020404-two-long

Two New Proposed DRBG Algorithms Based on Any Approved Hash Function

John Kelsey, NIST, January 2004

1 Introduction

I've been asked by the other members of the X9.82 working group to design a replacement for the HASH_DRBG, which I analyzed in a previous note. I've gone a little further than asked, and designed two very closely related DRBGs. They are identical except for two internal functions. This basic structure can fit other DRBGs, and in fact, I am planning to submit a block-cipher-based DRBG separately, following the same structure.

Designing a DRBG based on a hash function apparently requires making some assumptions that don't have anything much to do with the normal requirements of the hash function. A pretty routine thing to do with a DRBG is to crank out some large sequence of outputs, which must be indistinguishable from random bits for all practical purposes, from a relatively small internal state. Although hash functions are used in a lot of places as random oracles, or to derive keys or do other things requiring some kind of pseudorandomness property, this is very seldom justified by any clear specification of what is being expected of the hash function.

My two designs are based on building a pseudorandom function family (PRF) from the underlying hash function. A PRF is a keyed function which an attacker can't distinguish from an ideal random function, given some reasonable limits on how much computation he can do and how many times he can query the function. I first work on the assumption that I can build such a PRF in a couple of different ways, and then build a mechanism for generating pseudorandom bits from that, in a way that only weakly relies on the function really behaving like a PRF. I like building a PRF because it's much easier to understand the security claims and assumptions this way, than with most other hash-based DRBG designs I have seen. I build the mechanism using the PRF in a somewhat careful way (using it in output feedback mode, with a limited number of outputs per request), because this makes the security of the DRBG depend, not on the PRF construction resisting adaptive chosen input attack, but instead on its resisting known-input attack with pseudorandom inputs.

The common structure for these DRBGs is intended to make the design very clean and easy to analyze, and also easy to use to build other DRBGs, based on entirely different mechanisms, in a straightforward way.

2 Preliminaries

2.1 Pseudocode

I'm going to use a kind of simplified Python for pseudocode; this should be reasonably easy to read. Uppercase letters are bitstrings, lowercase letters are small (32 bits, usually) integers, or very rarely, real numbers like probabilities. To define a function, I'll use

```
# I should describe the function a bit here.
def function(parameter1, parameter2, parameter3=None):
    some stuff
    some more stuff
    ...
```

The only thing that may look puzzling about this is the

parameter3=None bit. That's how Python specifies an optional parameter with a default value. If the caller doesn't send anything in, then parameter3 takes on the value None (something like NULL in C or Nil in Pascal).

Python operates a lot on bytes. So, if I say X[:10] I mean the first ten bytes of X, and if I say X[10:20], I mean bytes ten through nineteen of X, and if I say X[20:], I mean all the bytes of X from byte twenty up.

I'm not going to get into defining classes and such, though that would make the description simpler. (The code I have at the end of this note is in Python, and uses the whole class notation.)

Comments are anything that appears after a pound sign, #, on a line.

I will use the || symbol to mean concatenation of bit strings. I will use the function encode(i) to mean "encode the 32-bit unsigned integer i as a bitstring, using network byte order." If I say X xor Y, that's exclusive oring two bitstrings together. That's not defined for bitstrings of different lengths.

I will use x^y to mean "x to the power y" and x^{y+z} to mean "x to the power (y+z)."

Because I'm much more of a programmer than a mathematician, I'll use variables with lots of letters in them sometimes.

Because I'm describing this in pseudocode at a fairly casual level, I'm going to just assume scope of variables. If I define X in a specific DRBG, then all the functions described for that DRBG can have access to X.

If I want to describe a constant byte string, I'll write it like this

0x00 0x00 0x00 ... 0x00 (35 bytes)

This is a 35-byte string of binary zeros.

2.2 Hash Function Specifics

Both DRBGs are built on hash functions. I am assuming the basic structure of the SHA family of hashes here. That is, all the work is done by a compression function, Compress(U,V), where U is an N byte chaining value, and V is an M byte message block. In general, SHA family hashes work as follows:

a. Pad the message to an integer number of message blocks (a multiple of M bytes) in an unambiguous way, which includes the length of the message before padding in the last message block. This always adds at least nine bytes, and may add as many as 72 bytes.

b. Let X[0,1,2,...,R-1] be the padded message, broken into M-byte blocks.

c. Let H = I, the initial chaining value of the hash function.

d. For j = 0 to R-1:
        H = Compress(H,X[j])

e. The final hash is H, possibly truncated to a shorter output length. (SHA384 and SHA224 both truncate in this step.)

2.2.1 Notation

I'll use the following notation to describe the hash algorithm used.

a. N = the number of bytes in the chaining value input and compression function output.

b. M = the number of bytes in the message block input to the compression function.

c. Compress(X,Y) is the compression function, applied to chaining variable input X and message block input Y.

d. hash(X) is the full hash function applied to the bitstring X.

2.3 Security Levels

We talk a lot about security levels in X9.82. As closely as possible, what we mean by a K-bit security level is the same set of security tradeoffs we expect from a block cipher with no known attacks and a K-bit key. That is, if you do p*2^K encryptions' worth of work, you ought to have about a p probability of breaking the cipher.

Breaking a DRBG means either predicting some parts of the output you haven't seen, or distinguishing an output string from random. In general, we want the probability of an attacker doing either of those to be pretty closly bounded by the fraction of the 2^K work he's done, where K is the security level of the DRBG. We're probably not as precise about this definition as we should be; we care a lot more about prediction attacks than distinguishing attacks, but the distinguishing attacks can sometimes be important, and "indistinguishable from random" is pretty-much the gold standard for random number generation.

Both DRBGs here claim a security level equal to the number of output bits of the hash function. Thus, a DRBG based on SHA256 claims to be capable of supporting up to 256 bits of security.

2.4 Entropy and Unguessability

If X is an unguessable value, that means the attacker can't afford to guess it given his security level. That means that guessing it is subject to, at best, the same kind of tradeoff as guessing a K-bit key, when K is the security level of the DRBG.

In X9.82, we use min-entropy (minus the base-2 log of the highest probability in the distribution) as our entropy measure. A string with K bits of min-entropy is guaranteed to be unguessable to someone who can't defeat a K-bit security level; if such a string is hashed to produce a K-bit key, an attacker is never better off guessing the string than guessing the key.

A DRBG is in a secure state when its working state is unguessable and it's ready to generate outputs that are indistinguishable from random to an attacker who can't defeat a K-bit security level, where K is the security level of the DRBG.

2.5 Some Arbitrary Parameters

In my work on X9.82, I've assumed some parameters that take on values like "as many outputs as we'll ever really see" or "negligible probability."

I've assumed that no innocent party ever does more than 2^{64} of

anything. That is, if you're not trying to attack someone, I assume you don't do more than 2^{64} outputs, 2^{64} reseeds, etc.

I've assumed that events that happen to a single user, device, or DRBG instance with probability less than 2^{-64}, or events that happen somewhere among the whole population of DRBG users with probability less than 2^{-32}, can usually be ignored.

3 The Common Structure

Any DRBG algorithm that looks like the ones we've defined in X9.82 ends up supporting three public functions:

a. Initialize(seedString) initializes the DRBG's working state. If the seedString is unguessable, then the DRBG ends up in a strong state.

b. Reseed(seedString) reinitializes the DRBG's working state. If the seedString is unguessable, then the DRBG ends up in a secure state regardless of how it started. If the DRBG is in a secure state when Reseed(seedString) is called, it ends up in a secure state regardless of the contents of seedString.

c. Generate(bytes, optionalInput=None) generates pseudorandom bytes, using a state that's essentially been reseeded with the optionalInput if it exists. Generate has several security requirements:

> (i) If the DRBG starts in a secure state, then regardless of optionalInput, it must be in a secure state while generating the pseudorandom bytes and it must end up in a secure state.
>
> (ii) If the optionalInput is unguessable, then regardless of what state the DRBG started in, it must be in a secure state while generating pseudorandom bytes, and it must end up in a secure state. ("Prediction Resistance")
>
> (iii)If the DRBG is in a secure state when the pseudorandom bytes are generated, then those bytes must be indistinguishable from random bytes at the DRBG's security level, and the DRBG must end up in a secure state.
>
> (iv) If the DRBG's state is compromised after the Generate() function completes, previous outputs must still be indistinguishable from random, and previous working states must still remain secure. ("Backtracking resistance")

In the two DRBGs I've designed, these three functions are supported by three internal functions, which do all the cryptographically interesting stuff. These are _Setup(), _Renew(seedString), and _Crank().

d. _Setup() initializes the DRBG's working state to some constants. The state after Setup() is done is completely known to any attacker, but is ready to be used.

e. _Renew(seedString) generates a new DRBG working state, so that if either the DRBG started in a secure state, or the seedString is unguessable, then the DRBG ends in a secure state.

f. _Crank() generates one block of pseudorandom output of fixed length, and updates the working state. If the DRBG is in a secure state, then _Crank() generates outputs that cannot be distinguished from random, and leaves the DRBG in a different secure state.

3.1 Pseudocode for the Three External Functions

```
# Get the DRBG to a secure starting point from the seedString.
def Initialize(seedString):
     _Setup()
     _Renew(seedString)

# Get the DRBG to a secure starting point if either the DRBG started
# in a secure state, or if the seedString is unguessable.
def Reseed(seedString):
     _Renew(seedString)

# Generate the requested number of outputs, up to 2^{32} bytes,
# updating the state first with optionalInput if it's provided.
def Generate(bytes,optionalInput=None):

     # Restrict outputs to 2^{32} or fewer bytes of outputs.
     if bytes>2^{32}:
          Raise an error.

     # If we have optional input, let it make our state secure.
     # If _Renew() meets its security requirements, then we meet
     # requirements (i) and (ii) in the list above.
     if optionalInput<>None:
          _Renew(optionalInput)

     # Generate the bits requested.  If _Crank() meets its security
     # requirements, then we meet requirement (iii), above.
     tmp = ""
     while len(tmp)<bytes:
          tmp = tmp || _Crank()

     # Update the state to prevent backtracking.  If _Renew() meets
     its security requirements, we meet requirement (iii), above.
     if optionalInput<>None:
          _Renew(optionalInput)
     else:
          _Renew("")

     # Return requested bits
     return tmp[:bytes]
```

The following two sections describe two closely-related DRBGs. Both use the above three definitions, along with specific definitions of the internal functions to build a complete DRBG.

4 HMAC_DRBG

4.1 Overview

The core idea of HMAC_DRBG is to use HMAC as a PRF. HMAC is widely assumed to be a PRF, and it is widely used this way, so this seems reasonable enough. Once we have a PRF, it's not difficult to design a DRBG, in this case by running the PRF in output feedback mode.

In this section, HMAC(K,X) refers to HMAC computed with key K and input string X. N refers to the number of output bytes from the underlying hash function, and thus from the HMAC construction.

For reference: HMAC(K,X) = hash(K xor opad || hash(K xor ipad || X))

where

K = an N-byte block with M-N bytes of binary zeros appended to the end

opad = 0x5c 0x5c 0x5c ... 0x5c (N bytes)

ipad = 0x36 0x36 0x36 ... 0x36 (N bytes)

4.2 Pseudocode for the Three Internal Functions

```
def _Setup():
    K = 0x00 0x00 0x00 ... 0x00 (N bytes)
    X = 0x00 0x00 0x00 ... 0x00 (N bytes)

def _Crank():
    X = HMAC(K,X)

def _Renew(seedString):
    K = HMAC(K,X || 0x00 || seedString)
    _Crank()
    if seedString <>"":
       K = HMAC(K,X || 0x01 || seedString)
       _Crank()
```

4.2.1 Expanding the Pseudocode

The pseudocode in section 3, above, describes precisely how each DRBG function call is translated to one or more of these calls. In words:

Initializing the DRBG just amounts to setting its working state to some constants and calling _Renew(seed).

Reseeding the DRBG just calls _Reseed(seed).

Generating outputs calls _Renew(optionalInput) if the optionalInput was provided. Then it calls _Crank() repeatedly to generate the requested number of outputs. After all outputs have been generated, it calls _Renew() once more, with either the optionalInput, or with an empty string.

4.3 Analysis of _Crank()

_Crank() has a single requirement: If it starts out in a secure state, then it outputs an unguessable N-byte bitstring, and ends up in a new secure state.

The security of _Crank() depends directly on the security of HMAC. If HMAC is a PRF with security level K and 2^{32} or fewer queries permitted, then a sequence of 2^{32} or fewer outputs will be indistinguishable from random. Why is this true? There are two cases to consider. If X repeats during an output sequence, then _Crank() will be distinguishable from random regardless of the properties of HMAC, because the whole sequence will begin to repeat after that. If X doesn't repeat, but an attacker can distinguish the output sequence from random, he can also distinguish HMAC from a random function.

4.3.1 No Collisions

Suppose I have an algorithm A() to distinguish a sequence of 2^{32} successive _Crank() outputs from an ideal random string. I can trivially turn this into an algorithm for distinguishing HMAC from a PRF:

a. Generate a random starting value X[0].
b. For i = 1 to 2^{32}, request X[i+1] = PRF(X[i])
c. Apply A() to X[0,1,2,...,2^{32}], and return the result.

This applies to any similar PRF construction.

4.3.2 Collisions

Suppose I want to wait for a collision to happen internally, so that I can distinguish the DRBG outputs without knowing how to distinguish HMAC from a PRF. Until the collision, I cannot distinguish the outputs from random.

The smallest N we deal with is 20 bytes, for SHA1; that's 160 bits. With 160 bits, and 2^{32} calls to _Crank(), we have a probability of about 2^{-97} of seeing a collision in a given Generate() output; this probability is small enough to ignore.

4.3.3 Is HMAC Really a PRF?

HMAC is widely assumed to be a PRF. Attacking it in the model of an attacker with huge numbers of F_K() queries to distinguish it from random appears very difficult, but this doesn't prove that it is a PRF. (This is the sort of assumption that can never be more than an assumption, unless someone develops an attack, and shows that it was a mistaken assumption.)

The structure of HMAC as used in output generation is like this:

F_K(X) = Compress(K0, Compress(K1, X||padding))

where
I = the initial chaining value for the hash function
K0 = Compress(I,K xor opad)
K1 = Compress(I,K xor ipad)

Now, the output we see is the result of this outer Compress() function. And that function always starts with the same chaining value (for the same key), and gets a random-looking N-bit input in its message input block. Note that many of those input bits are fixed padding bits; for SHA1, that means that 352/512 message input bits are fixed and known to the attacker. However, this doesn't seem to lead to any attack. In an adaptive chosen input attack, the goal would be to find some property of these outputs from the outer Compress() function that deviated from random. I don't see any useful way to proceed in attacking this as a PRF, and I am aware of no result that would rule out HMAC being a PRF.

4.3.3.1 Do We Care?

The requirements on a PRF are stronger than what we need for this DRBG to be secure. The attacker never gets a chosen-input attack on HMAC when it's being used for generating outputs, and no more than 2^{32} outputs are ever generated under a single key K.

4.4 Analysis of _Renew(seedString)

_Renew(seedString) derives a new K and X from a seedString. This looks like:

K = HMAC(K,X || 0x00 || seedString)
X = HMAC(K,X)
if seedString<>"":

```
K = HMAC(K,X || 0x01 || seedString )
X = HMAC(K,X)
```

There are two cases of interest:

a. The DRBG is in a secure state. In that case, the attacker knows neither K nor X. The new value for K is indistinguishable from random if the attacker can't distinguish HMAC from a random function. If seedString is not empty, this same basic operation is done a second time; again, with the current K unknown to an attacker, the new K is likewise unknown. This is true even when the attacker chooses seedString to try to cause problems with this.

b. The DRBG is not in a secure state, but the seedString is unguessable to the attacker. In this case, the new value for K is indistinguishable from random if HMAC with a fixed known key K does a good job of distilling entropy.

4.4.1 Distilling Entropy in General

Informally, a function does a good job of distilling entropy at a K-bit security level if, when it's given an unguessable input string, it produces a result indistinguishable from random.

No fixed function can do a good job of distilling entropy for all possible input distributions. For example, let F() be any fixed function that maps long strings down to 160 bits. Suppose a computationally very powerful attacker chooses a set of 2^{160} values A[0,1,...,2^{160}-1] so that F(A[i]) is the same for all A[i]. Now, we can have an input string that's unguessable, by choosing one of the A[i] at random. But F(A[i]) is completely known.

For our purposes, I assume that input strings that bear entropy are not entirely chosen by some attacker; instead, the attacker facing an input string with K bits of min-entropy may have chosen part of the string, but doesn't have enough control over it to exert the kind of control necessary to affect the probability of collisions in a reasonable cryptgraphic hash function.

I'll note that an alternative approach is to assume that the attacker *can* exert that much control, but then restrict the security level of the DRBG that relies upon an N*8 bit hash function to (N/2)*8 bits.

4.4.2 Distilling Entropy in HMAC

Because of the way the SHA hashes are constructed, many input strings will lose a small amount of entropy in processing. Consider an input string to be distille by HMAC-SHA1:

```
seedString = 344-bit random block || 512 constant bits
```

When this is processed, the interior hash will look like:

```
Compress(Compress(K1, X || 0x00 || 344 random bits ),512 constant bits)
```

Typically, this will lose about a bit of entropy, meaning that even with 344 bits of entropy in the input, we get only about 159 in the output.

A longer message is worse; each additional 512 bit block is another iteration. With 2^{32} 512-bit blocks after the 344-bit random block has been processed, we expect about 2^{127} possible states.

Now, this is apparently very hard to use in an attack, because an attacker has to do something like 2^{191} compression function computations to learn thes 2^{127} possible states. However, I couldn't convince myself there was no attack possible here, so I did the key update twice whenever we have a non-empty seedString.

4.4.3 Distilling Entropy in _Renew(seedString)

Because _Renew() generates K twice, we have a lot of confidence that it distills at least N*8 bits of entropy from seedString. In fact, it should usually distill nearly 2*N*8 bits of entropy. In the worst case where the seedString is 344 bits random bits, followed by 2^{32} 512-bit blocks, we get (starting from a known K,X and using HMAC-SHA1):

```
K = HMAC(K,X || 0x00 || seedString ) ==> about 2^{129} possible states
X = HMAC(K,X) ==> One possible state given K; 2^{129} possible
K = HMAC(K,X || 0x01 || seedString ) ==> about 2^{129} possible states
                                         given previous K, 2^{160}
                                         without knowledge of previous
                                         K.
X = HMAC(K,X)
```

At the end of this, K is indistinguishable from random, and so is X. An attacker can guess the intermediate value of K with 2^{129} work; each such guess leads to one guess for the intermediate value of X. But then, K is randomized; put in one of 2^{129} or so possible new states. Then, X is randomized according to the final K. Guessing the final K,X pair requires about 2^{258} work.

4.5 Spiraling Down

Consider the situation where the HMAC_DRBG is initialized once, and then is used to generate 2^{64} output blocks without reseeding or being given additional input. It's clear that the DRBG spirals down to a smaller number of possible states.

Before a Generate() call, K has about 2^{160} possible states, and X has about two possible states per K. If Generate() is called with no optionalInput, and a request is made for 2^{32} output blocks, we can see X as spiraling down just a little. However, in practice, each iteration renews X with all the entropy from K. These iterations can never decrease the entropy in K, which isn't being updated. Thus, we don't generally spiral down below 160 bits of entropy when we generate long output sequences.

At the end of the Generate() call, _Renew() is called to provide backtracking resistance. K is regenerated, based on both K and X; we might expect K to lose about a bit of entropy here, on average, but X also carries a great deal of entropy from K. Then, X is rederived from K and the previous X. We can ideally see this as a mapping from 320 bits to 320 bits, and when that is applied to 2^{160} different values, we lose very little entropy; only about one pair of inputs will be expected to collide.

For these reasons, I am not concerned with the HMAC_DRBG spiraling down to a too-small set of states.

4.6 Guessing the State

The HMAC_DRBG working state includes N*8 secret bits and N*8 public, changing bits; it tries to support an N*8 bit security level.

Some DRBGs can be attacked by guessing 2^{K-R} possible working

states, where K is the security level of the DRBG, and then waiting for the DRBG to generate about 2^R outputs. At this point, it is likely that one of those outputs was generated by a state in the attacker's list of possible working states, and this permits recovery of the DRBG working state.

The HMAC_DRBG has as many bits of secret state as it does bits of security. An attacker trying to guess possible working states to see if his output values match those from the HMAC_DRBG has to guess the entire key K; this requires an N*8-bit guess; therefore, this kind of guessing attack gives the attacker no advantage over straightforward guessing of the HMAC key.

4.7 Efficiency and Implementations

HMAC can be computed either with standard library calls to the underlying hash function, or more efficiently with access to the underlying compression function calls.

The HMAC_DRBG implemented with standard hash library calls will require four compression function calls per N byte output. Implemented with direct access to the compression function, the HMAC_DRBG will require two compression function calls per N byte output.

5 KHF_DRBG

5.1 Overview

The core idea of KHF_DRBG is to construct a more efficient PRF from the compression function of the underlying hash function, in such a way that the DRBG can be implemented using either a standard library call to the hash function, or more efficiently using lower-level access to the hash compression function. We construct a PRF by computing

PRF(K,X) = hash(K0 || PAD_0 || (X || PAD_1) xor K1)

which can equivalently be computed, for SHA1, SHA256, and SHA512, as:

PRF(K,X) = Compress(K0',((X || PAD_1) xor K1) || PAD_2)

In this DRBG, we use the hash function in a simple way to distill entropy from an input string; the goal is that if the input string is unguessable, then the output string is indistinguishable from random. This function is called hash_df(string,bytes), and is defined as:

```
def hash_df(string,bytes):
     tmp = ""
     i = 0
     while len(tmp)<bytes:
          tmp = tmp || hash(encode(bytes)|| encode(i) || string )
     return tmp[:bytes]
```

This is also defined in X9.82 Part 3, and is currently used in other DRBG designs.

N is the number of bytes of output from the compression function, M is the number of bytes of input into the message block of the compression function, and L is the length of the smallest number of bytes that can appear in a message, so that the message can still be hashed with a single compression function call. For concreteness, in SHA1, N=20, M=64, and L=55.

Note that N is also the size of hash output for SHA1, SHA256, and SHA512, but not SHA224 or SHA384. In the case where the truncated hash is used, the security level of the DRBG must not be claimed to exceed the number of bits in the hash function output. This is important because of some details of how entropy distillation is done, as discussed below.

5.2 Pseudocode for the three internal functions

```
def _Setup():
     PAD_0 = 0x00 0x00 ... 0x00 (M-N bytes; for SHA1, that's 44 bytes)
     PAD_2 = 0x00 0x00 ... 0x00 (L-N bytes; for SHA1, that's 35 bytes)
     K0 = 0x00 0x00 ... 0x00 (N bytes)
     K1 = 0x01 0x01 ... 0x01 (L bytes)
     X = 0x02 0x02 ... 0x02 (N bytes)

def _Crank():
     X = hash(K0 || PAD_0 || (X || PAD_2) xor K1 )
     return X

def _Renew(seedString):
     T = ""
     while len(T)<N+L:
          T = T || _Crank()
     T = T xor hash_df(seedString,N+L)
     K0 = T[:N]
     K1 = T[N:N+L]
     _Crank()
```

5.2.1 Expanding the Pseudocode

The pseudocode in section 3, above, describes precisely how each DRBG function call is translated to one or more of these calls. In words:

Initializing the DRBG just amounts to setting its working state to some constants and calling _Renew(seed).

Reseeding the DRBG just calls _Reseed(seed).

Generating outputs calls _Renew(optionalInput) if the optionalInput was provided. Then it calls _Crank() repeatedly to generate the requested number of outputs. After all outputs have been generated, it calls _Renew() once more, with either the optionalInput, or with an empty string.

Note that everything about the KHF_DRBG is identical to the HMAC_DRBG, except for these three internal functions.

5.3 Analysis of _Crank()

As with the HMAC_DRBG, if the attacker doesn't know K0 or K1, and X doesn't cycle, the DRBG generates a strong pseudorandom output string if the PRF construction above is actually a PRF, with no more than 2^{N*8} work and no more than 2^{32} online queries allowed.

The PRF construction I am using is defined above. For SHA1, SHA256, and SHA512, this can be rewritten in a more informative way:

```
F(K0',K1,X) = Compress(K0', ((X || 0x00 0x00 ... 0x00) xor K1) || PAD_1 )
```

where PAD_1 is the last nine bytes of input to the message block of the compression function; this is defined as the proper padding string

for the underlying hash, for a message of length M+L bytes.

In words: K0' is the chaining variable input to the Compress() function. K1 is L bytes long; for SHA1, it's 55 bytes. We pad X out with zeros to be the same size as K1, then XOR them together. We pad the result with nine constant bytes, and then compute the compression function output. (The nine constant bytes let us compute this function with standard library calls to a hash function, which makes non-performance-critical implementations much easier to do.)

5.3.1 No Collisions

Suppose I have an algorithm A() to distinguish a sequence of 2^{32} successive _Crank() outputs from an ideal random string. I can trivially turn this into an algorithm for distinguishing my keyed function from a PRF:

a. Generate a random starting value X[0].
b. For i = 1 to 2^{32}, request X[i+1] = PRF(X[i])
c. Apply A() to X[0,1,2,...,2^{32}], and return the result.

This applies to any similar PRF construction.

4.3.2 Collisions

Suppose I want to wait for a collision to happen internally, so that I can distinguish the DRBG outputs without knowing how to distinguish my keyed function from a PRF. Until the collision, I cannot distinguish the outputs from random.

The smallest N we deal with is 20 bytes, for SHA1; that's 160 bits. With 160 bits, and 2^{32} calls to _Crank(), we have a probability of about 2^{-97} of seeing a collision in a given Generate() output; this probability is small enough to ignore.

4.3.3 Is My Keyed Hash Function Really a PRF?

The goal is to build a PRF. That means an attacker who knows nothing about K0 or K1 is given up to 2^{32} chosen inputs to either this function, or an idealized random function, and 2^{N*8} compression function computations he's allowed to do offline. At the end of this, he has to guess whether he's been interacting with this function or the idealized random function. If he has more than a negligible advantage over just flipping a coin in deciding, then this function is not a PRF.

My keyed function has somewhat different properties than HMAC. An attacker doesn't directly know any bits of input to the compression function except the last 72. However, he knows large numbers of XOR differences between the inputs. For 2^{32} values, he gets about 2^{63} input differences, and in a chosen input attack, he can use this in some circumstances to get 2^{63} input pairs with some useful difference pattern.

I've spent some time trying to find ways to distinguish this from a random function, but I haven't seen anything so far. In general, the compression function of any good hash function is designed to be pretty robust against straightforward differential attacks. On the other hand, because the message expansion function of SHA1 is linear, the attacker knows the precise input XOR differences in the expanded message words. But while he can choose from a set of up to about 2^{319} possible input differences in principle, and he can request 2^{63} of them in a structure, he doesn't have a lot of control of

them; the message expansion of SHA1 spreads the differences out over many words. I haven't tried to work out the minimum number of changed message words the attacker can get, but I believe it's going to be hard for him to avoid getting about half the expanded message words changed with any input difference at all. And all of this seems, intuitively, to be attacking SHA1 in just about the way its designers anticipated. Finally, chosen input attacks define a PRF, but my DRBG doesn't permit them.

If there are linear characteristics through the compression function with bias of about 2^{-16} or more, an attacker can use this to distinguish my keyed function from a random function, and this will directly be applicable to cryptanalyzing the DRBG. It seems hard to preserve any linear characteristic through the entire compression function, because of the impact of both the carry bits and the nonlinear functions, and because the rotations mix bit positions together quickly. But I honestly haven't considered this in enough depth to have much confidence in my answer yet.

A chosen-input attacker can saturate one word of input, causing it to run through all possible values. The message expansion will extend this to saturating several expanded message words, but most of the expanded message words will not run through all possible values, or even have the summation-to-zero property; they will simply take on a bunch of random-looking values. Inside the round function, a saturation property will last only a few steps before being lost; after ten steps, I'm almost certain the A,B,C,D,E state won't even have the summation-to-zero property.

I will keep thinking about these attacks. Maybe I'll have an interesting insight. For now, I am very confident that distinguishing my keyed function, running in output feedback mode, from a random output sequence will be harder than guessing an N*8 bit key.

5.4 Analysis of _Renew()

_Renew(seedString) is very simple to analyze.

5.4.1 Starting in a Secure State

Suppose the DRBG starts in a secure state. Two things happen:

a. A new output of N+L bytes is generated from the DRBG's working state. If the working state is secure, then this output is indistinguishable from random.

b. The seedString is processed with hash_df to generate an N+L byte input string. An attacker could be in control of this, at least partially.

c. The two are XORed together, the first N bytes are used for K0, and the last L bytes for K1.

Even if the attacker is in complete control of the N+L byte string from step (b), the DRBG ends up in a secure state. (Suppose this weren't true. Then the attacker would have an algorithm for distinguishing N+L byte DRBG outputs from random, by using them in this way to generate new N+L byte DRBG outputs that weren't in a secure state.)

5.4.2 Starting Compromised

If hash_df(seedString,N+L) does a good job of distilling entropy, so that an unguessable seedString leads to an output indistinguishable from random, then again, the DRBG ends up in a secure state. (Suppose this weren't so. Then, an attacker could distinguish the hash_df() outputs from random when the seedString was unguessable.)

5.4.3 Distilling Entropy in hash_df

hash_df uses the underlying hash function to distill entropy. Each call of hash_df can, under some circumstances, lose a few bits of entropy. For example, if the hash is SHA1, and the input is

seed = 448 bit random number || 2^{32} 512-bit blocks

then the result of each 160-bit output from hash_df will have about 2^{129} possible values.

I don't think this is useful in an attack, because the attacker has to do 2^{192} or so work to learn these 2^{129} possible values. More to the point, the above PRF construction has a total of at least 160+440 = 600 bits of key material. I am convinced that this does not pose a threat to the security of the KHF_DRBG.

5.5 Spiraling Down

Consider the situation where the KHF_DRBG is initialized once, and then is used to generate 2^{64} output blocks without reseeding or being given additional input. It's clear that the DRBG spirals down to a smaller number of possible states.

The DRBG is initialized with N*8 bits of entropy. This is expanded to 600 bits of key material, and further expanded to 760 bits of state, in the worst case. For all the hashes, this enormous amount of state ensures that, over the number of iterations any DRBG user will do in practice, the entropy loss is negligible.

5.6 Guessing the State

The KHF_DRBG working state includes (N+L)*8 secret bits and N*8 public, changing bits; it tries to support an N*8 bit security level.

Some DRBGs can be attacked by guessing 2^{K-R} possible working states, where K is the security level of the DRBG, and then waiting for the DRBG to generate about 2^R outputs. At this point, it is likely that one of those outputs was generated by a state in the attacker's list of possible working states, and this permits recovery of the DRBG working state.

The KHF_DRBG has many more bits of secret working state than its security level. No guessing attack can succeed without guessing all of these bits, as far as I can see. There appears to be no hope for this kind of attack to work.

5.7 Efficiency and Implementations

My keyed function can be computed either with standard library calls to the underlying hash function, or more efficiently with access to the underlying compression function calls.

With standard library calls, each N byte output requires two compression function calls. With a lower-level implementation that has direct access to the compression function, each N byte output requires only one compression function call.

The keyed hash is always defined as

F(K,X) = hash(K0 || <M-N bytes of zeros> ||
(X || <L-N bytes of zeros) xor K1)

Optimized lower-level versions must always be true to that specification.

5.7.1 Efficient Versions

For hash functions where the hash output and the compression function output are the same length, the efficient version is simple:

I = the hash initial chaining variable

K0' = Compress(I,K0 || <M-N bytes of zeros>)

FinalPad = padding and length bytes for this hash function, when hashing an M+L byte message. (For all the SHA hashes, this is the same nine byte string.)

F(K,X) = Compress(K0', ((X||<L-N bytes of zeros>) xor K1) || FinalPad)

The speedup comes from caching K0', which would otherwise be recomputed each time.

For SHA1, SHA256, and SHA512, this is straightforward. For SHA224 and SHA384, the same more efficient version works, but in that case, N is the number of bytes of compression function output.

6 Conclusions

I've described my two closely-related DRBGs, and given some of my reasons for designing them this way. I think HMAC_DRBG should replace the current keyed hash DRBG, and KHF_DRBG should replace the current HASH_DRBG.

An application developer choosing which of these two DRBGs to use can make a decision based on his comfort with HMAC, vs. his desire for greater speed and state.

The HMAC_DRBG has the nice property that the public cryptographic community has a lot of faith in HMAC; when someone wants to specify a PRF, that's virtually always the one they choose.

The KHF_DRBG has much more state, and seems to me to be more likely to resist prediction attacks because of that.

7 A Python Implementation

```
##########
# Two New Hash DRBGs
##########

import sha,hmac,struct

def xorString(x,y):
    if len(x)<>len(y):
        raise ValueError,"Unequal lengths!"
    z = ""
    return "".join([chr(ord(x[i])^ord(y[i])) for i in range(len(x))])

def hash_df(bytes,seed):
    tmp = ""
```

```
    i = 0L
    while len(tmp)<bytes:
        tmp += sha.sha(struct.pack("!LL",bytes,i)+seed).digest()
        i += 1
    return tmp[:bytes]

class PRF_DRBG(object):
    def __init__(self):
        raise ValueError,"Abstract base class--don't instantiate directly!"
    def initialize(self,seed):
        self._renew(seed)
        self.ready=1
    def reseed(self,seed):
        if not self.ready: raise ValueError,"Not ready to reseed!"
        self._renew(seed)
    def generate(self,bytes,optionalInput=None):
        if not self.ready: raise ValueError,"Not ready to generate!"
        if bytes<0 or bytes>2L**32:
            raise ValueError,"Invalid number of bytes requested!"
        if optionalInput<>None:
            self._renew(optionalInput)
        tmp = ""
        while len(tmp)<bytes:
            tmp += self._crank()
        if optionalInput<>None:
            self._renew(optionalInput)
        else:
            self._renew("")
        return tmp[:bytes]

class HMAC_DRBG(PRF_DRBG):
    def __init__(self):
        self.ready = 0
        self.hashLen = 20
        self.K = "\x00"*self.hashLen
        self.X = "\x00"*self.hashLen
    def _crank(self):
        self.X = hmac.HMAC(self.K,self.X,sha).digest()
        return self.X
    def _renew(self,seed):
        self.K = hmac.HMAC(self.K,self.X + "\x00" + seed,sha).digest()
        self._crank()
        if seed<>"":
            self.K = hmac.HMAC(self.K,self.X + "\x01" + seed,sha).digest()
            self._crank()


class KHF_DRBG(PRF_DRBG):
    # SHA1, SHA224, SHA256: |K1|=55
    # SHA384, SHA512: |K1|=119
    def __init__(self):
        self.ready = 0
        self.K1Len = 55
        self.K0Len = 20
        self.K0 = "\x00"*self.K0Len
        self.K1 = "\x01"*self.K1Len
        self.X = "\x02"*self.K0Len
        self.Xpad = "\x00"*(64-9-self.K0Len)
        self.Kpad = "\x00"*(64-self.K0Len)
    def _crank(self):
        self.X = sha.sha(self.K0+self.Kpad+\
                         xorString(self.K1,self.X+self.Xpad)).digest()
        return self.X
```

```
    def _renew(self,seed):
        T = ""
        while len(T)<self.K0Len+self.K1Len:
            T += self._crank()
        T = xorString(T[:self.K0Len+self.K1Len],\
                      hash_df(self.K0Len+self.K1Len,seed))
        self.K0 = T[:self.K0Len]
        self.K1 = T[self.K0Len:self.K0Len+self.K1Len]
        self._crank()
```

8 Appendix B: Test Values

8.1 Test Value Code

I added a dump() method to both DRBGs, requiring them to dump their working state. I then used the following function to generate test cases:

```
def makeTestValues(drbg,steps=10):
    seed = "\x00"*20
    s2 = "\x01"
    for i in range(steps):
        print "Initialize with ",hexstr(seed)
        drbg.initialize(seed)
        drbg.dump()
        print "Generate 10 bytes."
        output = drbg.generate(10)
        print "Output = ",hexstr(output)
        drbg.dump()
        print "Generate 10 bytes with input ",hexstr(s2)
        output = drbg.generate(10,s2)
        drbg.dump()
        print "Reseed with ",hexstr(seed)
        drbg.reseed(seed)
        drbg.dump()
        seed = drbg.generate(20)
        s2 = drbg.generate(1)
```

This has many limitations; it won't tell you if your implementation is screwing up odd lengths, or whatever. But it will give you a nice way of checking an implementation for simple problems.

The only hash I've implemented these with so far is SHA1.

8.2 HMAC_DRBG

```
Initialize with  000000000000000000000000000000000000000000
self.K =  da438663a54f45cb975ff88306d89dbb8c59c2d9
self.X =  746b6df02eba0a55e9e31a22145331ffc85ea0a2
Generate 10 bytes.
Output =  c8d7010562961b25a1e9
self.K =  b8b2175189c4292013be7843f3abd04179e8a265
self.X =  009b84add5013e58865d90d248b6658fee75262b
Generate 10 bytes with input  01
self.K =  458cc16b6bee69c21cfaa74b66fb4bb5973d708b
self.X =  8af315aaf8c95869532e85ce9ab901fd8a634b48
Reseed with  000000000000000000000000000000000000000000
self.K =  290edd57b159fdd48bf30e60c03994bdaa545d02
self.X =  f9f240e6e14d167b535c8af9a0297b801155cdae
Initialize with  f632da001e06fdea3cdbbd4d12502fffc9ff3157
self.K =  d6320271b25f44e5f4bf0972fae2894c88dd1c50
self.X =  55d71af3b91c7fcd05ad6bd3ae877ee092357f21
```

```
Generate 10 bytes.
Output =  d66682505ea02b95852d
self.K =  a1e9ce1ba2d32009ba23e9dc657bc343cc839436
self.X =  11fcec940f6b8b0e74354c6a8a965f372d74b337
Generate 10 bytes with input  ea
self.K =  74ea15ee64b4cf2c9037783b0a39ba66cffb8f74
self.X =  9ce4a0477cbd698bceabe0bc51083d6765077eb2
Reseed with  f632da001e06fdea3cdbbd4d12502fffc9ff3157
self.K =  0120c1054507f82f16f1bbf7b79db5fc4ec9e9a3
self.X =  02c70d9f250edb5474ce293181edc50b05162d65
Initialize with  b74ebd085ffe1ae1f9c6e34257fc8832728bdc10
self.K =  b4d71a53b72ba429f38ca3996991260d75d4d39f
self.X =  499500c3696ed156ecb5e2f05369de4d92c4acb9
Generate 10 bytes.
Output =  d5910f345ea51c676499
self.K =  28122b3a55fac3569638582a3358c48f8bb61500
self.X =  e2f6b2933baddb33bf48c4c61f5f4673b34143a1
Generate 10 bytes with input  a4
self.K =  3ebd183c4722817505f56b89bc1d32773493677f
self.X =  3f25dd57a37a20ae1aae338e186ace60a1d82797
Reseed with  b74ebd085ffe1ae1f9c6e34257fc8832728bdc10
self.K =  96cf6de4cd84184ddc6809d81e59ff647551e2e3
self.X =  cef1fe7f75d4e31014ee2fdf668c3b9bb238544e
Initialize with  7ca3de5df870bfb5acb4f9fcccd43f538ab06957
self.K =  290d28c399859731a504a41434dc1afaac96f758
self.X =  9eb596f5f4f71acae81595ab192a4cf95b1f76e4
Generate 10 bytes.
Output =  5739f632810efc4edc81
self.K =  3a29064e1e56b0764d3973ea4bedac9c99c32118
self.X =  80649fe6d006f1d16c5eea175beab0806a1851dc
Generate 10 bytes with input  5f
self.K =  8022af0eb3da9a7b2a6fcbed169521c8c6ab5a9b
self.X =  8b0ceb3109c8a884a60219e50f848ffde245b9ff
Reseed with  7ca3de5df870bfb5acb4f9fcccd43f538ab06957
self.K =  4336cc705bf216fe35ffa7ac191a51d1b2f1609a
self.X =  969d476db473e0956b8ce245391977e009a8625a
Initialize with  d3e229092fc44a4df09d0d968cc5f9fe20e9bd3f
self.K =  598f9621c2c8bdc7d060525ffcae84e6c444dcdd
self.X =  6612217e2f2a11aaa096d3a432079f781bd040da
Generate 10 bytes.
Output =  2d860ec773ba40498e90
self.K =  bc66914d480421e45e57572209f7c9a419224184
self.X =  2170579d7c56af85f1de2ce51ba0b342a47d6f0d
Generate 10 bytes with input  04
self.K =  d11713dc0900607a97478242aca66a2a135982e2
self.X =  1d1c5085cd20fb5352533c739f60216ad26f2c3b
Reseed with  d3e229092fc44a4df09d0d968cc5f9fe20e9bd3f
self.K =  b56f829ffbb59a61184d37acf2ea229ea3fc99da
self.X =  621583f07bbc35d1968c0300ed5caaf510682028
Initialize with  167924af4c3f987a5e0b61c1d3c98bdcbd5290a5
self.K =  902c4df4ceececbdbc023f2cb32e3a11ee2693ca
self.X =  66507819a2b8b0461616dcb02acb7af19c4dd9db
Generate 10 bytes.
Output =  86ccbde52e2165c52c18
self.K =  a46140cf7e491579d8d75f944853c0f2897879b5
self.X =  3b29fbe2ed26d4b27767917b20ebbebd492a3492
Generate 10 bytes with input  5a
self.K =  81fa6394d2a46f5f86f89514ce66cc15f8f20a61
self.X =  8aafbb53857b4ac077087023c421becde796069a
Reseed with  167924af4c3f987a5e0b61c1d3c98bdcbd5290a5
self.K =  43f337a2075b6bc83dac55b50c70d5e745ddfea8
self.X =  fe52e446762004ba09b2d1965984fea7330879e5
Initialize with  285fce04ebda7bc84f4b153fda90d77a600accc4
```

```
self.K =  169d8a351fbbd1e65662ca054c1d673bbe155a18
self.X =  736e7ce27b2a36243568c490b9861e2b8af24ea3
Generate 10 bytes.
Output =  e0b019de5ebebfb9f1b8
self.K =  06eaafe0b0361980a74aa91ca6c710f79447d533
self.X =  7662ad2b1410cee97cb5c20cee17b675e7df79c1
Generate 10 bytes with input  a4
self.K =  657138fd4a6369618deb0ea2e51ab3735e1efad1
self.X =  565cbb5f1ef3a36aac3742a69c978cd8e7d27923
Reseed with  285fce04ebda7bc84f4b153fda90d77a600accc4
self.K =  e7d0e493c5eef52809bb60d79e0560093c1001d3
self.X =  4319000f67b9cc4dcfcd236a5bac6559e1662bfc
Initialize with  717d2094d36f16e4f310e9afb6f4a3de4501c1b2
self.K =  03b2727dc8d0bf76e2a68eba1e61c6a74d7e3ac1
self.X =  08fbdaf961d48081c5f169e2b942e869c57071be
Generate 10 bytes.
Output =  624e10fad397c567ba1c
self.K =  fd89709c194a24ab8b79644bdd0521f2fa1ce885
self.X =  f19d091dd1bdd6b562a6eb3422d83aa5ebfcc099
Generate 10 bytes with input  23
self.K =  dacac124d74a27cab7c070576c1b76cf67b07a7d
self.X =  506b5aec800fa61ce951df87a016b0d34ddf239d
Reseed with  717d2094d36f16e4f310e9afb6f4a3de4501c1b2
self.K =  ed0aacc1c134099eb1e578c2250e627eac6c1ab4
self.X =  a991d55344e3d19429642335794c8a849a164dad
Initialize with  bfe275cdeefa9ed34b6483746631969663e7dbbb
self.K =  fb3fc003f41f867f85087bc627ca7511461b882e
self.X =  d52d8858d2b39fac2379c02053251e31ee00b390
Generate 10 bytes.
Output =  3d6a6ef44bb6a2d54990
self.K =  3f4884a6205d62c8519e84eca90f0f1017c9d0be
self.X =  329976e2b797d1178fc4d7fac47c99a4752f9ba2
Generate 10 bytes with input  e5
self.K =  da511d8d3a45ce7b637548d0a272dbdc43982d5f
self.X =  a74d6fb3b8bbb8c10c5334c251c4d8c218d3c615
Reseed with  bfe275cdeefa9ed34b6483746631969663e7dbbb
self.K =  2863c1701bcfb0e95034311b6563a12ca410d9a5
self.X =  109f31a53d01317a2033b624346bbead2755c7e0
Initialize with  4e04ca09373bcfb63f1b1525a1179df9ee95b922
self.K =  35349776def8d43b3d54c0e06d7a5cb612fdcbc3
self.X =  33df42996a0aced640ed415a50eb629af75a27f4
Generate 10 bytes.
Output =  1af0b2b45b1a4d2a6991
self.K =  93d77aa3b743f8fb65b6aa0f7b19afd6ce1de69e
self.X =  63b2e82eccc7a337fd2aa627074b6e9486b4dccf
Generate 10 bytes with input  f8
self.K =  4c054003217010d3fff371083ea2b0e0ece1bfb2
self.X =  d2cf6f3c37dad075e03053bcd001ce09aaf1efeb
Reseed with  4e04ca09373bcfb63f1b1525a1179df9ee95b922
self.K =  fc22ce7cdb80262952a62755517e570f6648c56d
self.X =  ff1a5ccd7dfba07d61c1702c0066c19ae2289cfc
```

8.3 KHF_DRBG

```
Initialize with  000000000000000000000000000000000000000000
self.K0 =  3ada0d3db8af9f448f04459541abffe91a1d69d8
self.K1 =
064b6b91ac504130f30189fdbeff83f6c9fd2a48d44ddad4d031c6a7c346088edfcb0e843b492431623f
74ac1cf02aa10f3759865504c1
self.X =  fa492bdaccdc710a0cff7a2ec6bcc4142e3b5f05
Generate 10 bytes.
Output =  4322c7c1b0e1b74c45ab
self.K0 =  e0bb158853ab29a119dbd6daa4b453d3f62a9b3d
```

```
self.K1 =
070c3b8b0631474892b58fa6587d7b9a3ed8d8784ae5019e38216a0555967c3a8ea42d9fcbc30200ce98
d46dabcc05ff92e18af8e34fc4
self.X =  48a4d8044d0b407edc29ab0609d90bdcf54b4da6
Generate 10 bytes with input  01
self.K0 =  7ff2ec172acf960374d9cce71592bb05d0cfbfb1
self.K1 =
2243c693f14c73e228bdb264b2a145bff4ba0cf5601563f8430902af7f7fa4ab0a4e2ee5ac5d3b42787a
7bddafb2d2cad6af0ae18c3b1d
self.X =  c6fe41b607c01581a8e2bde25e1805c64507735a
Reseed with  00000000000000000000000000000000000000000000
self.K0 =  6816b451cdb76a073b1871a6dcd30d1d4015e6b4
self.K1 =
f56586907986408608951 1b3b8bd3a78809b7e938a3f11848ef7e47aed44c712c7fd3f9c17d3acfb42f8
9902bde1c990255c142c931f37
self.X =  2b761db506c41ef6bf695c0087bc284df0597cd9
Initialize with  91f6a6bef642f36f741442efe6f95366b8978971
self.K0 =  16054c531788bc36a125aac1939c6721b606d608
self.K1 =
7c913106b916c994f6a181bc2aced9ab4e0277ef426904a588008c6fd5119c4b67e0bab20c3ff77dffe0
c36b5cffad95b616ac71958f90
self.X =  7af8dbc9db38f1e0f2d33c8219551e4f1e6149fa
Generate 10 bytes.
Output =  bf49dd4a446712c88a14
self.K0 =  c22a9964b05323cd57817525ecdbadc7a27105c4
self.K1 =
0be648d24c9fda8f435a4ce3435b6b8f18687ee500d5f20fa09df6b74a73e41c155ae6212a445bb153b2
487b16aea473c8aa3f7ef8d157
self.X =  72fc792277bb7b9a36fa5a049f2da3d69d0736ab
Generate 10 bytes with input  1e
self.K0 =  b9f4afa57fb1a1ce4f02c1a4c48cf94ac48b72a9
self.K1 =
e632a5777bf238962859171f2d43a6acf3085519842dc489512846fc2190010e224fc52e13afed819cdd
5f4df94505d7f6a154c7425885
self.X =  b03b118d99fcab2f6b7a6aa15138312d8996999a
Reseed with  91f6a6bef642f36f741442efe6f95366b8978971
self.K0 =  d6b6b2d5724bc00a58fb3cc81ccdc4593546673d
self.K1 =
bb37437bc1d5091e5c490f5bb1deed08a0a397d22697528519d7ba2f3d9f88f575117e56976ee16b1d0a
a1a2d815b161343ff5595b0241
self.X =  4118bdc1c858acde4584534ae1537cc9b3998f42
Initialize with  7220b62935dc720e314933ec3cbff3aa98c1fc14
self.K0 =  1e4e6ec5921d686cc3f9f61d2c1e86add224cecb
self.K1 =
ce06dd0b9fbd643b91cc1fb66b86dce34f335cdcf6e27aff2585c03daebaed1bea0abcab8f68bfa1b28d
335a159f54070f099a080085b6
self.X =  583badc76a47a91d86b5151174b464399f98d228
Generate 10 bytes.
Output =  7b66219cfbc27cd63be5
self.K0 =  94e2d4990fbf94c4eda0cce8238ed39d41bb1008
self.K1 =
47dbef802f1af43a79e79f682e3121acbe88a0cfe07f68a08dd05af13b3687c3a4e474cee751d84c4053
be7f454aa14944bf57e8e98c5b
self.X =  56c1cc84eb6a4b2c16b1712efdeed3a64dc71d70
Generate 10 bytes with input  9c
self.K0 =  c06f6ec90406e43dad492182da07164da989171b
self.K1 =
effd3325da81f3942286caf8f1990cc2a798a9a52c689c514d56483ecddcab5d3d7c1574ce87355c5daa
2fd6989fc95d6f0173b8a99808
self.X =  3a5b483bd1795b2094ab3865866845354607d320
Reseed with  7220b62935dc720e314933ec3cbff3aa98c1fc14
self.K0 =  409f9cefb0ffbb29a1ab3a10a138244c8c99695b
self.K1 =
```

28aec993ce1b7af3dba1a28d515e8380e2cdecc9f0d3eb5bb86112e7679d212e01ad56c8eee5ee56db67
653219999b722465d463fbe29c
self.X = e4dec57f5ad265a1692787a41214c5714092cfda
Initialize with af7d55b462a216e5cf7074ae1ec926b3744161f5
self.K0 = 652cdd8f6812b1a3666411695651801a64a17c3a
self.K1 =
668555764bd9b596269a6ed6c3e79b7cefe61ec463404db89d739265b366285a2eecfc101addbc80e749
7f6f86ec18faa94fc795e7bc40
self.X = 9ab644e9cd0252905644affd069587f704debfd7
Generate 10 bytes.
Output = 712059e3f04f9f78faf7
self.K0 = 43cecda4478359fae97ca225600dddd662a6e095
self.K1 =
6228f444392e2a1913c69c00a3df73f93d64f028672026afe568c7bd55a5d5cffc0151d2ad7d658905f8
e7be0c9710e0d6f6b69945e0f5
self.X = ee125c7949210875536928818150f9d4ea8d6318b
Generate 10 bytes with input da
self.K0 = 3b17903d84a868b136328045c3e79cc43d111777
self.K1 =
a17ac48f109c4b8ad364467c6c071f3e47df159fc6ddcb811323e06e239ed1021f02589706ffe58119b5
9244e396e4a77e2ad82ba5f2de
self.X = a369233a9682e14ac700f6853b1349f993c62a17
Reseed with af7d55b462a216e5cf7074ae1ec926b3744161f5
self.K0 = 4ac165f1e7a0da926a0f230eb205123975335c76
self.K1 =
bdaee2ca790ad94594f86534b18e5b1f39fbe29930c26ac8ef03b3b389f5365b9fbc20f7154e10a85e77
4f120f680123eb7dee4633a761
self.X = 6356cc399ec3dd2d2724bbfb45752b44282365c6
Initialize with c9a098457f33f699b02ebde84fc21580a6321fdc
self.K0 = 3fc37645e176869c03346abf930ab8c576270407
self.K1 =
699309a1f90ac0f53fce38d456675643ad7348c953198ba810f18fc1cbae3178fb051ac329860a7493c1
49aeabe73d8a3fc16a44d4b941
self.X = 1be9bbe8a0a5872b546f50dbc0a67d3da7700189
Generate 10 bytes.
Output = 326048ae86c07d584c5b
self.K0 = d71c98b36e0d4c4819f3cef1eafb89c39d506944
self.K1 =
524c8ca0f99819fba39733c42cb1dc72298529082b7f294c093221a1df109bc2dc41163c8a1af13f0d02
6c1429f843a727d34950a25f9a
self.X = ababbaf00b1bdc28d9faca030343c986aa964633
Generate 10 bytes with input 09
self.K0 = 58f99520fa1c34429ec8885b350cb2d7718c87df
self.K1 =
d105dccf704316b093bdf01e40e147edf62a4df581766a7b20d65c84097d8feb42f0ef8686688addbc09
a8535458929ee48e4e94126e13
self.X = 41a392974292f54eb3cfce4a4a3aec77e1306bf4
Reseed with c9a098457f33f699b02ebde84fc21580a6321fdc
self.K0 = 9dabc8f5951d40644ad9630bd00e8eb64cb3e7e0
self.K1 =
56a4c302739152a8ceb134c94f57f954a5cc7a98ce6cde466d5c7bd3ea20e429f8d5ce427be795664f6c
e96a761c3ba3c03bcb7e1df633
self.X = 3f087db63c8bc7e9d06bee983876285f87651a63
Initialize with 542413c263c4fb43dca3fbbe528e7cc4396883d5
self.K0 = ee77a04c04ca9d4520433feaa2a5b81fd9bb7839
self.K1 =
b5952657c5aa1609d659f51a0ae74951fe56da6f6adb4522603272dc02bf70e16a0a8c2b90f539b6e967
f69c7ff18e5db3bfe0fb1def05
self.X = 7172851df5cc4841227407499101069l2c176e0e
Generate 10 bytes.
Output = 7e34e3730905042814d5
self.K0 = 8e10ef1c470be660149c230f7fb2a60031b4f748
self.K1 =

2a61eb0f38d84dac194938f3a0e4680f097b70db39a96ce4b658545254c61be739780a08204644a19d64
123ee9c371cc617d7a844f38a5
self.X = 777cc6e8cbef98490dca1e008cee513eb7a0cf5c
Generate 10 bytes with input f2
self.K0 = 65271a8801110e9503a401d87e308a93130a95c3
self.K1 =
d20428db2392a71e7997f96a1ca69e651e224bfd2ae7bf57661e1132b16b7b51c869eb6d48bd93cfe028
4bd14e2e4e1d923ac502a8b2bf
self.X = b3cd61533d402477bdddd893516a510d5c285915
Reseed with 542413c263c4fb43dca3fbbe528e7cc4396883d5
self.K0 = 4a950c5e1dde7bfae3b971f556b5eebf513c651a
self.K1 =
667443cb18b52a6285f984527b70703efac9b4a32f35d921c90830b704765f6d49685e3fee418de7b5b5
d43329341053a7ef7283697f13
self.X = d34eebd86dba2eb8e33eb7cb57df0178a91bd86b
Initialize with abf9928cdd90924bfad37432db552c816c364cad
self.K0 = fc1268cb3bb3a15803bb565d1c6cd9d4c7f5ef43
self.K1 =
eae26a0992c5464af07c9b8cc837b290082586455bebefb6dd732424190f3d7d51a04f1a34b0d3839f23
cd9293fb1328378a4420c5c940
self.X = 2f91b912310546ad11c2626e42acc86bdc7e17dc
Generate 10 bytes.
Output = 63dc941611cb507c5550
self.K0 = 41450bf4e88344c3165e30017c6efa060589050d
self.K1 =
766dad9c8bccf3a8ff2702450008e3e00e90f4a3e61afc067fa453e2ecd08b376204326548de065784fc
3c8fd55f04062ee12470611628
self.X = 3958daf2d54e355899e30b222b2b1174bfcc32db
Generate 10 bytes with input 0f
self.K0 = 7a93c03bb983cd95e4ca410f1e2a617d5f8f7cd9
self.K1 =
1258892aaa757dcc44e95d7a41c4a2dca92a9132b796b4527420da99c5ff3d1bad0928ba0746e668fb3f
fb13f4eca70148f248568899b8
self.X = 9754109c9faed2d52204de8b5f453b647a4048e0
Reseed with abf9928cdd90924bfad37432db552c816c364cad
self.K0 = e337963327d54b14688a09b98d3c51bb140e120a
self.K1 =
91cd5f0cdc587cb70156fbdec16fe8802f28c6dce191b0a89abf06cd22baa2e6c85c61a39a1c1b1157bc
b823b1457a93cd5aaa04847e70
self.X = afdd1bde32b6565809f4b688461519cf41857f1a
Initialize with 65cc0fa7a0bf38b637de3d6fc6522b7a4f203797
self.K0 = d7a07aacf6a1fe4a1e9d3bdf5903c68a3e5c76b8
self.K1 =
5e864aa18ede2afc2018ef06c2aa3fce3467fe1379aeaedc40015ef13b620adcb558005fff73f8a0fda8
9c63c000d680347230f679b04f
self.X = cc07b1cdfb44633bee170bdaf5be760a45c3e41e
Generate 10 bytes.
Output = 0270fe62e8202eda0eae
self.K0 = 33fc2fa2fca0582d32dddc66569c0f5f4abd4972
self.K1 =
39d3a5f95ece78ff7d069ee4907e24dae127fc6de9e2f865f1750fb6d2e42c7f0a4a0366b25b5b3109a2
8660a5ae9f453e5ac47dd2f2af
self.X = 5cb3f3f55cf630c6da44239c39e8c96662321c0d
Generate 10 bytes with input b2
self.K0 = c2be46972e3637d42cd3482a9b61894ed0b20c8b
self.K1 =
b47ff73cdeb85f393f080dafd8201f1f41eb59c4e95c1aa8aad6ce5315a12e1d371be4776c2df5fdd644
9a189697a027e0114e4de9885f
self.X = 4cff70a18a1e2d0867213bc17ce91b73200ad70e
Reseed with 65cc0fa7a0bf38b637de3d6fc6522b7a4f203797
self.K0 = adaab40017e0a0d54609b3b5e55aed495700a4bf
self.K1 =
4b299b86de40c204f9e2d752f766f583a33dffb27e739368fa3a65e202312a3c5ddc07bf1b4cbf91c50b

```
16c61174c3e5ce8683b62ee166
self.X =  a83c3d1dd52e922a5c947cb01acfbc5070be5022
Initialize with  1472eab4e8968723cbd54e8f4ecdeaf6a4674723
self.K0 =  933475d4bdc1e56916227cb29600c9e2826da3a8
self.K1 =
9cbb6f09dbd2dc53cc5892364745e5e7f7848edf92c97e85e0bb9ad5891cdb919b5d01da40c55127d596
19203842962bc68e1241e7005d
self.X =  478475b2318128fe17a21a5dbb75892a7a23114f
Generate 10 bytes.
Output =  384ee6926077d860b1e8
self.K0 =  fed80bd234a0f6edf96841ba8a34c4c1b6baa74e
self.K1 =
32078d7aff91f57d67960a63b8ec1ccaf1054d81731d50eb6cf85bf3b924a273d3532b51f988993dc21f
cc9c55a93618e3976d3c1b92b9
self.X =  2a77965a90a65927904d15411ddec7d5644bea9d
Generate 10 bytes with input  9e
self.K0 =  6b63aad9cc1fd29ce874992ef75f740e6bf309cc
self.K1 =
f98950d7a4f3a6401d9e22a4602aefcc1092e5c63208229480d964b14aa3432615fadb6df45acac8867d
60fbe46bcde2a3bcd27216984a
self.X =  313ffe3b3e9ccdac2f340f762bda11c8b309a1ec
Reseed with  1472eab4e8968723cbd54e8f4ecdeaf6a4674723
self.K0 =  cee82fa0477e4b7790b9324fd5f48843843e20e3
self.K1 =
e7abced7196776856d9737db7a44c2dadfbffdabedf932e6bafcd549533a32ad7b9519678cfbafda052a
c26eb6bb60075b607e77849a55
self.X =  518fd42dc3c86513cbf97b5355eb47834faeec01
Initialize with  b749c704ba27e7c6069b5f60ae16fe35f045d71b
self.K0 =  835e1799407f1157114c338dcd85d49d35f76334
self.K1 =
1daa0cd30b038680146 0f7a0ceccb293aedbcf4813cb7752add44f2754d782aac173cd13bb85e2f6b492
0a7ac02dcfd4e23e13b3999f93
self.X =  6791cdc9c5627459da32c122d0921101197f2572
Generate 10 bytes.
Output =  95448c85ed031a77b2c1
self.K0 =  4e081e6edf6178a3896920472efded498d955a83
self.K1 =
5d67bf42bfb83ec772dcb135e1645a17a7cdaaae04d065e4d6936ac4d561541522b3d12ebd3abe6b8ad4
4677d194d58ffd501dc62a3179
self.X =  9384b91fedec857c3751f32154d7f0334ab88865
Generate 10 bytes with input  17
self.K0 =  9eb9279a362070b0f57fe27949879ff6aec499da
self.K1 =
66b90c26e25c6fc4648a2812e7e0a5c36741600e135254bafea6e10df4bb56862867998961be7da5a5ee
01ad8c54d9d2e0992542440c61
self.X =  525493bdc4cf81abb34247c340f46d7d8a22fa3f
Reseed with  b749c704ba27e7c6069b5f60ae16fe35f045d71b
self.K0 =  73cb311c0668d7ba0107bcbdad4daf1185b92e21
self.K1 =
0f3b80d3386930082d4f564535f9b4527bbb7b7517b9e80dbb4429be7655e0c38d3b7169c5f8345d18ef
1c6faacc4a406324de57ea21b5
self.X =  18683be763f26c1ab37c387368a013e922407a93
```

8.4 Spin Tests

This code can be used to give the DRBG a lot more chances to fail.

```
def spinTest(drbg,steps=1000,seed=None):
    if seed==None: seed = "%s"%time.time()
    drbg.initialize(seed)
    print "Starting Seed = ",hexstr(seed)
    for i in xrange(steps):
        seed = drbg.generate(20+i,seed)
```

```
    print "Ending Value:",hexstr(drbg.generate(10))
```

The results I generated were

```
HMAC_DRBG
new_hash_drbgs.spinTest(new_hash_drbgs.HMAC_DRBG(),seed="\x00"*10)
Starting Seed =  00000000000000000000
Ending Value: 0b5d546d778393c12880

KHF_DRBG
>>> new_hash_drbgs.spinTest(new_hash_drbgs.KHF_DRBG(),seed="\x00"*10)
Starting Seed =  00000000000000000000
Ending Value: f1f79f6f564b95e98c0d
```

These values represent generating every output length from 21 to 10020 bytes, and reseeding with the same lengths.