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 written in the probabilistic 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.

We present the first machine-checked proof of a crucial cryptographic security property of a pseudo-random 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, by Campagna (2006), is not peer-reviewed. The second, by Hirose (2008), 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.

After this, we plan to prove additional cryptographic properties of HMAC-DRBG, including backtracking resistance. No paper proofs exist for any of these additional properties.

Additionally, we plan to connect our proof of security with an existing proof of correctness to create an end-to-end machine-checked proof chain of functional correctness and security of HMAC-DRBG, which has never been done before.

TODO: include precise theorem statement

II. PROBLEM BACKGROUND AND RELATED WORK

Andrew Appel's group has done the most significant related work in the area. Appel (2015) presents a full formal machine-checked verification of a C program: the OpenSSL implementation of SHA-256. Beringer et al. (2015) 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 and HMAC-DRBG (Campagna (2006), Hirose (2008)), though the former hasn't appeared in a peer-reviewed venue and the latter is very complicated.

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 (Barthe (2011)).

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. (2013) 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. Dorre and Klebanov (2015) 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) 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.

(TODO: fix these citations)

III. 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, 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 the secret key K for HMAC, and an internal value V which is updated with each block of output. The administrative information, consisting of the security strength of this instantiation, and a flag that indicates whether this instantiation requires prediction resistance.

HMAC-DRBG consists of four main functions, *Instantiate*, *Generate*, *Update*, and *Reseed*, and another function we don't model called *getEntropy*. Refer to NIST 800-90A for the pseudocode. (TODO: summarize the functions in my thesis)

IV. INFORMAL PROOF

TODO: write this up. (The draft paper has an outline of the formal proof.)

V. FOUNDATIONAL CRYPTOGRAPHY FRAMEWORK SUMMARY

TODO: cover the following topics:

1. Shallowly-embedded into Coq
2. Game-based crypto proofs
3. Comp monad (probabilistic computations)
4. Relating pairs of games using probabilistic relational Hoare logic (postconditions, *comp_spec*)
5. Oracles, hidden state, and oracle computations (OracleComp)
6. Modeling the adversary as any abstract probabilistic polynomial time algorithm
7. Writing games
8. FCF example

VI. FORMAL PROOF OUTLINE

A. Starting HMAC-DRBG definitions

The core inner loop of HMAC-DRBG that generates pseudorandom blocks. (TODO: compare and contrast each definition with the corresponding NIST pseudocode)

```
(* 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.
```

We combined the *Generate* and *Update* functions into one program, since they're always called together.

```
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

Adam Petcher did some prior work on proving the security of the core loop seen above, run once. His definition is similar to ours, but more abstract. Here f is any PRF.

```
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.
```

We reuse his proof techniques to replace the PRF with a random function, then the random function with random bits. The first is bounded by `PRF_Advantage`, the second by the collision bound. (TODO: elaborate on this)

C. Extending prior work

We extend his work to deal with the following complications in the full HMAC-DRBG and model real-world usage.

- Additional functions: *Instantiate*, *Generate*, and *Update*.
- Multiple calls to *Generate* and *Update*.
- We need to now consider whether the adversary is adaptive or non-adaptive.
- The V is updated after each call.
- The K is updated after each call, so the PRF is re-keyed.

D. Proof of indistinguishability

Here are the statement and the proof in Coq. Note that the proof relies on four main lemmas, which we discuss below.

```
Theorem G1_G2_close :
  | Pr[G1_prg_original] - Pr[G2_prg] | <=
  (numCalls / 1) * Gi_Gi_plus_1_bound.
Proof.
  rewrite G1_Gi_0_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.
```

TODO: discuss pseudorandom functions, random functions, uniformly sampling random bits.

TODO: Indistinguishability definition

E. Hybrid argument

F. Main games

TODO: talk about what a game is

Updating the v immediately after re-keying the PRF, in the same oracle call, is hard to reason about. This is because at the beginning of each call, by the hybrid argument, we can assume that the (K, V) have been randomly sampled. Now we have to replace the key with a random key to reason about updating the v. The solution is simple: we move each v-update to the beginning of the next call and prove that the new sequence of programs is indistinguishable to the adversary. (TODO explain this with pictures)

```
(* [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'')).
```

The adversary, an abstract probabilistic polynomial time algorithm. It takes a list of blocks and returns a guess.

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

The first game: modeling normal usage of the PRG with the PRF.

```
(* 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).
```

The end game: everything returned is a uniformly sampled random bit vector.

```
(* 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.
```

Hybrid games. TODO: define notation, talk about why hybrid and why in this order, talk about oracles

Game i uses RBs for all calls less than i and PRF for all calls greater than or equal to i. Call numbering starts at 0. It passes i to oracle i, which chooses the appropriate oracle to use.

```
(* oracle i *)
(* number of calls: first call is 0, last call is (
    numCalls - 1) for numCalls calls total
G0: PRF PRF PRF
G1: RB PRF PRF
G2: RB RB  PRF
G3: RB RB  RB
there should be (S numCalls) games, so games are
    numbered from 0 through numCalls *)
Definition Oi_prg (i : nat) (sn : nat * KV) (n : nat
    )
  : Comp (list (Bvector eta) * (nat * KV)) :=
  [callsSoFar, state] <-2 sn;
  let GenUpdate_choose := if lt_dec callsSoFar i (*
      callsSoFar < i *)
                          then
                              GenUpdate_rb_intermediate

                          (* first call does not
                              update v, to make
                              proving equiv. easier*)
                          else if beq_nat callsSoFar 0
                               then GenUpdate_noV
                               else GenUpdate in
  (* note: have to use intermediate, not final
      GenUpdate_rb here *)
  [bits, state'] <-$2 GenUpdate_choose state n;
  ret (bits, (S callsSoFar, state')).

(* game i (Gi 0 = G1 and Gi q = G2) *)
Definition Gi_prg (i : nat) : Comp bool :=
  [k, v] <-$2 Instantiate;
  [bits, _] <-$2 oracleMap _ _ (Oi_prg i) (0, (k, v)
      ) maxCallsAndBlocks;
  A bits.
```

The PRF adversary. It uses the existing adversary so we can go from PRF to RF, which is much easier to reason about. (TODO: explained already below) Game i here is more complicated because we can pass in any oracle to use on only the ith call, instead of the the hard-coding in game i above. (TODO: talk about motivation for this)

```
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: like previous oracle i, but uses the provided oracle 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')).
```

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_0_equal : Pr[G1_prg] == Pr[Gi_prg 0].`

   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_oc* 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].`

   We write the ith hybrid in terms of the ith oracle-replaced hybrid using the PRF oracle. Outputting random bits on calls $< i$ and the PRF afterward is equivalent to outputting random bits on calls $< i$, using the PRF oracle on call i, and using the PRF oracle afterward.

   ```
   n = 4, i = 2
   call #  : 0  1  2  3
   Gi_prg 2: RB RB PRF PRF
   Gi_prf 2: RB RB PRF PRF
   ```

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

   In the ith call to the *GenUpdate* oracle, replace the pseudorandom function (PRF) oracle used with the random function (RF) oracle.

   ```
   n = 4, i = 1
   call #  : 0  1  2  3
   Gi_prf 2: RB RB PRF PRF
   Gi_rf 2:  RB RB  RF PRF
   ```

   The difference in probability that any adversary can distinguish between the PRF and the RF is defined by cryptographers to be upper-bounded by a quantity called `PRF_Advantage`. We call this adversary `PRF_Adversary`.

   ```
   Definition PRF_Advantage_Game i : Rat :=
     PRF_Advantage RndK ({0,1}^eta) f eqdbl eqdbv (
         PRF_Adversary i).


      PRF_Advantage =
   fun (D R Key : Set) (RndKey : Comp Key)
     (RndR : Comp R) (f : Key -> D -> R)
     (A : OracleComp D R bool) =>
   | Pr[PRF_G_A RndKey f A] - Pr[PRF_G_B RndR A] |
   ```

   The PRF adversary is given an oracle (PRF or RF with equal probability) and can call it as many times as it wants on whatever inputs it chooses. It needs to guess whether the oracle was the PRF or the RF.

   We want to get our existing hybrid game i in this format, so we simply construct this PRF adversary by passing the oracle we are given to the abstract PRG, then returning what our PRG adversary returns.

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

   Using `PRF_Adversary` allows us to swap out the PRF for the random function, which is much easier to reason about, at the cost of adding `PRF_Advantage` to our final bound.

It is a little questionable that this difference in probability (`PRF_Advantage`) is defined to be the same upper bound no matter how many times the oracle in question is called within `GenUpdate` (and it is called $2 + numBlocks$ times, not just once).

3.
```
Gi_rf_rb_close : forall (i : nat),
 | Pr[Gi_rf i] - Pr[Gi_prg (S i)] | <=
     Pr_collisions.
```

We replace the random function oracle in the ith call with an oracle that simply outputs random bits. We want to upper-bound the difference in probability that any adversary can distinguish a list of n things outputted by a random function (where each output is used as the next input) from a list of n uniformly sampled random bitvectors.

```
n = 4
Gi_rf 2:   RB  RB  RF PRF
Gi_prg 3:  RB  RB  RB PRF
```

We show that it is the probability that there is a collision in the random function's inputs, which is a list of length n. Intuitively, the random function acts exactly like random bits, except for the pathological case where one of the randomly-sampled outputs O happens to be one of the previous inputs. Then, when it is fed in as an input, $RF(O)$ yields its previous output, since it was "cached." In fact all outputs will repeat from then on, leading to a "cycle."

For example, take 101 as a fixed initial input, and everything after it as an output. $*000*$ denotes the bad event of the first repeated input. Note the following cycle.

$101, 000, 011, 001,$
$*000*, *011*, *001*,$
$*000*, *001*, ...$

The analogous proof in `PRF_DRBG` is `PRF_DRBG_G3_G4_close`.

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.
```

```
n = 4, i = 2
Gi_prf 2: RB RB PRF PRF PRF
Gi_rf 2:  RB RB RF  PRF PRF
```

The PRF advantage for hybrid game i is defined to be the normal PRF advantage using the constructed PRF adversary on the ith game. Note that here the PRF advantage is parametrized by i, whereas in a non-hybrid argument, it would simply be the cryptographer-defined upper bound of `PRF_Advantage`.

```
Definition PRF_Advantage_Game i : Rat :=
  PRF_Advantage RndK ({0,1}^eta) f eqdbl eqdbv (
      PRF_Adversary i).
```

We can prove this theorem by simply unfolding the definitions of `Gi_prf` and `Gi_rf`, because they are both in the form stipulated by `PRF_Advantage`. The RF game passes the oracle to an adversary, which returns a guess.

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

The PRF game uniformly samples a random key for the adversary, then passes the adversary the PRF oracle using that key (which it cannot see).

```
Definition Gi_prf (i : nat) : Comp bool :=
  k <-$ RndK;
  [b, _] <-$2 PRF_Adversary i _ _ (f_oracle f _ k)
      tt;
  ret b.
```

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

We would like to use a constant `PRF_Advantage` and eliminate the i, so we arbitrarily pick $i = 0$.

```
n = 4, i = 0
Gi_prf 0: PRF PRF PRF PRF
Gi_rf 0:  RF   PRF PRF PRF
```

The `PRF_Advantage`s are the same for all i except $i = n$ (the number of calls to the `GenUpdate` oracle). For $i = n$, `PRF_Advantage n` $= 0$, since everything has been replaced with random bits, so the two hybrids are equal. Hence the $\leq$ in the theorem.

```
n = 4, i = 4
Gi_prf 4: RB RB RB RB
Gi_rf 4:   RB RB RB RB
```

Identical until bad section.

Note. This section is outlined but the theorems are not explained as fully as in the previous section. I'll finish this in the thesis.

`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 form using the PRF adversary, passing it the 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 in `Gi_rf`l. The bad event is that there are duplicates in the inputs to the *i*th oracle call.

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 `Gi_rb`.

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]
```

   The difference of two games is difficult to work with. We prefer to work with one game, which we can then massage and prove equivalent to other games. Therefore, we apply Bellare's "fundamental lemma of game-playing." It upper-bounds the probability that the adversary can distinguish between `Gi_rf_bad` and `Gi_rb_bad` by the probability that the bad event occurs in `Gi_rb_bad`, which gives us a single game to work with.

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

   As explained in `Gi_rf_rb_close` above, the probability of a bad event happening in the RB game is bounded by the probability of collisions in a list of length (n+1) of randomly-sampled bit vectors.

   To prove this, after transforming and simplifying `Gi_rb_bad` via many intermediate games, we apply the collision bound found in `PRF_DRBG`.

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 ].
```

   This is the statement of Bellare's fundamental lemma. Below, we prove the two assumptions in the theorem, yielding the conclusion.

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 ].
```

   This is the first assumption needed to apply the fundamental lemma: show that the two games have the same probability of returning bad (that is, the bad event has the same probability of happening).

   Concretely, this is true for `Gi_rb_bad` and `Gi_rf_bad` because the probability of the bad event in both is the probability of duplicates in a list of length *n* of uniformly-randomly-sampled bit vectors.

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).
```

   I don't work with it probabilistically; working in the program logic (TODO explain what this means), the goal becomes this:

```
comp_spec (fun b1 b2 : bool * bool => b1 = (a,
    false) <-> b2 = (a, false))
  (Gi_rb_bad i) (Gi_rf_dups_bad i)
```

   This is the second assumption needed to apply the fundamental lemma: given that the bad event does not happen, the distributions of the outputs of the two games are identical (that is, "identical until bad").

   Concretely, this is true for `Gi_rb_bad` and `Gi_rf_bad` because if there are no duplicates in the *i*th oracle call's inputs, clearly the random function behaves exactly like uniformly sampling random bitvectors. So their output should be indistinguishable (or identical, if you provide each run with the same "coins" of randomness).

Both `Gi_rb_rf_return_bad_same` and `Gi_rb_rf_no_bad_same`, when unfolded, essentially assert that some combined postcondition relates the `PRF_Adversary` executions, one using the random bits oracle and one using the random function that preserves duplicate.

`Gi_rb_rf_return_bad_same`, when unfolded, looks like this:

```
comp_spec eq
  (a <-$
   (PRF_Adversary i) (list (Blist * Bvector eta))
     (list_EqDec (pair_EqDec eqdbl eqdbv))
        rb_oracle nil;
   x <-$ ([b, state]<-2 a; ret (b, hasInputDups
      state)); ret snd x)
  (a <-$
   (PRF_Adversary i) (list (Blist * Bvector eta))
     (list_EqDec (pair_EqDec eqdbl eqdbv))
        randomFunc_withDups nil;
   x <-$ ([b, state]<-2 a; ret (b, hasInputDups
      state)); ret snd x)
```

The postcondition here is equality (`comp_spec eq`). That is, both computations return the same value for whether the bad event happened.

And `Gi_rb_rf_no_bad_same`, when unfolded, looks like this:

```
  comp_spec (fun b1 b2 : bool * bool => b1 = (a,
      false) <-> b2 = (a, false))
    (z <-$
     (PRF_Adversary i) (list (Blist * Bvector eta))
       (list_EqDec (pair_EqDec eqdbl eqdbv))
           rb_oracle nil;
     [b, state]<-2 z; ret (b, hasInputDups state))
    (z <-$
     (PRF_Adversary i) (list (Blist * Bvector eta))
       (list_EqDec (pair_EqDec eqdbl eqdbv))
           randomFunc_withDups nil;
     [b, state]<-2 z; ret (b, hasInputDups state))
```

The postcondition here is that if `hasInputDups state = false` (that is, the bad event didn't happen), then both computations return the same output bits (the `a` in `b1 = (a, false)<-> b2 = (a, false)`).

So, the combined postcondition on the `PRF_Adversary` computation (which returns the bits and state) simply combines the two postconditions above. Stated as a separate lemma,

```
Theorem PRF_Adv_eq_until_bad : forall (i : nat),
  comp_spec
    (fun a b : bool * list (Blist * Bvector eta) =>
      let (adv_rb, state_rb) := a in
      let (adv_rf, state_rf) := b in
      let (inputs_rb, outputs_rb) := (fst (split
          state_rb), snd (split state_rb)) in
      let (inputs_rf, output_rf) := (fst (split
          state_rf), snd (split state_rf)) in
      hasDups _ inputs_rb = hasDups _ inputs_rf /\
      (hasDups _ inputs_rb = false ->
       state_rb = state_rf /\ adv_rb = adv_rf))

    ((PRF_Adversary i) (list (Blist * Bvector eta))
      (list_EqDec (pair_EqDec eqdbl eqdbv))
          rb_oracle nil)

    ((PRF_Adversary i) (list (Blist * Bvector eta))
      (list_EqDec (pair_EqDec eqdbl eqdbv))
      randomFunc_withDups nil).
```

The postcondition looks intimidating, but is simply the conjunction of the two postconditions discussed above. A computation has duplicates in the oracle inputs only if the other computation does. And, if there are no duplicates, then the oracle states are equal and the adversary guesses are equal. I use that lemma to prove both the identical until bad conditions, and now we only have to worry about proving the one lemma `PRF_Adv_eq_until_bad`.

`PRF_Adversary` generates the pseudorandom bits, then returns the indistinguishability adversary's guess.

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

So, we can push the postconditions into the generation of the pseudorandom bits, specifically into *oracleCompMap_inner*, and use that to prove `PRF_Adv_eq_until_bad`. The inner lemma is:

```
Theorem oracleCompMap__oracle_eq_until_bad_dups :
    forall (i : nat) b b0,
    comp_spec
     (fun y1 y2 : list (list (Bvector eta)) * list (
         Blist * Bvector eta) =>
       hasDups _ (fst (split (snd y1))) = hasDups _
            (fst (split (snd y2))) /\
       (hasDups _ (fst (split (snd y1))) = false ->
        snd y1 = snd y2 /\ fst y1 = fst y2))

     ((z <--$
       oracleCompMap_inner
         (pair_EqDec (list_EqDec (list_EqDec eqdbv))
           (pair_EqDec nat_EqDec eqDecState))
         (list_EqDec (list_EqDec eqdbv)) (Oi_oc' i)
         (0, (b, b0)) maxCallsAndBlocks; [bits, _
             ]<-2 z; $ ret bits)
        (list (Blist * Bvector eta)) (list_EqDec (
            pair_EqDec eqdbl eqdbv))
        rb_oracle nil)

     ((z <--$
       oracleCompMap_inner
         (pair_EqDec (list_EqDec (list_EqDec eqdbv))
           (pair_EqDec nat_EqDec eqDecState))
         (list_EqDec (list_EqDec eqdbv)) (Oi_oc' i)
         (0, (b, b0)) maxCallsAndBlocks; [bits, _
             ]<-2 z; $ ret bits)
        (list (Blist * Bvector eta)) (list_EqDec (
            pair_EqDec eqdbl eqdbv))
        randomFunc_withDups nil).
```

The postcondition is the same as in `PRF_Adv_eq_until_bad`, except that instead of saying that the adversary guesses are the same, we say that the generated pseudorandom bits are the same, which clearly implies the former. (Intuitively, doing proofs relating two probabilistic programs is like proving things about the deterministic programs given the same "tape" of randomness.)

The proverbial buck stops here; we don't push the postcondition back further into `oracleCompMap_inner` (which iterates chosen `GenUpdate` oracles), `Oi_oc` (which chooses which `GenUpdate` oracle to use), the `GenUpdate` oracles, or `Gen_loop`, which uses the provided random bits or random-function-with-duplicates oracle to generate bits. If we did that, we would have to reason about many computations extraneous to the oracle, whereas we only need to reason about the properties of the oracle and how it handles bad events.

We use a powerful theorem called `fcf_oracle_eq_until_bad` to "strip away" the computations that use the oracle so we can just reason about the oracle. Informally, `fcf_oracle_eq_until_bad` states that postconditions in our "identical until bad" form are true if we can prove three side conditions:

1. If the two oracles start in the same state, if the bad event either did not happen or did happen in

both states (meaning NOT (the bad event happened in one state and not the other)), then the same identical-until-bad postcondition relates one run of the each of the oracles.

2. For the first oracle, if its state starts bad, it stays bad.

3. For the second oracle, if its state starts bad, it stays bad.

This identical until bad postcondition is (TODO write out). The full statement of the theorem is very long, so it is included in the appendix. The proof, done by Petcher (2015), is also included in the appendix.

Because this proof doesn't depend on the details of the computations, only the oracle, and our oracles here are the same as in the corresponding proof in `PRF_DRBG` (which is `PRF_A_randomFunc_eq_until_bad`), so we can simply re-use that proof to prove `oracleCompMap__oracle_eq_until_bad_dups`.

Here's how we prove the side conditions. (Numbers correspond to above numbering.)

1. To prove the first condition: first, we unfold the definition of `randomFunc_withDups` in the statement of the theorem.

```
y <-$
     match arrayLookup D_EqDec xs a with
     | Some y => ret y
     | None => { 0 , 1 }^eta
     end; ret (y, (a, y) :: xs)
```

   Then we do a case analysis on whether the newest input is a duplicate (that is, is already in the state of the random function oracle that preserves duplicates). In the Coq tactic language, that's `case_eq (arrayLookup _ xs a)`.

   The first case is easy. If the element `a` is **not** in the random function's state `xs`, then the random function uniformly randomly samples a bitvector, so the expression simplifies to be identical to that of `rb_oracle`:

```
      comp_spec
    (fun y1 y2 : Bvector eta * list (D * Bvector
        eta) =>
     hasDups D_EqDec (fst (split (snd y1))) =
     hasDups D_EqDec (fst (split (snd y2))) /\
     (hasDups D_EqDec (fst (split (snd y1))) =
         false ->
      snd y1 = snd y2 /\ fst y1 = fst y2))
    (y <-$ { 0 , 1 }^eta; ret (y, (a, y) :: x2))
    (r <-$ { 0 , 1 }^eta; ret (r, (a, r) :: x2))
```

   We were already given the first postcondition as a hypothesis. The second postcondition holds because indeed, there are no duplicates (by the case analysis) and the return values are equal.

   The second case is more involved. If the element `a` **is** in the random function's state `xs`, then we know there are duplicates in the random function's state. So, since the two oracles start with the same state, the `rb_oracle` also has duplicates in its state. After simplifying, our new goal is to prove this:

```
      comp_spec
    (fun y1 y2 : Bvector eta * list (D * Bvector
        eta) =>
     hasDups D_EqDec (fst (split (snd y1))) =
     hasDups D_EqDec (fst (split (snd y2))) /\
     (hasDups D_EqDec (fst (split (snd y1))) =
         false ->
      snd y1 = snd y2 /\ fst y1 = fst y2)) (ret (b,
          (a, b) :: x2))
    (ret (b0, (a, b0) :: x2))
```

   So, we use the tactic `fcf_spec_ret`, which says, "we're done manipulating the two games and now they simply return things; let's prove that the postcondition relates their two return values."

   The second part of the postcondition is easy to discharge:

```
     (hasDups D_EqDec (fst (split (snd y1))) =
         false ->
      snd y1 = snd y2 /\ fst y1 = fst y2)
```

   It starts with the assumption that there are no duplicates in the entire state. But, by our case analysis earlier, we are in the case where there are duplicates in the tail of the state, which implies that there are duplicates in the entire state. So, we can eliminate this case.

   The first part of the postcondition,

```
     hasDups D_EqDec (fst (split (snd y1))) =
    hasDups D_EqDec (fst (split (snd y2)))
```

   requires us to prove that whether the `randomFunc_withDups` oracle has duplicates equals whether the `rb_oracle` has dups. This follows because they started with the same initial state, which has duplicates. `hasDups (thing1 :: x2)= hasDups (thing2 :: x2)` since `hasDups x2`

2. For the `randomFunc_withDups` oracle, the state is append-only. So, if the state starts out with duplicates, no matter what we query or what we return, the state will continue to have duplicates.

3. For the `rb_oracle`, the state is also append-only. So, as above, if the state starts out with duplicates, no matter what we query or what we return, the state will continue to have duplicates.

The commented Coq proof of this can be found in the appendix.

H. Proof graph

To add, after I break up `Gi_rb_rf_no_bad_same`. I've already made one using `coq-dpdgraph`.

I. Current limitations of the formal proof

We abstracted or ignored a few parts of HMAC-DRBG.

- NIST specifies that *Instantiate* produces the (K, V) by calling *Update* using hardcoded constant (K, V) and some additional entropy. The entropy might not be uniformly random, so to reason about this construction, we would have to show that HMAC is an "entropy extractor." This seems not worth the trouble, so we assume that *Instantiate* samples (K, V) uniformly at random.
- *Update* includes a call to *Reseed* if the reseed counter is greater than reseed count, or the DRBG's state is compromised. (*Reseed* provides prediction resistance by injecting entropy unknown to the adversary into the DRBG). We remove the call to *Reseed* because again, it's hard to reason about entropy, so our DRBG is not prediction resistant. Also, reseed count is a very large number that means you would naturally reseed about once every million years.
- We need to figure out how to adapt the indistinguishability proof to a nonadaptive adversary that picks any list of *blocksPerCall* and *numCalls* beforehand, not the hardcoded *maxBlocksPerCall* and *maxCalls*. One can imagine a pathological DRBG that outputs its key for only $\mathit{numCalls} = 7$, for example.
- We need to figure out how to adapt the indistinguishability proof to an adaptive adversary. This adversary can choose the number of calls and number of blocks per call, then once it receives output, can call again as many times as it wishes with parameters of its choice. One can imagine a pathological DRBG that "encourages" an adversary to input 7 as *blocksPerCall* and outputs its secret key after 3 such calls, so that if it discovers that 7 is a "bad" input, it keeps inputting it. A non-adaptive adversary would have negligible chance of making this experiment or this discovery.
- We ignore the "additional input" and "personalization string" parameters of *Generate*, *Update*, and *Instantiate*. These are generally used for fork-safety. If you fork the DRBG process, the child process starts with exactly the same internal state, so one would update the state with something unique to that process, generally the process ID.
- Entropy failure

J. Comparison to existing proof of HMAC-DRBG security

To write after I've read the Hirose paper.

- 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

Formally verifying HMAC-DRBG helped us notice several NIST design decisions that made our job either harder or easier.

- NIST re-keys the PRF with a length-extended input. This is good because HMAC can take inputs of any length, and all previous inputs for HMAC with that key were of fixed length (since HMAC has a fixed output). So we know the new key won't collide with previous outputs.
- Updating the v immediately after re-keying the PRF, in the same oracle call, is hard to reason about. This is because at the beginning of each call, by the hybrid argument, we can assume that the (K, V) have been randomly sampled. Now we have to replace the key with a random key to reason about updating the v. The solution is simple: we move each v-update to the beginning of the next call and prove that the new sequence of programs is indistinguishable to the adversary. (TODO explain this with pictures)
- In *Update*, NIST does (++ [v]) instead of (v ::). The former is intuitive on paper; the latter is much easier to reason about by induction in Coq.

VII. ADDITIONAL PROPERTIES TO PROVE ABOUT HMAC-DRBG

We plan to prove that HMAC-DRBG possesses backtracking resistance. That is, if it is compromised at time T, the adversary still cannot distinguish HMAC-DRBG previous output from ideal random strings. That is, if the adversary is given a *Compromise* oracle that it can call after a certain number of calls to the *GenUpdate* oracle to reveal the PRG's internal (K, V), the adversary still cannot distinguish the previous output from ideal random strings. We plan to do this proof and reuse much of our existing work on indistinguishability. We're working on formalizing the definitions and working out some issues regarding v-updating and nonadaptive vs. adaptive adversaries.

NIST also specifies that HMAC-DRBG possesses the complementary property of prediction resistance. That is, if it is compromised at time T, the adversary will find it difficult to distinguish HMAC-DRBG's future output from ideal random strings. Naively, if given (K, V) at time T, the adversary should be able to compute all

future DRBG output and thus distinguish. Thus, the *Reseed* function is supposed to ensure prediction resistance by injecting fresh entropy into the DRBG that is unknown to the adversary. (It refreshes the (K, V) by re-HMACing them with some entropy appended to the previous (K, V))

However, proving things about this is difficult. Dodis et al. (2013) proved that the Linux PRNG was insecure and did not possess prediction resistance. However, they had to do tricky reasoning about how much entropy was injected at once and where. They had to reason about, for example, the difference between injecting entropy as five bits per call over ten calls, versus fifty bits in one call, and relative difficulties for the adversary.

VIII. LINKING OUR CRYPTO SPEC WITH THE FUNCTIONAL SPEC

Naphat Sanguansin '16 proved functional correctness of the mbed TLS implementation of HMAC-DRBG using a different functional specification, also written in Gallina. To create a truly end-to-end proof of correctness, we must prove that our cryptographic specification of HMAC-DRBG is the same as his specification.

A. Proving functional correctness of HMAC-DRBG

To add: VST overview. Hoare logic overview. Summary of Naphat's proof. A diagram of the whole project, similar to the HMAC diagram. My prior work for HMAC.

B. Bridging the gaps between the specifications

We must prove lemmas about at least the following differences between the specifications.

1. Naphat's code has essentially the same core loop (*Gen_loop*) and *Generate* and *Update* code. However, it's surrounded by many layers of error checking code, e.g. for `prediction_resistance_request`. We need to prove equivalence modulo error checking.

2. Naphat's code also models failures in the type for the entropy stream. It is an infinite stream of either bit or failure. Also, it is not necessarily ideal randomness. We need to relate FCF's sampling uniformly at random to this more realistic entropy model. Adam Petcher's thesis provides some theorems about the operational semantics of FCF that can help us with this.

IX. PLAN FOR THE NEXT TWO MONTHS

There are 21 admitted lemmas left to prove that I'm confident about, and 1 lemma left that I'm sure is true, but am not sure how to prove. I planned to have proved them by the beginning of March. However, since it is now the beginning of March, and I'm traveling for 1.5 weeks for graduate school visits, it is plausible that I can have the indistinguishability proof all proven by the end of March.

Matt and I have started working on the backtracking resistance proof. It relies heavily on reusing our work on the indistinguishability proof, modulo adaptiveness of adversary. So, we will concurrently work on writing an informal proof of that and formalizing the specifications and definitions in FCF. I will likely not do the formal proofs until April or July, but there shouldn't be too many, given that we are reusing the indistinguishability proofs.

Also, we need to figure out how to possibly adapt the indistinguishability proof to a nonadaptive adversary that picks any list of *blocksPerCall* and *numCalls* beforehand, not the hardcoded *maxBlocksPerCall* and *maxCalls*.

I plan to start writing my thesis in the beginning of April and finish a draft by the end of the third week of April, then revise it. In April I'll look more into doing the spec equivalence proof, but probably not do it. I plan to do this proof in hopefully less than one month during July.

X. FUTURE WORK

HMAC is slow. AES is fast. Thus, AES CTR-DRBG, which uses AES in CTR mode, is much more widely used in practice than HMAC-DRBG (e.g. Amazon uses AES CTR-DRBG). It would be practically useful to formally verify HMAC-DRBG, but also theoretically interesting. How do our proofs generalize? (Some things would be different; e.g. HMAC is assumed a PRF, and AES is assumed a pseudo-random permutation. Their respective DRBGs are slightly different as well. Some things would break; e.g. to refresh the key without a collision, NIST cannot length-extend the input.) Can we build a general framework for verifying DRBGs? How automated can it be? Is the concrete security bound better?

Also, OpenSSL's team is picking or designing a new PRNG–we hope our work encourages implementors to co-design with formal methods researchers.

XI. CONCLUSION

To add.

ACKNOWLEDGMENTS

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

[1] Affeldt, Reynald, David Nowak, and Kiyoshi Yamada. "Certifying assembly with formal cryptographic proofs: the case of BBS." *Electronic Communications of the EASST* 23 (2009).

[2] Appel, Andrew W. "Verification of a cryptographic primitive: SHA-256." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37, no. 2 (2015): 7.

[3] Barker, Elaine, and John Kelsey. "NIST Special Publication 800-90A: Recommendation for random number generation using deterministic random bit generators." (2012).

[4] Barthe, Gilles, Benjamin Grgoire, Sylvain Heraud, and Santiago Zanella Bguelin. "Computer-aided security proofs for the working cryptographer." In *Advances in CryptologyCRYPTO* 2011, pp. 71-90. Springer Berlin Heidelberg, 2011.

[5] Beringer, Lennart, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. "Verified correctness and security of OpenSSL HMAC." In *24th USENIX Security Symposium (USENIX Security 15)*, pp. 207-221. 2015.

[6] Campagna, Matthew J. "Security Bounds for the NIST Codebook-based Deterministic Random Bit Generator." *IACR Cryptology ePrint Archive 2006* (2006): 379.

[7] Dodis, Yevgeniy, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. "Security analysis of pseudo-random number generators with input:/dev/random is not robust." In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security*, pp. 647-658. ACM, 2013.

[8] Drre, Felix, and Vladimir Klebanov. "Pseudo-Random Number Generator Verification: A Case Study." *Proceedings, Verified Software: Theories, Tools, and Experiments (VSTTE)* (2015).

[9] Hirose, Shoichi. "Security analysis of DRBG using HMAC in NIST SP 800-90." In *Information Security Applications*, pp. 278-291. Springer Berlin Heidelberg, 2008.

[10] Petcher, Adam, and Greg Morrisett. "The foundational cryptography framework." *In Principles of Security and Trust*, pp. 53-72. Springer Berlin Heidelberg, 2015.

[11] Sanguansin, Naphat. "Verification of a Deterministic Random Bits Generator." Independent work paper, Princeton University. 2016.

Appendix A: Definitions and proofs

Will add worked examples of Coq proofs using FCF tactics.