Machine-checking the universal verifiability of ElectionGuard

Thomas Haines[1], Rageev Goré[2], and Jack Stodart[2]

[1] Norwegian University of Science and Technology, Trondheim, Norway
{firstname.lastname}@ntnu.no
[2] The Australian National University, Canberra, Australia
{firstname.lastname}@anu.edu.au

Abstract. ElectionGuard is an open source set of software components and specifications from Microsoft designed to allow the modification of a number of different e-voting protocols and products to produce public evidence (transcripts) which anyone can verify. The software uses ElGamal, homomorphic tallying and sigma protocols to enable public scrutiny without adversely affecting privacy. Some components have been formally verified (machine-checked) to be free of certain software bugs but there was no formal verification of their cryptographic security.

Here, we present a machine-checked proof of the verifiability guarantees of the transcripts produced according to the ElectionGuard specification. We have also extracted an executable version of the verifier specification, which we proved to be secure, and used it to verify election transcripts produced by ElectionGuard. Our results show that our implementation is of similar efficiency to existing implementations.

Keywords: verifiable e-voting · interactive theorem provers · code extraction.

1 Introduction

Electronic voting has been in use for at least the last fifty years; however, the nature of elections makes it very hard to verify whether the electronic components are behaving as they should. Current best practice is to ensure that each software component creates publicly verifiable evidence that its output is correct with respect to certain criteria. A cascade of such processes then guarantees that the whole process is end-to-end-verifiable [1]. Such systems invariably require using increasingly complicated cryptographic techniques.

A particular group of techniques for adding verifiability to electronic voting is homomorphic tallying in which each vote is encrypted under a homomorphic encryption scheme to produce a ciphertext with all ciphertexts publicly tallied (without decrypting them) to produce an encryption of the tally. The tally ciphertext is then decrypted by the authorities. Zero-knowledge proofs can be used to prove publicly that the encrypted ballots are well-formed and that the tally was decrypted correctly. This does not suffice for overall verifiability since we do

not know whether the collected ballots contain the votes intended by the eligible voters but is rather evidence that the collected ballots are counted correctly. There are various ways to extend homomorphic tallying to have end-to-end verifiability, so that the counted ballots are guaranteed to be the intended ballots of the eligible voters, which we omit for brevity. The most famous deployed e-voting scheme using homomorphic tallying is surely the online voting system Helios [2], which is used by the International Association for Cryptologic Research.

ElectionGuard is a set of open-source software components and specifications released by Microsoft in 2019 [3]. It is designed to quickly allow ballot-collecting devices (such as ballot-marking devices and optical scanners) to work with trustees (so called because they are trusted to maintain privacy) to produce public evidence. Specifically, to allow such devices to produce evidence that the encrypted ballots are well-formed, that the ballots were correctly tallied, and that the announced result was correct with respect to the tally. All references to ElectionGuard in this document are to version 0.85.[3]

An election in the context considered by ElectionGuard is a protocol involving Election Officials, Trustees, Voters, and Interested Citizens. The Trustees are responsible for generating the required cryptographic keys and then decrypting the encrypted tally at the end of the election. The Election Officials have numerous responsibilities including providing the Ballot Marking Devices, Electronic Ballot Boxs, and Electronic Poll Books.

Prior to election day, the ballot style is determined by the Election Officials and the Trustees generate their cryptographic keys (including what threshold is required to decrypt); for brevity we elide much of the other preparation and refer the reader to the ElectionGuard documentation.[4] In the booth, the Voter selects her candidates using the Ballot Marking Device. The Ballot Marking Device then creates both a paper ballot and an electronic Cast Vote Record (CVR). It assigns the ballot a unique ID number, encrypts the CVR and constructs a non-interactive zero-knowledge proof that the ballot is well-formed. In addition to the paper ballot, the voter is also provided with a tracker which contains a human-comparable hash of the encrypted ballot. The voter is then given the option to cast or spoil the ballot. If the ballot is cast, the paper ballot is added to the ballot box and the CVR is added to the Electronic Ballot Box. If the ballot is spoiled the Ballot Marking Device must prove to the voter that it correctly encrypted the voter's selection (and the voter must create another ballot).

After the election, the encrypted ballots are homomorphically aggregated; the paper ballots are also tallied. Prior to tallying, the zero-knowledge proofs should be checked by trustees to ensure that we tally only well-formed ballots. The Trustees then decrypt the aggregated ciphertexts containing the election result. Voters need not check the result, however a diligent voter should check that any ballots (cast or spoiled) which match the trackers they posses appear

[3] https://github.com/microsoft/electionguard/wiki/Informal/ElectionGuardSpecificationV0.85.pdf

[4] For simplicity, we describe a fairly narrow use case of ElectionGuard.

correctly on the bulletin board. Any observer can check the well-formedness of voter encryptions and correct tallying by running a computer program called a verifier over the published transcripts, and ElectionGuard includes a reference verifier. The goal of this process is to give the voters high confidence that their ballot was recorded correctly and that the collected ballots are correctly tallied. The security guarantees from voter-initiated checks ensure individual verifiability. The checks performed by any interested party ensure universal verifiability.

The checks that the voter needs to make are simple. But the checks that the verifier software must make are considerably more complicated. Indeed, ElectionGuard is useful only if we can be sure that a verifier that correctly implements the ElectionGuard verifier specification will indeed give the security guarantees claimed by ElectionGuard. As we point out next, this critical relationship between the specification and implementation is easy to break.

1.1 Implementation issues in E-Voting Systems

The theoretical foundations of verifiable electronic voting has matured greatly. Simple schemes, such as ElectionGuard, using homomorphic tallying are well-known and theoretically easy to construct. Nevertheless, we are now seeing small but critical bugs in the implementation of such schemes. For instance, the Swiss Post system, while not itself using homomorphic tallying, contained many of the same components, many of which were broken despite extensive review [4]. This is the tip of the proverbial iceberg in terms of failures and issues in allegedly end-to-end verifiable systems; other examples have included the iVote system deployed in the Australian state of New South Wales [5], and the e-voting system used in national elections in Estonia [6]. In addition, many general issues have been discovered [7–9] which need to be carefully avoided in any implementation. Most of these issues were present in the Helios e-voting system [2]. Thus even simple systems are prone to critical software errors.

General software development aims at increasing security through a process of best practice which is not specific to the particular goal of that software. This kind of development avoids many kinds of errors, including but not limited to, division by zero, off-by-one errors, syntax errors, and resource errors. Various organisations offer services for checking that software is developed according to these standards and indeed this is commonly done for e-voting software. It is interesting, then, that the bugs mentioned previously occurred even though the software, in many cases, was certified according to these best practice standards.

The reason that the bugs slipped through is due to their nature. These bugs are not standard programming bugs which might be caught by standard best practice techniques. Rather, the code does not correctly capture the logical flow of the protocol, as required by the cryptographic primitives. Compounding these issues is that many of the bugs were present in the specification as well as the code. So, at present, the problem of securely deploying electronic voting does not appear to be primarily about improving theory or requiring more secure programming; rather, it appears to be improving our ability to check that the specification and implementation contain the logical flow they should.

A key observation here is that while end-to-end verifiability protects against bugs in the software running the elections, it transfers the correctness requirements to the software that checks (verifies) the produced evidence. This is an excellent trade since the software required to check the evidence is much simpler and multiple independent verifiers can be developed. However, the independence of the verifiers is normally only skin deep since they are implementations of a common specification which may itself be incorrect. (The common practice to have under-graduate computer science students implement the independent verifiers is unlikely to result in insightful detection of errors in the cryptographic specification.) Our work here can be viewed as a formal proof that the specification is cryptographically secure and that our extracted verifier tests that our encoding of the specification is compatible (interoperable) with the existing implementations, as well as being another independent verifier itself.

1.2 Contribution

We have formally verified the (universal) verifiability of the ElectionGuard specification: that is, we have encoded the specification in Coq and proved its cryptographic soundness. Specifically, we prove special soundness (that if any adversary is able to produce multiple accepting transcripts then the collected ballots must be counted correctly) which is known to imply soundness [14].

We extend previous work on formally verifying the verifier of Helios by:

- Creating a richer type system to allow the ballots to be encoded into Coq;
- Defining a stand-alone verifier for these ballots and the associated proofs;
- Proving this verifier to be correct; that is to have correctness, special soundness, and honest-verifier zero-knowledge.

This compares to prior work [13] where the various components of the verifier were defined and proven secure in Coq but then composed in the extracted version. The fact that the components should compose correctly is trivial (it amounts to saying that the logical conjunction of n statements is true if all the statements are true) but defining it formally in Coq results in complicated types.

These contributions allow us to extract our formally verified verifier into an executable verifier which is comparable in efficiency to existing implementations. We used this verifier to verify transcripts produced by ElectionGuard. We have not proved any privacy properties of the ElectionGuard system but we have proved the honest-verifier zero-knowledge of the verifier in Coq.

1.3 Interactive theorem provers

Interactive theorem provers are pieces of software that check that mathematical "proofs" are correct. A human encodes the mathematical theorem and (purported) proof within the language of the interactive theorem prover and the interactive theorem prover checks the proof using a given finite collection of proof-rules. Trust rests upon three pillars: first, the code base for interactive

theorem provers is usually very small and has been scrutinised by many experts, typically over decades; second, most interactive theorem provers produce a machine-readable proof of the claimed theorem and these can be checked either by hand or by a different interactive theorem prover; third, interactive theorem provers typically enjoy extremely rigorous mathematical foundations, which have withstood decades of peer review. Many interactive theorem provers transliterate (extract) correct proofs into ML, Haskell, Scheme or OCaml programs.

The main impediment to using interactive theorem proving and code extraction is the rather steep learning curve involving exotic mathematical logic(s) and the associated proof-rules. Consequently, interactive theorem provers mostly remained in an academic setting [10, 11], and were rarely considered for real life software-engineering. Recent debacles, such as heartbleed[5], have led companies and researchers to focus on avoiding bugs by using formal verification, to the point where it is now gaining momentum in mainstream development.

In this work, we used the interactive theorem prover Coq [12] to: encode specifications; verify (machine-check) that (functional) programs are correct with respect to these encoded specifications; and extract the code corresponding to the verified functional programs.

1.4 Verification and Code Extraction Via Coq

We now explain how to use the interactive theorem prover called Coq [12] to: encode specifications; encode functional programs; and to verify them correct against these encoded specifications to finally extract corresponding code.

Below, we exemplify one way to produce verified programs via Coq using addition of two natural numbers. As in the sequel, we first give a natural language definition as might be found in a mathematics text, then its encoding into Coq, followed by an explanation of the encoding. Doing so is important as it helps to ensure that the encoding really does do the job we intend it to do.

Definition 1. *The set mynat is the smallest set formed from the clauses:*

1. *the term O is in mynat;*
2. *if the term n is in mynat then so is the term S n;*
3. *nothing else is in mynat.*

```
Inductive mynat : Set :=
  | O : mynat              (* O is a mynat *)
  | S : mynat -> mynat.    (* S of a mynat is a mynat *)
```

Here, the first line encodes that *mynat* is of type *Set* and the vertical bar separates the two subclauses of the encoding. The terms O and S are known as constructors and anything in between "(*" and "*)" are comments. The first subclause illustrates that the colon can also be read as set membership $\in$ while the second clause illustrates that the constructor S is actually a function that

[5] http://heartbleed.com/

accepts a member from *mynat* and constructs another member of *mynat* by prefixing the given member with S. Thus the explicit mention of n in the natural language definition is elided. Clause (3) of the natural language definition is encoded by the declaration `Inductive`. Intuitively, the natural numbers are the terms $O, (S\ O), (S\ (S\ O)), \cdots$ corresponding to $0, 1, 2, \cdots$.

Definition 2 (Specification of addition). *Adding O to any natural number m gives m, and for all natural numbers n, m, and r, if adding n to m gives r then adding $(S\ n)$ to m gives $(S\ r)$.*

```
Inductive add: mynat -> mynat -> mynat -> Prop :=
 | addO: forall m, (add O m m)
 | addS: forall n m r, add n m r -> add (S n) m (S r).
```

Here, the notation $mynat \to mynat \to mynat \to Prop$ encodes that *add* is ternary and that it is a "Proposition" which returns either true or else false, but in intuitionistic logic rather than classical logic. Our specification of addition is encoded as a ternary predicate *add n m r* that is true iff "adding n to m gives r", based purely on the only two ways in which we can construct the first argument: either it is O, or it is of the form $(S\ \cdot)$. The "extraction" facilities of Coq allow us to produce actual code in OCaml, Haskell, or Scheme. The encoding below is our hand-crafted function *myplus* in which the "where" keyword allows an infix symbol + for *myplus* and $\Rightarrow$ (not $\to$) indicates the return value of the function:

```
Fixpoint myplus (n m: mynat) : mynat :=
    match n with
    | O   => m
    | S p => S (p + m)
    end
    where "p + m" := (myplus p m).
```

Our function is correct if it implies the specification below.

Theorem 1. *For all natural numbers n, m, r, if $r = myplus\ n\ m$ then add $n\ m\ r$ is true.*

```
Theorem myplus_correct:
  forall n m r: mynat, (r = myplus n m) -> (add n m r).
Proof.
   induction n. intro m. intro r. intro H. simpl in H.
   subst r. apply addO. intros m r H. rewrite H.
   simpl myplus1. apply addS. apply IHn. reflexivity.
Qed.
```

The text shown between the words `Proof` and `Qed` consists of commands typed in by the user to guide Coq to the proof of the theorem. That is, the user interacts with Coq to obtain the proof, with Coq checking each step to ensure that it is acceptable. The Coq extraction mechanism turns our function "myplus" into Ocaml, Haskell or Scheme code giving us a program which is provably correct with respect to our specification of addition.

We can also reason about our specification itself inside Coq. For example, the theorem below encodes that our definition of addition is commutative:

Theorem 2. *For all natural numbers n, m, r, if add n m r then add m n r*

```
Theorem add_commutative:
  forall n m r: mynat, (add n m r) -> (add m n r).
Proof. ... Qed.
```

In the sequel, we give all of our theorems in both plain text and in Coq to enable the reader to confirm that our encodings do indeed capture our intentions.

1.5 Protecting against flaws in code and specifications

Haines *et al.* [13] suggested combining techniques for verifiable e-voting and formal verification of software. The idea is that the key component, at least for integrity, in a verifiable e-voting system is not the e-voting software but the verifier that checks the public evidence produced by the e-voting software; if the verifier is correct (and used) then the properties it guarantees will hold regardless of any bugs present in the e-voting software. This is useful because the verifier is a far simpler and more self contained than the e-voting software. Rivest [1] called this "Software independence" but the term is perhaps slightly misleading because there is still a fundamental reliance on the software that implements a correct verifier.

If the verifier is the key entity to verify (machine-check), the next logical question is what properties of the verifier need to be checked? Specifically, Haines *et al.* argued that the logical properties of the verifier are what need to be checked. In the context of e-voting systems built largely upon zero-knowledge proofs, the key property of the verifier is soundness. That is, the verifier should not accept the transcript unless the statement is true, at least with overwhelming probability. Collectively, this means that the integrity of a deployed e-voting scheme can be reduced to the strong guarantees of correctness provided by interactive theorem provers rather than the new and understudied e-voting scheme.

Haines *et al.* demonstrated the feasibility of this approach by creating several machine-checked sub-verifiers for the Helios e-voting system which collectively sufficed for universal verifiability. They achieved this by providing the logical machinery to easily prove secure the sigma protocols used in e-voting; we reuse this machinery in our work. The similarities in ElectionGuard and Helios mean many of the underlying components (sub-verifiers) are similar but in our work we take care of the various differences and extend Haines *et al.*'s work to prove the completed verifier secure rather than the sub-verifiers.

1.6 Residual trust assumptions

The residual trust assumptions differ between the different aspects of our contribution. In general, the work in Coq has fairly low trust assumptions whereas the extracted verifier has higher trust assumptions.

The correctness of the work in Coq depends on the correctness of Coq but also that we have correctly defined verifiability and ElectionGuard. The definition of verifiability takes the well established form of special soundness. The soundness

of the ElectionGuard definition is resolved by proving that it satisfies verifiability; the compatibility of the definition is demonstrated by showing that it can handle real ElectionGuard transcripts.

The extracted verifier incurs several additional trust assumptions and for this reason we suggest that our extracted verifier should be one of many. First, the extraction facility in Coq has not been formally verified and this could introduce errors. In addition, we replace some of the inefficient Coq arithmetic functions with significantly faster native OCaml functions. Finally, since deployed sigma protocols are made non-interactive via the Fiat-Shamir transform, this transform also needs to be checked for correctness to ensure the deployed elections are secure.This is not challenging to do manually despite the prevalence of careless implementations. It would be nice to prove the correctness of the Fiat-Shamir transform inside an interactive theorem prover but unfortunately this would involve rewinding random oracles which is not currently supported in any prover known to the authors.

2 Machine checking the verifiability of ElectionGuard

In this section we will introduce our Coq specification which encapsulates the relevant parts of ElectionGuard. We will aim to provide sufficient detail to give an overview of what we did without completely overwhelming readers who are unfamiliar with Coq. It is important to provide such details so that readers can check that our Coq encodings do actually capture what we claim we capture. For conciseness we will not provide details of the sigma protocols and interested readers may consult [13].

2.1 ElectionGuard Elections

An election in the context of ElectionGuard consists of a fixed number of contests with one or more candidates in each contest. This style of voting varies between plurality voting and approval voting depending on the number of candidates which are allowed for selection. We assume for simplicity that each voter is allowed to select exactly one candidate in each contest: this is easy to change but doing so unduly complicates the presentation. We use `numContests` as the number of contests in any given election.

ElectionGuard uses ElGamal in Schnorr groups. We abstract our verifier over any group `G` of prime order since the exact group does not matterfor the security reduction. For a given group `G`, the ElGamal ciphertext space is the product group `DG.G` of `G` and `G`: a product group is the group-theoretic analogue of the Cartesian product where all operations are taken component-wise. In the Coq examples that follow `G` and `DG.G` will refer to the sets that underly these groups.

Running example: here we give a running example to show how we encode ballots into Coq using the digit 1 to signify "preferred candidate" and using 0 to signify "unpreferred candidate". Suppose we have an election with three contests with four candidates in the first contest, three candidates in the second contest,

and two candidates in the third contest. To vote for a candidate in the first contest, a voter has to create a list of natural numbers of length four containing exactly one 1 with the others all 0. The list entries are then mapped into the group `G` (which is the message space of the encryption scheme) before being encrypted to give ciphertext members of `DG.G` so we use E(1) and E(0) to stand for "encryption of 1" and "encryption of 0" respectively. For example, the vector [E(1),E(0),E(0),E(0)] of length four is a vote for the first candidate out of the four candidates in contest 1 where E(1) ∈ `DG.G` and E(0) ∈ `DG.G` . Suppose that the election has two cast ballots with the first ballot cast for candidates 1, 2, and 1 in the three respective contests and the second ballot cast for candidates 2, 1, 2. respectively, as shown below:

Contests	1	2	3
Ballot 1	[E(1),E(0),E(0),E(0)]	[E(0),E(1),E(0)]	[E(1),E(0)]
Ballot 2	[E(0),E(1),E(0),E(0)]	[E(1),E(0),E(0]	[E(0),E(1)]

We now describe how we encoded such ballots into Coq using vectors and product types.

2.2 Vector and Product Types

We assume that the reader is not an expert in Coq and therefore explain how we encoded ballots into Coq in some detail. There is nothing particularly original in our encoding but it may appear complicated to a naive reader.

We encode most of our information in vectors which are defined in Coq via the command `vector type length` where `type` is the the type of the elements of the vector and `length` is the length of the vector: thus `vector int 3` encodes that the vector contains integers and is of length three. To maintain generality, the declaration `vector nat numContests` tells Coq that each vector is of length `numContests` and contains natural numbers `nat`. Our running example of an election with three contests with four, three, and two candidates, respectively, would be a vector called `numSel` = [4,3,2] of type `vector nat 3`. The functions `Vhead` and `VTail` are provided by Coq to allow us to split a vector (a list) into its components, so `Vhead [4,3,2]` would return `4` and `Vtail [4,3,2]` would return `[3,2]`.

In Coq, if `A` and `B` are two arbitrary types, then the type `prod A B` contains all pairs (a, b) such that a is of type `A` and b is of type `B`. If `A` and `B` are of type `Set` then so is `prod A B`.

An (encrypted) ballot, such as Ballot 1 above, is an ordered tuple where each member of the tuple is itself a tuple of ciphertexts. We define it in Coq as shown below. The type of a ballot is a nested product of vectors of ciphertexts where the depth of the product is the number of contests and at each layer of the product it contains a vector of ciphertexts of length equal to the number of candidates in that contest. For example, continuing our previous example above, a ballot would be of type `vector DG.G 4 * vector DG.G 3 * vector DG.G 2`. It is easy to see that the ballots in our example, $([E(1), E(0), E(0), E(0)], [E(0), E(1), E(0)], [E(1), E(0)]$ and $([E(0), E(1), E(0), E(0)], [E(1), E(0), E(0], [E(0), E(1)])$, are of this type.

The type `ballot` is defined in Coq as a set of functions which accept a vector of natural numbers of length `numContests`. By making `ballot` depend upon the argument `numContests`, we tell Coq that the type of the vector input depends on `numContests`. The function also specifics that the type it outputs will be a set. The function `fun ...` consists of two clauses depending upon the value of the natural number `numContests`:

Base case: the first is for the base case when the number of contests is zero in which case we simply return the empty set.

Inductive case: the second is for when the number of contests is non-zero when we recursively define the result as the Cartesian product of two sets whose types are, respectively, `vector DG.G (Vhead v)` and `ballot (Vtail v)`. The first set has type `vector DG.G (Vhead v)` where `(Vhead v)` is the number of candidates in the first contest. The second set has type `ballot (Vtail v)` as returned by `ballot` on all remaining contests.

```
Fixpoint ballot (numContests : nat) :
    vector nat numContests -> Set :=
  match numContests with
  | 0%nat => fun _ => Empty_set
  | _     => fun v => prod (vector DG.G (Vhead v)) (ballot (Vtail v))
  end.
```

A ballot on its own may be ill formed and contain a large number of votes for each candidate. ElectionGuard, therefore, requires that each ballot come with a cryptographic proof that it is well formed. We define the type of the ballot proof as shown below.

```
Fixpoint ballotProof (numContests : nat) :
    vector nat numContests -> Type :=
  match numContests with
    | 0%nat => fun _ => Empty_set
    | _  => fun v1 => prod (ProofTranscript (OneOfNSigma (Vhead v1)))
                           (ballotProof (Vtail v1))
  end.
```

In essence, it is a nested tuple where each element in the tuple corresponds to the cryptographic proof, in the form of a sigma-protocol transcript, that the corresponding ciphertexts are all encryptions of zero or else one and that the summation of the ciphertexts is equal to one. That is, in this context, the votes for each candidate are either yes or no and exactly one candidate has a yes vote.

The sigma protocol transcript type is returned by the function `ProofTranscript` which takes a sigma protocol and returns the type of its transcripts. In this case, the sigma protocol is `OneOfNSigma` which takes a natural number n and returns a sigma protocol to check that n chiphertexts are all encryptions of one or else zero and the product of the ciphertexts is one.

Once all the ballots are homomorphically combined, the authorities decrypt the summation of all ballots. They do this by each using their share of the secret key to produce decryption factors. These decryption factors can then be publicly combined by anyone to decrypt the ciphertext. We define the type of the decryption factors as shown below.

Algorithm 1: Election Verifier

Data: numTrustees numCast numContests
Data: vector containing the number of selections in each contest numSel
Data: group generator g, public key shares pks
Data: castBallots ballotProofs decFactors decProofs
Result: *valid*
valid := true;
acc := Encryption of zero;
for *(ballot,proof)* ∈ *(castBallots,ballotProofs)* **do**
  if *proof of correct encryption for ballot are invalid* **then**
    valid := false;
  end
  acc := *acc* × ballot
end
if *If decProofs are invalid for decFactors with respect to acc* **then**
  valid := false;
end

Fig. 1. Algorithm of the Verifier

```
Fixpoint decryptionFactors (numContests numTrustees : nat) :
  vector nat numContests -> Set :=
  match numContests with
    | 0%nat => fun _ => Empty_set
    | S n'  => fun v1 => prod (vector (vector G numTrustees) (Vhead v1))
                              (decryptionFactors numTrustees (Vtail v1))
  end.
```

Since we do not trust the authorities (trustees) to honestly decrypt the result, ElectionGuard uses sigma protocols to prove that the decryption factors are computed correctly. We define the type of these proofs as below.

```
Fixpoint decryptionProof (numContests numTrustees : nat) :
    vector nat numContests -> Type :=
  match numContests with
    | 0%nat => fun _ => Empty_set
    | S n'  => fun v1 => prod
        (ProofTranscript (BallotDecSigma (Vhead v1) numTrustees))
        (decryptionProof numTrustees (Vtail v1))
  end.
```

2.3 Verifier

We will largely skip over the details of our implementation of the verifier because we have proven its cryptographic soundness and have checked that it is compatible with ElectionGuard, and as such, the exact details are not particularly important.

At a high level, the verifier is defined in Figure 1 (for simplicity we use parameters implicitly); it takes in the election parameters, the cast ballots and various cryptographic proofs and decryption factors. It then checks that the

cryptographic proofs that the ballots are well formed are valid and that the crytographic proofs of correct decryption for the summation of all the ballots are valid. The Coq variant is shown below. `bVforall2` takes a predicate p on two values, and two vectors v, v' of the same length m and checks that $p(v[i], v'[i])$ is true for all i in 1 to m.

```
Definition Verifier
  (* Parameters *)
  (numTrustees numCast numContests : nat)
  (numSel : vector nat numContests)
  (g : G) (pks : vector G numTrustees)

  (* Cast ballots *)
  (castBallots : vector  (ballot numContests numSel) numCast)

  (* Proofs of correct encryption *)
  (ballotProofs : vector (ballotProof numContests numSel) numCast)
  (decFactors : decryptionFactors numTrustees numSel)
  (decProofs : decryptionProof numTrustees numSel) : bool :=
  let pk := (g, VG_prod pks) in
  let tally := Vfold_left (multBallots numContests numSel)
                          (zeroBallot numContests numSel) castBallots in
  (* Check proof of correct encryption *)
  (bVforall2 (BallotVerifier pk numContests numSel)) castBallots ballotProofs
  (* Checks proof of correct decryption *) &&
  DecryptionVerifier g pks numContests numSel tally decFactors decProofs.
```

We describe each component of the Coq definition:

`numtrustees`: is the number of election authorities participating in the election;
`numCast`: is the number of ballots cast in the election;
`numContests`: is the number of contests in the election;
`numSel`: is a vector containing the number of candidates in each contest;
`g`: is the generator of the underlying Schnorr group `G` for ElGamal;
`pks`: is the vector of length `numtrustees` containing elements from `G` ie the public keys of the authorities;
`ballot numContests numSel`: is the set of all ballots for `numContests` contests with `numSel` candidates in each contest.
`castBallots`: of type `vector (ballot numContests numSel) numCast` is then the vector of length `numCast` containing each ballot of type `(ballot numSel)`;
`multBallots numContests numSel:` is a function which forms the multiplication of two ballots in `ballot numContests numSel` by multiplying the ciphertexts component-wise.

2.4 Machine Checked Verifiability

In this subsection we will present our main theorem about the validity of the verifier. We will present it first in more standard notation and then in Coq notation. A reader familiar with sigma protocols will notice that it takes the form of cryptographic special soundness.

Recall that a zero knowledge proof demonstrates that a statement s belongs to a particular language, and it is common to use R to denote the relationship between statements and witnesses. Special soundness says that if any adversary

can produce two accepting transcripts for different challenges then it is possible to extract a witness w from those transcripts efficiently such that $(s, w) \in R$. Bellare and Goldreich give the standard definition of proofs of knowledge in their work "On Defining Proofs of Knowledge" [15]. They define knowledge error, which intuitively denotes the probability that the verifier accepts even when the prover does not know a witness. It has been shown that a sigma protocol satisfying special soundness is a proof of knowledge with negligible knowledge error in the length of the challenge, as stated next.

Theorem 3. *A sigma protocol $\mathcal{P}$ for relation $\mathcal{R}$ with challenge length t is a proof of knowledge with knowledge error 2^{-t}.*

The intuition for why special soundness implies soundness is straightforward. Special soundness says that, for any given commitment, if the adversary can answer for two different challenges then the adversary must know a witness for the statement. This implies that if no witness is known there must be at most one challenge for which the adversary could successfully respond. The chance of drawing the single challenge for which the adversary can successfully respond is negligable in the security parameter. (The formal argument in the case of a proof of knowledge is slightly different and can be found in [14].)

The reader may also find that the upcoming Theorem 5 has a slightly odd feel. The proofs of correct encryption and decryption intuitively have a temporal ordering, the protocol even specifies that the trustees should check the proofs of correct encryption before decrypting. However, since we are defining the verifier for the public information after the election is concluded, we can fold these proofs into one large proof for simplicity. Formally, we are allowed to do this because the properties of sigma protocol are invariant under parallel composition [14], which was proven to be true for the formalisation of Sigma protocols we use in [13].

Theorem 4. *For all number of trustess, number of cast ballots, number of contests, for all ballot formats, generators, public key shares, cast ballots, for all decryption factors, if there exists an adversary $\mathcal{A}$ which can produce accepting proofs for the verifier for two different challenges on the same commitment then the ballots are all correctly formed and the summation is correctly decrypted.*

The Coq theorem is stated slightly differently, we show that the existence of two accepting proofs with two different challenges on the same commitment implies that the ballots are all correctly formed and the summation is correctly decrypted. Since this holds for any two transcript of this form it clearly holds for any $\mathcal{A}$ producing transcripts of this form. We show this modified theorem in Theorem 5.

Theorem 5. *For all number of trustees, number of cast ballots, number of contests, for all ballot formats, generators, public key shares, cast ballots, for all decryption factors, for all pairs of accepting proof transcripts, if the pair of proof transcripts have two different challenges on the same commitment, then the ballots are all correctly formed and the summation is correctly decrypted.*

Theorem 5 is encoded into Coq as shown below.

```
Theorem VerifierCorrect :
forall
   (numTrustees numCast numContests : nat)
   (numSel : vector nat numContests)
   (g : G)(pks : vector G numTrustees)
   (* Cast ballots *)
   (castBallots : vector  (ballot numSel) numCast)
   (* Proofs of correct encryption *)
   (balProf1 balProf2 : vector (ballotProof numSel) numCast)
   (decFactors : decryptionFactors numTrustees numSel)
   (decProf1 decProf2 : decryptionProof numTrustees numSel)
   (result : tally numSel),

let pk := (g, VG_prod pks) in
let summation := Vfold_left (multBallots numSel) (zeroBallot numSel)
                            (castBallots) in

(* The tally and the decryption factors are consistent *)
ResultDecFactorsConsistent numTrustees g numSel summation result decFactors
-> Verifier numSel g pks castBallots balProf1 decFactors decProf1
-> (* Conditions for special soundness *)
    Verifier numSel g pks castBallots balProf2 decFactors decProf2 ->
    Vforall2 (ballotProofDis numSel) balProf1 balProf2 ->
    Vforall2 (ballotProofComEq numSel) balProf1 balProf2 ->
    decryptionProofDis numTrustees numSel decProf1 decProf2 ->
    decryptionProofComEq numTrustees numSel decProf1 decProf2 ->

Vforall (BallotCorrectlyFormed pk g numSel) castBallots /\
BallotCorrectlyDecrypted pk numSel summation result.
```

`Vfold_left` Takes a binary function, an initial value, and a vector and reduces the vector to a single value. In this case it multiplies all the encrypted ballots using the function `multBallots`. The proof of Theorem 5 follows from the soundness of the underlying sigma protocols; in essence we extract witness to all the underlying statements and show that they collectively imply that all the encrypted ballots are well-formed and the aggregation of all encrypted ballots is correctly decrypted.

3 Using the Extracted Verifier

Having defined the verifier we could use it inside Coq to check election transcripts, but unfortunately, this is prohibitively slow. Instead, we make use of the Coq extraction facility to produce OCaml code which matches the Coq specification. This extraction facility is the subject of the CertiCoq project [16] which aims to verify its correctness. Our verifier is proven secure for any Schnorr groups in the sense that the reductions and logical proofs hold for any such group; of course, if the decisional Diffie-Hellman problem is easy in the chosen group then privacy is lost. We note that our extracted verifier replaces the Coq implementation of arithmetic with the native OCaml implementation for efficiency.

We now encounter an issue, the reference verifier released with ElectionGuard[6] does not appear to be compatible with the parameters in the ElectionGuard specification.[7] The reference verifier works for a limited set of safe prime groups

[6] https://github.com/microsoft/electionguard-verifier

[7] https://github.com/microsoft/electionguard/wiki/Informal/ElectionGuardSpecificationV0.85.pdf

whereas the specification requires a Schnorr group which is not a safe prime group. To test our verifier, we therefore changed the parameters from those in the specification to the 1536-bit group from the reference verifier. We produced test cases (in the form of JSON files) with the reference verifier; we then wrote code to parse this JSON and feed it into our verifier. Our verifier accepted on the test cases and rejected on the incorrect inputs we tried.

3.1 Efficiency

Our extracted verifier with some underlying Coq functions replaced by OCaml counterparts is twice as efficient as the reference verifier provided by ElectionGuard. The time to verify is dominated by the number of ciphertexts which is the total number of candidates in all contests multiplied by the number of voters. Our verifier takes about 50 seconds per 1000 ciphertexts, so for an election with one million ciphertexts, it would take roughly 14 hours. This compares to the reference verifier which takes 110 seconds per 1000 ciphertexts. We were surprised that our verifier was faster; the OCaml implementation we use of the mathematics is faster by a factor of two which might explain the difference.

Note, our current encoding is a first attempt and mimicks the underlying mathematics as closely as possible to ensure that the encoding does not contain transliteration errors. Our encoding can be further optimised for speed and parallelised if required, but this requires further work.

The performance of our ElectionGuard verifier on the test cases, while comparable in efficiency to other implementations, is significantly slower than the machine-checked verifier for Helios created by Haines *et al.* [13]. This is due to the use of a safe prime group in the ElectionGuard reference verifier even though the specification requires a Schnorr group. If we replace that safe prime group with a Schnorr group of comparable security, as used by Helios, but with prime order of around 256 bits, our implementation would be ∼6 times faster than it currently is, meaning that an election of one million ciphertexts would take only 2 hours to verify. The ElectionGuard specification mandates a Schnorr group with a prime order of around 256 bits, so in a real election, our verifier would be faster than it was on tests produced by the reference implementation.

4 Conclusion

In this work we machine-checked the verifiability specification of ElectionGuard to be cryptographically sound. We achieved this by encoding the specification inside the interactive theorem prover Coq and then proving that it has cryptographic soundness. In addition, we proved the zero-knowledge properties of the verifier. We extracted an executable version of the verifier specification which is of comparable efficiency to existing implementations and used it to verify election transcriptions produced by the reference implementation of ElectionGuard.

Acknowledgments. This work was supported by the Luxembourg National Research Fund (FNR) and the Research Council of Norway for the joint project SURCVS

References

1. Rivest, R.L.: On the notion of 'software independence' in voting systems. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **366** (2008) 3759–3767
2. Adida, B.: Helios: Web-based open-audit voting. In van Oorschot, P.C., ed.: USENIX Security Symposium, USENIX Association (2008) 335–348
3. Microsoft: Electionguard (2019)
4. Haines, T., Lewis, S.J., Pereira, O., Teague, V.: How not to prove your election outcome. In Oprea, A., Shacham, H., eds.: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Jose, CA, USA, May 17-21, 2020, IEEE (2020) 784–800
5. Halderman, J.A., Teague, V.: The New South Wales iVote system: Security failures and verification flaws in a live online election. In: International Conference on E-Voting and Identity, Springer (2015) 35–53
6. Springall, D., Finkenauer, T., Durumeric, Z., Kitcat, J., Hursti, H., MacAlpine, M., Halderman, J.A.: Security analysis of the Estonian internet voting system. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM (2014) 703–715
7. Bernhard, D., Cortier, V., Pereira, O., Smyth, B., Warinschi, B.: Adapting Helios for provable ballot privacy. In: ESORICS. Volume 6879 of Lecture Notes in Computer Science., Springer (2011) 335–354
8. Cortier, V., Smyth, B.: Attacking and fixing Helios: An analysis of ballot secrecy. Journal of Computer Security **21** (2013) 89–148
9. Bernhard, D., Pereira, O., Warinschi, B.: How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In: ASIACRYPT. Volume 7658 of Lecture Notes in Computer Science., Springer (2012) 626–643
10. Gonthier, G.: The four colour theorem: Engineering of a formal proof. In Kapur, D., ed.: Computer Mathematics, Berlin, Heidelberg, Springer Berlin Heidelberg (2008) 333–333
11. Geuvers, H., Wiedijk, F., Zwanenburg, J.: A constructive proof of the fundamental theorem of algebra without using the rationals. In: Selected Papers from the International Workshop on Types for Proofs and Programs. TYPES '00, Berlin, Heidelberg, Springer-Verlag (2002) 96–111
12. Bertot, Y., Castéran, P., Huet, G., Paulin-Mohring, C.: Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions. Texts in theoretical computer science. Springer (2004)
13. Haines, T., Goré, R., Tiwari, M.: Verified verifiers for verifying elections. In: ACM Conference on Computer and Communications Security, ACM (2019) 685–702
14. Damgård, I.: On Σ-protocols (2002)
15. Bellare, M., Goldreich, O.: On defining proofs of knowledge. In: CRYPTO. Volume 740 of Lecture Notes in Computer Science., Springer (1992) 390–420
16. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq. In: The Third International Workshop on Coq for Programming Languages (CoqPL). (2017)

A Coq source

Our code is available via the link below:

https://github.com/gerlion/secure-e-voting-with-coq