Theorem Provers to Protect Democracies

Anonymous Author(s)

ABSTRACT

Electronic voting requires some trusted setup from an election commission to bootstrap the voting process. One such trusted setup is generating group parameters, i.e., group generator of a finite cyclic group, public key, private key, etc. In theory, computing group generator is not a very difficult problem and in fact, there are many algorithms to compute group generators. However, election verifiability –every step must be accompanied by some evidence that can be checked by independent third party to ascertain the truth– rules out many of these algorithms because they do not produce evidence of correctness, with their result. In general, if a software program that is used for computing group generators for an election encodes any of these ruled out algorithms, it can be catastrophic and possibly undermine the security of whole election.

In this work, we address this problem by using Coq theorem prover to formally verify the group generator `Algorithm A.2.3`, specified in `National Institute of Standards and Technology (NIST), FIPS 186-4 (Digital Signature Standard)`, with a proof that it always produces a correct group generator. Algorithm A.2.3 is a highly sought method to compute group generator(s) in a verifiable manner because its outcome can be established independently by third parties. Our formalisation captures all the requirements, specified in Algorithm A.2.3, using the expressive module system of Coq theorem prover. We evaluate the group generator algorithm/function inside the Coq theorem prover itself to produce group generators, only trusting the Coq theorem prover and its evaluation mechanism. Our formalisation can be used to validate group generators produced by unverified program written in Java, C, C++, etc. It can be accessed from GitHub: https://anonymous.4open.science/r/Formally_Verified_Verifiable_Group_Generator-1104.

CCS CONCEPTS

• **Security and privacy** → **Formal methods and theory of security**;

KEYWORDS

Formal Verification, Verifiable Group Generator, Cryptography, Electronic Voting, Coq Theorem Prover, Safe Computation, SHA-256, Fermat's Little Theorem

1 MOTIVATION

Electronic voting is getting popular in many countries, and the reason for its popularity is cost-effective, faster result, high voter turn out, and accessible for disabled voters. Undeniably, electronic voting has helped, for example, Australia to ease the logistic challenges of elections because of its massive land size and sparse population and save millions of dollars. Despite all these benefits, electronic voting is an arduous effort because a minuscule possibility of going anything wrong in a software or hardware, which is extremely hard to detect, could lead to an undesirable situation.

There is a history of bugs, varying from trivial to critical, in the electronic voting software programs employed for democratic elections, e.g., Scytl/SwissPost [8] (used in Swiss election), Voatz [16] (used in West Virginia, USA election), Democracy Live Online Voting System [15] (used in Delaware, West Virginia, and New Jersey, USA election), Moscow Internet Voting System [6] (used in Russia election), The New South Wales IVote System [4, 9] (used in New South Wales, Australia election). Most of these software programs establish their correctness by means of testing, which does not capture all the possible scenarios. In addition, these software programs are proprietary artefacts and are not allowed to be inspected by the members of general public [1]. Interestingly, most of these bugs were found by researchers, who inspected the code after the source code was made public, even though the companies developing these proprietary software programs had claimed that they were bug free. In the paper, How Not to Prove Your Election Outcome [8], Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague demonstrated, amongst several flaws in the (Java) source code of Scytl/SwissPost e-voting solution, the problem of independence of generators (two generators g, h are independent if no one knows the discrete logarithm $log_g h$). The severity of problem can be understood that it could allow a corrupt authority to change the ballots but produce a valid shuffle proof that verifies, i.e., every thing is good, which clearly is not the case.

We argue that all the components of electronic voting software programs should be developed with utmost rigour, using formal verification. In addition, these components should be open sourced so that anyone can inspect the source code to verify the claims about the source code, but more importantly, any third party should be able to substantiate all these claims independently.

We focus on the problem of computing independent generators, specifically in the context of electronic voting [8] to strengthen the democratic process. We establish the independence of generators by computing them using a well specified method (`Algorithm A.2.3`), encoded in Coq theorem prover, that takes some inputs and returns a generator. Later, we release these inputs publicly so that any third party can ascertain that it is indeed the case. The rationale is that when two generators g, h are produced by some public data, it is difficult, or computationally infeasible, to know their discrete logarithm. More importantly, we keep the trusted computing base bare minimum, i.e., only Coq theorem prover [17] and its evaluation mechanism.

1.1 Our Contribution

We have made the first significant impact in the direction of bootstrapping an election parameters by using formal verification, and thereby significantly reducing the gap between theory and practice. The current practices of electronic voting contains multiple layers of humongous amount of text documentation, ranging from high level descriptions to low level implementation details. Generally, the higher layer contains English prose, and the lower we move, it becomes more precise in terms of the technical details. In theory,

these multiple layers are supposed to align with each other, but in practice, they don't and thereby leading to a significant gap and source of errors. While having these documents are certainly an excellent thing, relying on these documents for correctness of an implementation is not a good idea. Therefore, we use the most precise system available, i.e., formal system (constructive logic), to encode an election voting bootstrapping software program to eliminate this gap and enable far greater confidence in the election bootstrapping process. In our formal system, the English prose about the correctness proof of an implementation itself becomes an implementation (a theorem which we need to prove).

In this work, we encode the generation algorithm, `FIPS 186-4 Algorithm A.2.3`, and verification algorithm, `Algorithm A.2.4`, [11] in Coq theorem prover [17] and prove their correctness. In addition, the efficient encoding of generation and verification allows us to evaluate the generator algorithm/function and verification algorithm/function in the Coq theorem prover itself, thereby reducing the trusted computing base to only the Coq theorem prover and its evaluation mechanism. Our formalisation contains:

(1) encoding of `Verifiable Canonical Generation of the Generator g (FIPS 186-4, Algorithm A.2.3)`, with a correctness proof that it always computes the correct generator.
(2) encoding of `FIPS 186-4, Algorithm A.2.4`, known as `Validation Routine`, to check the validity, or correctness, of generators, computed according to the protocol described in the Algorithm A.2.3.
(3) efficient encoding `SHA-256 (FIPS 180-4)` [12] hash algorithm, with usual correctness properties, needed in previous two steps (step 1 and 2) that can compute the hash value of any arbitrary string inside the Coq theorem prover within a reasonable amount of time, without running out of memory.
(4) `Fermat's little theorem` formalisation, required to prove the correctness of generation algorithm (FIPS 186-4, Algorithm A.2.3) and validation (verification) algorithm (FIPS 186-4, Algorithm A.2.4).

In addition, we use coqprime [18] to establish the primality (primeness) of a number inside the Coq theorem prover. Our formalisation contains 28000 lines of Coq code (5000 lines of code comes from the `src` directory containing the formalisation of SHA-256, Fermat's Little Theorem, Generator (Algorithm A.2.3), Generator Verification (Algorithm 2.3.4) and 23000 lines of code comes from primality directory which contains the Coq certificates, generated using coqprime, of various prime numbers).

1.2 Notations

Throughout this paper, we assume that p and q are large prime numbers such that $p = k * q + 1$ for some natural number k (when k = 2 these primes are called `Sophie Germain primes` or `Safe primes`), G_q is the subgroup of Z_p^*, g is the generator of the subgroup G_q of order q, i.e., $g^q \equiv 1 \pmod{p}$. This set-up is also known as the Schnorr Group [14]. Two generators $g, h \in G_q$ are independent, or `independent generators`, if no one knows their discrete logarithm $k \in Z_q$, where $k = log_g h$. The goal of this paper to compute `independent generators` in a way that any third party can establish the correctness independently (*public verifiability*).

1.3 Background

We briefly explain commitment scheme to introduce the importance of independence of generators in the context of electronic voting. Commitment scheme, a key requirement in most electronic voting software programs, were first introduced by Blum [3]. The problem is: two parties, Alice and Bob, who do not trust each other, and are possibly hostile to each other, but want to reach an agreement using a coin flip via telephone. The caveat is that they do not see each other's outcome and therefore they can cheat, without getting caught. Blum solved this problem by forcing both parties, Alice and Bob, to `commit` the secret value of their coin flip and make it public, by giving it to each other or publishing it to public bulletin board. Once both parties have the committed value of each other, they reveal the secret values to each other. Both parties check each others claims by matching them against their published commitments, and Alice or Bob wins the toss, depending on the call. In a nutshell, commitment scheme forces the parties, participating in a online protocol, to behave honestly, even though they have huge incentive to deviate from the protocol.

Now, we demonstrate that how not following the *independence of generators assumption* for the `Pedersen Commitment Scheme` [13] in an electronic voting can undermine the security of whole election, as shown in [8]. In most electronic voting software programs, Pedersen Commitment Scheme, or some generalisation [2], is used. It is defined as: when a party wants to commit a message $m \in Z_q$, it chooses two independent generators $g, h \in G_q$, a random $r \in Z_q$ and computes:

$$C(m, r) = g^m * h^r$$

The idea behind the Pedersen commitment scheme is that $C(m, r)$ does not reveal any information about m, known as `perfectly hiding`. Moreover, a committer cannot open a commitment $c, C(m, r)$, of a message m to any other message m' ($m' \neq m$), known as `computationally binding`, unless she knows the discrete logarithm k ($log_g h$). The perfectly hiding property ensures that no other party can guess anything about the message m from the committed value, $C(m, r)$, while the computationally binding property forces the committer to be honest and reveal the original message m. The key requirement of the Pedersen commitment scheme is that g and h should be *independent*, i.e., no one should know the k, such that $k = log_g h$. Because if such k is known, then the committer can open the commitment, $C(m, r)$, of the message m to an arbitrary message m' ($m' \neq m$).

$$\begin{aligned} C(m, r) &= g^m * h^r \\ &= g^m * (g^k)^r \text{ (substituting the } h = g^k) \\ &= g^m * g^{k*r} \\ &= g^{m+k*r} \end{aligned} \tag{1}$$

Now, the committer wants to open $C(m, r)$ to an arbitrary message m', so they compute r':

$$\begin{aligned} C(m, r) &= C(m', r') \\ g^{m+k*r} &= g^{m'+k*r'} \\ m' + k * r' &= m + k * r \\ r' &= (m + k * r - m') * k^{-1} \end{aligned} \tag{2}$$

The committer can open the same commitment in many different ways and therefore defeats the purpose of commitment. In a nutshell, if the Pedersen commitment implementation used in a electronic voting does not follow the key requirement of independence of generators, it can break the security of whole election, as demonstrated by the Haines et al. [8] in Scytl/SwissPost source code where a corrupt authority can change the ballots ballots without getting caught.

1.4 Coq Introduction

Coq is an interactive theorem prover (computer program) that allows users to encode (mathematical) definitions, express specifications (true statements) about the definitions, and formally prove that the definitions imply the the specifications.

We give a simple example by encoding natural numbers in Coq, writing a specification, and proving it formally, in Coq. We define natural number in Coq as (inductively) defined set *Nat* formed using the following two clauses:

(1) the term Zero is in the set *Nat*
(2) if a term *n* is in the set *Nat*, then *Succ n* is in the set *Nat*

We encode the set *Nat* in Coq by defining it as an inductive data type, shown in `Listing 1`, with two constructors, *Zero* and *Succ*. The constructor *Zero* represents the natural number 0, and the constructor *Succ* is actually a function that accepts a member from Nat and constructs another member of Nat by prefixing the given member with *Succ*, e.g., we can represent the natural number 1 as *Succ Zero*, the natural number 2 as *Succ* (*Succ Zero*), so on and so forth.

Listing 1: Definition of Natural Number

```
Inductive Nat :=
| Zero : Nat
| Succ : Nat -> Nat.
```

Next, we define addition on the set *Nat*. In Coq, *Fixpoint* is a keyword that is used to define a recursive function, followed by the name of the function (*plus* in our case and shown in `Listing 2`) and arguments of the function (*m* and *n* in our case). The function *plus* takes two natural numbers *m* and *n* and returns their sum. The function *plus*, shown in `Listing 2`, is Coq encoding of the following clauses:

(1) plus Zero n = n
(2) plus (Succ m) n = Succ (plus m n)

The function *plus* is defined by a case distinction on *m*. When *m* is *Zero*, it returns *n* because adding Zero to any natural number *n* is the same as *n*, and when *m* is *Succ m'*, for some natural number *m'*, it returns *Succ* (*plus m' n*) because adding (*Succ m'*) to any natural number *n* is the same as adding *m'* to *n* and then applying *Succ* to the result, i.e., *Succ* (*plus m' n*).

Listing 2: Addition of two Natural Numbers

```
Fixpoint plus (m n : Nat) :=
  match m with
  | Zero => n
  | Succ m' => Succ (plus m' n)
  end.
```

Now that we have encoded the set *Nat* in Coq and defined the plus function, we can write a specification for the plus function. We define the specification as a predicate *plus_commutative*, shown in `Listing 3`, which takes two natural numbers *m* and *n* and returns a proof that *plus m n = plus n m* (Coq is based on the constructive logic [5] and therefore the notion of truth is equivalent to producing a witness, that leads to program extraction [10] in OCaml/Haskell/Scheme. However, we are not using the extraction facility of the Coq theorem prover). We state the specification by using the keyword *Theorem*, followed by the name of the predicate (*plus_commutative*) and then we state the predicate (*forall n m, plus n m = plus m n*). We start the proof by using the keyword `Proof` and end the proof with the keyword `Qed`. In between, we write *tactics* to prove the specification. Coq provides tactics, a domain specific language, to build complicated proofs. For example, we use *induction, intros, simpl, rewrite, reflexivity* tactics to build the proof that *plus* is commutative. In fact, tactics are very convenient to build complex proofs [7].

Listing 3: Specification of Natural Number Addition

```
Theorem plus_commutative :
  forall n m, plus n m = plus m n.
Proof.
  induction n.
  + intros ?.
    simpl.
    rewrite plus_zero.
    reflexivity.
  + intros ?.
    simpl.
    rewrite plus_succ.
    rewrite IHn.
    reflexivity.
Qed.

Theorem  plus_zero :
  forall n, plus n Zero = n.
Proof.
  (* proof terms omitted *).
Qed.


Theorem plus_succ :
  forall n m, plus n (Succ m) = Succ (plus n m).
Proof.
 (* proof terms omitted *)
Qed.
```

The proof *plus_commutative* establishes that addition on natural number is commutative, and therefore we can be confident in our implementation of the *plus*. Similarly, to increase the confidence, we can prove many other properties about the *plus*, e.g., *plus* is associative, etc. However, the key point in formal verification is to ensure that specifications capture the right intention, otherwise it does not mean anything.

REFERENCES

[1] Australian Electoral Commission. 2013. Letter to Mr Michael Cordover, LSS4883 Outcome of Internal Review of the Decision to Refuse your FOI Request No.

LS4849. (2013). avaliable via http://www.aec.gov.au/information-access/foi/2014/files/ls4912-1.pdf, retrieved February 8, 2022.

[2] Stephanie Bayer and Jens Groth. 2012. Efficient Zero-Knowledge Argument for Correctness of a Shuffle. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 263–280.

[3] Manuel Blum. 1983. Coin Flipping by Telephone a Protocol for Solving Impossible Problems. *SIGACT News* 15, 1 (jan 1983), 23–27. https://doi.org/10.1145/1008908.1008911

[4] Andrew Conway, Michelle Blom, Lee Naish, and Vanessa Teague. 2017. An Analysis of New South Wales Electronic Vote Counting. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW '17)*. Association for Computing Machinery, New York, NY, USA, Article 24, 5 pages. https://doi.org/10.1145/3014812.3014837

[5] Thierry Coquand and Gérard Huet. 1986. *The calculus of constructions*. Ph.D. Dissertation. INRIA.

[6] Pierrick Gaudry and Alexander Golovnev. 2020. Breaking the Encryption Scheme of the Moscow Internet Voting System. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020 , Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers*. Springer-Verlag, Berlin, Heidelberg, 32–49. https://doi.org/10.1007/978-3-030-51280-4_3

[7] Georges Gonthier et al. 2008. Formal proof–the four-color theorem. *Notices of the AMS* 55, 11 (2008), 1382–1393.

[8] Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. 2020. How not to prove your election outcome. In *2020 IEEE Symposium on Security and Privacy (SP)*. 644–660. https://doi.org/10.1109/SP40000.2020.00048

[9] J. Alex Halderman and Vanessa Teague. 2015. The New South Wales IVote System: Security Failures and Verification Flaws in a Live Online Election. In *Proceedings of the 5th International Conference on E-Voting and Identity - Volume 9269 (VoteID 2015)*. Springer-Verlag, Berlin, Heidelberg, 35–53. https://doi.org/10.1007/978-3-319-22270-7_3

[10] Pierre Letouzey. 2008. Extraction in coq: An overview. In *Conference on Computability in Europe*. Springer, 359–369.

[11] National Institute of Standards and Technology. 2013. Digital Signature Standard (DSS). (2013). avaliable via https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf, retrieved February 8, 2022.

[12] National Institute of Standards and Technology. 2015. Secure Hash Standard (SHS). (2015). avaliable via https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf, retrieved February 8, 2022.

[13] Torben Pryds Pedersen. 1991. Non-interactive and Information-theoretic Secure Verifiable Secret Sharing. In *Annual international cryptology conference*. Springer, 129–140.

[14] C. P. Schnorr. 1991. Efficient Signature Generation by Smart Cards. *J. Cryptol.* 4, 3 (jan 1991), 161–174. https://doi.org/10.1007/BF00196725

[15] Michael Specter and J. Alex Halderman. 2021. Security Analysis of the Democracy Live Online Voting System. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3077–3092. https://www.usenix.org/conference/usenixsecurity21/presentation/specter-security

[16] Michael A. Specter, James Koppel, and Daniel Weitzner. 2020. The Ballot is Busted Before the Blockchain: A Security Analysis of Voatz, the First Internet Voting Application Used in U.S. Federal Elections. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1535–1553. https://www.usenix.org/conference/usenixsecurity20/presentation/specter

[17] The Coq Development Team. 2019. The Coq Proof Assistant, version 8.10.0. (Oct. 2019). https://doi.org/10.5281/zenodo.3476303

[18] Laurent Théry and Guillaume Hanrot. 2007. Primality Proving with Elliptic Curves. In *Theorem Proving in Higher Order Logics*, Klaus Schneider and Jens Brandt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 319–333.