Formally Verified Verifiable Group Generators

No Author Given

No Institute Given

Abstract. Electronic voting requires a trusted setup from an election commission to bootstrap the voting process, and one such trusted setup is producing a group generator of a finite cyclic group. In theory, computing a group generator is not a very difficult problem, and in fact, there are many algorithms to compute group generators. Election verifiability, however, rules out many of these algorithms because they do not produce evidence of correctness with their result.
In this work, we present a formally verified implementation of the group generator algorithm *A.2.3* and algorithm *A.2.4*, specified in National Institute of Standards and Technology (NIST), FIPS 186-4, in the Coq theorem prover. A.2.3 and A.2.4 are highly sought methods to compute and verify group generator(s), respectively, because their outcomes can be established independently by third parties (verifiability). Our formalisation captures all the requirements specified in algorithm A.2.3 and algorithm A.2.4 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, thereby only trusting the Coq theorem prover and its evaluation mechanism. It is, however, very slow and takes 30 minutes to produce a group generator for a 3072 bit prime number, so we also extract Haskell code from our Coq formalisation, and the extracted Haskell code takes few seconds for a 3072 bit prime number.

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

1 Motivation

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 [20] (used in Swiss election), Voatz [31] (used in West Virginia, USA election), Democracy Live Online Voting System [30] (used in Delaware, West Virginia, and New Jersey, USA election), Moscow Internet Voting System [16] (used in Russia election), iVote System [22] [13] (used in New South Wales, Australia election). Most of these software programs establish their correctness by means of testing, but testing does not capture all the possible scenarios. In addition, these software programs are proprietary artefacts, and therefore, members of general public –including researchers– are not allowed to inspect them [5]. 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. We argue that all the components of electronic voting software programs should be developed with utmost rigour, using the existing formal verification techniques. In addition, these components should be open sourced so that anyone can inspect the source code to verify the claims about the source code.

In this paper, we focus on the problem of computing independent generators, specifically in the context of electronic voting [20]. Algorithm A.2.3 is a well established method for computing independent generators (generators g and h are independent if no one knows a value k such that $h = g^k$. The value $k = log_g h$ is also known as discrete logarithm). Algorithm A.2.3 takes some public random data and produces generators. Later, all the public random data is published at a public bulletin-board 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 (To the best of our knowledge, we are not aware of any security proof for A.2.3).

1.1 Our Contribution

In this work, we turn the English proses describing the algorithm A.2.3 and algorithm A.2.4 into formal implementations using the Coq theorem prover [32], and as a result, all the correctness assertions, again expressed in English proses, of algorithm A.2.3 and algorithm A.2.4 turn into theorems, which need to proven formally in the Coq theorem prover. In essence, we use the most precise system available, i.e., formal system (constructive logic) to encode the algorithm A.2.3 and algorithm A.2.4 into computer programs, and thereby significantly reducing the gap between theory and practice. Our formalisation contains 5000 lines of Coq code and proofs of:

1. encoding of algorithm *A.2.3* with a correctness proof that it always produces correct generators.
2. encoding of algorithm *A.2.4* to check the validity of generators, computed according to the protocol described in the previous step (step 1).
3. encoding of *SHA-256 (FIPS 180-4)* [25] hash algorithm, required in the previous two steps (step 1 and 2).
4. encoding of *Fermat's little theorem*, required to prove the correctness of generation algorithm (A.2.3) and validation algorithm (A.2.4).

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 a subgroup of Z_p^*, g is a generator of G_q of order q, i.e., $g^q \equiv 1 \pmod{p}$. This set-up is also known as the *Schnorr* group [27]. Two generators $g, h \in G_q$ are independent, or *independent generators*, if no one knows their discrete logarithm $k \in Z_q$, where $k = log_g h$.

1.3 Background

We briefly explain commitment scheme to introduce the importance of independent generators in the context of electronic voting. Commitment scheme, a key requirement in most electronic voting software programs, were first introduced by Blum [8]. The problem is: two parties Alice and Bob that do not trust 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 if they have huge incentive to deviate from the protocol.

We now highlight that how not following the *independent generators assumption* for the *Pedersen commitment scheme* [26] in an electronic voting can undermine the security of whole election, as shown in [20]. In most electronic voting software programs, Pedersen commitment scheme, or some generalisation [6], 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 the commitment c of a message m to any other message m' $(m' \neq m)$, known as *computationally binding*. The perfectly hiding property ensures that no other party can guess anything about the message m from the committed value c, 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, the committer can open the commitment c of the message m to an arbitrary message m' $(m' \neq m)$, thereby breaking the computationally binding assumption.

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

Now, the committer wants to open $C(m, r)$ to an arbitrary message m', so she computes 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. [20] in Scytl/SwissPost source code where a corrupt authority can change the ballots ballots without getting caught.

1.4 The Coq Theorem Prover

The Coq theorem prover [33] is an interactive computer program that allows users to encode mathematical definitions, express specifications (true statements) about the definitions, and formally prove that definitions imply, or satisfy, the specifications. A user first defines a mathematical object (definition) and then figures out some true statements (specifications) about the mathematical object. Once the user has the definition and specifications, she needs to prove that the definition satisfies the specifications (proof writing). In general, amongst these three steps, the proof writing step is the most challenging and important step. During the proof writing, the user interacts with the Coq theorem prover via *tactics*, a domain specific language to ease the proof writing. Even though the Coq theorem provides some amount of automation and proof search, most of the time it requires human assistance to finish the proof. It is one the most popular and robust theorem prover, and one of the reasons for its popularity is dependent types that allow to encode any mathematical statement in the logic of Coq. It has been used to verify many real world software projects and mathematical artefacts, e.g., CompCert [24], a C compiler used by many companies[1], FiatCrypto [15], a cryptographic library used Chrome, Android, and CloudFlare, ConCert [4], a smart contract certification framework, etc.

Definitions, Specifications, and Proofs In this section, we demonstrate how to encode definitions, write specifications, and formally prove that definitions satisfy the specifications in the Coq theorem prover by an example of natural numbers (natural numbers is the set $\{0, 1, 2, 3, \dots\}$). The set of natural numbers Nat is the set of elements defined by the following clauses:

- *Zero*, represents 0, is in the set Nat ($Zero \in Nat$)

[1] https://www.absint.com/partners.htm

– if n is in the set Nat, then *Succ* n, represents $n + 1$, is in the set Nat (if $n \in Nat$, then $Succ\ n \in Nat$)

This representation of natural numbers is called inductively defined set where the base case, or starting point, is *Zero* (constant symbol), and if we have a natural number n, then we can get the next natural number by prefixing *Succ* (unary function symbol) to n. Using this encoding of natural numbers, we can represent 1 by *Succ Zero*, and 2 by *Succ (Succ Zero)*, and so on. We encode the set *Nat* in Coq as inductively defined data type, shown in Listing 1.1, with two constructors: *Zero* and *Succ*.

Listing 1.1. Definition of Natural Number

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

With the definition of natural numbers at our disposal, we can define addition, subtraction, multiplication, division, and many other mathematical functions on natural numbers in the Coq theorem prover. However, we only define the addition function, shown in Listing 1.2, to demonstrate the concept. In Coq, *Fixpoint* is a keyword for defining function, followed by the name of the function (*plus*), arguments (m and n), and the body (definition) of the function. The body of *plus*, shown in Listing 1.2, is Coq encoding of the following two clauses:

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

The first clause amounts to $0 + n = n$ and the second clause amounts to $(1 + m) + n = 1 + (m + n)$.

Listing 1.2. Addition of two Natural Numbers

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

Now, it is time to define some specification (true statement) about the addition operation on natural numbers. From mathematical literature, we know that the addition operation on natural numbers is associative and commutative. For simplicity, we prove that our definition of addition on natural numbers is commutative, shown in Listing 1.3, and it is precisely the *specification* for our addition operation.

Listing 1.3. Addition is Commutative

```
Theorem plus_commutative : forall n m, plus n m = plus m n.
Proof. induction n; intros m; rewrite IHn. Qed.
(* some proof terms omitted *)
```

In Coq, *Theorem* is a keyword to define a specification, followed by the name (*plus_comm*). It says that for any two arbitrary natural numbers m and n, encoded as *forall n m*, adding n and m is the same as adding m and n, encoded

as *plus n m = plus m n*. To write a formal proof of this statement, we tell the Coq theorem prover that we want to use the tactic language by stating the keyword *Proof*. After finishing the proof, we use the keyword *Qed* to inform Coq that our proof is finished. The moment Coq sees *Qed*, it ensures that the proof is complete and correct according to its underlying logic. If we try to use Qed without finishing the proof, Coq will not let us do so. Therefore, once we have Qed for any specification, by just trusting the Coq theorem prover we can be sure that our proof is correct. However, at this point, the curious reader can argue that why should we trust Coq because it is also a computer program, similar any other computer program? Trusting Coq hinges on two things: (i) the meta theory, *Calculus of Inductive Construction*, of Coq [35] and (ii) its OCaml implementation [33]. The meta theory of Coq has been scrutinised for decades by many mathematicians and so far it has withstood all the scrutiny. The OCaml implementation of Coq has a very small kernel and has been inspected by many computer scientists. In addition, there is a recent effort to develop more rigorous implementation for Coq proof checking [29].

The keywords *induction, intros, rewrite* are tactics that are used to prove the statement that addition on natural numbers is commutative. The proof written between the keywords *Proof* and *Qed* amounts to the following:

- Base case: when n is Zero, we need to prove that *plus Zero m = plus m Zero*
- Inductive case: when n is Succ, we assume *plus m n = plus n m* and we need to prove *plus (Succ m) n = plus n (Succ m)*

It is called proof by induction, and tactics are used to make the complicated process of proof writing easy.

2 Verifiable Group Generator

Verifiability –every step can be ascertain independently by a third party– is a key requirement in electronic voting. It ensures that the election is not manipulated by an adversary and correctly reflects the intentions of voters. One way to establish the independence of generators, in context of electronic voting, is to produce them using some public data. Later publish all the data so that everyone can establish the claim that these generators are independent (recall that when two generators g, h are produced by some public data, it is difficult, or computationally infeasible, to know their discrete logarithm).

NIST has published an algorithm A.2.3, shown in *Algorithm* 1, that produces generators in a verifiable way. The algorithm computes a generator by taking inputs p and q (prime numbers such $p = k * q + 1$ for $k \geq 2$), a *domain_parameter_seed* (a seed from which p and q have been generated), and an *index* (an 8 bit unsigned integer) that can be used as a marker to compute generators for different purposes. For example, if $index = 0$ then the generator is used for for digital signatures, if $index = 1$ then the generator is used for key establishment, etc. Anyone can take this publicly available data and check using

the verification (validation) algorithm, shown in *Algorithm* 2, that the claim is true or not.

Briefly, the generation *Algorithm* 1 iterates a 16 bit unsigned integer counter *count* from 1 to $2^{16} - 1$ and loop back to 0 (overflow), ensured by line 9 test if *count* has reached the maximum value. In each iteration, we concatenate the *domain_parameter_seed*, literal string "*ggen*", *index*, and *count* and assigns this concatenated string to a variable U (line 11). In line 12, we compute the hash value, in our case *SHA-256*, of U and assigns it to a variable W. Finally, we compute g (W^k *mod* p) and check if it is less than 2, line 14. If so, we go to line 8, increment the *count* and run the iteration one more time. Otherwise, we have found a generator g of order q, i.e., $g^q \equiv 1 \pmod{p}$. We can prove that g has order q by Fermat's little theorem.

Proof:

$$
\begin{aligned}
(g^q \bmod p) &= ((W^k \bmod p)^q \bmod p) && (3)\\
&= W^{k*q} \bmod p && (4)\\
&= W^{p-1} \bmod p \text{ (* p = k * q + 1 *)} && (5)\\
&= 1 \bmod p \text{ (* Fermat's little theorem *)} && (6)
\end{aligned}
$$

Similarly, the verification *Algorithm 2* takes everything that *Algorithm* 1 takes as input with an additional input a generator g, that we want to validate. The verification algorithm is very similar to generation algorithm, except in the beginning, line 3, line 5, and line 7, there are checks to rule out invalid generators. Line 3 ensures that *index* is between 0 and 255, line 5 ensures that g is between 2 and $p - 1$ (inclusive), line 7 ensures that the order of subgroup is q. The steps from line 9 to line 18 are same as the generation algorithm. Line 20 ensures that if we have found a generator, it better matches with the input generator g and if so we return *VALID*, otherwise we return *INVALID*.

2.1 Coq Formalisation of Generation Algorithm

We explain the Coq encoding of Algorithm 1, *compute_gen* shown in *Listing 1.4*. The *compute_gen* is a very liberal encoding of the Algorithm 1 because it takes an extra argument an (unary) natural number n. Moreover, *count* is not represented as 16 bit unsigned integer but a (binary) natural number. (In addition, it does not mention anything about the primes p, q, *domain_parameter_seed*, and *index*. They have been elided for presentation simplicity and we explain more about them in the next section). The rationale to start with a very liberal implementation is to keep it simple and readable. In addition, it ensures the proof for functional correctness is straightforward. The extra n is used as a fuel to ensure the termination, or more precisely makes the Coq type checker accept our definition; *count* not being 16 bit unsigned number because it keeps the implementation generic and makes the proof of functional correctness easy (we have marked it *Local* to ensure that no client, outside of the file where *compute_gen* is defined, can access it).

Algorithm 1 Verifiable canonical generation of a generator g. It takes two large primes p and q ($p = k * q + 1$), *domain_parameter_seed* (the random seed used during the generation of p and q), *index* (a bit string of length 8 that represents an unsigned integer).

procedure VERIFIABLE-GENERATOR-PROCEDURE($p, q, domain\_parameter\_seed, index$)
  Result: *status, g*
  if *index* is incorrect **then** return **INVALID**
  end if
  N = len(q)
  k = (p - 1)/q
  count = 0
  count = count + 1
  if count = 0 **then** return **INVALID**
  end if
  U = domain_parameter_seed || "ggen" || index || count
  W = Hash (U)
  g = W^k mod p
  if g < 2 **then**, go to step 8
  end if
  Return **VALID** and the value of g
end procedure

Algorithm 2 Validation routine when the Algorithm 1 is used to compute a generator g. It takes same inputs as Algorithm 1 with the generator g itself.

procedure VERIFIABLE-GENERATOR-VALIDATION($p, q, domain\_parameter\_seed, index, g$)
  Result: **VALID or INVALID**
  if *index* is incorrect **then** return **INVALID**
  end if
  if g $\notin$ [2, p) **then** return **INVALID**
  end if
  if $g^q \neq 1$ (mod p) **then** return **INVALID**
  end if
  N = len(q)
  k = (p - 1)/q
  count = 0
  count = count + 1
  if count = 0 **then** return **INVALID**
  end if
  U = domain_parameter_seed || "ggen" || index || count
  W = Hash (U)
  computed_g = W^k mod p
  if computed_g < 2 **then**, go to step 12
  end if
  if computed_g = g **then** return **VALID**
  elsereturn **INVALID**
  end if
end procedure

Listing 1.4. Generic group generator

```
Local Fixpoint compute_gen (n : nat) (count : N):Tag :=
  match n with
  | 0%nat => Invalid
  | S n' =>
    let U := append_values count in
    let W := sha256_string U in
    let g := Npow_mod W k p in
    if g <? 2 then
    compute_gen n' (count + 1)
    else Valid g
  end.
```

Finally, to emulate the exact behaviour of Algorithm 1, we define another function *compute_generator* that calls *compute_gen* by instantiating n with $2^{16} - 1$ and *count* with 1. All the correctness proofs that we have established for the *compute_gen* also hold for the *compute_generator* function, albeit for a specific value of n $(= 2^{16} - 1)$ and *count* $(= 1)$. Now the reader can verify that *compute_generator* is the exact encoding of the Algorithm 1, except the details of *domain_parameter_seed*, primes p and q, and *index*.

Now we focus on the details of *domain_parameter_seed*, primes p and q, and *index*. The Coq theorem prover has a very expressive (dependent) type system that can be used to encode and verify mathematical proofs. In our formalisation, we use the powerful module system of Coq. We orchestrate our development in such a way that if a client wants to call *compute_generator*, it has to ensure that all the requirements of *Algorithm 1* are met, which is only possible because of the powerful (dependent) type module system of Coq theorem prover. The novelty of our approach is that we define *compute_generator* inside a module *Comp*, shown in *Listing* 1.5, that itself takes another module type *Prime*, shown in the *Listing* 1.6, as an input. If any client wants to call *compute_generator* to compute generators, first it has to construct an element of module type *Prime* and pass this element as an argument to the module *Comp*, and precisely this step forces the client to ensure the correctness. It is only possible because of the expressive (dependent) type system of Coq theorem prover.

Listing 1.5. Module *Comp*, indexed by another module *Prime*, where the function *compute_generator* is defined and can be used by a client to compute a generator, only if the client can construct a concrete instance of module *Prime*.

```
Module Comp (P : Prime).
  Let p := P.p.
  Let q := P.q.
  Let k := P.k.
  Let domain_parameter_seed :=
    P.domain_parameter_seed.
  Let ggen : N := P.ggen.
  Let index := P.index.
```

```
  Local Fixpoint compute_gen (n : nat) (count : N) :
  Tag := (* definition omitted *)

  Definition compute_generator :=
    compute_gen (2^16-1) 1.
End Comp.
```

The module type *Prime* has 6 data points –represent as *Parameters*– p, q, k, *ggen*, *index*, and *domain_parameter_seed*. In addition, the correctness assumptions, represented as *Axiom*, are:

1. p is a prime, represented by *Axiom prime_p: prime p*
2. The bit length of prime p is greater than or equal to 1024, represented by *Axiom p_len: 1024 ≤ N.size p*
3. q is a prime, represented by *Axiom prime_q: prime q*
4. The bit length of q is greater than or equal to 160, represented by textttAxiom q_len: 160 ≤ N.size q
5. p is a safe prime, represented by *Axiom safe_prime: p = k * q + 1*
6. k is greater than or equal to 2, represented by *Axiom k_ gteq_2: 2 ≤ k*
7. *ggen* equal to $0x6767656e$ (it is a part of the FIPS 186-4 specification), represented by *Axiom ggen_hyp: ggen = 0x6767656e*
8. *index* is between 0 and 255 (8 bit unsigned integer), represented by *Axiom index_8bit: 0 ≤ index < 0xff*.

Listing 1.6. Module Type *Prime* that captures all the data and the correctness criterion, specified in *FIPS 186-4, A.2.3*.

```
Module Type Prime.
  Parameters (p q k
    domain_parameter_seed ggen index : N).
  Axiom prime_p : prime (Z.of_N p).
  Axiom p_len : 1024 <= N.size p.
  Axiom prime_q : prime (Z.of_N q).
  Axiom q_len : 160 <= N.size q.
  Axiom k_gteq_2 : 2 <= k.
  Axiom safe_prime : p = k * q + 1.
  Axiom ggen_hyp : ggen = 0x6767656e.
  Axiom index_8bit : 0 <= index < 0xff.
End Prime.
```

To get a concrete instance of *Prime*, a client needs to instantiate all data points, represented as *Parameter*, and discharge all the proof obligations, represented as *Axiom*. Once we have a concrete instance, *Pins*, of module type *Prime*, we can use *compute_generator*, shown in the *Listing* 1.7, function to compute a generator.

Listing 1.7. Concrete Instance

```
(* get a concrete instance *)
Module genr := Comp Pins.
```

```
(* Call the function to get a generator *)
Time Eval vm_compute in genr.compute_generator.
```

2.2 Proof of Correctness

Proof of theorem *generator_in_range_2_p* is very straightforward and merely a fact that we compute the generator g modulo p. Similarly, proof of the theorem *generator_verifier_correctness* is also straightforward and basically by induction on n, the fuel, and then instantiating n with 2^{16}. However, the second theorem *correct_compute_generator* is a corollary of Fermat's little theorem, that itself alone is a subject of study and formalisation, Chan and Norrish [11] in HOL4 [28], Théry and Hanrot in Coq [34], Boyer and Moore [9] in ACL [10]. We explain the details of Fermat's little theorem in the next section.

Listing 1.8. Correctness of generator algorithm and its connection with verification algorithm

```
(* generator in range 2 to p - 1 *)
Theorem generator_in_range_2_p : forall (g : N),
  g = compute_generator -> 2 <= g < p.

(* g generates a subgroup of order q *)
Theorem correct_compute_generator : forall g,
  g = compute_generator -> g^q mod p = 1.

(* compute_generator and verify_generator agree *)
Theorem generator_verifier_correctness : forall g,
  g = compute_generator <-> verify_generator g = true.
```

3 Fermat's Little Theorem

Fermat's little theorem is stated in two flavours: (i) for any integer a, when p is prime then $a^p \equiv a \pmod{p}$ and (ii) when a is not divisible by p, then we have $a^{p-1} \equiv 1 \pmod{p}$. Our Coq proof simply follows the proof of Euler using binomial theorem, explained at Wikipedia [14]. We sketch the proof here for completeness. Proof by induction on a, (i) *base case:* $0^p \equiv 0 \pmod{p}$ and (ii) *induction case:* we assume the induction hypothesis $a^p \equiv a \pmod{p}$ and prove $(a+1)^p \equiv a^p + 1 \pmod{p}$. The induction case can be proved by the fact that every term of binomial expansion of $(a+1)^p$ is equal to $0 \pmod{p}$, except the first term, $\binom{p}{0} * a^p$, and the last term, $\binom{p}{p} * 1^p$.

3.1 Coq formalisation

Fermat's little theorem has been formalised number of times in various theorem provers, e.g., Chan and Norrish [11] in HOL4 [28], Théry and Hanrot in Coq

[34], Boyer and Moore [9] in ACL [10]. All these formalisation, including ours, are based on the proof of Euler, except [11] which is based on counting necklaces [17].

Listing 1.9. Two flavours of *Fermat's little theorem*. The first one, *fermat_little_simp*, encodes $a^p \equiv a \pmod{p}$, while the second one, *fermat_little_coprime*, encodes when a is not divisible by p, then we have $a^{p-1} \equiv 1 \pmod{p}$

```
(* a^p = a mod p *)
Theorem fermat_little_simp : forall a p : nat,
prime p ->  a^p mod p = a mod p.

(* a^(p-1) = 1 (mod p), when a and p are coprime *)
Theorem fermat_little_coprime : forall (a p : nat),
 prime p -> a mod p <> 0 -> a^(p-1) mod p = 1
```

The theorem *prime_pow_exp*, shown in *Listing* 1.10, encodes the fact that for any integer a and prime p, $(a+1)^p = a^p + 1 \pmod{p}$ and captures the inductive step. We establish it by expending the terms of the binomial expansion of $(a+1)^p$, encoded as *Nat.pow (a + 1) p*. Finally, we show that every term, by the theorem *binom_mod_p_bound*, of the binomial expansion of $(a+1)^p$ is equal to 0 (mod p), except the first term, $\binom{p}{0} * a^p$, and the last term, $\binom{p}{p} * 1^p$. Fermat's little theorem is core of our formalisation and the correctness criterion.

Listing 1.10. Binomial Proof

```
(* (x + 1)^p % p = x^p + 1 (mod p) *)
Theorem prime_pow_exp : forall (a p : nat),
   prime p -> (a + 1)^p mod p = (a^p + 1) mod p.

(* pCk mod p = 0 for 1 <= k < p *)
Theorem binom_mod_p_bound : forall p k : nat,
   prime p -> k <= p ->
   (binomial_exp p k) mod p = 0 <-> 1 <= k < p.
```

For space reasons, we do not explain our SHA-256 encoding in detail, but our Coq encoding is fairly standard because we just follow the instructions in FIPS 180-4. Contrary to other implementations of SHA-256 in Python, Haskell, OCaml, etc., we prove all the invariants formally in Coq. Basically, we follow the correct by construction principal in encoding the SHA-256 algorithm. In addition, our encoding of SHA-256 is very efficient and we can compute SHA-256 hash inside the Coq theorem prover.

4 Experimental Result

We evaluate our implementation, to test the feasibility of our approach, on the test cases provided by NIST, FIPS 186-4. We take 12 test cases, primes ranging from 1024 to 3072 bit (currently used in practice). These test cases are written in 12 separate Coq file, ranging from *Generator_1.v* to *Generator_12.v*. The reader

may notice that the prime proofs are admitted, and the reason is that we want to measure the true time for computing a group generator. The reader can see the *primality* directory to ensure that all the p and q, used in *Generator_1.v* to *Generator_12.v*, are indeed a prime number. The Coq evaluation mechanism (*vm_compute*) takes: (i) 1 minute for 1024 bit prime, (ii) 10 minutes for 2048 bit prime, and (iii) 30 minutes for 3072 bit prime on Apple M1 16 GB Laptop. However, election commissions have more resources and we believe that with the powerful computers, the time can be reduced significantly. Nonetheless, it is definitely unreasonable amount of time to compute group generators, and in practice it takes a few seconds to compute group generators with OCaml/Haskell/Python. To test this hypothesis, we have developed a script that contains 25 lines of Python code to evaluate the time for computing group generators, and it takes less than one second to produce a group generator for all primes, ranging from 1024 to 3072 bit. We therefore extract Haskell from our Coq formalisation and use the Haskell compiler GHC (Glasgow Haskell Compiler) [23] to get an executable whose performance is on par with (unverified) Python implementation. However, it enlarges the trusted code base by including GHC. One way to keep the trusted codebase minimum and get performance by using CertiCoq [3] to transform the Coq code into a C code, and subsequently use CompCert to compile the C code to get machine code, but CertiCoq does not support Coq module extraction yet.

The reader would notice that it is not very difficult to compute a group generator. In fact, is it worthy of formalisation if it can be done in few lines of Python/Haskell/OCaml code (any unverified language)? The difficult part is not generation but making sure that all the specifications (preconditions) has been followed correctly during the generation. However, we cannot ensure this in Haskell/OCaml/Python/Java, or in fact any unverified language. This is particularly important in contexts such as elections, where the security of the system may be compromised if the group generator is not generated correctly.

5 Related Work

Our work is highly influence by coqprime [34], certifying prime numbers in the Coq theorem prover, and treating theorem provers as a computation tool. There are many verified cryptographic project and we give here a brief overview. Haines et al. [18] [21] [19] have used the Coq theorem prover extensively to encode and prove properties of various constructs used in electronic voting. Erbsen et al. [15] have developed a high-performance library for curve P-256 in the Coq theorem prover. Later, they extracted their formalisation in C code by developing a small programming language; it is used in BoringSSL library. This work only includes the Coq theorem prover in the trusted code base for formalisation. HACL* [36] is a formally verified cryptographic library in F* programming language which compiles down to C code. It has been integrated within Mozilla's NSS cryptographic library, and it includes SMT solver in the trusted code base. On the contrary, we trust only the standard the Coq theorem prover, which has

a proof-checking kernel significantly smaller. Also, there is a formally verified type checker [29] for the Coq theorem prover, so we can use the formally verified type checker, rather than the Coq type checker, to type check our proofs. Chen et al. [12] have formally verified the Curve25519 by combination of the Boolector SMT solver and the Coq theorem prover. Again, this work includes SMT solver as a trusted code base. SHA-3 [2] has been formally verified in Jasmin [1] programming language. It includes that the SHA-3 hash function is in-differentiable from a random oracle and functional correctness. In addition, provably protected against timing attacks in an idealised model of timing leaks, but it is not a mature tool as the Coq theorem prover. Lennart et al. [7] formally verified the OpenSSL implementation of HMAC with SHA- 256 correctly implements the FIPS functional specification in the Coq theorem prover.

References

1. Almeida, J.B., Barbosa, M., Barthe, G., Blot, A., Grégoire, B., Laporte, V., Oliveira, T., Pacheco, H., Schmidt, B., Strub, P.Y.: Jasmin: High-assurance and high-speed cryptography. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1807–1823. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134078, https://doi.org/10.1145/3133956.3134078
2. Almeida, J.B., Baritel-Ruet, C., Barbosa, M., Barthe, G., Dupressoir, F., Grégoire, B., Laporte, V., Oliveira, T., Stoughton, A., Strub, P.Y.: Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of sha-3. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1607–1622. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3363211, https://doi.org/10.1145/3319535.3363211
3. 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)
4. Annenkov, D., Nielsen, J.B., Spitters, B.: Concert: a smart contract certification framework in coq. Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (Jan 2020). https://doi.org/10.1145/3372885.3373829, https://dx.doi.org/10.1145/3372885.3373829
5. Australian Electoral Commission: 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
6. Bayer, S., Groth, J.: Efficient Zero-Knowledge Argument for Correctness of a Shuffle. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 263–280. Springer (2012)
7. Beringer, L., Petcher, A., Ye, K.Q., Appel, A.W.: Verified correctness and security of OpenSSL HMAC. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 207–221. USENIX Association, Washing-

ton, D.C. (Aug 2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer
8. Blum, M.: Coin Flipping by Telephone a Protocol for Solving Impossible Problems. SIGACT News **15**(1), 23–27 (jan 1983). https://doi.org/10.1145/1008908.1008911, https://doi.org/10.1145/1008908.1008911
9. Boyer, R.S., Moore, J.S.: Proof Checking the RSA Public Key Encryption Algorithm. The American Mathematical Monthly **91**(3), 181–189 (1984)
10. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic press (2014)
11. Chan, H.L., Norrish, M.: A String of Pearls: Proofs of Fermat's Little Theorem. Journal of Formalized Reasoning **6**(1), 63–87 (Jan 2013). https://doi.org/10.6092/issn.1972-5787/3728, https://jfr.unibo.it/article/view/3728
12. Chen, Y.F., Hsu, C.H., Lin, H.H., Schwabe, P., Tsai, M.H., Wang, B.Y., Yang, B.Y., Yang, S.Y.: Verifying curve25519 software. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 299–309. CCS '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2660267.2660370, https://doi.org/10.1145/2660267.2660370
13. Conway, A., Blom, M., Naish, L., Teague, V.: 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 (2017). https://doi.org/10.1145/3014812.3014837, https://doi.org/10.1145/3014812.3014837
14. Editors, W.: Proofs of Fermat's Little Theorem (2022), avaliable via https://en.wikipedia.org/wiki/Proofs_of_Fermat's_little_theorem, retrieved April 19, 2022
15. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1202–1219 (2019). https://doi.org/10.1109/SP.2019.00005
16. Gaudry, P., Golovnev, A.: Breaking the Encryption Scheme of the Moscow Internet Voting System. In: Bonneau, J., Heninger, N. (eds.) Financial Cryptography and Data Security. pp. 32–49. Springer International Publishing, Cham (2020)
17. Golomb, S.W.: Combinatorial proof of Fermat's "little" theorem. The American Mathematical Monthly **63**(10), 718 (1956)
18. Haines, T., Goré, R., Tiwari, M.: Verified Verifiers for Verifying Elections. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 685–702 (2019)
19. Haines, T., Goré, R., Sharma, B.: Did you mix me? Formally Verifying Verifiable Mix Nets in Electronic Voting. In: 2021 IEEE Symposium on Security and Privacy (SP)
20. Haines, T., Lewis, S.J., Pereira, O., Teague, V.: How not to prove your election outcome. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 644–660 (2020). https://doi.org/10.1109/SP40000.2020.00048
21. Haines, T., Pattinson, D., Tiwari, M.: Verifiable Homomorphic Tallying for the Schulze Vote Counting Scheme. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 36–53. Springer (2019)
22. Halderman, J.A., Teague, V.: The New South Wales iVote System: Security Failures and Verification Flaws in a Live Online Election. In: Haenni, R., Koenig, R.E., Wikström, D. (eds.) E-Voting and Identity. pp. 35–53. Springer International Publishing, Cham (2015)

23. Hall, C.V., Hammond, K., Partain, W., Peyton Jones, S.L., Wadler, P.: The glasgow haskell compiler: A retrospective. In: Proceedings of the 1992 Glasgow Workshop on Functional Programming. p. 62–71. Springer-Verlag, Berlin, Heidelberg (1992)
24. Leroy, X.: Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 42–54. POPL '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1111037.1111042, https://doi.org/10.1145/1111037.1111042
25. National Institute of Standards and Technology: Secure Hash Standard (SHS) (2015), avaliable via https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf, retrieved February 8, 2022
26. Pedersen, T.P.: Non-interactive and Information-theoretic Secure Verifiable Secret Sharing. In: Annual international cryptology conference. pp. 129–140. Springer (1991)
27. Schnorr, C.P.: Efficient Signature Generation by Smart Cards. J. Cryptol. **4**(3), 161–174 (jan 1991). https://doi.org/10.1007/BF00196725, https://doi.org/10.1007/BF00196725
28. Slind, K., Norrish, M.: A brief overview of hol4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) Theorem Proving in Higher Order Logics. pp. 28–32. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
29. Sozeau, M., Boulier, S., Forster, Y., Tabareau, N., Winterhalter, T.: Coq coq correct! verification of type checking and erasure for coq, in coq. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371076, https://doi.org/10.1145/3371076
30. Specter, M., Halderman, J.A.: Security Analysis of the Democracy Live Online Voting System. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 3077–3092. USENIX Association (Aug 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/specter-security
31. Specter, M.A., Koppel, J., Weitzner, D.: 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). pp. 1535–1553. USENIX Association (Aug 2020), https://www.usenix.org/conference/usenixsecurity20/presentation/specter
32. Team, T.C.D.: The coq proof assistant, version 8.10.0 (Oct 2019). https://doi.org/10.5281/zenodo.3476303, https://doi.org/10.5281/zenodo.3476303
33. Team, T.C.D.: The coq proof assistant, version 8.15.0 (Oct 2022). https://doi.org/10.5281/zenodo.1003420, https://zenodo.org/record/1003420
34. Théry, L., Hanrot, G.: Primality Proving with Elliptic Curves. In: Schneider, K., Brandt, J. (eds.) Theorem Proving in Higher Order Logics. pp. 319–333. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
35. Thierry, C., Gérard, H.: The calculus of constructions. Information and computation **76**, 95–120 (1988)
36. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: Hacl*: A verified modern cryptographic library. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1789–1806. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134043, https://doi.org/10.1145/3133956.3134043