The Notorious PRG: Formal verification of the HMAC-DRBG pseudorandom number generator

Katherine Ye, advised by Andrew W. Appel[1] and and Matthew Green[2]

[1] *Princeton University*
[2] *Johns Hopkins University*

We have proved, with machine-checked proofs, that the output produced by HMAC-DRBG is indistinguishable from random by a computationally bounded adversary. We proved this about a high-level specification of a simplified version of HMAC-DRBG written in the probabilistic language provided by the Foundational Cryptography Framework (FCF), which is embedded in the Coq proof assistant. We have also proven on paper that HMAC-DRBG is backtracking-resistant. Our work comprises the first formal verification of a real-world PRG. Our functional specification can be linked to a proof of functional correctness of mbedTLS's C implementation of HMAC-DRBG. This will allow our proofs of cryptographic security properties to transfer to this implementation.

CONTENTS

I. Introduction 2

II. Related work 2

III. HMAC-DRBG overview 3
- A. Security properties 4

IV. Proofs 4
- A. Indistinguishability proof for one call 4
- B. Indistinguishability proof for a simplified HMAC-DRBG 7
  - 1. Definitions 7
  - 2. Proof outline 8
  - 3. Formalization 13
- C. Proof of backtracking resistance 14
- D. On proving prediction resistance 16
- E. On adaptivity of the adversary 16

V. Overview of the Foundational Cryptography Framework 16

VI. Formal proof outline 17
- A. Starting HMAC-DRBG definitions 17
- B. Prior work on abstract PRF-DRBG 17
- C. Proof of indistinguishability 17
  - 1. Main games 18
- D. List of lemmas 19
- E. Proof graph 24
- F. Current limitations of the formal proof 24
- G. Comments on the Coq development 25
- H. Comments on HMAC-DRBG's design 25

VII. Linking our crypto spec with the functional spec 25
- A. Bridging the gaps between the specifications 26

VIII. Future work 26

IX. Conclusion 26

X. Acknowledgments 27

References 27

A. Definitions and proofs 27

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 pseudorandom number generator (PRNG) is an easy way to break the entire cryptosystem. A weak PRNG will generate output predictable to an attacker, allowing her to guess private keys, yet most statistical tests will not be able to detect the regularities in the output. 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 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, by Campagna (2006), is not peer-reviewed, and does proofs in the random oracle model. The second, by Hirose (2008), was peer-reviewed, but they do not prove security of HMAC-DRBG for multiple calls of the *Generate* function, nor do they include the state updating in their *Generate* function. (Nevertheless, we arrive at almost the same bound for one call to *Generate*, which is a vote of confidence for both of us.) Neither paper's proofs has been machine-checked or linked to an implementation of HMAC-DRBG.

We have also proven on paper that HMAC-DRBG is backtracking-resistant. No paper proofs exist for this. We plan to modify our computer-checked indistinguishability proof to hold for this property.

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 is entirely novel work.

II. RELATED WORK

Andrew Appel's group has done the most similar 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 fully verifying HMAC-DRBG.

In addition, there exist paper proofs of the security of CTR-DRBG and HMAC-DRBG (Campagna (2006), Hirose (2008)), though (as mentioned earlier) the former hasn't appeared in a peer-reviewed venue and does proofs in the random oracle model, and the latter does not prove security over multiple calls to *Generate* or account for state updating.

In the area of checking game-based proofs of cryptographic security within a proof assistant, there are two main tools. The long-established tools are EasyCrypt and its cousin CertiCrypt (Barthe (2011)), only the latter of which is foundational, but it is no longer maintained. A newer tool, is the Foundational Cryptography Framework (Petcher (2015)). The EasyCrypt tutorial (Barthe (2014)) proves the pseudorandomness of a simple stateful PRG. The PRG is very similar to the core loop construction discussed in Section REFERENCE, and the bound is also very similar (it is essentially the same one described in Hirose (2008)). The tutorial's concrete bound is quantified over "For all PRG-distinguisher D that makes at most qn queries to its next oracle," so their PRG doesn't involve updating the PRF's key and then making more queries.

In the general area of formalizing PRNGs, several crypto papers analyze the security of PRNGs and propose new security properties. Dodis et al. (2013) 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. Dörre 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.

Affeldt (2009) prove that their assembly implementation of the Blum-Blum-Shub PRNG satisfies "unpredictability" (equivalent to pseudorandomness) in the game-playing style in Coq. They do so directly on an assembly implementation of the 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.

III. HMAC-DRBG OVERVIEW

A pseudorandom number generator is used to stretch a small amount of randomness into a large amount of pseudorandomness, 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 that proven to be a pseudorandom function (PRF) given that HMAC's internal hash function is a PRF.

The following section summarizes HMAC-DRBG's description in NIST SP 800-90A.

HMAC-DRBG possesses an internal state consisting of two pieces of administrative information, which are constant across calls, and two pieces of secret information, which are updated across calls.

The internal state consists of:

- Administrative information:
  - The security strength of the PRG's instantiation. The *Instantiate* function obtains the appropriate amount of entropy for that security strength.
  - A prediction resistance flag that indicates whether this instantiation requires prediction resistance. Prediction resistance is explained in the next section.
- The working state, which we often refer to as (k, v).
  - The secret key k of length c for the internal PRF, which is HMAC. It is updated at least once each time the PRG generates pseudorandom bits by calling the *Generate* function.
  - An internal value v of length c, which is updated each time the PRG generates another block of pseudorandom bits of length c.

HMAC-DRBG consists of four functions, *Update*, *Instantiate*, *Reseed*, and *Generate*, and another function we don't model called *getEntropy*. For simplicity, we omit the *additional_input* and *personalization_string* parameters from *Instantiate*. Our proof of security does not include them. Refer to NIST SP 800-90A for the full pseudocode.

Let f denote HMAC and f_k denote HMAC partially applied with the key k. Let $||$ denote concatenation. Here we will not use mutable variables as NIST does, but write our code in a more functional style, which will make it more readable.

Update refreshes (k, v), optionally using some *data*, which can be fresh entropy for prediction resistance.

```
**Update**($data, k, v$) :=
    $k_1 \leftarrow f_k(v || 0x00 || data)$
    $v_1 \leftarrow f_{k_1}(v)$
    if $(provided\_data = nil)$, $ret(k_1, v_1)$
    $k_2 \leftarrow f_{k_1}(v_1 || 0x01 || data)$
    $v_2 \leftarrow f_{k_2}(v_1)$
    $ret\ (k_2, v_2)$
```

Instantiate initializes (k, v) and *reseed_counter*.

```
**Instantiate**($entropy, nonce$) :=
    $seed \leftarrow entropy || nonce$
    $k \leftarrow 0x00 \dots 00$
    $v \leftarrow 0x01 \dots 01$
    $(k_1, v_1) \leftarrow Update(seed, k, v)$
    $reseed\_counter \leftarrow 1$
    $ret\ (k_1, v_1, reseed\_counter)$
```

Reseed could be called for two reasons. First, *reseed_counter* could have exceeded *reseed_interval*. This is rare, since *reseed_interval* is set to high values in practice; its maximum value is 2^{48}, meaning it would naturally reseed once every couple of million years. (Indeed, in practice, one does not want a PRG to reseed often. This would give an attacker more opportunities to compromise the entropy source.) More commonly, *Reseed* is called when a PRG's state could have be compromised, and requires fresh entropy to be mixed in.

```
**Reseed**($k, v, entropy$) :=
    $(k_1, v_1) \leftarrow Update(seed, k, v)$
    $reseed\_counter \leftarrow 1$
    $ret\ (k, v, reseed\_counter)$
```

Generate generates n blocks of pseudorandom bits.

```
**Generate(k, v, n)** :=
    if $reseed\_counter > reseed\_interval$,
        $ret$ "reseed required"
    $temp_0 \leftarrow nil$
    $v_0 \leftarrow v$
    while $len(temp_i) < n$
        $v_{i+1} \leftarrow f_k(v_i)$
        $temp_{i+1} \leftarrow temp_i || v_{i+1}$
    (say the loop above ran $j$ times)
    $returned\_bits \leftarrow$ leftmost $n$ of $temp_j$
    $(k_1, v_{j+1}) \leftarrow Update(k, v_j)$
    $reseed\_counter_1 \leftarrow reseed\_counter + 1$
    $ret\ (bits, k_1, v_{j+1}, reseed\_counter_1)$
```

From the user's point of view, only the *Instantiate* and *Generate* functions are visible. The state is, of course, hidden. Typical usage would be a call to *Instantiate*, followed by any number of calls to *Generate* some number of pseudorandom bits. *Generate* automatically calls *Update* every time it is called, and *Generate* may call *Reseed*.

We discuss our proof of security for a simplified PRG, consisting of only one call to the internal loop in *Generate*, in section REFERENCE. We discuss our proof of security for a simplified version of HMAC-DRBG in section REFERENCE.

A. Security properties

The most important property for a PRG is that its output is indistinguishable from ideal randomness. This definition will be formally stated in the proof in section REFERENCE.

A good PRG ought to also possess *backtracking resistance* and *prediction resistance.*

If a PRG is backtracking resistant, then if the working state of the PRG is compromised at some time, any of its (previously unseen) output from before that time remains indistinguishable from ideal randomness.

This definition will be formally stated in the proof in section REFERENCE.

Prediction resistance is the complementary property. If a PRG is prediction resistant, then if the working state of the PRG is compromised at some time, after a certain recovery or reseeding interval, it is again indistinguishable from ideal randomness for any time afterward.

We do not prove that HMAC-DRBG is prediction resistant. This is because the definition is slippery, and because it is difficult to prove.

IV. PROOFS

In this section we will write proofs as a conventional cryptographer would. We prove three things: indistinguishability for a simple PRG constructed from a PRF, indistinguishability for a real-world PRG constructed from a PRF, and backtracking resistance for the same PRG.

For any game G that returns a boolean, let $\Pr[G]$ denote the probability that G returns true.

A. Indistinguishability proof for one call

Adam Petcher proved the security of the core loop construction in HMAC-DRBG's *Generate* function. We summarize his proofs here.

First, take any family of PRFs

$$F_c = \{f_k : \{0,1\}^c \to \{0,1\}^c\}_{k \in \{0,1\}^c}.$$

Construct a PRG

$$H : (n : \mathbb{N}) \to (k : \{0,1\}^c) \to (v : \{0,1\}^c) \to (bits : \{0,1\}^{cn})$$

It takes an input seed and applies the PRF to it, then uses the output as one block of pseudorandom bits and as the next input seed to the PRG. n is the number of blocks desired, k is the key for the PRF, and c (the input/output size of the PRF) is the security parameter.

```
$H(n, k, v) :=$
    if $n = 0$ then return $nil$ (the empty list)
    else return $f_k(v) || H(n-1, k, f_k(v))$.
```

For example, $H(0, k, v_0) = nil$, and

$$H(3, k, v_0) = f_k(v_0) || f_k(f_k(v_0)) || f_k(f_k(f_k(v_0))).$$

(Note one difference between H and the loop in *Generate*: H prepends generated blocks, whereas *Generate* appends generated blocks. We use the former construction because it is easier to reason about in Coq, but intuitively it is as secure as the latter.)

The user (or adversary) can request any number of blocks n polynomial in the security parameter c, and the length of the seed is c. Thus, the stretch of H is $l(n, c) = cn$. When we use H, we pass in $k \xleftarrow{R} \{0,1\}^c$.

We want to prove that the bits generated by H are pseudorandom.

Definition 1. *For all non-uniform, non-adaptive probabilistic polynomial time distinguishers D, number of blocks n that is polynomial in c, there exists a negligible function such that*

$$|\Pr[D(H(n, k, U_c) = 1] - \Pr[D(U_{cn}) = 1]| \leq neg(c, n),$$

where $U_c \xleftarrow{R} \{0,1\}^c$ (U_c is sampled uniformly at random from $\{0,1\}^c$), $U_{cn} \xleftarrow{R} \{0,1\}^{cn}$, and $k \xleftarrow{R} \{0,1\}^c$ (k is the key for the PRF).

The probability that the distinguisher returns 1 when given the PRG's output, $\Pr[D(H(n, k, U_c) = 1]$, is the same as the probability that the result of this real-world game is 1.

```
**G_real** :=
    $n \leftarrow D$ (the distinguisher picks a number of blocks)
    $k \xleftarrow{R} \{0,1\}^c$
    $v \xleftarrow{R} \{0,1\}^c$
    $bits \leftarrow H(n, k, v)$
    $ret\ D(bits)$
```

That is, $\Pr[D(H(n,k,U_c)=1)] = \Pr[G\_real]$

The probability that the distinguisher returns 1 when given a uniformly randomly sampled bit string of length cn is the same as the probability that the result of this ideal-world game is 1. Let $generate\_bitvectors\ n$ generate a list of n uniformly randomly sampled bitvectors of length c each.

```
G_ideal :=
    $n \leftarrow D$ (the distinguisher picks a number of blocks)
    $bits \leftarrow generate\_bitvectors\ n$
    $ret\ D(bits)$
```

That is, $\Pr[D(U_{nx})=1)] = \Pr[G\_ideal]$.

So, to bound the probability of distinguishing,

$$|\Pr[D(H(x,U_n)=1] - \Pr[D(U_{nx})=1]|,$$

we bound the distance between the two games,

$$|\Pr[G\_real] - \Pr[G\_ideal]|.$$

To bound this distance, we introduce an intermediate game, $G\_intermediate$. First, in the PRG, we replace the PRF with a random function

$$R_c = \{r : \{0,1\}^* \to \{0,1\}^c\}.$$

```
$H\_rf(n,v) :=$
    if $n = 0$ then return $nil$
    else return $r(v) || H(n-1, r(v))$.
```

Next, we define $G\_intermediate$ with the same structure as G_real, but using H_rf (the PRG using the random function) instead of H.

```
G_intermediate :=
    $n \leftarrow D$ (the distinguisher picks a number of blocks)
    $v \xleftarrow{R} \{0,1\}^c$
    $bits \leftarrow H\_rf(n,v)$
    $ret\ D(bits)$
```

Then we calculate the overall bound using this intermediate game. We first bound the difference between the real and the intermediate game, then between the intermediate and the ideal game.

Theorem 2.

$$|\Pr[G\_real] - \Pr[G\_ideal]| \leq$$
$$|\Pr[G\_real] - \Pr[G\_intermediate]|$$
$$+|\Pr[G\_intermediate] - \Pr[G\_ideal]|.$$

Proof. By the triangle inequality. □

Theorem 3.

$$|\Pr[G\_real] - \Pr[G\_intermediate]| \leq PRF\_Advantage.$$

Proof. We are not really doing math here; this follows by definition.

$PRF\_Advantage$ is the maximum probability that any non-uniform probabilistic polynomial-time distinguisher D_p can distinguish between a PRF (with input/output length c) and a random function (abbreviated RF from now on). It is given oracle access to one of them, which it can query a number of times polynomial in the security parameter c, and must output a guess.

$$|\Pr[D_p^{f_k(\cdot)}(1^c)=1] - \Pr[D_p^{r(\cdot)}(1^c)=1]| \leq PRF\_Advantage$$

We can define it equivalently in terms of the distance between games, as discussed above.

$$|\Pr[G\_Adv\_PRF] - \Pr[G\_Adv\_RF]| \leq PRF\_Advantage.$$

where the games are as follows:

```
G_Adv_PRF :=
    $k \xleftarrow{R} \{0,1\}^c$
    $b \leftarrow D_p^{f_k(\cdot)}$
    $ret\ b$

G_Adv_RF :=
    $b \leftarrow D_p^{r(\cdot)}$
    $ret\ b$
```

Since we are doing proofs of concrete security, we must use this definition of the difference between a PRF and a RF. If we were proving asymptotic security, we would reduce the security of the PRG to the security of the PRF (which we do here) by showing that given a non-uniform PPT adversary against the PRG, we could construct a non-uniform PPT adversary against the PRF. But this does not yield a concrete bound.

Then we rewrite our existing games to fit the form of some adversary being given a particular oracle.

```
G_real_2 :=
    $k \xleftarrow{R} \{0,1\}^c$
    $b \leftarrow PRF\_A^{f_k(\cdot)}$
    $ret\ b$

G_intermediate_2 :=
    $b \leftarrow PRF\_A^{r(\cdot)}$
    $ret\ b$
```

where PRF_A, given either the PRF oracle or the RF oracle, simply generates the pseudorandom bits using the oracle and calls the existing PRG adversary on them. Note that PRF_A no longer samples the key for the PRF.

```
PRF_A o :=
    $n \leftarrow D$
    $v \xleftarrow{R} \{0,1\}^c$
    $bits \leftarrow H\_oc(o,n,v)$
    $ret\ D(bits)$
```

and H_oc is a modified version of the PRG that takes an oracle and queries it, in place of using the PRF.

```
$H\_oc(o, n, v) :=$
    if $n = 0$ then return $nil$
    else return $o.query(v)||H(o, n-1, o.query(v))$.
```

Lemma 3.1.

$$|\Pr[G\_real\_2] - \Pr[G\_intermediate\_2]| \leq PRF\_Advantage.$$

Proof. By the definition of $PRF\_Advantage$, with $D_p := PRF\_A$. □

And our two rewrites don't change the probability that the adversary returns 1 in a particular game.

Lemma 3.2. $\Pr[G\_real] = \Pr[G\_real\_2]$.

Proof. By substitution and program equivalence. □

Lemma 3.3. $\Pr[G\_intermediate] = \Pr[G\_intermediate\_2]$.

Proof. By substitution and program equivalence. □

Thus, the theorem follows:

$$|\Pr[G\_real] - \Pr[G\_intermediate]| \leq PRF\_Advantage.$$

□

Next, we bound the difference between the intermediate and the ideal-world game. This requires making a combinatorial argument.

Theorem 4. $|\Pr[G\_intermediate] - \Pr[G\_ideal]| \leq \frac{n^2}{2^c}$.

Proof. We use Bellare's *fundamental lemma of game-playing* to bound the distance between these two games.

If two games are *identical until bad*, they are "syntactically identical except for statements that follow the setting of a flag bad to true" (Bellare (2004)).

Lemma 4.1. *(Fundamental lemma of game-playing) Let G and H be identical-until-bad games and let A be an adversary. Then*

$$|\Pr[G^A = 1] - \Pr[H^A] = 1]| \leq Pr[H^A \text{ sets bad}].$$

Proof. See Bellare (2004). □

Intuitively, the "bad event" in both $G\_intermediate$ and G_ideal is that there are duplicates in the inputs to the random function used in the former game. If there are no duplicates, then the random function behaves exactly like uniformly sampling random bits. If there are duplicates, then the random function becomes deterministic and starts to cyclically repeat outputs. Thus, we manipulate both games to expose this bad event.

Instead of using $G\_intermediate$, we modify $G\_intermediate\_2$, which we had proven equivalent (via program equivalence) to $G\_intermediate$ in Lemma 3.3. We replace the random function oracle with a random function oracle rf that keeps track of all (input, output) pairs, including duplicate inputs.

```
**G_intermediate_3** :=
    $b \leftarrow PRF\_A^{rf(\cdot)}$
    $ret\ b.$
```

Then we expose the bad event as the second element of the output. PRF_A now outputs the state of the oracle after many calls, though PRF_A itself does not have access to the state.

```
**G_intermediate_4** :=
    $(b, state) \leftarrow PRF\_A^{rf(\cdot)}$
    $ret\ (b, hasDups(inputsOf(state))).$
```

In each of the three games, the probability that the adversary returns a guess of $true$ is the same.

Since some games will now return 2-tuples of booleans instead of single booleans, for such games G, we define $\Pr_1[G]$ to be the probability that the first value of the tuple is true and $\Pr_2[G]$ analogously.

Lemma 4.2.

$$\begin{aligned}\Pr[G\_intermediate\_2] &= \Pr[G\_intermediate\_3] \\ &= \Pr_1[G\_intermediate\_4].\end{aligned}$$

Proof. Adding state and duplicate tracking to the oracle does not change its outputs. Outputting whether the bad event happened does not change the adversary's guess. □

We similarly rewrite G_ideal in the form of $G\_intermediate\_2$. We create an oracle rb that, on any input, returns a uniformly sampled bitvector, and records the (input, output) pair in its state. Then we rephrase G_ideal in terms of PRF_A using rb.

```
**G_ideal_2** :=
    $b \leftarrow PRF\_A^{rb(\cdot)}$
    $ret\ b$
```

Then we similarly expose the bad event.

```
**G_ideal_3** :=
    $(b, state) \leftarrow PRF\_A^{rb(\cdot)}$
    $ret\ (b, hasDups(inputsOf(state)))$
```

In each of the three games, the probability that the adversary returns a guess of $true$ is the same.

Lemma 4.3.

$$\Pr[G\_ideal] = \Pr[G\_ideal\_2] = \Pr_1[G\_ideal\_3].$$

Proof. By unfolding the definition of PRF_A and program equivalence. (In both G_ideal and G_ideal_2, we generate a list of pseudorandom bits by uniformly randomly sampling a bitvector n times.) Also, adding state and duplicate tracking to the oracle does not change its outputs, and outputting whether the bad event happened does not change the adversary's guess. □

The two games, with the same bad event exposed, are identical until bad. Bellare's "syntactically identical except for the bad event" definition is too vague to use, so we assert that these games are identical until bad by a definition formalized by Petcher (2015):

1. The probability of the bad event is the same in both games.
2. If the bad event does not happen, the distribution of the outputs is the same.

Lemma 4.4. *$G\_intermediate\_4$ and G_ideal_3 are identical until bad.*

Proof. The random function behaves like uniformly sampling a bitvector, until one of its outputs happens to be one of its previous inputs. □

Thus, we can apply the fundamental lemma of game-playing, which implies

$$|\Pr_1[G\_intermediate\_4] - \Pr_1[G\_ideal\_3]| \leq \Pr[G\_ideal\_3 \textit{ sets bad}] = \Pr_2[G\_ideal\_3].$$

This is convenient, because now we can just work with a single game, which allows us to apply our game-equivalence techniques. Now we only need to bound $\Pr_2[G\_ideal\_3]$. Since we only care about its second return value, we can discard the adversary's guess.

```
G_ideal_3B :=
    $(b, state) \leftarrow PRF\_A^{rb(\cdot)}$
    $ret\ hasDups(inputsOf(state))$
```

Lemma 4.5. $\Pr_2[G\_ideal\_3] = \Pr[G\_ideal\_3B]$.

Proof. The first return value is irrelevant. □

G_ideal_3B is unnecessarily complicated. We wrote it in the form of PRF_A in order to get it in the same form as $G\_intermediate\_4$ and prove that they were identical until bad. Now we return to the simpler form, which is essentially the original G_ideal.

```
G_ideal_4 :=
    $n \leftarrow D$
    $bits \leftarrow generate\_bitvectors\ n$
    $ret\ (hasDups(bits))$
```

Lemma 4.6. $\Pr[G\_ideal\_3B] = \Pr[G\_ideal\_4]$.

Proof. By inlining PRF_A, it becomes clear that the probability of G_ideal_3B is the same as generating a list of n uniformly-sampled bitvectors of length c and returning whether there is a duplicate in that list. We can ignore the guess of the distinguisher D.

To be more precise, $inputsOf(state))$ in G_ideal_3B is a list of $n-1$ such bitvectors (inputs) with a randomly-sampled seed v at the beginning, which is the first input. □

Thus, we need only upper-bound the probability that there is a duplicate in a list of n uniformly-sampled bitvectors of length c. This is the probability that the bad event happens.

Lemma 4.7.

$$\Pr[G\_ideal\_4] \leq \frac{n^2}{2^c}$$

Proof. There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of bitvectors in the list, and the probability that there is a collision in a pair is $\frac{1}{2^c}$. By the union bound, the probability that there is no collision is $\frac{n(n-1)}{2} \cdot \frac{1}{2^c} \leq \frac{n^2}{2^c}$. □

This completes our bound on the difference between $G\_intermediate$ and G_ideal. □

This bound on the probability of the bad event yields our final result:

Theorem 5. *The distance between the two games is*

$$|\Pr[G\_real] - \Pr[G\_ideal]| \leq PRF\_Advantage + \frac{n^2}{2^c}.$$

Proof. By the triangle inequality, combined with theorems 3 and 4. □

Thus, H is a secure PRG. Moreover, since we did our proof in the concrete security model, we have a better idea of how secure it is (modulo the black box of $PRF\_Advantage$) than if we had proven its asymptotic security.

B. Indistinguishability proof for a simplified HMAC-DRBG

1. Definitions

To prove indistinguishability for HMAC-DRBG, we must extend the proof for H to apply for multiple calls of H, with (k, v) being updated after each call.

As summarized in section REFERENCE, the HMAC-DRBG functions are $Instantiate$, $Generate$, $Update$, and $Reseed$. The state is (k, v).

We make two main simplifications to HMAC-DRBG (in addition to the simplifications we made when first describing HMAC-DRBG). First, we do not model reseeding the PRG, because we are not proving that it possesses prediction resistance. Second, we do not model non-ideal randomness. As a corollary, we assume that $Instantiate$ initializes k and v with ideal randomness, which makes the proof much easier. Lastly, for convenience, we only output lists of pseudorandom bits whose lengths are multiples of the block size c.

```
**Instantiate** :=
    $k \xleftarrow{R} \{0,1\}^c$
    $v \xleftarrow{R} \{0,1\}^c$
```

For the remaining two functions, *Generate* and *Update*, we note that *Generate* always calls *Update* at the end. Thus, we inline *Update* to produce one function called *GenUpdate_original*. (It is labeled *original* because we will be using a different version in our proof, which we cover in subsection REFERENCE.)

```
$GenUpdate\_original((k, v), n) :=$
    $(bits, v_1) \leftarrow Gen\_loop(k, v, n)$
    $k_1 \leftarrow f_k(v_1 || 00\dots00)$
    $v_2 \leftarrow f_{k_1}(v_1)$
    $ret\ (bits, (k_1, v_2))$
```

In *GenUpdate_original*, *Gen_loop* behaves like H (the core loop from *Generate*) but also separately returns the last block it generates. It does this because the last block becomes the new v in the (k, v) state.

```
$Gen\_loop(k, v, n) :=$
    if $n = 0$ then $ret\ (nil, v)$
    else
        $v_i \leftarrow f_k(v)$
        $(bits, v_{final}) \leftarrow Gen\_loop(k, v_i, n-1)$
        $ret\ (v_i || bits, v_{final})$
```

Similarly to H:

$$Gen\_loop(k, v_0, 0) = (nil, v_0)$$
$$Gen\_loop(k, v_0, 1) = (f_k(v_0), f_k(v_0))$$
$$Gen\_loop(k, v_0, 2) = (f_k(v_0)||f_k(f_k(v_0)), f_k(f_k((v_0))))$$

In our simplified model of HMAC-DRBG, which we will call HMAC-DRBG$'$, a typical run would look like *Instantiate* followed by some number of calls to *GenUpdate*, each call requesting an some number of pseudorandom blocks.

This suggests the definition of security (indistinguishability) for HMAC-DRBG$'$. A non-uniform PPT adversary can make a number of calls to GenUpdate (polynomial in c), requesting a number of blocks (polynomial in c) each time. It receives the list of blocks, which are the bitvectors it requested. Then it must output a guess as to whether the bits are ideally random or pseudorandom.

We need to now consider whether the adversary is adaptive or non-adaptive. That is, should the adversary be allowed to make choices (e.g. numbers of blocks to query) based on previous results (e.g. pseudorandom bits returned so far)? This is a new concern. Previously, when proving the security of H, the adversary could only make one call to H. So it couldn't make any decision after picking n (the number of blocks).

Here, we choose to prove security against a non-adaptive adversary. Proving security against an adaptive adversary, and formalizing that proof, would be substantially harder. We discuss reasons for this decision in section REFERENCE.

That said, here is the formal statement of what we want to prove.

Definition 6 (Indistinguishability for HMAC-DRBG'). *For all non-uniform, non-adaptive PPT distinguishers D, for all lists of natural numbers l of length polynomial in c (the list of numbers of blocks requested) that it outputs, let n be the total number of blocks requested. ($n := \sum_i^{|l|} l_i$)*

Then there exists a function negligible in c and l such that

$$|\Pr[G\_real] - \Pr[G\_ideal]| \leq neg(c, n).$$

The real-world game models the real-world usage we discussed earlier, again with a non-adaptive adversary.

```
***G_real*** :=
    $l \leftarrow A$
    $(k, v) \leftarrow Instantiate$
    $bits \leftarrow oracleMap(GenUpdate\_original, (k, v), l)$
    $ret\ A(bits)$
```

Given an input state, the function oracleMap repeatedly applies the provided oracle f on a list of inputs l, and returns its final state and a list of outputs. An oracle is a function that takes a state and an input and outputs a new state and an output. [1]

In the ideal-world game, GenRand takes the list l outputted by the adversary and generates $\sum_0^{|l|-1} l_i$ blocks of ideal randomness.

```
***G_ideal*** :=
    $l \leftarrow A$
    $bits \leftarrow genRandForEach(l)$
    $ret\ A(bits)$
```

2. *Proof outline*

Again, we must extend the proof for the security of H to the security of multiple calls of H with (k, v) being updated after each call. This problem often arises when proving the security of PRGs, and it is solved by using a powerful technique called the *hybrid argument*. We will

[1] For brevity, we will elide the code of most function definitions in this proof. We discuss some of them in the section on formalization (Section REFERENCE), and all of them are defined in the Coq development. This function is also named *oracleMap* in the Coq development.

briefly summarize the use of the hybrid argument in the setting of concrete security.

We want to bound the probability that some distribution D_1 can be distinguished from another distribution D_2. Thus, we define t intermediate, or hybrid, distributions H_i, with $H_0 := D_1$ and $H_t := D_2$. (t must be polynomial in the security parameter.)

Define the probability of A distinguishing between two distributions D_i, D_j to be

$$Adv_{D_i,D_j}(A) := |\Pr_{x \xleftarrow{R} D_i}[A(x) = 1] - \Pr_{x \xleftarrow{R} D_j}[A(x) = 1]|.$$

Then by the triangle inequality, the probability that a non-uniform PPT adversary A can distinguish D_1 and D_2 is bounded by the sum of the probabilities of distinguishing between adjacent hybrids.

$$Adv_{D_1,D_2}(A) \le \sum_{0}^{t-1} Adv_{H_i,H_{i+1}}(A)$$

The above discussion pertained to the general hybrid argument. Now we specialize it to our proof.

Let n be the number of calls the adversary makes to $GenUpdate$ (so $n := len(l)$) and l_i be the number of blocks of pseudorandom output that the adversary requests on that call (l_i is the ith element of the list). Calls are numbered from 1 to n.

We take the first hybrid H_0 to be the distribution of the bits generated by using the PRG normally; that is, using the PRF to generate pseudorandom bits. Subsequently, we define one new hybrid for each call to the PRG, so $t = n+1$. H_1 replaces the PRF used in the first call with ideal randomness, and H_i replaces the PRFs used in calls 1 through i (inclusive) with ideal randomness. The last hybrid, H_n replaces all PRFs with ideal randomness.

Thus, $\Pr[G\_real]$ equals $\Pr_{x \xleftarrow{R} H_0}[A(x) = 1]$.

Similarly, $\Pr[G\_ideal]$ equals $\Pr_{x \xleftarrow{R} H_n}[A(x) = 1]$.

Thus, the endpoint hybrids correspond correctly to the quantity defined in Theorem 6:

$$|Pr[G\_real] - \Pr[G\_ideal]| = Adv_{H_0,H_n}(A).$$

Example 6.1. *Let's define a notation for the ith hybrid and write out all the hybrids for some n. Take $n = 3$ (so the adversary makes 3 calls to $GenUpdate$ with some number of blocks requested). Then there are four hybrids.*

In H_1, the first call to $GenUpdate$ generates bits in Gen_loop by uniformly sampling from $\{0,1\}^c$ instead of calling f_k. Denote this as RB (ideally random bits). Subsequent calls use the normal Gen_loop construction, which calls f_k. Hence we denote H_1 as
RB PRF PRF.

call	*1*	*2*	*3*
H_0	*PRF*	*PRF*	*PRF*
H_1	*RB*	*PRF*	*PRF*
H_2	*RB*	*RB*	*PRF*
H_3	*RB*	*RB*	*RB*

The notation hides the fact that state being kept between calls and updated in calls. (k, v) is instantiated, then passed into call 1 of $GenUpdate$, updated there and passed into call 2 of $GenUpdate$, and so on for the rest of the calls.

Intuitively, we replace PRFs with RBs from left to right. The order matters. In H_0, (k, v) are passed in from $Instantiate$, which uniformly randomly samples them. In H_i, (k, v) are passed in to the call $i+1$ (the first call using the PRF) from call i (the last call using RB), which uniformly randomly samples k and v when they are updated. That makes it easier to reason about subsequent PRF-using calls, which we will have to do later.

Again, we are trying to bound

$$Adv_{D_1,D_2}(A) \le \sum_{0}^{t-1} Adv_{H_i,H_{i+1}}(A).$$

Now we bound the quantity being summed, which is the distance between adjacent hybrids. Intuitively, we have just reduced it to our earlier proof about the security of H, and the bound is the same.

Theorem 7 (Distance between adjacent hybrids.)**.** *Let l_i be the number of blocks that the adversary requests on the ith call. Then for all $i \in [0, n-1]$,*

$$Adv_{H_i,H_{i+1}}(A) \le PRF\_Advantage\_Max + \frac{(l_i+1)^2}{2^c}.$$

Proof. In place of the PRF, H_i uses ideal randomness for calls $\le i$. (Not just the bits are ideally random; the state is also updated with ideal randomness.) So its output is ideally random for calls $\le i$. Afterward, it uses the PRF.

In place of the PRF, H_{i+1} uses ideal randomness for calls $\le i+1$. So its output is ideally random for calls $\le i+1$. Afterward, it uses the PRF.

We phrase $Adv_{H_i,H_{i+1}}(A)$ in terms of games. Game i replaces the PRF with ideal randomness for calls $\le i$ and uses the PRF for calls $> i$.

```
G_hybrid(i) :=
    $l \leftarrow A$
    $(k, v) \leftarrow Instantiate$
    $bits \leftarrow oracleMap(oracle\_i(i), (0, (k, v)), l)$
    $ret\ A(bits)$
```

$oracle\_i(i)$ uses $GenUpdate\_rb$ (a version of $GenUpdate$ that uniformly samples bits instead of using the PRF) as the oracle for calls $\le i$ and $GenUpdate$ as the oracle for calls $> i$. (This function may be found in the Coq development as Oi_prg. From now on, we will denote Coq names in the format of ($^*Oi\_prg$).)

Lemma 7.1. *For all i,*

$$Adv_{H_i,H_{i+1}}(A) = |\Pr[G\_hybrid(i)] - \Pr[G\_hybrid(i+1)]|.$$

Proof. The bits generated in $G\_hybrid(i)$ are sampled from the distribution H_i. Those bits are then passed to the adversary A, which returns a guess. □

Between H_i and H_{i+1}, we introduce an intermediate hybrid H_i'. Intuitively, it acts like H_i, but with the PRF on call $i+1$ replaced by a random function. More specifically, for calls $\leq i$, just like H_i, H_i' uses ideal randomness in place of the PRF. On call i, H_i' uses a random function in place of the PRF. For calls $> i$, H_i' uses the PRF normally.

Now we phrase H_i' in terms of a game.

First we define a more general game much the same as above, but replacing the PRF on call $i+1$ with a provided oracle o.

```
G_hybrid_useO(o, i) :=
    $l \leftarrow A$
    $(k, v) \leftarrow Instantiate$
    $bits \leftarrow oracleMap(oracle\_i\_useO(o, i), (0, (k, v)), l)$
    $ret\ A(bits)$
```

$oracle\_i\_useO$ ($^*Oi\_oc'$) replaces the oracle used in call $i+1$ with the provided oracle, and behaves like $oracle\_i$ otherwise (replaces the PRF with ideal randomness for calls $\leq i$, and uses the PRF normally for calls $> i+1$). To do this replacement, we replace $GenUpdate$ in $oracle\_i\_useO$ with $GenUpdate\_oc$, which uses the provided oracle in place of the PRF.

Now we provide a random function as the oracle for G_hybrid_useO. So the bits $G\_hybrid\_useO(randomFunc, i)$ generates are sampled from H_i'. This serves as our intermediate game between $G\_hybrid(i)$ and $G\_hybrid(i+1)$. We will nickname it $G\_hybrid\_rf(i)$.

To get all the games in a uniform format, we will take $G\_hybrid'(i) := (G\_hybrid\_useO(f\_oracle, i))$ and $G\_hybrid'(i+1) := (G\_hybrid\_useO(rb\_oracle, i))$. f_oracle is the PRF oracle and rb_oracle is an oracle that uniformly samples random blocks on each query. Both store their input/output state. These two rewrites can be proven equal, and we will use the game and its oracle-rewritten form interchangeable. (The proof is omitted for brevity.)

As in Section REFERENCE, we bound the distance between $G\_hybrid(i)$ and $G\_hybrid(i+1)$ via the triangle inequality. The proof is also very similar to our proof in Theorem REFERENCE.

Lemma 7.2.

$$\begin{aligned}|\Pr[G\_hybrid(i) - \Pr[G\_hybrid(i+1)]| \leq \\ |\Pr[G\_hybrid(i) - \Pr[G\_hybrid\_rf(i)| \\ +|\Pr[G\_hybrid\_rf(i) - G\_hybrid(i+1)|.\end{aligned}$$

Proof. By the triangle inequality. □

Here is a visual explanation of this proof for a small example.

Example 7.1. *We continue our hybrid example from earlier (Example 6.1). Again $n = 4$.*

Reproducing the table from earlier, here are all four hybrids.

call	*1*	*2*	*3*
H_0	*PRF*	*PRF*	*PRF*
H_1	*RB*	*PRF*	*PRF*
H_2	*RB*	*RB*	*PRF*
H_3	*RB*	*RB*	*RB*

Take $i = 1$. Here are H_1, H_1' (the intermediate hybrid), and H_2. RF denotes the random function oracle.

call	*1*	*2*	*3*
H_1	*RB*	*PRF*	*PRF*
H_1'	*RB*	*RF*	*PRF*
H_2	*RB*	*RB*	*PRF*

Intuitively, the difference between H_1 and H_1' is upper-bounded by $PRF\_Advantage$ because each can be written as a game distinguishing the PRF from the RF. The difference between H_1' and H_2 is exactly the probability of collisions in the inputs of the random function.

For completeness, here is every hybrid and every intermediate hybrid. There are four hybrids and three intermediates. Note that the last hybrid has no intermediate, because there are no PRFs left to replace with RFs.

call	*1*	*2*	*3*
H_0	*PRF*	*PRF*	*PRF*
H_0'	*RF*	*PRF*	*PRF*
H_1	*RB*	*PRF*	*PRF*
H_1'	*RB*	*RF*	*PRF*
H_2	*RB*	*RB*	*PRF*
H_2'	*RB*	*RB*	*RF*
H_3	*RB*	*RB*	*RB*

Now we tackle each part of the bound separately in Lemma 7.2 separately.

Lemma 7.3.

$$\begin{aligned}|\Pr[G\_hybrid(i)] - \Pr[G\_hybrid\_rf(i)]| &\leq PRF\_Advantage\_i\ i \\ &\leq \max_i PRF\_Advantage\_i\ i.\end{aligned}$$

Proof. This proof is not as easy as the proof-by-definition in Lemma REFERENCE. There are two added complexities. Because the (k, v) is updated at an inconvenient place, in order to rewrite the two games above in the form of a $PRF\_Adversary$, we must move one of the lines of update code. Additionally, there is one $PRF\_Adversary$ for each i, so there are multiple $PRF\_Advantages$.

Here we work with $G\_hybrid'(i)$ instead of $G\_hybrid(i)$ because it's in the same form as G_hybrid_rf.

First, we deal with the (k, v)-updating. We reproduce the code of $GenUpdate\_original$ here.

```
$GenUpdate\_original((k, v), n) :=$
```

```
    $(bits, v_1) \leftarrow Gen\_loop((k, v), n)$
    $k_1 \leftarrow f_k(v_1||00...00)$
    $v_2 \leftarrow f_{k_1}(v_1)$
    $ret\ (bits, (k_1, v_2))$
```

Note that the PRF is re-keyed on the second line. Then, on the next line, the function queries the *new* PRF. Therefore, if we were to replace f_k with a PRF oracle, it would have to span the last line of *GenUpdate_original* and the first two lines of the next *GenUpdate_original* call, but not include the second call's v-update. This would be messy and difficult to reason about. It is cleaner to move each v-update to the beginning of the next call of *GenUpdate_original*, then prove that the outputs are still identically distributed. We accomplish this by splitting *GenUpdate_original* into two versions.

The first version, *GenUpdate_noV*, omits the last line, which updates v.

```
$GenUpdate\_noV((k, v), n) :=$
    $(bits, v_1) \leftarrow Gen\_loop((k, v), n)$
    $k_1 \leftarrow f_k(v_1||00...00)$
    $ret\ (bits, (k_1, v_1))$
```

The second version, *GenUpdate* starts by updating v and does not update v again.

```
$GenUpdate((k, v), n) :=$
    $v_1 \leftarrow f_k(v)$
    $(bits, v_2) \leftarrow Gen\_loop(k, v_1, n)$
    $k_1 \leftarrow f_k(v_2||00...00)$
    $ret\ (bits, (k_1, v_2))$
```

We prove that the pseudorandom output produced by this sequence of calls

$$[GenUpdate\_original, GenUpdate\_original, ...]$$

is identically distributed to the pseudorandom output produced by this new sequence of calls

$$[GenUpdate\_noV, GenUpdate, GenUpdate, ...].$$

(Note that the intermediate (k, v) state between the PRG calls is *not* the same, but the two outputs are still identically distributed.) The proof is omitted because it is a proof of program equivalence, but it is present in the Coq development as *GenUpdate_v_output_probability*.

We rewrite *G_real* to replace *GenUpdate_original* with the two new versions.

```
**G_real_noV** :=
    $l \leftarrow A$
    $(k, v) \leftarrow Instantiate$
    $(head\_bits, state') \leftarrow$
        $GenUpdate\_noV((k, v), (head\ l))$
    $(tail\_bits, \_) \leftarrow oracleMap(GenUpdate, state', (taill))$
    $ret\ A(head\_bits :: tail\_bits)$
```

Lemma 7.4. $\Pr[G\_real] = \Pr[G\_real\_noV]$.

Proof. By program equivalence on the outputs. □

Pretend that this is what *G_real* has been all along, and that all subsequent rewrites of *G_real* have followed this structure. (We can simply add this as a game in the beginning. The v-updating is mentioned here, and not in the beginning of the proof, because it is better motivated in context.)

Now, as we originally desired, we can write each *GenUpdate* (and *GenUpdate_noV*) to query a provided oracle instead of the PRF.

```
$GenUpdate\_oc(o, (k, v), n) :=$
    $v_1 \leftarrow o.query(v)$
    $(bits, v_2) \leftarrow Gen\_loop\_oc(o, k, v_1, n)$
    $k_1 \leftarrow o.query(v_2||00...00)$
    $ret\ (bits, (k_1, v_2))$
```

Note that after the PRF is re-keyed, we do *not* further query it in this call of *GenUpdate*. This is much neater than before.

Finally, we can rewrite $G\_hybrid'(i)$ and $G\_hybrid\_rf(i)$ in terms of *PRF_Adversary*.

```
**PRF_Adversary(o, i)** :=
    $l \leftarrow A$
    $(k, v) \leftarrow Instantiate$
    $bits \leftarrow oracleMap(oracle\_i\_useO(o, i), (0, (k, v)), l)$
    $ret\ A(bits)$

**G_hybrid_oc'(i)** :=
    $k \xleftarrow{R} \{0, 1\}^n$
    $(b, \_) \leftarrow PRF\_Adversary(f\_oracle(k), i)$
    $ret(b)$

**G_hybrid_rf_oc(i)** :=
    $(bits, \_) \leftarrow PRF\_Adversary(randomFunc\_oracle, i)$
    $ret(b)$
```

Lemma 7.5. *for all* i,

$$|\Pr[G\_hybrid\_oc'(i)] - \Pr[G\_hybrid\_rf\_oc(i)]| \leq PRF\_Advantage(i).$$

Proof. More precisely, *PRF_Advantage* is parametrized by the *PRF_Adversary* passed in, which is parametrized by the hybrid index i. The bits generated differ depending on i.

This is similar to Lemma REFERENCE, but does not reduce to it. □

As is evident from the lemma above, there are multiple PRF adversaries, one for each i. So, we take the final bound to be the greatest of the *PRF_Advantage*s:

$$PRF\_Advantage\_max := \max_i PRF\_Advantage(i).$$

This completes our proof bounding the distance between hybrid i and the intermediate hybrid for i. □

Now we bound the distance between the intermediate hybrid and the next real hybrid.

Lemma 7.6. *for all i,*

$$|\Pr[G\_hybrid\_rf(i)] - \Pr[G\_hybrid(i+1)]| \leq \frac{(l_i+1)^2}{2^c}.$$

Proof. We define the bad event, show that the two games are identical until bad, and bound the probability of the bad event happening.

Definition 8. *The **bad event** is that there are duplicates in the inputs to the oracle used on the $i+1$th call. Intuitively, we need to "zoom in" on the inputs and outputs to the oracles in only the $i+1$th call of each game. Then, the bad event becomes the same as the bad event discussed in Theorem REFERENCE, in the proof about the simpler PRG.*

Theorem REFERENCE gives a detailed overview of why the bad event is bad, the fundamental lemma of game-playing, the definition of "identical until bad", the technique of exposing the bad event, and the $\Pr_1$ and $\Pr_2$ notation. This section will omit that discussion.

As before, we expose the bad event in both games by returning both the generated bits and a boolean indicating whether the bad event occurred in the oracle's state.

First, we expose the bad event in $G\_hybrid(i+1)$, rewritten to use an oracle that returns ideal randomness when queried and keeps track of all input-output pairs. The second return value of $PRF\_Adversary$ is the oracle's state. (Implicit program equivalence: $G\_hybrid(i+1)$ is the same as replacing the oracle on the $i+1$th call with random bits.)

```
G_hybrid_rb_oc_2(i) :=
    $(bits, state) \leftarrow PRF\_Adversary(rb\_oracle, i)$
    $ret(b, hasDups(inputsOf(state))$
```

Similarly, we expose the bad event in G_hybrid_rf, rewritten to use a random function oracle that keeps track of duplicate inputs. (Again, on the $i+1$th call, it uses this RF oracle.)

```
G_hybrid_rf_oc_2(i) :=
    $(bits, state) \leftarrow PRF\_Adversary(randomFunc\_withDups, i)$
    $ret(b, hasDups(inputsOf(state))$
```

Lemma 8.1. *for all i, $G\_hybrid\_rf\_oc\_2(i)$ and $G\_hybrid\_rb\_oc\_2(i)$ are identical until bad.*

Proof. They satisfy the two requirements. The reasoning is the same as in Theorem REFERENCE, because we only have to reason about the one call between the computations that uses the two different oracles. The only difference between here and Theorem REFERENCE is that the oracle is queried either one or two additional times when we update k and v at the end of $GenUpdate$.

1. If the bad event does not happen, their output distributions are identical. If there are no duplicates in the inputs to the oracle (rb_oracle or $randomFunc_withDups$), they behave the same.
2. The probability of the bad event happening is the same in both. Again, this is obvious.

Incidentally, this lemma is very obvious on paper, but took a lot of effort to formalize. We had to do a tricky induction over the list of blocks l that the adversary requested. See the discussion of $oracleCompMap\_\_oracle\_eq\_until\_bad\_dups$ in Section REFERENCE. □

Then we may bound the distance between the two games by the probability of the bad event happening.

Lemma 8.2. *for all i,*

$$|\Pr_1[G\_hybrid\_rf\_oc\_2(i)] - \Pr_1[G\_hybrid\_rb\_oc\_2(i)]| \leq \Pr_2[G\_hybrid\_rb\_oc\_2(i)].$$

Proof. Because we just proved that the two games are identical until bad, we apply the fundamental lemma, which yields this result. □

Now we need only reason about the bad event in the second game, which replaces the PRF with ideal randomness in the $i+1$th oracle call. We rewrite the game into several simpler forms and prove that each preserves the probability of the bad event happening, then reduce it to the bound proven earlier in Lemma REFERENCE.

Intuitively, we only care about the computation that queries the oracle, which is one iteration of $GenUpdate\_oc$. We can discard the other calls to $GenUpdate\_oc$. By the ordering of the hybrid argument, we know that the state coming into this call is ideally random. (This was an important reason we picked this hybrid ordering.)

Thus, we inline $GenUpdate$ and replace Gen_loop with $genRandForEach$ (since we replaced the PRF with ideal randomness).

```
G_hybrid_rb_oc_bad_1(i) :=
    $l \leftarrow A$
    $firstInput\_v \xleftarrow{R} \{0,1\}^n$
        (because the $v$ coming in is ideally random)
    $outputsAsInputs \leftarrow tail(genRandForEach(l))$
        (the $v$ is the first input; we throw away one
        element, which is an output)
    $v_2 \xleftarrow{R} \{0,1\}^n$
    $keyInput \leftarrow v_2||0\dots0$
        (updating the key)
    $ret(hasDups([keyInput, firstInput\_v]||outputsAsInputs)$
```

By NIST's design, the PRF is re-keyed as such:

$$k_1 \leftarrow f_k(v||0\dots0)$$

$(v||0\ldots0)$ can never be a duplicate input for that particular PRF, because the PRF was always queried previously with inputs of length c, and this input is longer. (Our guess is that NIST made this design decision so that for that PRF, the new key would always be ideally random.) Therefore, we can remove the key input from our list of possible duplicate inputs.

We are left with essentially the same situation as in Lemma REFERENCE. The total number of inputs is the sum of the total number of blocks requested (minus 1 because the last one is an output; plus 1 for the initial v), plus one more input because *GenUpdate* made one more query to the oracle to update v at the end. (The v-updating is not present in *GenUpdate_noV*, but the bound still holds.)

```
G_hybrid_rb_oc_bad_2(i) :=
    $l \leftarrow A$
    $inputs \leftarrow generate\_bitvectors(1 + sum(l))$
    $ret(hasDups(inputs))$
```

Each rewrite preserves the probability of the bad event happening.

Lemma 8.3. *for all* i,

$$\Pr_2[G\_hybrid\_rb\_oc\_2(i)] = \Pr[G\_hybrid\_rb\_oc\_bad\_1(i)]$$
$$= \Pr[G\_hybrid\_rb\_oc\_bad\_2(i)]$$

Proof. This is one lemma which is intuitively obvious, but subtle. It's easy to inline *GenUpdate* wrong by hand, mistake an output for an input, or make an off-by-one error. (The last error doesn't matter much in the final bound, but it's still good to be accurate.)

Here is where formal verification shines. In the explanation above, we combined 5-6 possible intermediate games into 2 games for brevity; in Coq, we can write out each one and prove each adjacent pair equivalent. These proofs can be found in the Coq development under the theorem *Gi_rb_bad_collisions*. □

Finally, we bound the probability of the bad event happening.

Lemma 8.4. *for all* i,

$$\Pr[G\_hybrid\_rb\_oc\_bad\_2(i)] \leq \frac{(l_i+1)^2}{2^c}$$

Proof. The game is in almost exactly the same form as G_ideal_4, so we reuse the collision probability bound with $n := 1 + sum(l)$. □

This completes our bound on the distance between hybrid $i+1$ and the intermediate hybrid for i. □

This completes our bound on the distance between adjacent hybrids H_i and H_{i+1}.

□

Theorem 9 (Distance between first and last hybrid.).

$$Adv_{D_1,D_2}(A) \leq t \cdot PRF\_Advantage\_Max + \sum_0^{t-1} \frac{(l_i+1)^2}{2^c}$$
$$\leq t \cdot (PRF\_Advantage\_Max + \frac{(maxBlocks+1)^2}{2^c}).$$

Proof. The first inequality follows by the triangle inequality combined with Theorem 7.

The second inequality follows by the fact that the PRG imposes an upper limit on the number of blocks that may be requested per call. This is 2^{19} bits for HMAC-DRBG, according to NIST SP 800-90A (39).

This is a negligible quantity for sufficiently large c. Usually c is 256, since HMAC's internal hash function is typically SHA-256.

This completes our bound on the total probability of distinguishing between the real-world and ideal-world games. □

Hirose (2008) estimates the probability of being able to distinguish for the core loop to be around TODO.

I'm not sure what *PRF_Advantage_Max* is. Under the assumption that HMAC is a PRF, the max probability of being able to distinguish ought to be negligible in the number of queries made. I'll estimate this as TODO.

Assuming real-world usage: the adversary makes 30 billion requests of 10,000 blocks each. The DRBG is never compromised or reseeded. Then the probability of the adversary being able to distinguish is about TODO.

3. *Formalization*

These are more code-related. Should I talk about them later?

Formalization reminded me of one subtlety: there are multiple PRF adversaries and that we need to take

Formalization helped me check the indices,

Some things I got wrong: v_2 vs v_1 in *GenUpdate*; relating which hybrids and intermediate hybrids

In more detail:

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)

Is there some way to pseudocode *oracleCompMap_outer*?

```
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.
```

New oracles: Oi_prg Oi_oc

C. Proof of backtracking resistance

It is not enough for a real-world DRBG to possess pseudorandomness. As a motivating example, say Alice and Bob want to communicate secretly. Perhaps they are sneaky vegans who want to trade insider information about the future stock prices of Earth Balance Inc. Thus, they use a simple hybrid cryptosystem to communicate, consisting of some public-key encryption scheme P and some symmetric-key encryption scheme S. (Such schemes are used in practice; one example is OpenPGP. They are used because the symmetric-key scheme is typically faster than the public-key scheme.)

On Bob's end, he obtains Alice's public key through some trusted authority, generates a fresh symmetric key, encrypts the message using the symmetric key, encrypts the symmetric key using the public key, and sends both encryptions to Alice.

On Alice's end, she receives both encryptions. She first decrypts the second encryption with her private key to obtain the symmetric key, then uses the symmetric key to decrypt the message. At that point, Alice and Bob communicate only within the scheme S. After a certain number of messages, for good hygiene, one generates a new symmetric key, encrypts that using the old symmetric key, and sends it over. From then on they use the new symmetric key to communicate.

Symmetric keys are essentially generated by a DRBG, with possibly some deterministic function applied to the pseudorandomness afterward. Say the DRBG only possesses the property of pseudorandomness. The DRBG's output here must be kept secret, but if only the output becomes public, then the adversary only knows the symmetric key for time T. However, if the DRBG is fully compromised at time T (meaning that an adversary receives its key and seed at that time), the adversary can also apply the DRBG's $Generate$ function to generate the symmetric keys to encrypt and decrypt messages sent *after* time T. And the adversary may be able to run the DRBG "backward" to generate the symmetric keys used to encrypt and decrypt messages sent *before* time T.

We give a formal definition of an adversary not being able to run a PRG "backward".

Definition 10. ***Backtracking resistance.*** *Informally, if a PRG is compromised at time T, the adversary still cannot distinguish its previous output from ideal random strings.*

Specializing to HMAC-DRBG: any non-uniform non-adaptive PPT adversary A 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). Even given this additional state, the adversary has a negligible probability of distinguishing the previous output from ideal randomness. This definition is based on the definition given by Barak (2005).*

Here is the real-world game. Since A_{BR} (the backtracking resistance adversary) is non-adaptive, there is no separate Compromise *oracle; the length of the list implicitly indicates the number of calls after which A wishes to see the internal state (k, v) of the PRG. Then it is given the generated pseudorandom bits and the final (k, v) state.*

```
G_real_BR :=
    $l \leftarrow A$
    $(k, v) \leftarrow Instantiate$
    $(bits, (k_f, v_f)) \leftarrow$
        $oracleMap(GenUpdate\_original, (k, v), l)$
    $ret\ A_{BR}(bits, (k_f, v_f))$
```

In the ideal-world game, A_{BR} is given a string of ideal randomness of the same length, as well as (k, v) that are ideally random.

```
G_ideal_BR :=
    $l \leftarrow A$
    $(k, v) \leftarrow Instantiate$
    $bits \leftarrow genRandForEach(l)$
    $ret\ A_{BR}(bits, (k, v))$
```

The PRG modeled on HMAC-DRBG is backtracking-resistant if for all non-uniform non-adaptive PPT A_{BR} that asks for a polynomial number of blocks over a polynomial number of GenUpdate *calls, A_{BR} has a negligible chance of distinguishing between the worlds.*

$$|\Pr[G\_real\_BR] - \Pr[G\_ideal\_BR]| \leq neg(n, l).$$

By this definition of backtracking resistance, the real-world and ideal-world games are very similar to the respective games in the definition of pseudorandomness, discussed in section REFERENCE. The only difference is that in the real-world game, the new adversary is given the final state, and in the ideal-world game, the new adversary is given an ideally random state. Thus, intuitively, the proof of backtracking resistance can reuse many of the steps and reductions we used to prove pseudorandomness.

Theorem 11. *HMAC-DRBG is backtracking resistant.*

$$\begin{aligned}
&|\Pr[G\_real\_BR] - \Pr[G\_ideal\_BR]| \\
&\leq t \cdot PRF\_Advantage\_Max + \sum_{0}^{t-1} \frac{(l_i + 1)^2}{2^c} \\
&\leq t \cdot (PRF\_Advantage\_Max + \frac{(maxBlocks + 1)^2}{2^c}).
\end{aligned}$$

Proof. This is a proof sketch, not the full proof.

Note that in every proof we reuse from the pseudorandomness proof, we have to check that the result holds for an adversary that also receives the state. We run into such a problem in the first step.

We reuse Lemma REFERENCE to move each v-update in $GenUpdate$ to the beginning of the next call. Because this is a program equivalence proof, the result holds as-is on the output. However, the *state* at the end of a $GenUpdate\_noV$ call is not the same as it would be at the end of a $GenUpdate$ call.

This breaks backtracking resistance. Referring to $GenUpdate\_original$ (in Section REFERENCE), the adversary would receive v_1, not v_2. But if A_{BR} knows that the last block of the PRG's output is v_1. So if A_{BR} is in the real-world game, it can simply check if this is true. This happening has a negligible probability of happening in the ideal-world game, since the v that A_{BR} receives is ideally random. To fix this, we simply update v_1 before giving the state to A_{BR} [2]

Here is the new game with v-updating.

```
G_real_BR_noV :=
    $l \leftarrow A$
    $(k, v) \leftarrow Instantiate$
    $(head\_bits, state') \leftarrow$
        $GenUpdate\_noV((k, v), (head\ l))$
    $(tail\_bits, (k_f, v_1)) \leftarrow$
        $oracleMap(GenUpdate, state', (taill))$
    $v_f \leftarrow f_k(v_1)$
    $ret\ A_{BR}(head\_bits :: tail\_bits, (k_f, v_f))$
```

Afterward, we take the same strategy. We define the same hybrids: H_i replaces the PRF on calls $\leq i$ with ideal randomness, and using the PRF normally afterward. Then we use the triangle inequality so that we only need bound the distance between adjacent hybrids H_i and H_{i+1}.

The first step in the bound between adjacent hybrids is the same as in the pseudorandomness proof. We use the same intermediate hybrid H_i': we rewrite $G\_real\_BR\_noV$ in the form of a game $G\_real\_BR\_noV\_RF(i)$ that passes a random function oracle to some PRF adversary, which uses ideal randomness in place of the PRF on calls $\leq i$, the provided oracle on call $i+1$, and the PRF afterward. We rewrite the beginning hybrid H_i in the same form, just passing the PRF adversary the PRF as an oracle instead.

The distance between H_i and H_i' is $PRF\_Advantage\_Max$ by definition and by taking the maximum $PRF\_Advantage$ over all hybrids (discussed in Section REFERENCE). In this step, we don't need to reason about the A_{BR} being given the state; no matter what happens, the bound is $PRF\_Advantage\_Max$ by definition.

The second step in the bound between adjacent hybrids is to rewrite the ideal-world game in the form of a game that instead passes the PRF adversary the oracle rb_oracle for all calls. (This oracle uniformly samples random bits and records the input-output pairs.) Then, we reason that this new game, $G\_real\_BR\_noV\_RB$, is related to the intermediate game because they are identical until bad. This is true. To see this, we consider the distribution of the output $(bits, (k_f, v_f))$ instead of just $bits$ as before.

1. The bad event (duplicates in the inputs to the provided oracle) has the same chance of happening in both games. This is true. Outputting the end state does not affect whether the bad event happens.

2. If the bad event does not happpen, the output distributions are identical between the games. We already proved this for $bits$. For (k_f, v_f), the key is always ideally random because its input is length-extended. v_f is ideally random because it is the first query of a freshly re-keyed PRF. This is true across both games.

At this point, we have essentially replaced the first hybrid with the last hybrid, adding on the collision bounds along the way. The last hybrid here is a slightly different game from the ideal-world game, whereas in the pseudorandomness proof the last hybrid was the same as the ideal-world game by program equivalence. The last hybrid here still possesses the last v-update after the $GenUpdate\_noV$ calls.

```
G_last_hybrid_BR :=
    $l \leftarrow A$
    $bits \leftarrow genRandForEach(l)$
    $k_f \xleftarrow{R} \{0,1\}^n$
    $lastV \leftarrow lastBlockOf(bits)$
    $v_f \leftarrow randomFunc\_withDups(lastV)$
    $ret\ A_{BR}(bits, (k_f, v_f))$
```

Clearly, since v_f is the first query to a random function, it is ideally random. Thus $(bits, (k_f, v_f))$ has the same distribution here as in the ideal-world game, so the distribution of A_{BR}'s guess is the same.

The bound on the probability that A_{BR} breaks backtracking resistance is the same as the bound on the probability that A can break pseudorandomness. This is not immediately obvious!

□

[2] One could change the ideal-world definition to give A_{BR} the last block of the ideally random output as v, but that would weaken the definition. Anyway, we still need to fix the v-updating in order to rewrite the game in terms of the PRF adversary, as discussed in Section REFERENCE.

D. On proving prediction resistance

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.

E. On adaptivity of the adversary

How would I prove this on paper?

V. OVERVIEW OF THE FOUNDATIONAL CRYPTOGRAPHY FRAMEWORK

We give a brief overview of FCF. For more details, a tutorial, and the theory behind it, see Petcher (2015).

As the abstract summarizes: "FCF provides an environment for developing and checking proofs of security of cryptographic schemes in Coq. ... Proofs may provide concrete bounds as well as asymptotic conclusions. FCF provides a language for probabilistic programs, a theory that is used to reason about programs, and a library of tactics and definitions that are useful in proofs about cryptography."

A cryptographic game may be viewed as an imperative, probabilistic program (Bellare (2004)). FCF defines the syntax of probabilistic programs as the type $\mathit{Comp}\ A$, the type of a probabilistic program that returns a value of type A. The most notable operation is Rnd, which produces n uniformly random bits. Other operations are sequencing statements in an imperative manner, repeating a statement, and returning a value.

In FCF, the standard non-uniform probabilistic polynomial time (PPT) adversary is modeled as a parameter $A : I- > \mathit{Comp\ bool}$. This denotes that the adversary A could be any non-uniform PPT adversary that takes as input some I and returns a guess of 0 or 1. Comp denotes that the adversary may flip coins internally. FCF has a mechanism to limit the adversary to a polynomial number of queries in some security parameter; refer to Petcher (2015) for details.

Cryptographic security definitions often involve giving the adversary some oracle that it can query, where the oracle must maintain some hidden state across multiple interactions with the adversary. Thus, FCF extends the Comp type, where $\mathit{Comp}\ A$ is a probabilistic program returning a value of type A, to the $\mathit{OracleComp}$ type, where $\mathit{OracleComp}\ A\ B\ C$ is a probabilistic program that returns a value of type C, and has access to an oracle that takes a value of type A and returns a value of type B. (The type of the oracle itself includes a state as input and a new state as output.) Oracle computations include the standard returning and binding operations, as well as an oracle query operations and an operation that can run a program using a different oracle that can access the current oracle.

Refer to Petcher (2015) for examples of simple games written in FCF and a walkthrough of proving security of a simple encryption scheme. A game typically has type $\mathit{Comp\ bool}$, because it is a probabilistic computation that usually returns an adversary's guess.

In game-playing proofs, we commonly do two things. First, we might make a slight change to a game then prove the former game equivalent to the latter, where "equivalent" can be taken in two senses. Either they are equivalent programs that return the same outputs, or they have computationally indistinguishable output distributions. Or, we might seek to bound the distance between two games by some negligible functions, where "distance" refers to the probability that some computationally-bounded adversary in both games can distinguish between two different distributions.

To aid with this kind of reasoning, FCF provides a program logic called Probabilistic Relational Postcondition Logic (PRPL). PRPL behaves like a Hoare logic, except there are no preconditions, and we write a specification relating pairs of probabilistic programs instead of a specification on a single deterministic program.

As a short example demonstrating PRPL, take the following proof goal in Coq.

```
Theorem PRPL_demo : forall (n m : nat),
  comp_spec (fun a b => a = fst b)
           (x <-$ {0,1}^n;
            ret x)
           (c <-$ {0,1}^n;
            d <-$ {0,1}^m;
            ret (c,d)).
Proof.
  intros.
```

`comp_spec` denotes a PRPL specification, or postcondition, that we want to prove about the two probabilistic programs that follow. Namely, we want to prove that the output of the first program is equal to the first output of the second program. `x <-$ {0,1]^n` denotes sampling a bitvector of n bits uniformly at random.

To continue the proof, we note that the first lines of each program yield the same output if both programs are given the same coins to flip, so we may "skip" them, and the relation that holds on the remaining programs will be the same. (This is the intuitive description. A more rigorous and detailed description of every tactic may be found in the *Tactics.v* file in the FCF repository, linked in the appendix.)

```
fcf_skip.
```

Now this is the new proof state. Note the two remaining programs.

```
n : nat
m : nat
b : Bvector n
H0 : In b (getSupport ({ 0 , 1 }^n))
H : In b (getSupport ({ 0 , 1 }^n))
============================
 comp_spec (fun (a : Bvector n)
               (b0 : Bvector n * Bvector m)
               => a = fst b0)
   (ret b)
   (d <-$ { 0 , 1 }^m; ret (b, d))
```

The first line of the second program is irrelevant because our specification on the outputs does not mention it. Therefore, we can eliminate that line with the following tactic.

```
fcf_irr_r.
```

This is the remaining proof state. Note that *b* is the the same between both programs, which is what we wanted to prove.

```
n : nat
m : nat
b : Bvector n
H0 : In b (getSupport ({ 0 , 1 }^n))
H : In b (getSupport ({ 0 , 1 }^n))
b0 : Bvector m
H1 : In b0 (getSupport ({ 0 , 1 }^m))
============================
 comp_spec (fun (a : Bvector n)
               (b1 : Bvector n * Bvector m)
               => a = fst b1)
   (ret b) (ret (b, b0))
```

When we have whittled down both programs such that they both only return values, we can use the following tactic to apply the specification to the return values and prove that it holds.

```
fcf_spec_ret.
```

In this case, FCF is smart enough to figure out that the specification holds, so it discharges the goal. We have proven that the given specification relates these two probabilistic programs.

```
Qed.
```

VI. FORMAL PROOF OUTLINE

A. Starting HMAC-DRBG definitions

The core inner loop of HMAC-DRBG that generates pseudorandom blocks.

```
(* 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)

TODO: move the explanation of identical until bad (in *oracleCompMap*) here

C. 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_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.
```

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

TODO: Indistinguishability definition

1. *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) (O, (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 *i*th 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 *i*th 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 O
           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.

D. 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 *G*1 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 models 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 *G*2 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
   ```

   More formally,

   ```
    Theorem Gi_normal_prf_eq_compspec :
   forall (l : list nat) (i : nat) (k1 k2 v : Bvector
        eta),
    comp_spec
      (fun (x : list (list (Bvector eta)) * (nat * KV
          ))
         (y : list (list (Bvector eta)) * (nat * KV)
             * unit) =>
       fst x = fst3 y)
      (oracleMap (pair_EqDec nat_EqDec eqDecState) (
          list_EqDec eqdbv)
         (Oi_prg i) (O, (k1, v)) l)

      ((oracleCompMap_inner
          (pair_EqDec (list_EqDec (list_EqDec eqdbv))
             (pair_EqDec nat_EqDec eqDecState))
          (list_EqDec (list_EqDec eqdbv)) (Oi_oc' i)
          (O, (k2, v)) l) unit unit_EqDec (* note: k'
              s differ. we aren't using this one *)
         (f_oracle f eqdbv k1) tt).
   ```

   This theorem is quantified over two important things: the list l of blocks that the adversary requests per call, and the number of the hybrid i. Intuitively, given that the two computations (one using `oracleMap`, the other using `oracleCompMap_inner`) output equal pseudorandom bits for a list l, if we make another call, the two outputs for that call will be equal too. This is true because there are three cases, according to the structure of `Oi_oc'`. Let n denote the length of l.

   (a) $n+1 > i$: for the new call, we use the PRF oracle in both computations.

   (b) $n+1 = i$: for the new call, in the `oracleMap` computation, we use the PRF oracle, and in the `oracleCompMap_inner` computation, we use the provided oracle. The caller is providing `f_oracle`, which is the PRF oracle.

   (c) $n+1 < i$: for the new call, we use the RB oracle in both computations.

   To prove this in Coq, we have to encode the insight that the theorem is inductive over **appending** to the list, rather than pre-pending (`cons`ing) to the beginning of the list. Thus, we induct on the reverse of the list l.

   We also have to deal with the special case of the first call, since we must use the special `GenUpdate_noV` oracle on that call. So we set $calls > 0$, perform the induction on the reverse of the list, then add the first call onto the head of the list.

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.
   ```

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.

$101, 000, 011, 001,$
$*000*, *011*, *001*,$
$*000*, *001*, \ldots$

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.

   The idea here is much the same as `Gi_normal_prf_eq`.

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`. The bad event is that there are duplicates in the inputs to the ith 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$_i$nner*, 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)
         (O, (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)
         (O, (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.

E. Proof graph

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

F. 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 $numCalls = 7$, 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

G. Comments on the Coq development

There are 17 main lemmas and many small ones. Not all are proven, but I've proved the four most difficult ones and am confident about the rest, which are mostly game equivalence proofs. There are 22 admitted lemmas (including medium-sized lemmas but not the smallest ones.

The development spans 2000 lines of code. Adam's proof for the core loop spans less than 1000 lines.

Besides proving the admitted lemmas, here's what's left to do in the near future:

- Adapt the indistinguishability proof to a nonadaptive adversary that picks any list of *blocksPerCall* and *numCalls* beforehand, not the hardcoded *maxBlocksPerCall* and *maxCalls*. The paper proof here already holds for the more general adversary.

- Reuse the indistinguishability proofs to show backtracking resistance.

- The spec equivalence proof. (Discussed in the following section.)

H. 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. LINKING OUR CRYPTO SPEC WITH THE FUNCTIONAL SPEC

Sanguansin (2016) proved functional correctness of the mbed TLS implementation of HMAC-DRBG using a different functional specification, also written in Gallina. He did this proof using the Verified Software Toolchain (VST) framework, which provides a logic for reasoning that the outputs of some C function satisfy some postcondition given that the inputs satisfied some precondition.

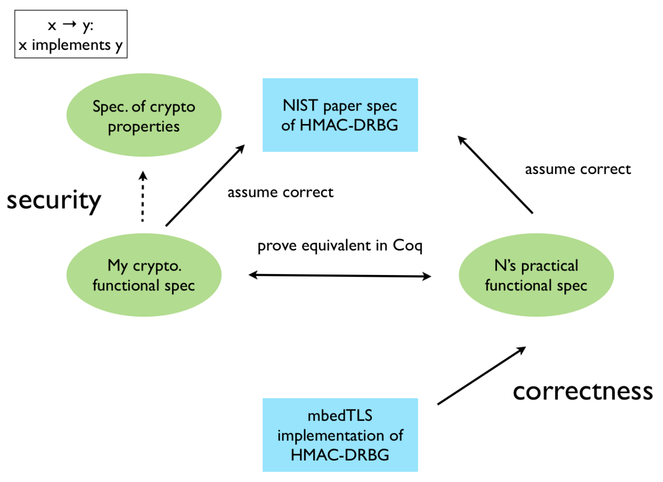
x → y:
x implements y
Spec. of crypto properties
NIST paper spec of HMAC-DRBG
security
assume correct
assume correct
My crypto. functional spec
prove equivalent in Coq
N's practical functional spec
correctness
mbedTLS implementation of HMAC-DRBG

Figure 1. A fully verified HMAC-DRBG.

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. This is the labeled arrow in Figure 1.

A. Bridging the gaps between the specifications

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

1. Sanguansin (2016)'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. Sanguansin (2016)'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 ideally random. We need to relate FCF's sampling uniformly at random to this more realistic entropy model. Petcher (2015)'s doctoral thesis provides relevant theorems about the operational semantics of FCF that can help in proving this relation.

Beringer (2015)'s verification of HMAC was similarly structured. They started with separate proofs of cryptographic security (in this case, that HMAC is a PRF given that SHA is a PRF) and of functional correctness (of an OpenSSL implementation of HMAC). Thus, they also had two separate specifications of HMAC, one using dependent types to enable cryptographic proofs in FCF, and one closely modeling the NIST standard for HMAC. Thus, they also had to do a similar proof of equivalence between the specifications. (I was the one who did this proof, with help from Beringer. See Ye (2014) for a detailed description of the proof.) This gives us confidence that such a proof can be done, and that proving equivalence does not require adding too many weakening assumptions on security or correctness.

VIII. 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 pseudorandom 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?

All of this effort was predicated on the fact that HMAC-DRBG's specification will not change, and mbedTLS's implementation of HMAC-DRBG does not change. The test of time will show how modular our proofs are.

Also, there have been rumors that OpenSSL's team is picking or designing a new PRNG. In this situation, and in similar situations, we hope our work encourages implementors to co-design their libraries with formal methods researchers.

IX. Conclusion

We have contributed a paper proof of the pseudorandomness of a simplified version of HMAC-DRBG. Hirose (2008) had already written paper proofs, but they were complicated and not linked to functional specifications or implementations. No prior proof considered real-world use of HMAC-DRBG (or any similar DRBG) over multiple calls to the Generate function, or including updating of the state. Also, no prior proof has proven backtracking resistance for this type of DRBG.

We have mostly formally verified the pseudorandomness of a simplified version of HMAC-DRBG. Simple PRGs have been verified before, but they were on the order of verifying the core loop in *Generate* (e.g. in the EasyCrypt documentation), not real-world PRGs.

We have contributed knowledge about the challenges and benefits of formally verifying PRGs, and especially about proving things about hybrid arguments. The process of writing a machine-checked proof of security unearthed several subtle errors, such as off-by-one errors in hybrid numbering and the presence of multiple PRF adversaries.

We plan to link this security proof to a functional correctness proof. Thus, mbedTLS's implementation of HMAC-DRBG will be fully certified.

X. ACKNOWLEDGMENTS

I'd like to thank the following people for their help.

Adam Petcher provided a lot of help by walking me through using FCF, doing the proof of security for the core loop of the PRG, and debugging many of my proofs via (lengthy) email.

Matt Green came up with the proof of security for the simplified HMAC-DRBG, asked many tough questions about the Coq development, and patiently walked me through the crypto math.

Andrew Appel provided lots of help with Coq proofs and suggestions for the future proof of equivalence. I'm glad he had the confidence that I would be able to do and verify crypto math (though it's not done yet!). Andrew has been a very active, supportive, and patient advisor during my JP, the grad school application process, and this thesis.

Lennart Beringer came to all of my thesis meetings and went above and beyond by helping me debug my proofs on Sundays.

Naphat Sanguansin did the work that would allow my proofs to actually hold on an implementation of HMAC-DRBG.

I want to thank Matt and Andrew again for suggesting a thesis topic that was cool and well-motivated, as well as providing detailed feedback on drafts of this thesis.

Thanks to everyone in 2D for general encouragement, cooking great food that sustained me at late hours, sending me thesis jams (thanks, Angela), and camping out with me in the dining room (hi, Jaehwan).

Thanks to Jean Juang, Frederic Koehler, and Xiaoyan Han for being great friends and thesis buddies, and Carolyn Chen and Pallavi Koppol for being great friends during the grad school application process.

Thanks to the Recurse Center for being a community of excellent people who are happy to listen to me ramble about research.

[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] Barak, Boaz, and Shai Halevi. "A model and architecture for pseudo-random generation with applications to/dev/random." *Proceedings of the 12th ACM conference on Computer and communications security.* ACM, 2005.

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

[5] Barthe, Gilles, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. "Computer-aided security proofs for the working cryptographer." In *Advances in Cryptology—CRYPTO* 2011, pp. 71-90. Springer Berlin Heidelberg, 2011.

[6] Barthe, Gilles, et al. "Easycrypt: A tutorial." *Foundations of Security Analysis and Design VII.* Springer International Publishing, 2014. 146-166.

[7] Bellare, Mihir, and Phillip Rogaway. "Code-Based Game-Playing Proofs and the Security of Triple Encryption." *IACR Cryptology ePrint Archive* 2004 (2004): 331.

[8] 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.

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

[10] Dodis, Yevgeniy, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. "Security analysis of pseudorandom 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.

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

[12] Fouque, Pierre-Alain, David Pointcheval, and Sbastien Zimmer. "HMAC is a randomness extractor and applications to TLS." *Proceedings of the 2008 ACM symposium on Information, computer and communications security.* ACM, 2008.

[13] Goldreich, Oded. *Foundations of cryptography: a primer.* Vol. 1. Now Publishers Inc, 2005.

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

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

[16] Petcher, Adam. *A Foundational Proof Framework for Cryptography.* Diss. 2015.

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

[18] Ye, Katherine. "Formally proving equivalence between abstract and concrete specifications of HMAC." Junior paper, Princeton University. 2014.

Appendix A: Definitions and proofs

My repository.

FCF repository.

HMAC repository.

Will add worked examples of Coq proofs using FCF tactics.