Formally verifying a pseudo-random number generator

Katherine Ye, advised by Prof. Andrew W. Appel[1] and and Prof. Matthew Green[2]
[1] *Princeton University*
[2] *Johns Hopkins University*

We have proved, with machine-checked proofs, that the pseudorandom output produced by HMAC-DRBG is indistinguishable from random by a computationally bounded adversary. We proved this (@?) about a high-level specification of HMAC-DRBG, (@in what language?) provided by the Foundational Cryptography Framework (FCF), which is embedded in the Coq proof assistant. We also plan to prove that HMAC-DRBG is backtracking-resistant and prediction-resistant. After these proofs are done, we will prove equivalence between our high-level specification of HMAC-DRBG and a different specification used to prove its functional correctness of the mBED TLS C implementation of HMAC-DRBG. This will allow our proofs of cryptographic security properties to transfer to the C implementation of HMAC-DRBG.

I. INTRODUCTION

Most modern cryptosystems rely on random numbers, which they use to generate secrets that need to be known to users and unknown and unpredictable to attackers. Reducing the entropy of a cryptosystem's pseudo-random number generator (PRNG) is an easy way to break the entire cryptosystem. The PRNG will generate output predictable to an attacker, allowing her to guess private keys, yet the bits may still look random. Plus, the rest of the cryptosystem will function normally, since PRNGs tend to be single self-contained components. These two factors make PRNGs very attractive targets to attackers.

The attack has indeed happened in practice (through due to a programming error and not malice), and with devastating consequences. Luciano Bello discovered that the random number generator in Debian OpenSSL, a widely-used cryptographic library, was predictable, allowing attackers to easily guess keys. Debian advised all users to regenerate keys, though some high-profile users didn't. For example, compromised SSH keys were used to access Spotify, Yandex, and gov.uk's public repositories on GitHub.

Despite the importance of PRNGs, surprisingly little work exists on proving them secure, either by proving on paper that certain widely-used PRNGs are secure, or by verifying with computer-checked proofs that implementations of these PRNGs satisfy their specifications.

Our project aims to do both. We aim prove correctness and security of an OpenSSL implementations of two widely-used PRNGs, CTR-DRBG and HMAC-DRBG. We will do this by proving that the implementation satisfies a functional specification (a high-level implementation in code) of the random number generator, which we trust satisfies its paper specification. Then we will prove that the functional specification guarantees the expected cryptographic properties, most notably that a computationally bounded attacker cannot distinguish generated bits from random bits. These properties will flow from the functional spec down to the compiled code via a verified compiler, CompCert. The entire verification will be machine-checked by Coq, a trusted proof assistant.

This thesis attacks the cryptographic aspect of the problem. We hope to prove that PRNGs called CTR-DRBG and HMAC-DBG generate bits that are indistinguishable from random, that the PRGs are backtracking-resistant, that they eventually recover from compromises of internal state, and other important properties. We will write functional specifications of these PRGs in the environment of the Coq proof assistant, then prove that they possess the desired properties by using the Foundational Cryptography Framework in Coq.

(@@@ revise last paragraphs)

II. STATEMENT OF MAIN RESULT

Thing

III. PROBLEM BACKGROUND AND RELATED WORK

(@fix these citations)

Andrew Appel's group has done the most significant related work in the area. Appel (2015) [1] presents a full formal machine-checked verification of a C program: the OpenSSL implementation of SHA-256. Appel et al. (2015) [2] build on this work to do the same for HMAC, adding a proof of security that relies on the security of SHA. We plan to use the same approach for CTR-DRBG and HMAC-DRBG.

In addition, there exist paper proofs of the security of CTR-DRBG [3] and HMAC-DRBG [4], though they haven't appeared in peer-reviewed venues.

In the area of checking game-based proofs of cryptographic security within a proof assistant, there are two main tools: EasyCrypt and its cousin CertiCrypt (neither of which is foundational), and the Foundational Cryptography Framework. [5]

In the general area of formalizing PRNGs, several crypto papers analyze the security of PRNGs and propose new security properties, e.g. Dodis et al. [6] who propose the robustness property and show that the built-in Linux PRNG, /dev/random, is not robust.

There's not much prior work on formal verification of PRNGs in our style. Drre and Klebanov (2015) [7] focus on verifying that a PRNG uses all its entropy. They perform this logic-based information flow verification using the KeY system for Java, which uses symbolic execution. This only defends against one particular attack (that of squandering entropy) and does not guarantee functional correctness or other security properties we may care about, such as indistinguishability from randomness and backtracking resistance.

Affeldt (2009) [8] do include game-playing proofs of provable security in Coq. However, they focus on doing so directly on an assembly implementation of a PRNG, not on a high-level functional specification. They also verified their own assembly implementation, not a widely-used existing one. One advantage of this approach is that it avoids mismatches between the functional specification of the C code and the functional specification used for cryptographic proofs.

Our approach is unique because it provides an end-to-end and foundational verification that guarantees both correctness and security. Our stack consists of Coq, the Foundational Cryptography Framework, the Verified Software Toolchain (using separation logic), and CompCert (a verified C compiler). In addition, we verify an existing widely-used PRG implementation in C, making our approach more useful in practice.

IV. HMAC-DRBG OVERVIEW

A pseudorandom number generator is used to stretch a small amount of randomness into a large amount of pseudo-randomness, often for use in cryptosystems. HMAC-DRBG, formalized in NIST SP 800-90A [cite], is one such pseudorandom number generator. It generates output by iterating HMAC, a keyed-hash message authentication function widely believed to be difficult to invert [cite] and proven to be a pseudo-random function (PRF) given that HMACs internal hash function is a PRF. [mention key?]

HMAC-DRBG possesses an internal state consisting of four elements:

The working state, consisting of an internal value V which

HMAC-DRBG definition.

State: Secret (K, V), Public (?, prediction resistance, reseed counter)

Instantiate:

```
K <-
V <- ...
```

Generate

Update

Reseed

Entropy

V. INFORMAL PROOF

Informal theorem.

Proof.

VI. FCF SUMMARY

Embedding into Coq Game-based crypto proofs Comp monad (probabilistic computations) Relating pairs of games using probabilistic relational Hoare logic (postconditions, *comp_spec*) Oracles, OracleComp Adversary as abstract probabilistic polynomial time algorithm Writing games

VII. FORMAL PROOF OUTLINE

A. Starting HMAC-DRBG definitions

```
Gen_loop

GenUpdate


Definition GenUpdate_original (state : KV) (n : nat)
     :
  Comp (list (Bvector eta) * KV) :=
  [k, v] <-2 state;
  [bits, v'] <-2 Gen_loop k v n;
  k' <- f k (to_list v' ++ zeroes);
  v'' <- f k' (to_list v');
  ret (bits, (k', v'')).
```

B. Prior work on abstract PRF-DRBG

Definition

```
  Fixpoint PRF_DRBG_f (v : D)(n : nat)(k : Key) :=
    match n with
        | O => nil
        | S n' =>
          r <- (f k v);
            r :: (PRF_DRBG_f (injD r) n' k)
    end.

  Fixpoint PRF_DRBG_f (v : D)(n : nat)(k : Key) :=
    match n with
        | O => nil
        | S n' =>
          r <- (f k v);
            r :: (PRF_DRBG_f (injD r) n' k)
    end.
```

```
(* save the last v and output it as part of the
    state *)
Fixpoint Gen_loop (k : Bvector eta) (v : Bvector eta
    ) (n : nat)
  : list (Bvector eta) * Bvector eta :=
  match n with
  | O => (nil, v)
  | S n' =>
    let v' := f k (Vector.to_list v) in
    let (bits, v'') := Gen_loop k v' n' in
    (v' :: bits, v'')
  end.
```

Random bits to random function

Random function to collision

C. Extending prior work

To deal with multiple calls Non-adaptive adversary V updating Key updating Instantiate

D. Formal theorem

```
Theorem G1_G2_close :
  | Pr[G1_prg_original] - Pr[G2_prg] | <=
  (numCalls / 1) * Gi_Gi_plus_1_bound.
Proof.
  rewrite G1_Gi_O_equal.
  rewrite G2_Gi_n_equal.
  (* inductive argument *)
  specialize (distance_le_prod_f
        (fun i => Pr[Gi_prg i])
        Gi_Gi_plus_1_close numCalls).
  intuition.
Qed.
```

Pseudorandom functions, random functions, random bits

Indistinguishability definition

E. Hybrid argument

F. Main games

V updating

```
(* [GenUpdate_original, GenUpdate_original, ...] = [
    GenUpdate_noV, GenUpdate, Genupdate, ...] *)
  (* use this for the first call *)
Definition GenUpdate_noV (state : KV) (n : nat) :
  Comp (list (Bvector eta) * KV) :=
  [k, v] <-2 state;
  [bits, v'] <-2 Gen_loop k v n;
  k' <- f k (to_list v' ++ zeroes);
  ret (bits, (k', v')).

Definition GenUpdate (state : KV) (n : nat) :
  Comp (list (Bvector eta) * KV) :=
  [k, v] <-2 state;
  v' <- f k (to_list v);
  [bits, v''] <-2 Gen_loop k v' n;
  k' <- f k (to_list v'' ++ zeroes);
  ret (bits, (k', v'')).
```

Adversary

```
(* Non-adaptive adversary. *)
Variable A : list (list (Bvector eta)) -> Comp bool.
Hypothesis A_wf: forall ls, well_formed_comp (A ls).
```

Game 1

```
(* blocks generated by GenLoop *)
Variable blocksPerCall : nat.
(* number of calls to GenUpdate *)
Variable numCalls : nat.
Hypothesis H_numCalls : numCalls > 0.

Definition maxCallsAndBlocks : list nat := replicate
     numCalls blocksPerCall.

(* only first call uses GenUpdate_noV; assumes
    numCalls > 0 *)
Definition G1_prg : Comp bool :=
  [k, v] <-$2 Instantiate;
  [head_bits, state'] <-$2 GenUpdate_noV (k, v)
      blocksPerCall;
  (* call the oracle numCalls times, each time
      requesting blocksPerCall blocks *)
  [tail_bits, _] <-$2 oracleMap _ _ GenUpdate state'
       (tail maxCallsAndBlocks);
  A (head_bits :: tail_bits).
```

Game 2

```
(* simpler version of GenUpdate only requires
    compMap. prove the two games equivalent *)
Definition G2_prg : Comp bool :=
  [k, v] <-$2 Instantiate;
  bits <-$ compMap _ GenUpdate_rb maxCallsAndBlocks;
  A bits.
```

Game i and PRF adversary (see long comment above this section)

```
Definition PRF_Adversary (i : nat) : OracleComp
    Blist (Bvector eta) bool :=
  bits <--$ oracleCompMap_outer _ _ (Oi_oc' i)
      maxCallsAndBlocks;
  $ A bits.

(* ith game: use RF oracle *)
Definition Gi_rf (i : nat) : Comp bool :=
  [b, _] <-$2 PRF_Adversary i _ _ (randomFunc
      ({0,1}^eta) eqdbl) nil;
  ret b.
```

Oracle i: decides which oracle to use on the ith call

```
(* same as Oi_prg but each GenUpdate in it has been
    converted to OracleComp *)
(* number of calls starts at 0 and ends at q. e.g.
G1:     RB  PRF PRF
Gi_rf 1: RB RF  PRF (i = 1 here)
G2:     RB  RB  PRF *)
(* number of calls: first call is 0, last call is (
    numCalls - 1) for numCalls calls total
G0: PRF PRF PRF <-- Gi_prf 0
    RF  PRF PRF <-- Gi_rf 0
G1: RB  PRF PRF <-- Gi_prf 1
    RB  RF  PRF <-- Gi_rf 1
G2: RB  RB  PRF
    RB  RB  RF
G3: RB  RB  RB  <-- note that there is no oracle slot
     to replace here
    RB  RB  RB  <-- likewise
there should be (S numCalls) games, so games are
    numbered from 0 through numCalls *)
Definition Oi_oc' (i : nat) (sn : nat * KV) (n : nat
    )
  : OracleComp Blist (Bvector eta) (list (Bvector
      eta) * (nat * KV)) :=
  [callsSoFar, state] <-2 sn;
  [k, v] <-2 state;
  let GenUpdate_choose :=
      if lt_dec callsSoFar i (* callsSoFar < i *)
      then GenUpdate_rb_intermediate_oc
      else if beq_nat callsSoFar i (* callsSoFar = i
           *)
           then GenUpdate_oc  (* uses provided oracle
                (PRF or RF) *)
      else if beq_nat callsSoFar 0
           then GenUpdate_noV_oc (* first call does
               not update v *)
      else GenUpdate_PRF_oc in     (* uses PRF with
           (k,v) updating *)
  [bits, state'] <--$2 GenUpdate_choose (k, v) n;
  $ ret (bits, (S callsSoFar, state')).
```

Hybrid argument notation
Replace PRF with random function
Replace random function with random bits

G. List of lemmas

From the top down (roughly breadth-first traversal of the proof tree).

For the top-level theorem,

```
G1_G2_close : | Pr[G1_prg] - Pr[G2_prg] | <=
   (numCalls / 1) * Gi_Gi_plus_1_bound.
```

1. GenUpdate_v_output_probability :
   Pr[G1_prg_original] == Pr[G1_prg].

   If we move each v-update to the beginning of the next *GenUpdate* call, the games are equivalent, since the output the adversary sees is exactly the same. The rest of the proof will be done on the modified *GenUpdate*s.

2. G1_Gi_O_equal :
   Pr[G1_prg] == Pr[Gi_prg O].

   Recall that *G1* is the first game we defined. It simulates "worst-case" real-world use of HMAC-DRBG by a non-adaptive adversary by calling *GenUpdate* the maximum number of times, requesting the maximum number of blocks, and passing the output to the adversary. Since this is modeling real-world use, every call to *GenUpdate* uses HMAC (abstracted to be any PRF). This game is equivalent to the first hybrid, where every call to the *GenUpdate*$_o$*c* oracle uses the PRF.

3. G2_Gi_n_equal :
   Pr[G2_prg] == Pr[Gi_prg numCalls].

   Recall that *G2* is the second game we defined. It simulates how we would ideally like HMAC-DRBG to behave. It calls *GenUpdate_rb* the maximum number of times, requesting the maximum number of blocks, and passing the output to the adversary. *GenUpdate_rb* is a version of *GenUpdate* with every call to HMAC (the PRF) replaced by uniformly sampling a random bitvector. This game is equivalent to the last hybrid, where every call to the *GenUpdate_oc* oracle uses uniform random sampling.

4. Gi_Gi_plus_1_close :
   forall (n : nat),
   | Pr[Gi_prg n] - Pr[Gi_prg (S n)] | <=
   Gi_Gi_plus_1_bound.

   This is the important part of the proof. We prove that the difference between each adjacent hybrid is bounded by some constant, defined as such:

   ```
   Gi_Gi_plus_1_bound := PRF_Advantage_i +
       Pr_collisions.
   ```

   Adam's existing lemma handles the rest of the work, yielding the result that the difference between the first and the last hybrid is at most this bound times the number of hybrids.

For *Gi_Gi_plus_1_close*, number four above:

1. Gi_normal_prf_eq : forall (i : nat),
   Pr[Gi_prg i] == Pr[Gi_prf i].

2. Gi_prf_rf_close_i : forall (i : nat),
   | Pr[Gi_prf i] - Pr[Gi_rf i] | <=
   PRF_Advantage_Game i.

3. forall (i : nat), (* not true for i = 0 (and not needed) *)
   | Pr[Gi_rf i] - Pr[Gi_prg (S i)] | <=
   Pr_collisions.

For PRF Advantage (Gi_prf_rf_close), number two above:

1. Gi_prf_rf_close_i : forall (i : nat),
   | Pr[Gi_prf i] - Pr[Gi_rf i] | <=
   PRF_Advantage_Game i.

2. PRF_Advantages_lt : forall (i : nat),
   PRF_Advantage_Game i <= PRF_Advantage_Game 0.

Note on "identical until bad."

For the Bad event, collisions, or Gi_rf_rb_close, number three for Gi_Gi_plus_1_close above:

1. Gi_normal_rb_eq : forall (i : nat),
   Pr[Gi_prg (S i)] == Pr[Gi_rb i].

   Put Gi_prg into the same form using RB oracle

2.
   ```
   Gi_rf_return_bad_eq : forall (i : nat),
   Pr[Gi_rf i] == Pr[x <-$ Gi_rf_bad i; ret fst x].
   ```

   Expose the bad event

3.
   ```
   Gi_rb_return_bad_eq : forall (i : nat),
       Pr[Gi_rb i] == Pr[x <-$ Gi_rb_bad i; ret fst x].
   ```

   Expose the bad event

4.
   ```
   Gi_rb_rf_identical_until_bad : forall (i:nat),
   | Pr[x <-$ Gi_rf_bad i; ret fst x]
     - Pr[x <-$ Gi_rb_bad i; ret fst x] |
   <= Pr[x <-$ Gi_rb_bad i; ret snd x]
   ```

   Applying the fundamental lemma here

5.
   ```
   Gi_rb_bad_collisions : forall (i : nat),
   Pr  [x <-$ Gi_rb_bad i; ret snd x ] <=
      Pr_collisions.
   ```

   The probability of bad event happening in RB game is bounded by the probability of collisions in a list of length (n+1) of randomly-sampled (Bvector eta)

   Calculating probability of bad event: Applying Adam's collision bound

To prove `Gi_rb_rf_identical_until_bad` (number two above):

1.
   ```
   fundamental_lemma_h : forall (A : Set)
      (eqda : EqDec A) (c1 c2 : Comp (A * bool)),

      Pr  [x <-$ c1; ret snd x ] ==
      Pr  [x <-$ c2; ret snd x ] ->

      (forall a : A, evalDist c1 (a, false) ==
                       evalDist c2 (a, false)) ->

      forall a : A,
      | evalDist (x <-$ c1; ret fst x) a -
        evalDist (x <-$ c2; ret fst x) a |
         <= Pr  [x <-$ c1; ret snd x ].
   ```

   Cite Bellare

   dupsInIthInputCalls

   Bad event:

2.
   ```
    Gi_rb_rf_return_bad_same : forall (i : nat),
       Pr  [x <-$ Gi_rb_bad i; ret snd x ] ==
       Pr  [x <-$ Gi_rf_bad i; ret snd x ].
   ```

   See comments on this theorem Assumption 1: two games have same probability of returning bad

3.
   ```
    Gi_rb_rf_no_bad_same : forall (i : nat) (a : bool),
      evalDist (Gi_rb_bad i) (a, false) ==
      evalDist (Gi_rf_bad i) (a, false).
   ```

   Assumption 2: if bad event does not happen, the distributions of the outputs are identical Oracle identical until bad

H. Proof graph

Thing

I. Current limitations of the formal proof

Parts of HMAC-DRBG we abstracted and ignored

Instantiate (randomly-sampled key; entropy extractor) Reseeding (prediction resistance) Adaptive adversary Additional input Entropy failure

J. Comparison to existing proof of HMAC-DRBG security

Hirose stuff

how does it compare to our paper proof? our computer proof? Why is formalizing our proof so much work? (Number of hours and lines of code?)

K. Comments on HMAC-DRBG's design

Re-keying the PRF with a length-extended input: good Updating the v after re-keying the PRF: bad (++ [v]) instead of (v ::) bad

VIII. ADDITIONAL PROPERTIES TO PROVE ABOUT HMAC-DRBG

Backtracking resistance

Prediction resistance

IX. LINKING OUR CRYPTO SPEC WITH THE FUNCTIONAL SPEC

Prove equivalence VST overview Summarize Naphats proof Relating (Adams) sampling uniformly at random to (Naphats) entropy stream that includes failure Core loop equivalence (prediction resistance) Diagram of work

X. PLAN FOR THE NEXT TWO MONTHS

Finish proofs (how many? 15+) Backtracking resistance, then equivalence proof Deadline table

XI. FUTURE WORK

HMAC is slow, AES is fast How does this generalize to other DRBGs, say AES-DRBG (CTR mode)?

XII. CONCLUSION

We have presented the first machine-checked proof of a crucial cryptographic security property of a pseudorandom number generator. That is, we have written a precise specification of HMAC-DRBG's main functions in Coq, then we have proved that the probability that a non-adaptive probabilistic polynomial time adversary can distinguish HMAC-DRBG's output from uniformly-sampled random bits is negligible. We have also proved a concrete bound on the probability that the adversary can distinguish the two.

There exist only two prior proofs of security of HMAC-DRBG. The first, Champagna YEAR, assumes that HMAC is a random oracle, which is a very strong assumption. Thus, cryptographers generally don't trust proofs written in this model. Additionally, this paper is not peer-reviewed. The second, Hirose YEAR, was peer-reviewed, but its proof is lengthy and involved. Neither paper's proofs has been machine-checked or linked to an implementation of HMAC-DRBG.

We plan to extend our proof to deal with features of HMAC-DRBG that we have omitted, such as . After this, we plan to prove

No DRBG has been verified before HMAC-DRBG has never been end-to-end verified before (for security (indistinguishability) and correctness) No paper proof for HMAC-DRBGs backtracking resistance exists

ACKNOWLEDGMENTS

I'd like to thank Adam Petcher, Matt Green, Andrew Appel, Naphat Sanguansin, and Lennart Beringer for their help.

[1] Last, First, Middle. *Title.* Source

Appendix A: Title

Stuff