]`
Hopefully the above paragraph helps

The above desiderata can be realised using encrypted ballots, 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 [8] and verifiable shuffling [1]. (Separate techniques exist to prevent ballot box stuffing and to guarantee cast-as-intended.)

`[Thomas: do we have / need references for ballot box stuffing or cast-as-intended?]` I don't think so

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 [16] is a preferential, single-winner vote counting scheme that is gaining popularity due to its relative simplicity while retaining near optimal fairness [15]. 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,

$$m(x, y) = \text{ballots that rank } x \text{ higher than } y - \text{ballots that rank } y \text{ higher than } x.$$

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 [13] 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 [13] for details of the second part of the certificate as this will remain unchanged in the work we are reporting here.

Related Work.

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 $r : C \to \mathbb{N}$ 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 [13]. 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’’`

Following from previous section, we now give elaborated view of formalization. The cryptographic primitives are treated as black box, and assumed each primitive holds certain properties which are modelled as axiom in Coq [5]. We have total six axioms about decryption, (secret) permutation, and shuffle, and we invite the reader to examine the validity of these axioms, but for sake of completeness and detail we sketch out the details of encryption scheme.

The very first one among all these primitives are encryption function and decryption function with the trait, as suggested by name of axiom itself, that decryption is deterministic.

```
Inductive Group : Type :=
    group : Prime -> Generator -> Publickey -> Group.

Variable encrypt_message :
    Group -> plaintext -> ciphertext.

Variable decrypt_message :
    Group -> Privatekey -> ciphertext -> plaintext.

Axiom decryption_deterministic : forall (grp : Group)
  (privatekey : Privatekey) (pt : plaintext),
  decrypt_message grp privatekey (encrypt_message grp pt) = pt.
```

Intuitively, it means that if we take some plaintext, encrypt it, and decrypt it we should get the original plaintext. This axiom certainly forces us to decrypt the values honestly during formalization, but to make sure that the certificate, which will be discuss in a moment, can be checked by scrutinizers to audit the election irrespective of how it was constructed, we also produce the evidence in form of zero knowledge proof for honest decryption which can be verified later independently. In order to attain verifiability, we need primitives to build honest decryption zero knowledge proof to decouple the verification of election outcome from code inspection used to produce result. As it is evident from the names itself, these primitives are about constructing honest decryption zero knowledge proof object, checking if a plaintext is indeed honest decryption of claimed ciphertext using decryption zero knowledge proof object, with property that what it means to be when honest decryption proof checks out i.e. return true.

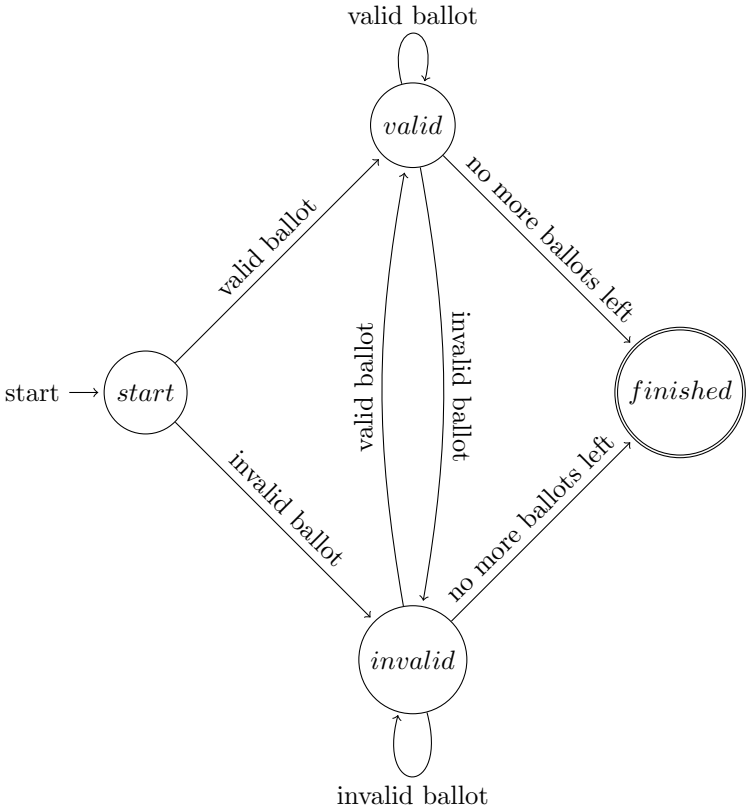
```
Variable construct_zero_knowledge_decryption_proof :
   Group -> Privatekey -> ciphertext -> DecZkp.

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

Axiom verify_true :
   forall (grp : Group) (pt : plaintext) (ct : ciphertext)
   (privatekey : Privatekey)
   (H : pt = decrypt_message grp privatekey ct),
   verify_zero_knowledge_decryption_proof grp pt ct
   (construct_zero_knowledge_decryption_proof grp privatekey ct)
   = true.
```

The axiom, `verify_true`, itself states that if plaintext, *pt*, is honest decryption of ciphertext, *ct*, then primitive `verify_zero_knowledge_decryption_proof` returns true for suitably constructed zero knowledge proof.

We have followed the similar line of arguments for other cryptographic primitives used in formalization including generation of secret permutation, the secret permutation is indeed a permutation, and ballots are indeed permuted by the claimed secret permutation by generating zero knowledge proof for each operation performed during the counting ballots. Once we have axiomatized cryptographic primitives, we plug them into our formalization to construct verifiable counting system.

At very high level, we can think of counting ballots as finite state machine or automaton which has following states

- starting state where all the ballots are uncounted, no invalid ballot, and margin function is zero.
- intermediate steps where we inspect a ballot from list of uncounted ballots, and depending on its validity we either update the margin (if ballot is valid), or move to pile to invalid ballots.
- final (accepting) state where no more uncounted ballots left, and use the computed margin to produce result



We use Coq's dependent inductive data type facility to capture the notion of finite state machine. To be precise, the outcome of a election is inhabitant of inductive data type used to express the different states of counting (finite state

machine) using various constructors, i.e. each constructor signifies some states of counting. Because of Coq's expressive (dependent) type system any arbitrary term can appear at type level, we can parameterised the counting (finite state machine) type by suitable terms to make sure that computation for constructing terms of type counting (finite state machine) executes correctly.

In order to capture the states of ballots and margin function during counting process, we introduce inductive data type **EState**, and the interpretation of it is,

- list of uncounted ballots which must be dealt with, and invalid ballots seen so far in counting process
- computed homomorphic margin so far

or decrypted margin function after all the ballots are counted, or a boolean function used to determine the winner.

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

We formalize the correct counting of homomorphic Schulze method by dependent inductive data type, **ECount**, which is parameterised by group, used for cryptographic primitive, list of cast ballots, and **EStates**. It is very crucial to notice that adding **EState** as parameter to **ECount** forces the computation constructing the term of type **ECount** to follow the each step of protocol correctly. The computation expressed above follows the same spirit as appending two dependent type vector indexed over natural numbers of length n, and m would always produce a vector of length (n + m). Since Coq's expressive type system allows us to represent this fact at type level only correct implementation of append would type check.

In order to show that protocol (counting) executed correctly, we need to show that it always ends up accepting state i.e. produces the term (inhabitant) of type **ECount grp bs (ewinners w)**.

```
Inductive ECount (grp : 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 (u : eballot) (v : eballot) (w : eballot)
           (b : pballot) (zkppermuv : cand -> ShuffleZkp)
```

```
          (zkppermvw : cand -> ShuffleZkp)
          (zkpdecw : cand -> cand -> DecZkp)
          (cpi : Commitment) (zkpcpi : PermZkp)
          (us : list eballot) (m nm : cand -> cand -> ciphertext)
          (inbs : list eballot) :
  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 (u : eballot) (v : eballot) (w : eballot)
            (b : pballot) (zkppermuv : cand -> ShuffleZkp)
            (zkppermvw : cand -> ShuffleZkp)
            (zkpdecw : cand -> cand -> DecZkp)
            (cpi : Commitment) (zkpcpi : PermZkp)
            (us : list eballot) (m : cand -> cand -> ciphertext)
            (inbs : list eballot) :
  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 (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 (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).
```

– The first constructor, **ecax**, marks the beginning the counting process by taking list of uncounted ballots, **us**, encrypted margin function, **encm**, decrypted margin function, **decm**, and zero knowledge proof, **zkpdec**, with proof that uncounted ballots, **us**, are same as cast ballots, **bs**, every entry in

decrypted margin is 0, and **decm** is indeed the honest decryption of **encm** as honest decryption proof checker returns true.

- The constructor, **cvalid**, takes ballot **u** from uncounted ballots, ballot **v** which is row permutation of **u** by secret permutation, **w** which is column permutation of **v** by same secret permutation, **b** which is decryption of w, zero knowledge proof, **zkppermuv**, of v is indeed row permutation of u, zero knowledge proof, **zkppermvw**, of w is column permutation of v, zero knowledge proof, **zkpdecw**, of b is honest decryption w, commitment, **cpi**, to secret permutation, and zero knowledge proof, **zkpcpi**, that we indeed committed to secret permutation with list of uncounted ballot, **us**, current margin, **m**, updated margin, **nm**, after adding ballot u homomorphically to current margin, **m**, and list of invalid ballots, **inbs**, seen so far with the usual proofs about validity data embedded in constructor.
- The constructor, **cinvalid**, is very similar to **cvalid** except rather than updating the running margin function, **m**, we move the ballot to list of invalid ballot, **inbs**.
- The constructor, **ecdecrypt**, marks the finish of counting all the ballot as it can be seen by state, **ECount grp bs (epartial ([], inbs) encm)**, with list of invalid ballots, **inbs**, final encrypted margin, **encm**, decrypted margin, **decm**, zero knowledge proof of honest decryption, **zkp** with usual proofs about the consistency of data in constructor.
- The final constructor, **ecfin**, takes decrypted margin, **dm**, boolean function, **w**, and d that delivers type level evidence of winning and losing for each candidate.

We formally state that for every list of ballots, we can find a boolean function that decides winners, but what more important is that the term of type **ECount grp bs (ewinners f)** captures each step of execution which can be produced as evidence, known as certificate, to scrutineers for inspection.

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

Coq extraction mechanism[12] allows us to extract its proofs into programs retaining all the terms which lives in Type universe, and erases every term from Prop universe. After extracting Coq proofs into OCaml[11] program, and plugging the cryptographic primitives into Unicrypt library via a thin wrapper written in OCaml using the library which calls Java function from OCaml (more details in next section), and executing it on set of ballots produces the following certificate (as promised earlier), but the numbers appearing are very large and snipped because of space constraint.

```
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..),
Decrypted margin [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: ..,
DM: [AA: 0 AB: 4 AC: 4 BA: -4 BB: 0 BC: 4 CA: -4 CB: -4 CC: 0],
Zero Knowledge Proof of Decryption: [..]
-------------------------------------------------------------------------------------------
DM: [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 strenght 4, 5-coclosed set: [(B,A),(C,A),(C,B)]
   for C: path A --> C of strenght 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)]
```

We have 3 candidates A, B, and C. In the beginning of computation, we publish the information embedded in constructor **ecax** i.e. encrypted zero margin, M, its decryption to show that all the entries are 0, and zero knowledge proof of honest decryption. In next line, we display the first ballot from uncounted pile, V, that we are about to count, pile of invalid ballots so far, I, running encrypted margin, M, row permutation, i.e. permuting each row by secret permutation, of ballot that we are counting, column permutation, i.e. permuting each column by same secret permutation, of ballot obtained from previous step after row permutation, decryption of final permuted ballot with zero knowledge proof of each operation. Depending on the validity of finally permuted ballot, either we update the margin function homomorphically, **cvalid**, or move it to pile of invalid ballots, **cinvalid**. We continue until every ballot is counted. Once counting is finished then we decrypt the fully constructed margin function with zero knowledge, **ecdecrypt**. In final step, we show the evidence provided by constructor **ecfin**, which included decrypted margin, winners(loser) with evidence about winning(losing).

Not only we provide certificate to verify the computation, but we provide proof of correctness against fully verified implementation of Schulze method [13] assuming that if there one to one correspondence between plaintext ballots and encrypted ballots, i.e. if plaintext ballot is valid then its encryption is also valid and if encrypted ballot is valid then it's decryption is also valid, then computing winners via plaintext ballot is same as encrypted ballot 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

5 Analysis

– trust base: library, wrapper, bindings, extraction into unverified BigInts
– security: probabilistic aspects ignored, attacker model
– applicability to real world scenarios

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. 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.
7. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *STOC*, pages 291–304. ACM, 1985.
8. 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.
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 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.
14. 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.
15. 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.
16. 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.
17. 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 $n \times$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 $n \times$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 $\{1, \ldots, q\text{ - }1\}$, 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 $matrix_b allot_v alid$ 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 [17]. 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.