Formally Verified Verifiable Group Generators

Mukesh Tiwari
Department of Computer Science and Technology, University of Cambridge, Cambridge, UK
mt883@cam.ac.uk

Abstract

Electronic voting requires some trusted setup from an election commission to bootstrap the voting process. One such trusted set up is generating group parameters, i.e., group generator of a finite cyclic group, public key, private key, etc. In theory, computing group generators 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 generators 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. We evaluate the group generator algorithm/function inside the theorem prover itself to produce group generators, only trusting the Coq theorem prover and its evaluation mechanism. Our formalisation can be used as an oracle to validate group generators produced by unverified program written in Java, C, C++, etc.

We evaluate our formalisation on the test cases provided by NIST. Our implementation produces a group generator in: (i) 1 minute for a 1024 bit prime, (ii) 10 minutes for a 2048 bit prime, and (iii) 30 minutes for for 3072 bit prime on a Apple M1 16 GB RAM machine. Our formalisation can be accessed from GitHub: `https://github.com/mukeshtiwari/Formally_Verified_Verifiable_Group_Generator`.

2012 ACM Subject Classification Security and privacy → Formal methods and theory of security

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

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Acknowledgements I would like to thank Timothy Griffin for his valuable comments about the earlier draft of this paper and Laurent Théry for inspiring this project and providing the prime certificate (because the author lacks the computing resources).

1 Introduction

There is a history of bugs, varying from trivial to critical, in the electronic voting software programs Scytl/SwissPost [20], Voatz [32], Democracy Live Online Voting System [31], Moscow Internet Voting System [12], The New South Wales IVote System [22, 9] used for democratic elections. Most of these software programs establish their correctness by means of testing, which does not capture all the possible scenarios. In addition, most of these software programs are proprietary artefacts and are not allowed to be inspected by the members of general public [2]. 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

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

paper, How Not to Prove Your Election Outcome [20], 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 [20] to strengthen the democratic process.

1.1 Our Contribution

In this work, we encode the generation algorithm, `FIPS 186-4 Appendix A.2.3`, and verification algorithm, `FIPS 186-4 Appendix A.2.4`, [24] in Coq therom prover [33] and prove their correctness. In addition, our efficient encoding of generation and verification allows us to evaluate the generator algorithm/function and verification algorithm/function in the theorem prover itself, thereby reducing the trusted computing base to only Coq evaluation mechanism. Our formalisation contains:

1. encoding of `Verifiable Canonical Generation of the Generator g (FIPS 186-4, Appendix A.2.3)`, with a correctness proof that it always computes the correct generator.
2. encoding of `Validation Routine (FIPS 186-4, Appendix A.2.4)` to checck the validity, or correctness, of generators, generated according to the protocol described in the step 1 (Appendix A.2.3).
3. efficient encoding `SHA-256 (FIPS 180-4)` [25] hash algorithm, with usual correcntess properties, needed for step 1 and 2 that can compute the hash value of any arbitrary string inside Coq theorem prover within 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, Appendix A.2.3), described in step 1, and validation (verification) algorithm (FIPS 186-4, Appendix A.2.4), described in step 2.

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 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 [29]. Two generators $g, h \in G_q$ are independent, or `independent generators`, if no one knows the 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

Commitment schemes, a key requirement in most electronic voting software programs, were first introduced by Blum [5]. The problem is: two parties, Alice and Bob, who do not trust each other, and 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, a 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 show that how not following the `independence of generators assumption` for the `Pedersen Commitment Scheme` [28] 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 of it [3], 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`, and a committer cannot open a commitment $c, = C(m, r)$, of a message m to any other message m', where $m' \neq m$ and known as `computationally binding`, unless she knows the discrete logarithm $k, = log_g h$. The first property, perfectly hiding, ensures that no other party can guess anything about the message m from the committed value, $C(m, r)$, while the second property, computationally binding, forces the committer to be honest and show 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 a k, such that $k = log_g h$. Because if such a k is known, then a committer can open the commitment, $C(m, r)$, of the message m to an arbitrary message m', where $m' \neq m$.

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

Now, the committer wants to open $C(m, r)$ to an arbitrary message m', so 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 defeating 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.

2 Verifiable Group Generator

One way to establish the `independence of generators` is to produce them using some public data, and later publish all the data so that everyone can establish the claim that these generators are independent. This is very important from the election security perspective because it strengthens the verifiability aspect of an election.

NIST has published an algorithm FIPS 186-4, Appendix A.2.3, shown in `Algorithm 1`, which produces a generator in a verifiable way. The algorithm computes a generator by taking input p and q, prime numbers such $p = k * q + 1$ for some $k \geq 2$, *domain_parameter_seed*, the seed from which p and q have been generated, and the *index*, an 8 bit unsigned integer that can be used as a marker to compute generators for different purposes, such as $index = 1$ for first (independent) generator, $index = 2$ for second (independent) generator, etc. The idea is that anyone can take this input data, available publicly, and check using the verification algorithm, shown in `Algorithm 2`, that claim is true or not, i.e., the claimed generator is produced by the given public data.

Briefly, the generation algorithm, Algorithm 1, iterates a counter *count*, 16 bit unsigned integer, from 0 to $2^{16} - 1$ and loop back to 0, ensured by line 9 test if *count* has reached the maximum value. In each iteration, it concatenates the *domain_parameter_seed*, the literal string "*ggen*", the *index*, and the *count* and assigns this concatenated string to the variable U (line 11). In line 12, we compute the hash value, in our case `SHA-256`, of the string U and assigns it to the variable W. Finally, we compute W^k mod p and check if it is less than 2, line 14. If so, then we go to line 8, increment the *count* and run the iteration one more time and repeat everything that we mentioned earlier. 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, which is stated in two flavours: (i) any integer a and prime p, $a^p \equiv a \pmod{p}$, and (ii) if a is not divisible by p, $a^{p-1} \equiv 1 \pmod{p}$.

$$
\begin{aligned}
(g^q \bmod p) &= ((W^k \bmod p)^q \bmod p) \text{ (* value of g from line 13 *)} && (3)\\
&= W^{k*q} \bmod p && (4)\\
&= W^{p-1} \bmod p \text{ (* p = k * q + 1 *)} && (5)\\
&= 1 \bmod p \text{ (* Fermat's little theorem *)} && (6)
\end{aligned}
$$

Similarly, the verification algorithm, Algorithm 2, takes everything that generation algorithm, Algorithm 1, takes as input with an additional input the generator, g, itself 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

We now explain the Coq encoding, `compute_gen_slow` shown in `Listing 1`, of Algorithm 1 (we also have a slightly optimised version, `compute_gen_fast`, that we have proven equivalent to `compute_gen_slow`, but due to the simplicity and straight forward encoding, we will focus on `compute_gen_slow`). The `compute_gen_slow` is almost straight forward encoding of the Algorithm 1, except it takes an extra argument a unary natural number n and *count* is

Algorithm 1 Verifiable canonical generation of a generator g. It takes two large primes p and q, such that $p = k * q + 1$, *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, used as a marker for generating g.

1: **procedure** VERIFIABLE-GENERATOR($p, q, domain\_parameter\_seed, index$) ▷ p and q are large prime numbers such that $p = k * q + 1$, the seed used during the generation of p and q, the index to be used for generating g (index is a bit string of length 8 that represents an unsigned integer)
2:     **Result**: *status, g*
3:     **if** *index* is incorrect **then** return **INVALID**
4:     **end if**
5:     N = len(q)
6:     k = (p - 1)/q
7:     count = 0
8:     count = count + 1
9:     **if** count = 0 **then** return **INVALID**
10:     **end if**
11:     U = domain_parameter_seed || "ggen" || index || count
12:     W = Hash (U)
13:     g = W^k mod p
14:     **if** g < 2 **then**, go to step 8
15:     **end if**
16:     Return **VALID** and the value of g
17: **end procedure**

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*. These parameters are very important and part of another module, and we explain this in the next section). The extra n is used a fuel to ensure the termination, or more precisely making Coq type checker accept our definition, *count* not being 16 bit unsigned number because it makes the proofs easy and more general. The function `compute_gen_slow` is more liberal encoding of Algorithm 1 and therefore we mark it as `Local` so that no client, outside of the file where `compute_gen_slow` is defined, can access it. Finally, to emulate the exact behaviour of Algorithm 1, we define another function `compute_generator` that calls `compute_gen_slow` by instantiating n with 2^{16} and *count* with 1. Now the reader can verify that that `compute_generator` is exact encoding of the Algorithm 1, except the details of p, q, *domain_parameter_seed*, and *index*. Now we focus on these details, and recall that we made a claim in the abstract, “usage of the expressive module system to capture all the requirement”. The novelty of our approach is that we define `compute_generator` inside the module `Comp`, shown in the `Listing 3`, that itself takes another module type `Prime`, shown in the `Listing 4`, 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 is the step that forces a client to ensure the correctness. The module type `Prime` has 6 data points, represented as `Parameters`, p, q, k, *domain_parameter_seed*, *ggen*, and *index*. In addition, the assumptions, represented as `Axiom`, are (i) p is a prime, `Axiom prime_p: prime p`, of bit length greater than or equal to 1024, `Axiom p_len: 1024` $\leq$ `N.size p`, (ii) q is a prime prime, `Axiom prime_q: prime q`, of bit length length greater than or equal to 160, `Axiom q_len: 160` $\leq$ `N.size q`, (iii)

Algorithm 2 Validation Routine when the Algorithm 1 is used to compute a generator g. It takes same inputs as Algorithm 1 –two large primes p and q, such that $p = k * q + 1$, *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, used as a marker for generating g– with the generator g itself.

1: **procedure** VALIDATION-OF-VERIFIABLE-GENERATOR($p, q, domain\_parameter\_seed, index, g$)
    ▷ p and q are large prime numbers such that $p = k * q + 1$, the seed used during the generation of p and q, the index used for generating g in A.2.3 (index is a bit string of length 8 that represents an unsigned integer), the value of g to be validated
2:     **Result**: **VALID or INVALID**
3:     **if** *index* is incorrect **then** return **INVALID**
4:     **end if**
5:     **if** g $\notin$ [2, p) **then** return **INVALID**
6:     **end if**
7:     **if** $g^q \neq 1 \pmod{p}$ **then** return **INVALID**
8:     **end if**
9:     N = len(q)
10:     k = (p - 1)/q
11:     count = 0 (* Note: count is an unsigned 16-bit integer.*)
12:     count = count + 1
13:     **if** count = 0 **then** return **INVALID**
14:     **end if**
15:     U = domain_parameter_seed || "ggen" || index || count
16:     W = Hash (U)
17:     computed_g = W^k mod p
18:     **if** computed_g < 2 **then**, go to step 12
19:     **end if**
20:     **if** computed_g = g **then** return **VALID**
21:     **else**return **INVALID**
22:     **end if**
23: **end procedure**

p is a safe prime, `Axiom safe_prime: p = k * q + 1`, (iv) k is greater than or equal to 2, `Axiom k_gteq_2: 2 ≤ k`, (v) *ggen* equal to $0x6767656e$ (it is a part of the FIPS 186-4 specification), `Axiom ggen_hyp: ggen = 0x6767656e`, and (vi) *index* is between 0 and 255 (8 bit unsigned integer), `Axiom index_8bit: 0 ≤ index < 0xff`. We would like to emphasize that in order to construct a concrete instance of the module type `Prime`, shown in Listing 5 `Pins`, a programmer does not need to know any theorem proving. The proof of these axioms are straight forward and only requires three tactics, `lia, vm_compute, reflexivity`, except for proving that p and q are prime numbers because it uses coqprime, which is also mechanical. Overall, a programmer does not need to know anything about theorem proving to compute a group generator.

▶ Remark 1. As a part of this project, we use the NIST test data to show the feasibility of our approach. In our examples, we prove that p and q are prime using the Coq-prime library, but we do not establish that the p and q are generated from the *domain_parameter_seed*. We leave this as a future work.

Listing 1 Generic group generator that takes an extra argument, a unary natural number n, to ensure the termination, or make the Coq type checker accept the definition of `compute_gen_slow`. It is marked as `Local` so no client can call it from outside of the file where it is defined.

```
Local Fixpoint compute_gen_slow (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_slow n' (count + 1) else Valid g
    end.
```

Listing 2 Specialised group generator, which emulates the Algorithm 1, that calls `compute_gen_slow` by instantiating the n with 2^{16} and the *count* with 1. It can be used a client to compute a group generator only if the client can construct an element of module type `Prime`, which captures all the correctness requirements to compute a generator in a verifiable manner.

```
Definition compute_generator :=
    compute_gen_slow (2^16) 1.
```

Listing 3 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).
Section Generator.

  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.

  Inductive Tag : Type :=
  | Invalid : Tag
  | Valid : N -> Tag.


```

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

  Definition compute_generator :=
    compute_gen_slow (2^16) 1.

End Generator.
End Comp.
```

Listing 4 Module type `Prime` that captures all the data and the correctness axioms, specified in `FIPS 186-4, A.2.3`. In order to construct an instance of `Prime`, a client has to instantiate the data, represented as `Parameter`, and discharge all the proofs, represented as `Axiom`. The proofs can be constructed simply by using three tactics `lia, vm_compute, reflexivity` and therefore, no prerequisite of theorem proving is needed to use our library.

```
(* https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf*)
(* Verifiable Canonical Generation of the Generator g  *)
(* We assume that that the prime p q has been
   generated by using  domain_parameter_seed *)

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.
```

Listing 5 Construction of concrete value of the module `Prime` by instantiating the p, q, *domain_parameter_seed*, and *index* by an example, defined in NIST test cases. A programmer does not need know any theorem proving to use our system.

```
Module Pins <: Prime.

(* Large Prime P)
Definition p : N :=
0xff600483db6abfc5b45eab78594b3533d550d9f1bf2a992a7a8da
a6dc34f8045ad4e6e0c429d334eeeaaefd7e23d4810be00e4cc1492
cba325ba81ff2d5a5b305a8d17eb3bf4a06a349d392e00d329744a5
179380344e82a18c47933438f891e22aeef812d69c8f75e326cb70e
a000c3f776dfdbd604638c2ef717fc26d02e17.

(* Large Prime Q *)
Definition q : N :=
0xe21e04f911d1ed7991008ecaab3bf775984309c3.

(* P = k * Q + 1 *)
Definition k : N := Eval vm_compute in N.div (p - 1) q.

(* Domain Parameter Seed, used for generating the prime p and q *)
Definition domain_parameter_seed : N :=
0x180180ee2f0ae4a7b3a1ab1b8414228913ef2911.

```

```
(* GGen *)
Definition ggen : N := 0x6767656e.

(* Index *)
Definition index : N := 0x79.

Theorem prime_p : prime (Z.of_N p).
(* See the primality directory. Admitted
to truly measure the group generation time *)

Theorem p_len : 1024 <= N.size p.

Theorem prime_q : prime (Z.of_N q).

Theorem q_len : 160 <= N.size q.

Theorem k_gteq_2 : 2 <= k.

Theorem safe_prime : p = k * q + 1.

Theorem ggen_hyp : ggen =  0x6767656e.

Theorem index_8bit : 0 <= index < 255.

End Pins.
```

Listing 6 Once a programmer can constructed the concrete instance, `Pins`, of module type `Prime`, she can use the `compute_generator` function to get a concrete generator.

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

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

Listing 7 Coq encoding of generic verification algorithm that checks if a generator is valid or not, depending on the data. It is very similar to the `compute_gen_slow`, except takes an extra parameter g, the generator itself. It is marked as `Local`, so no client can call it from the outside of the file where it is defined.

```
  Local Fixpoint verify_generator_rec (n : nat) (m g : N) : bool :=
    match n with
    | 0%nat => false (* reached the end *)
    | S n' =>
      let U := append_values m in
      let W := sha256_string U in
      let y := Npow_mod W k p in
      if y <? 2 then verify_generator_rec n' (m + 1) g
      else g =? y
    end.
```

Listing 8 Coq encoding of verification algorithm, Algorithm 2, that checks if a generator is valid or not. It does some initial checks to rule out invalid generators. It is very similar to the function `compute_generator`, except the extra argument g.

```
  (* procedure that computes the validity of a generator.
     It checks: (i) 2 <= g < p, (ii) g^q mod p = 1
     (iii) calls the  verify_generator_rec to check g *)
```

```
  Definition verify_generator (g : N) : bool :=
    if negb (andb (2 <=? g) (g <? p)) then false
    else if negb (Npow_mod g q p =? 1) then false
    else verify_generator_rec (2^16) 1 g.
```

We establish, shown in `Listing 9`, that whenever the function `compute_generator` returns a generator g, represented by `Valid g`, then (i) g is in range 2 to $p-1$, $2 \le g < p$, `gen_generator_range`, and the proof is merely a fact that we compute the generator g modulo p, $W^e \pmod p$, (ii) g is the generator the subgroup of order q, `correct_compute_gen`, and the proof a simple corollary of Fermat's little theorem, and (iii) the generation function `compute_gen` and the verification function `verify_generator` agree with each other, `generator_verifier_correctness`, and the proof basically follows from the definition of the generator function and verification function.

Listing 9 Correctness of generator algorithm and its connection with verification algorithm

```
(*
generators are in range 2 <= g < p. We can instantiate
n = 2^16 and m = 1 to get  Valid g = compute_generator
*)
Lemma gen_generator_range : forall (n : nat) (m g : N),
    Valid g = compute_gen_slow n m  -> 2 <= g < p.

(* g generates a subgroup of order q, Fermat's little theorem *)
Lemma correct_compute_gen : forall n m g,
    compute_gen_slow n m = Valid g -> Zpow_mod g q p = 1.


(* Proof that compute_generator and verify_generator agree *)
Lemma generator_verifier_correctness : forall g,
    Valid g = compute_generator <-> verify_generator g = true.
```

3 SHA-256

Now, we focus on the `SHA-256`, required in line 12 of Algorithm 1 and line 16 of Algorithm 2. SHA-256, a cryptographic hash function and specified in NIST, FIPS 180-4, can be informally defined as an easy to compute but hard to invert function that takes a message as input of length l bits, where $0 \le l < 2^{64}$, and produces an output string, message digest, of length 256 bits. To compute the hash of a message M, of length l bits, we append '1' (one) bit followed k '0' (zero) bits at the end of M, where k is the smallest and non-negative solution of the equation $l + 1 + k \equiv 448 \pmod{512}$. Finally, we append 64-bit block that represents the length of l at the end of M. An example that pads the message `"abc"`, shown below and taken from the NIST FIPS 180-4 document. The k here is 423 and l is 24, represented as 64 bits and append in the end.

$$\underbrace{01100001}_{\text{"a"}} \quad \underbrace{01100010}_{\text{"b"}} \quad \underbrace{01100011}_{\text{"c"}} \quad 1 \quad \overbrace{00\ldots00}^{423} \quad \overbrace{00\ldots0\underbrace{11000}_{\ell=24}}^{64}$$

Once we pad the message in this manner, our message length in bits should be divisible by 512, or message length in bytes should be divisible by 64 (in Coq formalism, we establish this formally). Briefly, `SHA-256` shown in `Algorithm 3`, takes a message M and returns the message-digest, or hash value, of M. The very first step of the is to initialise H, a list of 8 32-bit predetermined values according to the specification described in Section 5.3.3 of NIST FIPS 180-4. Next, we pad M, described previously, and break it into N blocks of size 512 bits. We process the each block by splitting them further into 16 32-bit blocks, represented as $M_0^i, M_1^i, ..., M_{15}^i$ and computing the value W_t for $0 \leq t \leq 63$ (line 6 to line 12). When t is less than or equal to 15, we simply copy the value of M_t^i into W_t (line 8), otherwise W_t is computed according some predefined function σ_0, σ_1 (line 10 and addition is defined modulo 2^{32}). Then we initialise the eight working registers a, b, c, d, e, f, g, h, by the values in H (line 14 to line 15). From line 16 to line 27, we compute values according to the previously computed values of eight registers and some predefined functions $\sum_1, \sum_0, Ch, Maj$. Finally. we update H by adding the values of eight registers, line 28 to line 36. Once we finish the processing of all N blocks, we append all the values of H to produce the message-digest of M.

3.1 Coq Formalisation of SHA-256

Now we explain the Coq formalisation. It is fairly standard, just followed the instructions in FIPS 180-4, except some optimisation to ensure that it can compute the hash of a message in reasonable amount of time. We assume that message is represented as list of bytes, in big-endien style – the most significant bit is stored in the left-most bit position. We also assume that message length, in bits, is less than 2^{64}. We pad the message according to the description, we gave in the previous section. We prove that the final padded message length, in bytes, is divisible by 64, `div_64`.

Listing 10 SHA-256 message preparation, as described in the beginning of this section.

```
(* message m is in big endien style *)
Context (m : list byte).

(* Length of message bytes *)
Let n := N.of_nat (List.length m).

(* Length of message, in bits *)
Let mbits := 8 * n.

(* Hypothesis that message length, in bits, is less than 2^64*)
Context {Hn : mbits < 2^64}.

(* Length of to be padded *)
Let k := Z.to_N (Zmod (448 - (Z.of_N mbits + 1)) 512).

(* Number of 64 byte, 512 bits, blocks in the message *)
Let ms := N.div (n + wt + 8) 64.

(* final padded message *)
Definition prepared_message : list byte :=
  m ++ message_padding (N.to_nat wt) ++ message_length_byte.

(* message + padding + message_length should be
   divisible by 64 (byte)*)
Lemma div_64 : N.modulo (n +  wt + 8) 64 = 0.
```

Algorithm 3 SHA-256

procedure SHA-256 PROCEDURE(M) ▷
  Result: **SHA-256** hash value of message M
  Set the initial hash value, H^0 (see the Listing 11, Definition H0)
  Pad the message M, describe above, and split it N blocks, $M^{(1)}, M^{(2)}, ..., M^{(N)}$
  for i = 1 to N **do**
    for t = 0 to 63 **do**
      if t $\leq$ 15 **then**
        $W_t = M_t^i$
      else
        $W_t = \sigma_1(W_{t-1}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$
      end if
    end for
    Initialise eight working registers a, b, c, d, e, f, g, and h with
    $a = H_0^{i-1}$, $b = H_1^{i-1}$, $c = H_2^{i-1}$, $d = H_3^{i-1}$,
    $e = H_4^{i-1}$, $f = H_5^{i-1}$, $g = H_6^{i-1}$, $h = H_7^{i-1}$
    for t := 0 to 63 **do**
      $T_1 = h + \sum_1(e) + Ch(e, f, g) + K_t + W_t$
      $T_2 = \sum_0(a) + Maj(a, b, c)$
      $h = g$
      $g = f$
      $f = e$
      $e = d + T_1$
      $d = c$
      $c = b$
      $b = a$
      $a = T_1 + T_2$
    end for
    Compute i^{th} intermediate hash value, H^i
    $H_0^i = a + H_0^{i-1}$
    $H_1^i = b + H_1^{i-1}$
    $H_2^i = c + H_2^{i-1}$
    $H_3^i = d + H_3^{i-1}$
    $H_4^i = e + H_4^{i-1}$
    $H_5^i = f + H_5^{i-1}$
    $H_6^i = g + H_6^{i-1}$
    $H_7^i = h + H_7^{i-1}$
  end for
  We produce the SHA-256 message-digest of the Message M by
  by concatenating $H_0^N||H_1^N||H_2^N||H_3^N||H_4^N|H_5^N||H_6^N||H_7^N$
end procedure

Listing 11 SHA-256 message digest calculation. We encode the for loop, from line 5 to line 37 in Alorithm 3, as fold_left primitive.

```
  Definition sha256 :=
    List.fold_left
    (fun H i => sha256_intermediate (W i) H)
    (upto_n (N.to_nat ms)) H0.

  (* 8 32-bit predefined list of values *)
  Definition H0 : list N := [
    0x6a09e667; 0xbb67ae85; 0x3c6ef372; 0xa54ff53a; 0x510e527f;
    0x9b05688c; 0x1f83d9ab; 0x5be0cd19].


(* Steps 2, 3, and 4 in section 6.2.2. *)
  Definition sha256_intermediate (W : list N) (H : list N) :=
    (* step 2 *)
    let a := nth 0 H 0 in
    let b := nth 1 H 0 in
    (* omitted *)
    let h := nth 7 H 0 in
    (* Step 3 *)
    let '(a, b, c, d, e, f, g, h) :=
    List.fold_left
    (fun '(a, b, c, d, e, f, g, h) t =>
      let T1 := h + (Sigma1 e) + (Ch e f g) +
                (nth t K 0) + (nth t W 0) in
      let T2 := (Sigma0 a) + (Maj a b c) in
      (* omitted *)
      let a := T1 + T1 in
      (a, b, c, d, e, f, g, h))
    (upto_n 64)
    (a, b, c, d, e, f, g, h)
    in
    (* step 4: compute Hi *)
    [a + (nth 0 H 0); b + (nth 1 H 0); ... (* omitted *)].
```

4 Fermat's Little Theorem

Fermat's little theorem is stated in two flavours (i) for any integer a, when p is prime then $a^p \equiv a \pmod{p}$, (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 [1]. 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 (mod p), except the first term, $\binom{p}{0} * a^p$, and the last term, $\binom{p}{p} * 1^p$.

4.1 Coq formalisation

Fermat's little theorem has been formalised number of times in various theorem provers, e.g., Chan and Norrish [8] in HOL4 [30], Théry and Hanrot in Coq [34], Boyer and Moore [6] in

ACL [7]. All these formalisation, including ours, are based on the proof of Euler, except [8] which is based on counting necklaces [17].

Listing 12 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 $a^{p-1} \equiv 1 \pmod{p}$

```
Lemma fermat_little_simp : forall a p : nat,
   prime (Z.of_nat p) ->
   Nat.modulo (Nat.pow a p) p = Nat.modulo a p.


Theorem fermat_little_coprime : forall (a p : nat),
   prime (Z.of_nat p) -> Nat.modulo a p <> 0 ->
   Nat.modulo (Nat.pow a (p - 1)) p = 1.
```

Listing 13 The theorem `prime_pow_exp` encodes the fact that for any integer a and prime p, $(a+1)^p = a^p + 1 \pmod{p}$. The proof hinges on the `binom_mod_p_bound`, which encodes the fact that every terms is $0 \pmod{p}$, except the first and last term.

```
(* (x + 1)^p % p = x^p + 1 (mod p) *)
Lemma prime_pow_exp : forall (a p : nat),
   prime (Z.of_nat p) ->
   Nat.modulo (Nat.pow (a + 1) p) p =
   Nat.modulo (Nat.pow a p + 1) p.

(* pCk mod p = 0 for 1 <= k < p *)
Lemma binom_mod_p_bound : forall p k : nat,
   prime (Z.of_nat p) -> k <= p ->
   Nat.modulo (binomial_exp p k) p = 0 <->
   1 <= k < p.

(* Definition of binomial coefficient *)
Fixpoint binomial_exp (n k : nat) : nat :=
  match n with
  | O => match k with
   | O => S O
   | S _ => O
   end
  | S n' => match k with
   | O => S O
   | S k' => binomial_exp n' k + binomial_exp n' k'
   end
  end.
```

5 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. 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. As we mentioned in abstract, the Coq evaluation mechanism takes 1 minute for 1024 bit prime, 10 minutes for 2048 bit prime, and 30 minutes for 3072 bit prime. We can extract an OCaml code from Coq formalisation and compile it to machine executable to get speed up, but it is not ideal and acceptable from the election security perspective. Therefore, we do not recommend extracting OCaml code because it enlarges the trusted computing base, the OCaml compiler. One way to get speedup is by implementing it in CakeML [23] that is guaranteed to be correct to the machine level.

6 Related Work

Our work is highly influence by coqprime [34], certifying prime nubmers in Coq theroem prover, and treating theorem provers as a computation tool. In a very similar spirit of coqprime, we compute group generator in Coq theorem prover. The usage of theorem prover in electronic voting is growing. Cortier, Filipiak, and Lallemand [10] formally verified the proof of privacy, receipt-freeness, and verifiability of a voting scheme BeleniosVS using the tool ProVerif [4]. Linear logic [16] has been used by DeYoung and Schürmann [11] to model the different entities in electronic voting as a resource. Pattinson and Schürmann [26] first proposed the idea of vote counting as mathematical proof. Pattinson and Tiwari formalised Schulze method [27] in Coq and extracted OCaml code to compute Schulze winner on a ballots. Haines, Goré, and Tiwari [18] developed a formally verified checker in Coq and extract OCaml code to verify the correctness of IACR election. To hide the preferences in a ballot, Haines, Pattinson, and Tiwari [21] developed a homomorphic Schulze method. Ghale, Goré, Pattinson, and Tiwari [13, 14] developed single transferable vote, used in Australia, in Coq and extracted Haskell code to count real world ballots. Ghale, Pattinson, Norrish, and Kumar [15] developed a certified checker for family of STV algorithm by specifying it in HOL4 and then obtain the machine executable versions for the tools by relying on the verified proof translator and the compiler of CakeML [23]. Haines, Goré, and Sharma [19] developed a formally verified mixnet, used in electronic voting, in Coq.

7 Future Work

The goal of our work is ensure the correctness and verifiability of electronic voting bootstrap phase. In this work, we assumed that the prime p and q have been generated using the seed, *domain_parameter_seed*. In future, we would like generate the *domain_parameter_seed* in some verifiable way and compute the primes p and q from it. [20] has already suggested,“In the NIST standard (Algorithm 1) a sufficiently controlled domain parameter seed, for instance the name of the election, combined with a hash function with sufficient domain would seem acceptable”, but we would like to investigate more on this.

References

1. Proofs of Fermat’s Little Theorem. avaliable via `https://en.wikipedia.org/wiki/Proofs_of_Fermat%27s_little_theorem`, retrieved February 8, 2022.
2. 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.
3. Stephanie Bayer and Jens Groth. Efficient Zero-Knowledge Argument for Correctness of a Shuffle. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 263–280. Springer, 2012.

4 Bruno Blanchet. *Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif*, pages 54–87. Springer International Publishing, Cham, 2014. doi:10.1007/978-3-319-10082-1_3.

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

6 Robert S Boyer and J Strother Moore. Proof Checking the RSA Public Key Encryption Algorithm. *The American Mathematical Monthly*, 91(3):181–189, 1984.

7 Robert S Boyer and J Strother Moore. *A Computational Logic*. Academic press, 2014.

8 Hing Lun Chan and Michael Norrish. A String of Pearls: Proofs of Fermat's Little Theorem. *Journal of Formalized Reasoning*, 6(1):63–87, Jan. 2013. URL: https://jfr.unibo.it/article/view/3728, doi:10.6092/issn.1972-5787/3728.

9 Andrew Conway, Michelle Blom, Lee Naish, and Vanessa Teague. An Analysis of New South Wales Electronic Vote Counting. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3014812.3014837.

10 Véronique Cortier, Alicia Filipiak, and Joseph Lallemand. BeleniosVS: Secrecy and Verifiability Against a Corrupted Voting Device. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 367–36714, 2019. doi:10.1109/CSF.2019.00032.

11 Henry DeYoung and Carsten Schürmann. Linear logical voting protocols. In Aggelos Kiayias and Helger Lipmaa, editors, *Proc. VoteID 2011*, volume 7187 of *Lecture Notes in Computer Science*, pages 53–70. Springer, 2012.

12 Pierrick Gaudry and Alexander Golovnev. 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*, page 32–49, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-51280-4_3.

13 Milad K. Ghale, Rajeev Goré, and Dirk Pattinson. A Formally Verified Single Transferable Voting Scheme with Fractional Values. In Robert Krimmer, Melanie Volkamer, Nadja Braun Binder, Norbert Kersting, Olivier Pereira, and Carsten Schürmann, editors, *Electronic Voting*, pages 163–182, Cham, 2017. Springer International Publishing.

14 Milad K Ghale, Rajeev Goré, Dirk Pattinson, and Mukesh Tiwari. Modular formalisation and verification of STV algorithms. In *International Joint Conference on Electronic Voting*, pages 51–66. Springer, 2018.

15 Milad K. Ghale, Dirk Pattinson, Ramana Kumar, and Michael Norrish. Verified Certificate Checking for Counting Votes. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments*, pages 69–87, Cham, 2018. Springer International Publishing.

16 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

17 Solomon W Golomb. Combinatorial proof of Fermat's "little" theorem. *The American Mathematical Monthly*, 63(10):718, 1956.

18 Thomas Haines, Rajeev Goré, and Mukesh Tiwari. Verified Verifiers for Verifying Elections. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 685–702, 2019.

19 Thomas Haines, Rajeev Goré, and Bhavesh Sharma. did you mix me? formally verifying verifiable mix nets in electronic voting. In *2021 IEEE Symposium on Security and Privacy (SP)*.

20 Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to Prove your Election Outcome. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 644–660, 2020. doi:10.1109/SP40000.2020.00048.

21 Thomas Haines, Dirk Pattinson, and Mukesh Tiwari. Verifiable Homomorphic Tallying for the Schulze Vote Counting Scheme. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 36–53. Springer, 2019.

22 J. Alex Halderman and Vanessa Teague. 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, page 35–53, Berlin, Heidelberg, 2015. Springer-Verlag. doi:10.1007/978-3-319-22270-7_3.

23 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179–191, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2535838.2535841.

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

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 Dirk Pattinson and Carsten Schürmann. Vote Counting as Mathematical Proof.

27 Dirk Pattinson and Mukesh Tiwari. Schulze Voting as Evidence Carrying Computation. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, pages 410–426, Cham, 2017. Springer International Publishing.

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

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

30 Konrad Slind and Michael Norrish. A brief overview of hol4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

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

32 Michael A. Specter, James Koppel, and Daniel Weitzner. 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)*, pages 1535–1553. USENIX Association, August 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/specter.

33 The Coq Development Team. The coq proof assistant, version 8.10.0, October 2019. doi:10.5281/zenodo.3476303.

34 Laurent Théry and Guillaume Hanrot. Primality Proving with Elliptic Curves. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics*, pages 319–333, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.