Formal Verification of Homomorphic Vote Counting Algorithms

Jakub Nabaglo, u5558578

15 July 2016

Abstract

It is crucial in a democracy that elections be independently verifiable and free from voter coercion. These goals are often contradictory as publishing ballots cast, even anonymised ones, may enable a corrupt individual to verify whether a voter followed their instructions.

The Shuffle-Sum protocol enables a central authority to use homomorphic encryption to tally encrypted ballots in a single transferable vote election, and to publish a certificate of validity of the count. The certificate can then be independently verified without knowing the plaintext of the ballots. This counting system meets the need for result verifiability and significantly reduces the potential for coercion. This is important in STV elections as they are particularly vulnerable to illegitimate influence.

Recognising a need for a formally verified specification of the Shuffle-Sum protocol, this report lays the groundwork by providing a formally verified specification of first-past-the-post counting using homomorphic encryption.

Chapter 1

Background

1.1 Asymmetric encryption

Asymmetric encryption allows two parties who have never met to communicate securely over a public channel. Each transmission requires the use of two keys: the public key and the private key of the recipient. The public key may be communicated over an unsecured channel to the sender of a transmission.[1] However, the private key must remain a secret of the message's recipient.

The public key may be easily computed from the private key, but the inverse must be computationally difficult.

The public key is sufficient to encrypt a message. However, the private key is required to decrypt it.

If Peta wished to securely communicate a message to Tony over a public channel, Tony would generate a public-private key pair and transmit his public key to Peta in plaintext. Peta would then use the public key to encrypt the secret, and would send it to Tony. Tony would then decrypt it using his private key. Malcolm, wishing to eavesdrop on the transmission, could intercept Tony's public key and the encrypted message, but would not be able to decrypt it, lacking Tony's private key and not having the computational power to infer it from the public key.

One asymmetric encryption algorithm is the Paillier cryptosystem. To generate a public-private key pair, one chooses p, q random large primes, such that $\gcd(pq, (p-1)(q-1)) = 1$. The constants $n = pq$ and $\lambda = \operatorname{lcm}(p-1, q-1)$ are then computed. A random integer $g \in \mathbb{Z}^*_{n^2}$ is selected such that $n | \operatorname{ord} g$. Finally, $\mu = ((g^\lambda \mod n^2) - 1)//n)^{-1} \mod n$ is computed[2]. The public key is then (n, g), whereas the private key is (λ, μ).[3]

Observe that it is computationally difficult to obtain the private key from the public

[1] Whilst communicating the key, it is necessary to prevent man-in-the-middle attacks by making sure that no third party tampers with the transmission. Other cryptographic techniques are concerned with this problem.

[2] '//' represents Euclidean division. The 'mod' operator has the same precedence as multiplication.

[3] Description from Katz and Lindell (2007).

key. To compute λ, one would have to find the values of p and q from n. This would involve factorising n, a calculation believed to be hard.

To encrypt a message $m \in \mathbb{Z}_n$, one selects a random $r \in \mathbb{Z}_n^*$, and computes the ciphertext as $g^m r^n \mod n^2$.

To decrypt a ciphertext $c \in \mathbb{Z}_{n^2}^*$, one computes $\mu((c^\lambda \mod n^2 - 1)//n) \mod n$.

Asymmetric encryption can also be used to reveal the contents of a particular transmission without revealing the decryption key. If Donald receives a message from Sarah, and wishes to prove the contents of a snippet of the message to Hillary, all he needs to do is send Hillary the plaintext. Hillary can then encrypt it herself using Donald's private key, and compare it to the ciphertext she intercepted. If they agree, then the plaintext sent by Donald is proven valid without revealing the contents of the rest of the communication.

1.2 Homomorphic encryption

Homomorphic encryption allows computation to be performed on the ciphertext, without the need for it to first be decrypted.

Consider addition of natural numbers. Let m, m' be natural numbers and let e be an encryption function that is homomorphic under addition. Then there exists a function f : ciphertext $\times$ ciphertext $\to$ ciphertext such that $f(e(m), e(m')) = e(m + m')$.

The Paillier cryptosystem is homomorphic under addition. Recall that encryption is defined as $e(x) = g^x r^n \mod n^2$ for a random $r \in \mathbb{Z}_n^*$. We define $f(c, c') = cc' \mod n^2$. Then

$$\begin{aligned} f(e(m), e(m')) &= f(g^m r^n \mod n^2, g^{m'} r^n \mod n^2) \\ &= (g^m r^n \mod n^2)(g^{m'} r^n \mod n^2) \mod n^2 \\ &= g^{m+m'} r^n \mod n^2 \\ &= e(m + m'). \end{aligned}$$

Hence, we can perform addition on Paillier ciphertexts without being able to decrypt them. Observe, however, that the result of the operation is also encrypted, and the private key used to encrypt the input is necessary to decipher it.

Homomorphic encryption remains an open area of research. Whilst today we can perform simple operations, such as addition and multiplication, on encrypted data with relative ease—provided they were encrypted with an appropriate algorithm—a technique called *fully homomorphic encryption* is in its early stages of development. Fully homomorphic encryption enables one to perform arbitrary computation on encrypted data: a data centre may one day be able to perform computation on its clients' data without having the ability to observe the contents. The first fully homomorphic encryption systems were implemented only recently. Gentry and Halevi's (2010) implementation of Gentry's (2009) scheme is an example. Current implementations of fully homomorphic encryption are remain inefficient beyond any practical use. Viable schemes are an open research problem.

1.3 Probabilistic encryption

Probabilistic encryption utilises randomness to map each plaintext to a number of different ciphertexts. It defends against brute force attacks, where the eavesdropper would use the public key to encrypt every possible plaintext and compare the resulting ciphertexts against segments of the transmission. By inflating the number of possible ciphertexts, probabilistic encryption renders such an approach infeasible.

The Paillier cryptosystem is an example of probabilistic encryption. Recall that encryption is performed by selecting a random $r \in \mathbb{Z}_n^*$, and computes the ciphertext as $g^m r^n \mod n^2$. The addition of a random r gives rise to n possible ciphertexts for each plaintext.

1.4 Coercion-resistant voting

Voter coercion, through promise of reward or threat of violence, is a hazard to any democratic system. Secret ballots are the first line of defence, preventing the coercer from easily confirming that their wishes were met.

Transparency is also important in democratic elections to ensure that they are not falsified. Many jurisdictions, Australia included, publish complete lists of votes cast at every election, enabling the public to independently verify the result.

However, this transparency may itself cause problems. In elections that employ preferential voting, there is a large number of possible permutations of candidates. A coercer may demand that the voter nominate a specific combination of preferences. They would then know, with a low probability of error, whether the voter followed their wishes. This is known as the 'Italian Attack'[4].

It thus becomes necessary to find a way to allow observers to verify election results without revealing the contents of individual ballots.

1.5 Single transferable vote

Single transferable vote is a preferential, proportional voting system that may be used to elect multiple candidates.

The procedure, up to minor variations depending by jurisdiction, is as follows:

1. A voter assigns a unique preference to each candidate and submits the ballot.
2. A quota q needed for election is computed. It is usually $q = \frac{v}{s+1}$, where v is the number of votes cast and s is the number of candidates to be elected.
3. A value of 1 is assigned to each vote cast.
4. All candidates in the election are declared to be in the running.

[4]Teague, Ramchen and Naish (2009)

5. If the number of seats is equal to or less than the number of candidates in the running, all candidates in the running are declared elected, and the count terminates.
6. The votes are tallied according to their first preferences. That is, the entire value of the vote is assigned to the candidate with the highest preference.
7. a) Any candidate who meets the quota is elected. For each ballot that lists them as the first preference, the value of the ballot is multiplied by $\frac{t-q}{t}$, where t is the total value of votes cast for that candidate. The second preference replaces the first preference. An elected candidate is no longer in the running.
   b) If no candidate meets the quota, the candidate with the lowest tally is eliminated. They are eliminated from all ballots, so the ballots that list them as the first preference are transferred to the second preference at full value. An eliminated candidate is no longer in the running.
8. The count repeats steps 5-7 until it terminates.

Single transferable vote elections are particularly vulnerable to the Italian Attack, as a voter may assign a preference to every candidate, and the number of possible permutations of preferences increases factorially with the number of candidates.

Australia uses single transferable vote for most of its elections[5]. The recent NSW senate election featured 151 candidates, or 8.63×10^{264} possible votes. Since the Australian Electoral Commission publishes all ballots cast, these elections are easily susceptible to coercion.

1.6 The Shuffle-Sum algorithm

The Shuffle-Sum algorithm, described by Benaloh et al. (2009), is a scheme for homomorphically tallying encrypted single transferable vote ballots.

The Shuffle-Sum algorithm assumes the existence of a central authority that is able to decrypt the ballots. It then provides a protocol for tallying the vote without the need to decrypt each individual ballot. Instead, homomorphic addition is used to combine the information from multiple ballots. The homomorphic sum may then be decrypted without revealing the contents of any individual ballot.

The encrypted ballots, together with a transcript of the tally, may be provided as proof that the count was conducted correctly. Such a certificate may be verified by a sceptical observer, who is able to independently reproduce all computation, including homomorphic addition; the only exception being decryption of the homomorphic sums—this can be verified by using the public key to encrypt the provided plaintext and comparing it against the ciphertext.

Benaloh et al. (2009) describe the three main steps of the Shuffle-Sum protocol:

[5]Instant-Runoff Voting, used for most lower house elections, is a special case of single transferable vote, where only one candidate is elected.

1. Each ballot is converted to a representation that eases computation of first-preference tallies. The first-preference tallies for each candidate are then homomorphically computed and decrypted.
2. If there exists a candidate who meets the quota, they are elected. The transfer value of their votes is computed using the decrypted first-preference tallies. The value of each ballot listing the elected candidate as first preference is multiplied by the transfer value.
3. If no candidate meets the quota, the candidate with the lowest tally is eliminated from all ballots. The elimination process is obfuscated, so the transcript of the count does not reveal the preference assigned to the candidate being eliminated.

Benaloh et al. (2009) provide a pen-and-paper argument of the correctness of the algorithm. However, I was unable to find a published formal specification of the Shuffle-Sum algorithm that is proven to agree with a specification of plaintext single transferable vote counting. The need to develop techniques for verifying such algorithms thus became apparent.

Chapter 2

Related works

While little work has recently been published on the Shuffle-Sum algorithm and homomorphic counting of single transferable voting, much has been published on the use of homomorphic encryption in elections more broadly.

Henderson, Torija and Noakes (2016) describe a system that allows every voter to verify that their vote has been counted correctly and that every entitled voter has cast their ballot. They do so while maintaining the secrecy of each voter's choice.

They define a new encryption scheme, based on the difficulty of the Learning with Errors problem, that is homomorphic under addition and multiplication. It allows errors to be introduced into ballots, such that they are cancelled out when they are combined, hence enhancing secrecy.

Cerveró et al. (2015) describe a homomorphic tallying system for electronic voting that makes use of elliptic curve cryptography. Similarly to this work, they produce a proof that the result of the election is correct.

Chapter 3

Methodology

To lay the groundwork for a formally verified specification of the Shuffle-Sum protocol, we provide a specification of homomorphic first-past-the-post vote counting, and formally verify it against plaintext first-past-the-post counting. This is performed in the proof assistant Coq.

In brief, we define plaintext first-past-the-post counting as follows:

1. Given a list of candidates and a list of ballots, we verify that all candidates are unique and that all ballots are valid. A ballot is valid if there exists exactly one candidate with a vote value of 1, and all other candidates have a vote value of 0.
2. We tally the votes: for every candidate the tally is the sum over the ballots of their vote values.
3. We declare the winners: a candidate is a winner iff no other candidate has more votes than them.

Encrypted first-past-the-post counting is similar:

1. Given a list of candidates and a list of ballots, we verify that all candidates are unique and that all ballots are valid. A ballot is valid if there exists exactly one candidate with a vote value of 1, and all other candidates have a vote value of 0.[1]
2. We tally the votes: for every candidate the tally is the homomorphic sum over the ballots of their vote values.
3. We decrypt the tally and declare the winners: a candidate is a winner iff no other candidate has more votes than them.

3.1 Limitations

The algorithm described assumes that we trust our list of encrypted ballots. This is not always true, as a central authority that provides the ballots could spoof them to

[1] In the specification, we decrypt every ballot to check its validity. Later, we describe a method of checking the validity of a ballot without revealing its contents, and prove that they are equivalent.

manipulate an election.

3.2 Plaintext first-past-the-post specification

We first define a new data type for a candidate. This is left as a variable, so it can be dependent on the implementation.

```
Variable cand : Type.
```

Plaintext ballots and plaintext tallies are maps from a candidate to a natural number.

```
Definition PT_Ballot := cand -> nat.
Definition PT_Tally := cand -> nat.
```

We define a data structure that allows us to retain information between the three stages of the count.

```
Inductive PT_Node :=
   PT_Checked:                PT_Node   (* Ballot checked state.             *)
 | PT_Tallied: PT_Tally    -> PT_Node   (* Ballot tallied. Stores tally      *)
 | PT_Winners: (list cand) -> PT_Node.  (* Winners declared. Stores winners. *)
```

We also define what it means for a ballot to be valid. A ballot is valid if it assigns a vote value of 1 to exactly one candidate in the running and 0 to all other candidates in the running.

```
Definition PT_ballot_valid (cnds : list cand) (b : PT_Ballot) : Prop :=
  exists (c : cand), In c cnds
                   /\ b c = 1
                   /\ (forall c' : cand, In c' cnds -> c <> c' -> b c' = 0).
```

We can now define the three stages of the count. The first ensures that all candidates are unique[2], and that all ballots are valid. If the input data is incorrect, then the count obviously cannot be certified.

```
Inductive PT_pf (cnds : list cand) (blts : list PT_Ballot) : PT_Node -> Prop :=
    PT_pf_ax :
          pd cnds
          (* ^ All candidates are unique. *)
       -> fold_right and True (map (PT_ballot_valid cnds) blts)
          (* ^ The ballots are valid. *)
       -> PT_pf cnds blts PT_Checked
```

[2] Accomplished by the pd proposition. Omitted here for brevity.

Observe that we take a list of candidates as input, as opposed to operating on the assumption that every object of type `cand` is in the running. This allows one to use any reasonable type, such as a string, for to represent a candidate, easing implementation. Obviously not every possible string is in the running, so we must allow the user to specify the candidates.

We then tally the votes.[3]

```
| PT_pf_tl : forall tlly,
    PT_pf cnds blts PT_Checked
    -> (forall c : cand,
        In c cnds -> tlly c = plus_list (apply_l c blts))
       (* ^ For each candidate, the tally is the sum of all the ballot *)
       (*   fields corresponding to that candidate.                    *)
    -> PT_pf cnds blts (PT_Tallied tlly)
```

Observe that we require that `PT_pf cnds blts PT_Checked` to certify this stage. This implies that the candidates and ballots have already been checked for validity. Hence, each ballot is a vote for exactly one candidate. Our tally may simply be the function sum of all ballots. Note that we ignore candidates not in the running—this makes for a weaker condition.

Finally, we declare the winners.

```
| PT_pf_wn : forall tlly wnrs,
      PT_pf cnds blts (PT_Tallied tlly)
    -> (forall c : cand, In c wnrs <-> (In c cnds
                  /\ forall c' : cand, (In c' cnds -> tlly c' <= tlly c)))
       (* ^ A candidate is a winner iff nobody got more votes than them. *)
    -> PT_pf cnds blts (PT_Winners wnrs).
```

The winners are a subset of the candidates. They are exactly the candidates who received the highest number of votes. If there exists one clear winner, there will only be one candidate in the list of winners. If there is a tie, the winners will contain multiple candidates.

3.3 Ciphertext first-past-the-post specification

In addition to the candidate, as defined previously, we define the data types for the ciphertext and its salt. The salt is for use in probabilistic encryption: it is the random value that enables one plaintext to be mapped to multiple ciphertexts.

```
Variable ciphertext : Type.
Variable salt : Type.
```

[3]The `apply_l` function is similar to `map`, but instead of applying a function to a list of items, it applies an item to a list of functions.

We also define the encrypt and decrypt functions, together with their relationship. Observe that if we omitted the salt, then every plaintext would be mapped to exactly one ciphertext. This would not be a valid assumption.

```
Variable encrypt : (nat * salt) -> ciphertext.
Variable decrypt : ciphertext -> (nat * salt).

Hypothesis decrypt_inverse : forall ns : nat * salt,
                                    decrypt (encrypt ns) = ns.
```

We define a homomorphic addition function.

```
Variable h_plus : ciphertext -> ciphertext -> ciphertext.
Hypothesis h_plus_homomorphism : forall (c c' : ciphertext),
      fst (decrypt (h_plus c c')) = fst (decrypt c) + fst (decrypt c').
```

Similarly to before, we define ballots, tallies, and a data structure to keep track of information between stages of counting; and we define a valid ballot.

Observe that to validate a ballot, we decrypt every field. This makes for a more intuitive specification. We later describe a validation method that does not reveal the contents of the ballot, and we prove that they are equivalent.

```
Definition CT_Ballot := cand -> ciphertext.  (* Ciphertext ballot. *)
Definition CT_Tally := cand -> ciphertext.   (* Ciphertext tally.  *)

Inductive CT_Node :=
   CT_Checked:                CT_Node   (* Ballot checked state.             *)
 | CT_Tallied: CT_Tally    -> CT_Node   (* Ballot tallied. Stores tally      *)
 | CT_Winners: (list cand) -> CT_Node.  (* Winners declared. Stores winners. *)

Definition CT_ballot_valid (cnds : list cand) (b : CT_Ballot) : Prop :=
  exists (c : cand), In c cnds
                   /\ fst (decrypt (b c)) = 1
                   /\ (forall c' : cand,
                       In c' cnds -> c <> c' -> fst (decrypt (b c')) = 0).
```

We define the three stages of the count, to match the three plaintext stages. This makes it easier to show that the two specifications are equivalent. The first stage checks that all candidates are unique and that all ballots are valid.

```
Inductive CT_pf (cnds : list cand) (blts : list CT_Ballot) : CT_Node -> Prop :=
    CT_pf_ax :
          pd cnds
          (* ^ All candidates are unique. *)
       -> fold_right and True (map (CT_ballot_valid cnds) blts)
          (* ^ The ballots are valid. *)
       -> CT_pf cnds blts (CT_Checked)
```

We then tally the votes. Observe that we do not require the tally to be the full homomorphic sum of the ballots. Rather, we require that the tally and the homomorphic sum have matching plaintexts—matching salts are not necessary, and ignoring them weakens the condition. However, if we set the tally to be the homomorphic sum of the ballots, this condition is satisfied, so in some implementations it is possible to verify it without performing decryption.

```
| CT_pf_tl : forall tlly,
    CT_pf cnds blts CT_Checked
    -> (forall c : cand, In c cnds -> fst (decrypt (tlly c))
                            = fst (decrypt (h_plus_list (apply_l c blts))))
       (* ^ For each candidate, the tally is the homomorphic sum of all *)
       (*   the ballot fields corresponding to that candidate.          *)
    -> CT_pf cnds blts (CT_Tallied tlly)
```

Finally, we declare the winners. It is at this stage that we must decrypt the tally.

```
| CT_pf_wn : forall tlly wnrs,
      CT_pf cnds blts (CT_Tallied tlly)
    -> (forall c : cand, In c wnrs <-> (In c cnds /\ forall c' : cand,
         (In c' cnds -> fst (decrypt (tlly c')) <= fst (decrypt (tlly c)))))
       (* ^ A candidate is a winner iff nobody got more votes than them. *)
    -> CT_pf cnds blts (CT_Winners wnrs).
```

3.4 Proof of equivalence

The theorem of equivalence was defined as follows:

```
Theorem pf_equiv
  : forall (cnds : list cand) (blts : list CT_Ballot) (node : CT_Node),
    CT_pf cnds blts node
    <-> PT_pf cnds (decrypt_ballots blts) (decrypt_node node).
```

The `decrypt_ballots` function decrypts a `list CT_Ballot` to produce a `list PT_Ballot`. Similarly, `decrypt_node` decrypts a `CT_Node` and returns a `PT_Node`.

Hence, the theorem states that `CT_pf` and `PT_pf` are equivalent up to encryption or decryption.

It is clear that our specification of plaintext first-past-the-post voting is more intuitive than the ciphertext specification. By proving equivalence between them, we showed that the ciphertext specification meets the intuitive requirements.

In addition, most election laws assume plaintext ballots. This theorem shows that the ciphertext specification is compliant with those laws.

To make the proof easier, the theorem was proven by matching corresponding steps of the specifications. Due to the inductive nature of the specifications, induction was used in the proof.

3.5 Independent verification

We show that it is possible produce a certificate of correctness of a count that an independent observer can verify without having the ability to decrypt the ballots.

Consider the following theorem:

```
Theorem pf_observer_check
  : forall (cnds   : list cand)
           (blts   : list CT_Ballot)
           (wnrs   : list cand)
           (b_sum  : CT_Ballot -> (nat * salt))
           (f_tlly : cand      -> (nat * salt)),
  pd_f cnds
  -> (forall b : CT_Ballot, In b blts
         -> fst (b_sum b) = 1 /\ encrypt (b_sum b) = h_plus_list (map b cnds))
  -> (forall c : cand,
      In c cnds -> encrypt (f_tlly c) = h_plus_list (apply_l c blts))
  -> (forall c : cand, In c wnrs <-> In c cnds
        /\ (forall c' : cand, In c' cnds -> fst (f_tlly c') <= fst (f_tlly c)))
  -> CT_pf cnds blts (CT_Winners wnrs).
```

Given a list of candidates, a list of encrypted ballots, and a list of winners we wish to verify that the winners were correctly determined from the candidates and the ballots. To assist us, we are also given: for each ballot, a decrypted homomorphic sum of its fields; and for each candidate, a decryption of their final tally.

The theorem states that the certificate is correct if it meets the following conditions that are easily computable by an independent observer[4]:

- All candidates are unique. To show that this is computable, we define a function `pd_f` that we prove to be equivalent to the proposition `pd`.
- For each ballot, the decrypted homomorphic sum of its fields (given) is 1. This is to ensure that the ballot is valid—the sum being 1 is equivalent to `CT_ballot_valid`. In addition, our encryption of the decrypted sum must match our own homomorphic addition. This is to prevent the decrypted sum being spoofed.

  Observe that this gives us a way to prove the validity of each ballot without revealing the contents of that ballot. All we must provide is the sum of all the fields in the ballot—it is valid iff the sum is 1. It is not possible to determine the vote from that sum.
- Our own encryption of the decrypted tally matches our own homomorphic sum of all the ballots. This is to prevent the decrypted tally being spoofed.
- The list of winners consists exactly of every candidate who received the highest number of votes according to the decrypted tally provided.

Since the theorem was proven, we have that the above conditions are sufficient to declare a certificate correct.

[4]Recall that encryption is easily performed by the observer as they have access to the public encryption key, but decryption is not easily performed, as they do not have the private decryption key.

Chapter 4

Proving Shuffle-Sum

The specification of homomorphic counting of first-past-the-post votes, the proof of its equivalence with plaintext counting, and the proof of the validity of an easily produced certificate lay the groundwork for proving the Shuffle-Sum protocol.

A formal proof of the compliance of Shuffle-Sum with traditional STV voting could be substantially similar, albeit significantly more complicated.

The specification, along with the corresponding plaintext specification, could be expressed in stages, to ease proving. The proof would then be inductive, matching each stage with its counterpart.

We could also define a certificate for the validity of a Shuffle-Sum count that does not reveal the votes, and formally prove its sufficiency.

Chapter 5

References

- Benaloh J., Moran T., Naish L., Ramchen K., and Teague V. 2009. 'Shuffle-Sum: Coercion-Resistant Verifiable Tallying for STV Voting'. *IEEE Transactions on Information Forensics and Security*, vol. 4, no. 4.
- Cevreró M. et al. 2015. 'Elliptic Curve Array Ballots for Homomorphic Tallying Elections'. *Lecture Notes in Computer Science*, vol. 9265, pp 334-347.
- Gentry C. and Halevi S. 2010. *Implementing Gentry's Fully-Homomorphic Encryption Scheme*. ⟨`http://eprint.iacr.org/2010/520`⟩
- Henderson T., Torija F., and Noakes A. 2016. *Distributed Homomorphic Voting*. ⟨`https://courses.csail.mit.edu/6.857/2016/files/30.pdf`⟩
- Katz J. and Lindell Y. 2007. *Introduction to Modern Cryptography*, 1st ed. pp. 385-393. CRC Press. ISBN 978-1584885511.
- Teague V., Ramchen K., Naish L. 2008. *Coercion-Resistant tallying for STV voting*. ⟨`http://static.usenix.org/events/evt08/tech/full_papers/teague/teague_html`⟩