Pushing the Computational Limit of the Coq Theorem Prover

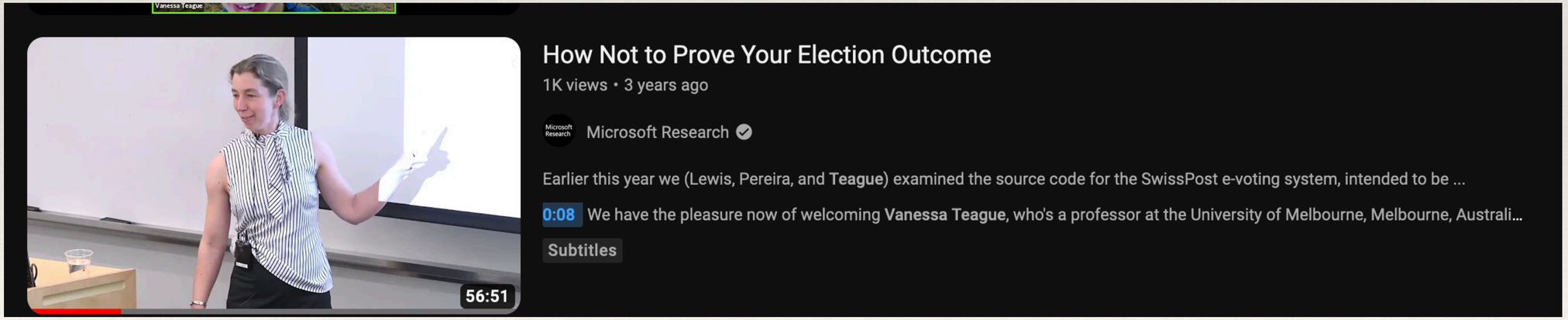
Mukesh Tiwari,
University of Cambridge,
Cambridge

UNIVERSITY OF CAMBRIDGE

How not to prove your election outcome

Publisher: IEEE Cite This PDF

Thomas Haines ; Sarah Jamie Lewis ; Olivier Pereira ; Vanessa Teague **All Authors**



Problem

We assume two large primes p and q such that $p = 2 * q + 1$ and a generator g such $g^q \equiv 1(\text{mod p})$

If I give you another generator h such that $h^q \equiv 1(\text{mod p})$, how do you know I do not know the discrete logarithm relation between g and h

Generation Algorithm (A.2.3) (FIPS 186-4)

1: **procedure** VERIFIABLE-GENERATOR-PROCEDURE($p, q, domain\_parameter\_seed, index$)
2: **Result**: *status*, g
3: **if** *index* is incorrect **then** return **INVALID**
4: **end if**
5: N = length of q
6: $k = (p-1)/q$
7: $count = 0$
8: $count = count + 1$
9: **if** $count = 0$ **then** return **INVALID**
10: **end if**
11: $U = domain\_parameter\_seed \,||\, \text{“ggen”} \,||\, index \,||\, count$
12: $W = \text{Hash}(U)$
13: $g = W^k \bmod p$
14: **if** $g < 2$ **then** go to step 8
15: **end if**
16: return **VALID** and the value of g
17: **end procedure**

Contribution

- Encoding of Generation Algorithm (A.2.3) and Verification (A.2.4) Algorithm
- Encoding of Fermat's Little Theorem
- Encoding of SHA-256

Encoding of Generation and Verification Algorithm (Easy Part)

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


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

Encoding of Generation and Verification Algorithm (Easy Part)

```
(* https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf*)
(* Verifiable Canonical Generation of the Generator g  *)
(* Input p, q, and domain_seed *)
(* We assume that that the prime p q has been validated and
   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.
```

Encoding of Generation and Verification Algorithm (Easy Part)

Instantiate Parameters with Data

```
Module Pins <: Prime.
Definition p : N := (* Prime *)
Definition q : N := (* Prime *)
Definition k : N := Eval vm_compute in N.div (p - 1) q.
Definition domain_parameter_seed : N :=  (* Domain Parameter Seed *)
Definition ggen : N := 0x6767656e.
Definition index : N := (* Index *)
```

Encoding of Generation and Verification Algorithm (Easy Part)

Discharge all Axioms

```
Theorem prime_p : prime (Z.of_N p).
(* Discharge the proof *)

Theorem p_len : 1024 <= N.size p.
(* Discharge the proof *)

Theorem prime_q : prime (Z.of_N q).
(* Discharge the proof *)
```

```
Theorem q_len : 160 <= N.size q.
(* Discharge the proof *)

Theorem k_gteq_2 : 2 <= k.
(* Discharge the proof *)

Theorem safe_prime : p = k * q + 1.
(* Discharge the proof *)
```

Encoding of Generation and Verification Algorithm (Easy Part)

Construct a Module

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

(* Call the function *)
Time Eval vm_compute in genr.compute_generator.
```

Proof of Correctness (Moderately Difficult Part)

$$g^q \equiv 1 (\text{mod } p)$$

```
Lemma correct_compute_genertor :
  forall g,  Valid g = compute_generator ->
  Zpow_mod (Z.of_N g) (Z.of_N q) (Z.of_N p) = 1%Z.
Proof.
```

Proof of Correctness (Moderately Difficult Part)

Generation and Verification algorithms agree with each other.

```
Lemma generator_verifier_correctness : forall g,
  verify_generator g = true <-> Valid g = compute_generator.
Proof.
```

Computing Generators inside Coq (Most Difficult Part)

```
Time Eval vm_compute in genr.compute_generator.


Finished transaction in 1479.475 secs (1478.121u,1.187s) (successful)
= genr.Valid
5142269046611298751053334471712018380952381659994149061069643310780863411117756830385169625171306902268731945265611175832982085711414921325430792621067001642054430322541316695452318115167970016625546426097672467620839422473921709026039725510992923909031023192174059668984872166791847785098121010517945765446961570007177822942898941331888153276962092525360573272974634978013864168046512739478574948414329466141821447534309755925230066981468451347043551719324995849899067726237010049836515585927730058820192474169847466483361570985526862054236113750881396542483192524341448209408531022129289703356623973794545446814225059247328981961060656844942121203760003864474654437004736370165730970677136513883966628087789535547496889264530799006158784007444380513118730444715144566680249530571284451878484157389189765788664403469491470323677756901984779279566477157877271477923138873212496519999904917186456515597600968771835406001603690
     : genr.Tag
```

Evaluation

We use Coq's evaluation mechanism to compute the group generator, thereby only trusting the Coq codebase.

- 1 minutes for 1024 bit prime
- 10 minutes for 2048 bit prime
- 30 minutes for 3072 bit prime

Best Part

You do not need to know Coq to use this software to compute generators!

Conclusion

In this work, I showed how to use the Coq theorem prover to implement the group generator algorithm (A.2.3), prove its correctness, evaluate it inside the theorem prover itself.

To bootstrap an election, election commission —or any responsible authority— can use my software to produce generators. They just need to publish the Coq scripts for every generator.

An auditor can audit the claims of the election commission by running the Coq scripts on their own computer