

Five DRBG Algorithms Based on Hash Functions and Block Ciphers

John Kelsey

NIST

July 2004

Overview

- Why So Many?
- Preliminaries
- Hash Based DRBGs
- Block Cipher Based DRBGs
- Wrapup

Preliminaries

Why So Many?

Properties of All DRBGs

Some Security Definitions

Five Symmetric DRBGs?

- Three hash-function based
- Two block-cipher based
- Why have so many?
 - Performance/security assumption tradeoffs.
 - Let designer use what he has available.
 - Minimize additional algorithm dependence.

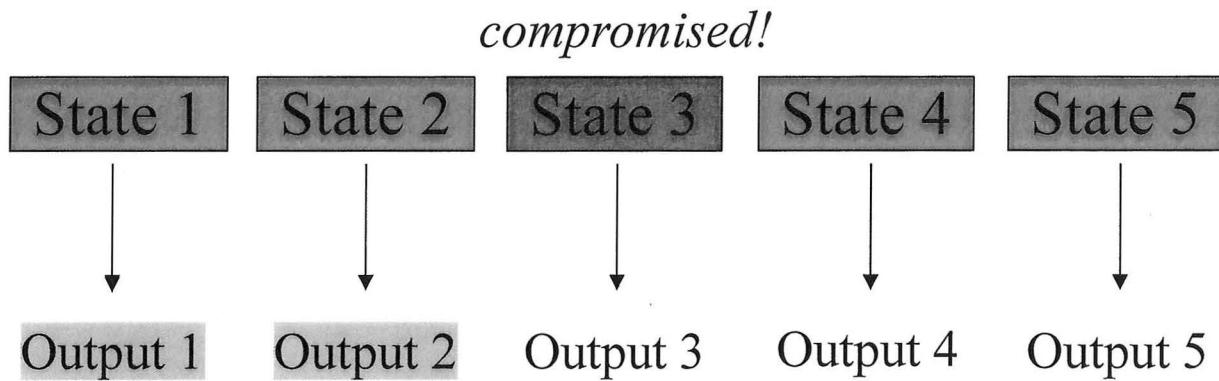
Preliminaries: Every DRBG Has....

- Security Level
 - 80, 112, 128, 192, or 256 bits
 - k -bit security level corresponds to a k -bit AES key
 - *Security level determines what mechanisms this DRBG can support.*
- *A Working State*
 - *At least $k+64$ bits, for security level k*
 - *Protected just like a key*
- *Assumption: No innocent party ever does more than 2^{64} of anything!*

Every DRBG Supports Three “Methods”

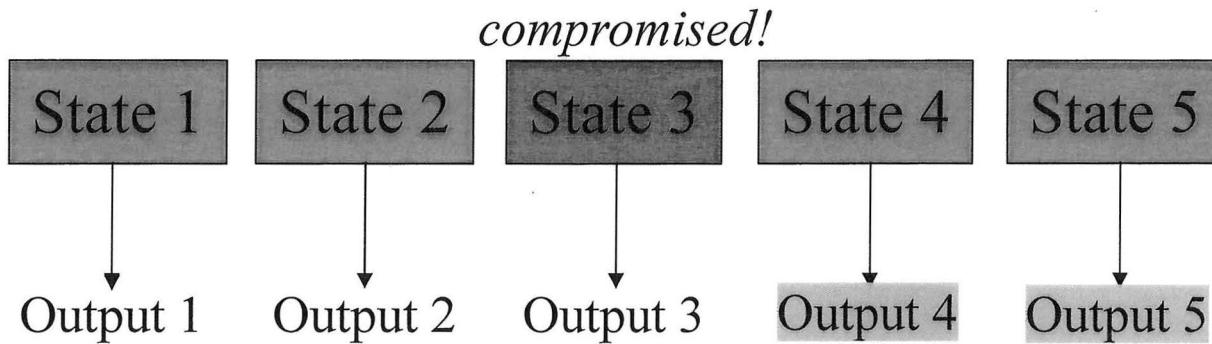
- *Instantiate*—Start the DRBG in a secure state.
- *Reseed*—Put the DRBG into a new, secure state.
- *Generate*—Produce pseudorandom output.
 - Update state after call for backtracking resistance.
 - Limit of 2^{32} bytes of output per request.
 - Limit of 2^{32} Generate requests.
 - Optionally accept additional input—prediction resistance.

Backtracking Resistance



- *Compromise of state has no effect on security of previous outputs.*
 - Example: Compromised State 3 has no effect on security of Outputs 1,2.
- *All our DRBGs provide backtracking resistance!*
 - Easy to do algorithmically
 - Per Generate call
- *Captured modules, forward secrecy*

Prediction Resistance



- *Compromise of state has no effect on security of later outputs.*
 - Example: Compromised State 3 has no effect on security of Outputs 4, 5.
- *Requires additional entropy*
 - Our DRBGs can support it per Generate call
- *Allows recovery from compromise or weak state.*

Basic Outline of All Symmetric DRBGs' Generate Calls:

- Process additional-input, if any
 - Update state with additional-input, if it exists.
Otherwise, skip this step.
- Generate the pseudorandom bits
 - Use current state to produce the bits as requested.
- Update state to provide backtracking resistance
 - If additional-input is present, use it;
 - Otherwise, update with just current state.

Entropy and Derivation Functions

- We assume inputs with at least k bits of min-entropy.
- We sometimes use derivation functions to process inputs:
 - Map input with k bits of min-entropy to random looking string of any desired length.
 - Ideally, indistinguishable outputs from random.
 - Practical requirement is no bad interaction with entropy source distributions or DRBG algorithms.

Hash-Based DRBGs

HMAC-DRBG

KHF-DRBG

Hash-DRBG

Preliminaries: The Compression Function

- Hash functions built on top of compression function:
 - Message padded to whole number of blocks, including length of input
 - Each message processed in turn
- Compression function parameters:
 - *Inlen* = message input size (512 for SHA1)
 - *Outlen* = hash output size (160 for SHA1)
- Note: *All our designs can be implemented with top-level hash interface, e.g., hash(X)*

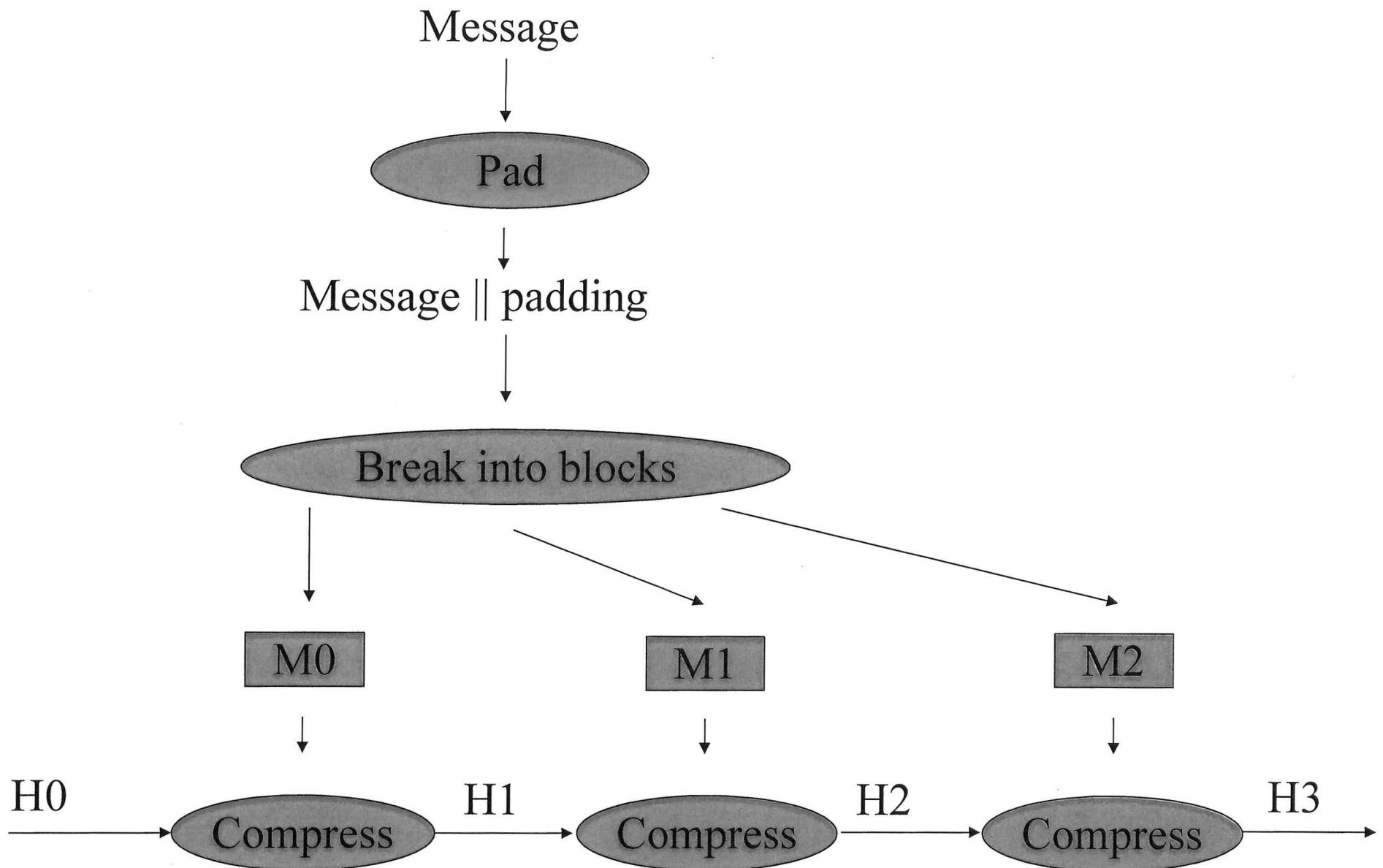


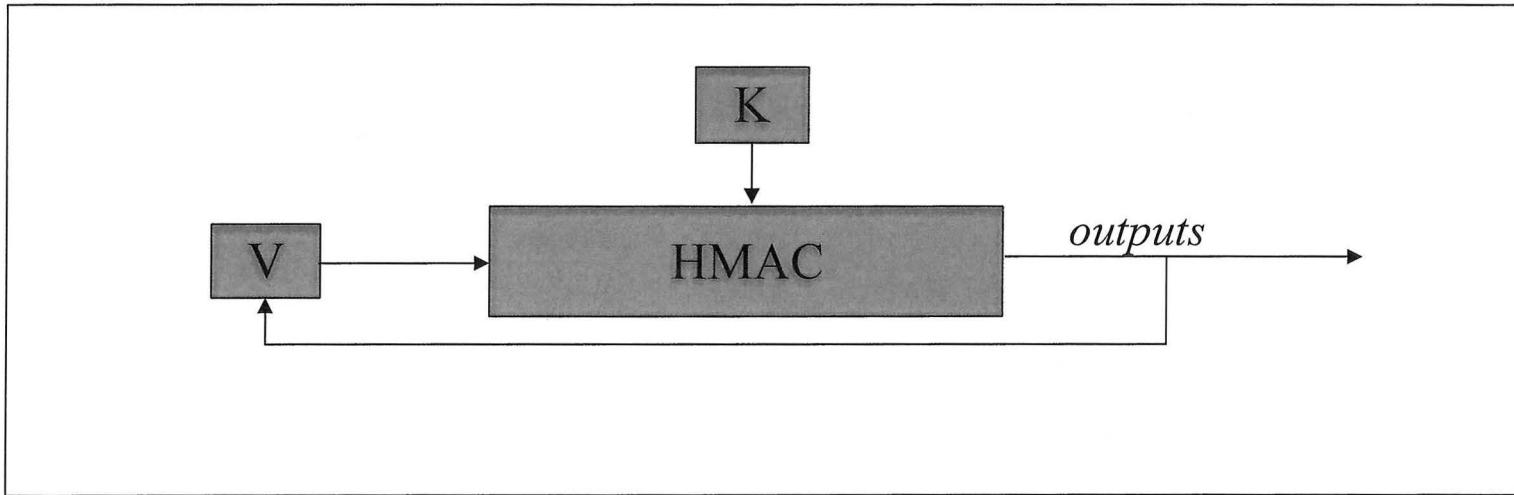
Illustration: Hashes and Compression Functions

Hash-Based DRBGs

Security Assumptions

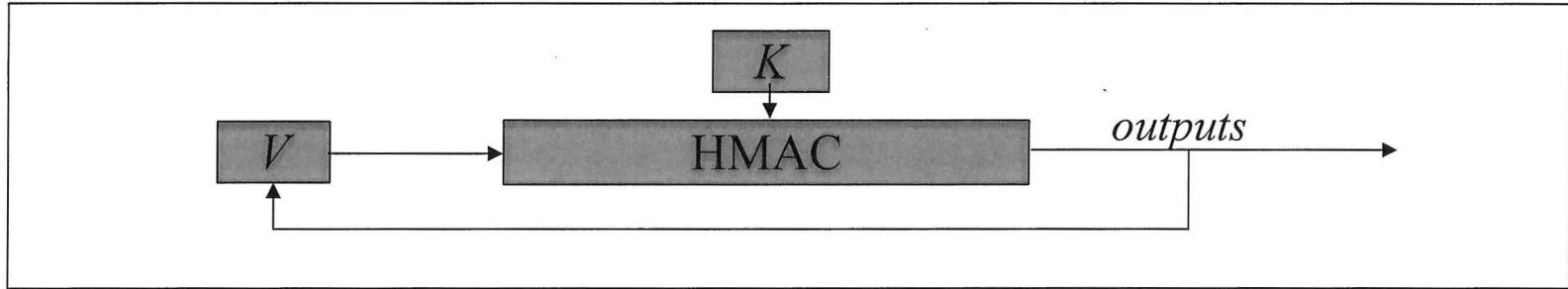
- Hashes designed for
 - Collision Resistance
 - Preimage Resistance
- DRBGs need pseudorandomness properties
- Possible that all our hash-based DRBGs are broken, but hashes are still okay
 - But for HMAC-DRBG, it would break HMAC as a PRF.
- Note: hashes used same way for key derivation, etc., all the time!

HMAC-DRBG



- Generation: Run HMAC in OFB-mode
 - Derive new HMAC key between generate calls
- Updating State: Apply HMAC to $V \parallel \text{inputString}$
- Security based on PRF assumption for HMAC

HMAC-DRBG: Generate



To produce N bits:

$tmp = \text{"")}$

while $\text{bitLength}(tmp) < N$:

$V = \text{HMAC}(K, V)$

$tmp = tmp \parallel V$

return leftmost N bits of tmp

HMAC-DRBG: Security of Generate Outputs

- If K good HMAC key, then...

Distinguishing Generate outputs from random
means

Distinguishing HMAC from random function

HMAC-DRBG: Updating State

- After state, given no additional input, we do:
$$K = \text{HMAC}(K, V \parallel 0x00)$$
$$V = \text{HMAC}(K, V)$$
- Backtracking resistance:
 - Learn previous K from new K == invert hash function
- Random selection of keys:
 - Distinguish new K from random w/o old K ==> Distinguish HMAC from random function
 - No cycling problems given our limits/assumptions

HMAC-DRBG: Updating With Input

- Instantiate, Reseed, and Generate: all use Update internal function

$$K = \text{HMAC}(K, V \parallel 0x00 \parallel \text{inputString})$$

$$V = \text{HMAC}(K, V)$$

$$K = \text{HMAC}(K, V \parallel 0x01 \parallel \text{inputString})$$

$$V = \text{HMAC}(K, V)$$

Question: Do we get required security properties?

HMAC-DRBG: Recovering From Compromise

- Suppose K known, input not:

$$K = \text{HMAC}(K, V \parallel 0x00 \parallel \text{inputString})$$

K is just result of hashing inputString with known prefix, then hashing result with known prefix:

Attacker who can't guess inputString should not know new K

Recall full procedure:

$$K = \text{HMAC}(K, V \parallel 0x00 \parallel \text{inputString})$$

$$V = \text{HMAC}(K, V)$$

$$K = \text{HMAC}(K, V \parallel 0x01 \parallel \text{inputString})$$

$$V = \text{HMAC}(K, V)$$

HMAC-DRBG: Resisting Chosen Input Attack

- Attacker chooses $inputString$, doesn't know K

$$K = \text{HMAC}(K, V \parallel 0x00 \parallel inputString)$$

$$V = \text{HMAC}(K, V)$$

$$K = \text{HMAC}(K, V \parallel 0x01 \parallel inputString)$$

$$V = \text{HMAC}(K, V)$$

- Attacker gets chosen input attack on HMAC
 - Few queries, never more than 2^{64}
 - Doesn't see outputs directly—can't see collisions!

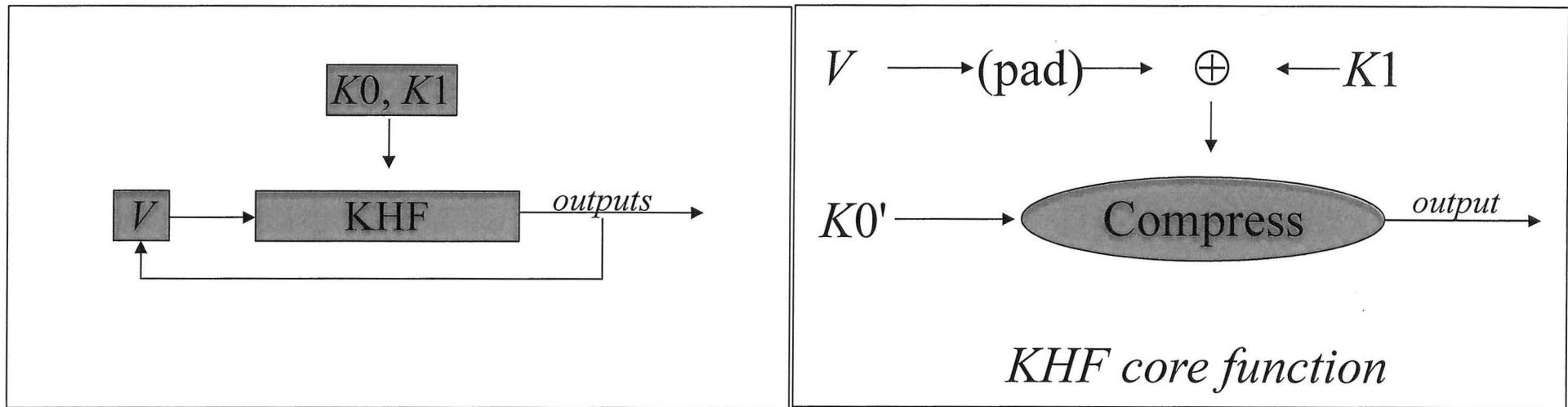
HMAC-DRBG: Performance

- *Overhead on each Generate call:*
 - 6 compress calls
- *Per outlen bits of output:*
 - 2 compress calls
- *Reseed, Instantiate:*
 - 12 compress calls

HMAC-DRBG: Summary

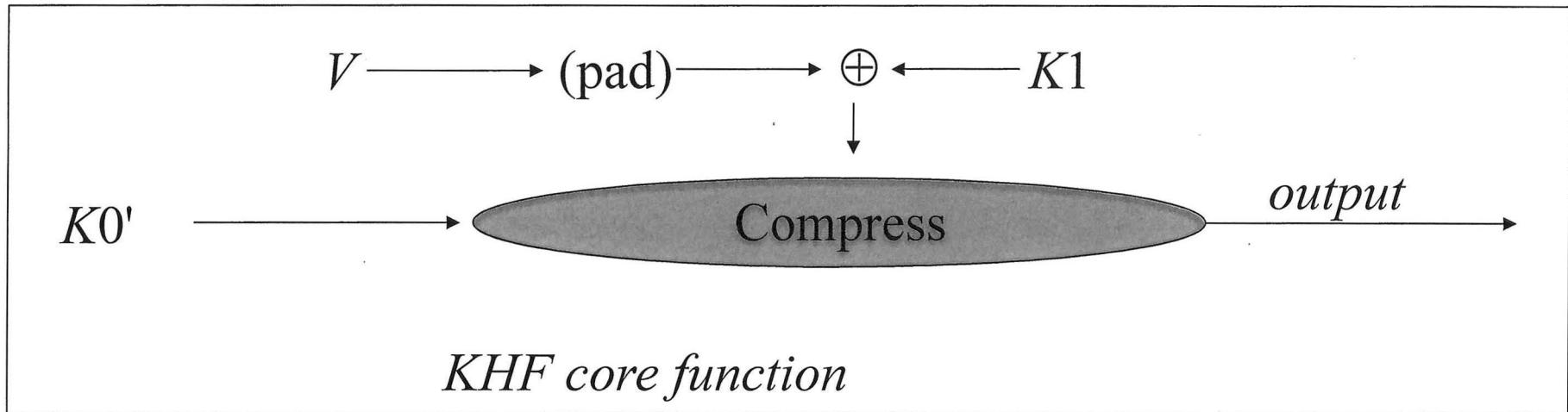
- HMAC-DRBG is:
 - Simple design
 - Makes easy assumptions on hash
 - Probably most robust hash-based design
- HMAC-DRBG Performance:
 - Slowest of hash-based DRBGs proposed

KHF-DRBG



- KHF core function takes one compress call
- Can be computed less efficiently with generic hash calls.
- Result: better performance, minimal number of input bits known to attacker

KHF as a PRF



- KHF is an attempt to make a PRF that's faster than HMAC—one compress call per KHF() call.
- Note:
 - Attacker knows only 72 bits of input to compression function
 - Attacker knows precise XOR differences within Generate call

KHF-DRBG: Security of Generate

- Same basic design as HMAC-DRBG.
 - Using OFB-mode instead of counter-mode means random-looking known-inputs only
 - Limits to number of queries
- Distinguishing Generate outputs from random means

Distinguishing KHF from random function

KHF-DRBG: Update

- Internal function *update* used for *Instantiate*, *Reseed*, and state update within *Generate*
- *In words:*
 - Generate a new key for KHF with KHF-DRBG
 - Generate a new key for KHF with hash_df
 - XOR the two together to get the new KHF key

KHF-DRBG: Update in pseudocode

Update(*inputString*):

tmp = “”

while bitLength(*tmp*) < *inlen* + *outlen* - 72:

V = KHF (*K0*, *K1*, *V*)

tmp = *tmp* || *V*

K0, *K1* = leftmost (*inlen* + *outlen* - 72) bits of *tmp*

XOR

hash_df (*inputString*)

V = KHF (*K0*, *K1*, *V*)

KHF-DRBG: Update Recovery from Compromise

- Suppose attacker knows (K_0, K_1) , not $\textit{inputString}$
- Attacker knows new (K_0, K_1) is
 - Known value XOR $\text{hash_df}(\textit{inputString})$
- If $\text{hash_df}(\textit{inputString})$ generates good KHF key given unguessable input,

then KHF-DRBG recovers from compromise.

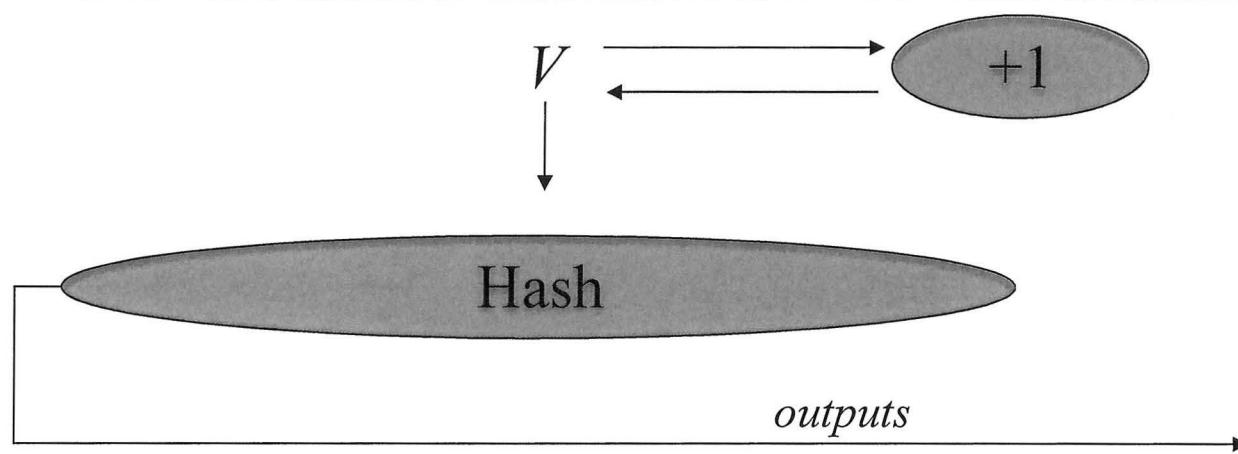
KHF-DRBG: Update Chosen Input Attack

- Suppose attacker chooses *inputString*, doesn't know (K_0, K_1) .
- Attacker knows new value is:
unknown pseudorandom value XOR known/chosen hash_df output
- Even if attacker allowed to choose *hash_df output*, can't mount chosen input attack w/o breaking KHF-DRBG generate.

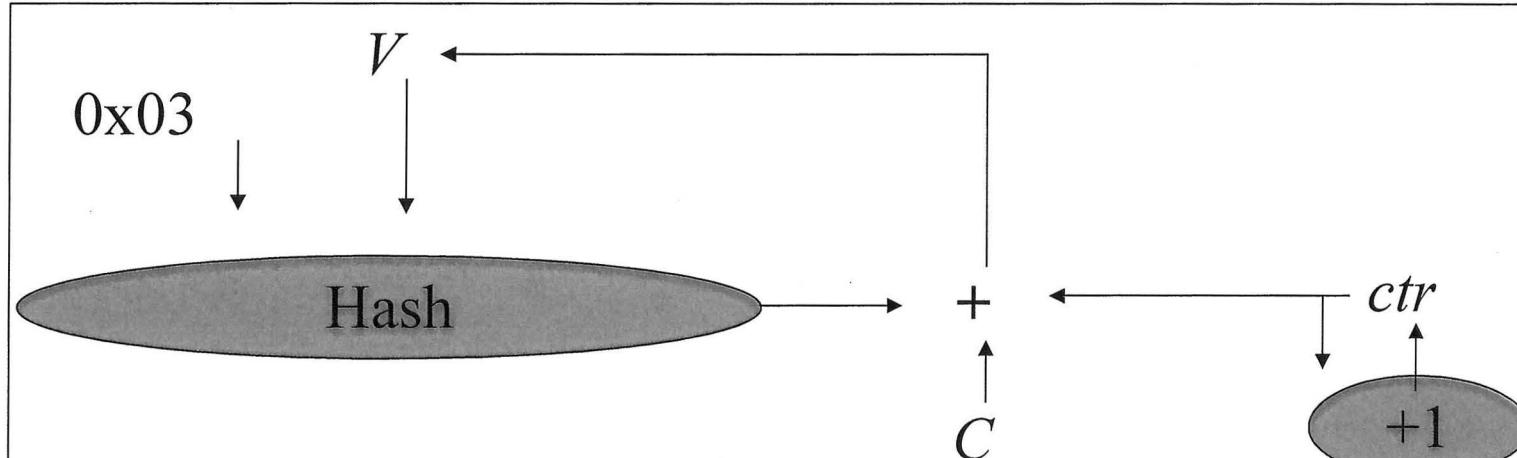
KHF-DRBG: Summary

- Same basic design as HMAC-DRBG: Use PRF in OFB-mode
- Update uses derivation function since KHF not defined on arbitrary-length inputs.
- Performance: *a little better than HMAC-DRBG*
 - Per call overhead (SHA1): 6 compress calls.
 - Per *outlen bit block*: *1 compress call*.
 - *Not parallelizable*
- *Arguably somewhat less robust than HMAC-DRBG (depends on which attacks)*

Hash-DRBG



Hashgen—core of Hash-DRBG



Hash-DRBG: Updating State

Hash-DRBG: History and Overview

- In some sense, derived from
 - FIPS-186 (DSA) PRNG
 - RSAREF/BSAFE PRNG
- Many revisions as requirements changed
- Good performance, but strong assumptions on hash function required

Note: *seedlen* is size of seed, always at least $k + 64$, where k is security level

Hash-DRBG: Security of Generate

- Output generation handled by $\text{Hashgen}(V, n)$:

$tmp = “”$

while $\text{bitLength}(tmp) < n$:

$tmp = tmp \parallel \text{hash}(V)$

$V = V + 1$

return leftmost n bits of tmp

- Security not closely related to hash fn properties
- Attacker sees many successive hash outputs, tries to learn V or *distinguish output sequence from random.*

Hashgen: Black Box Attacks

- Trivial attack (theoretical): If Hashgen visits 2^N states, attacker guesses $2^{seedlen-N}$ states, computes outputs, waits for match.
- Extends to whole Hash-DRBG:
 - Precompute $2^{seedlen-N}$ states and resulting outputs
 - Wait for outputs from 2^N states
 - Match and recover state
- Requires $seedlen \geq k+64$ for $k = security\ level$.

Hashgen and Hash Function Attacks

- Attacker facing hashgen:
 - Knows all but *seedlen* bits of *input* for each *output*
 - Knows relationships between each *input*
- *If compression function is random oracle, this is secure.*
- *No known or suspected weaknesses when used with SHA family of hashes.*

Hash-DRBG: Updating State in Generate

- At end of Generate, low *outlen* bits of V updated
$$V = (V + C + ctr + \text{hash}(0x03 \parallel V)) \bmod 2^{\text{seedlen}}$$
$$ctr = ctr + 1$$
- Backtracking resistance from hashing V
 - Hash with constant to avoid duplicating other hash computations
 - Computing previous V from new V given $C, ctr \Rightarrow$ inverting hash
- C is constant of size *outlen*
- ctr is 32-bit integer

Hash-DRBG: Instantiate and Reseed

- Instantiate and Reseed use `hash_df`:

Instantiate (*seed*):

$$V = \text{hash_df}(\textit{seed})$$

$$C = \text{hash}(0x00 \parallel V)$$

$$ctr = 0$$

Reseed (*seed*):

$$V = \text{hash_df}(0x01 \parallel V \parallel \textit{seed})$$

$$C = \text{hash}(0x00 \parallel V)$$

$$ctr = 0$$

Hash-DRBG Instantiate/Reseed: Recovery From Compromise

- *Does Instantiate get to a secure state? Does Reseed recover from compromise? Recall:*

$$V = \text{hash_df}(\text{seed})$$

or

$$V = \text{hash_df}(0x01 \parallel V \parallel \text{seed})$$

- *Suppose attacker can't guess seed*
 - *If hash_df gives good Hash-DRBG seed when input unguessable, we get secure state*
 - *V should look random w/o knowledge of seed*

Hash-DRBG: Chosen Input Attacks

- Reseed chooses new V as:

$$V = \text{hash_df}(0x01 \| V \| \text{seed})$$

- Generate chooses new V before generation as:

$$V = V + C + \text{ctr} + \text{hash}(0x02 \| V \| \text{inputString})$$

- Suppose attacker doesn't know V , knows seed or inputString

- hash_df has unguessable input string—good seed
 - Even if attacker chose output of hash, couldn't do anything to V
 - *But if can choose inputString to output V ...*

Hash-DRBG: Summary

- Hashgen is the core: runs hash function in counter mode
- Best performance of any hash-based DRBG
 - Per-call overhead: 1 compress call
 - Per *outlen-bit block*: 1 *compress call*
 - *Hashgen is parallelizeable*
- *Security based on more demanding assumptions.*
 - *Attacks on compression function more powerful...*
 - *...but no known attacks exist.*

Hash-Based DRBGs: Wrapup

- Do we need all three?
- Performance issues:
 - Per call overhead important in some applications
 - Per *outlen-bit block* important in others
- Security issues:
 - *HMAC-DRBG and KHF-DRBG expose hash function to fewer possible attacks.*
 - *Hash-DRBG exposes hash to much more powerful attacks, but gives better performance.*

Block Cipher Based DRBGs

AES-OFB

AES-CTR

TDEA-OFB

TDEA-CTR

Block Cipher Based DRBGs: Preliminaries

- Counter and OFB-modes.
- New key generated after each Generate request.
- State is always *keysize + blocksize*.
- *Can use derivation function or conditioned entropy bits.*
- *Choice of approved ciphers:*
 - *Best performance and security from AES.*
 - *Tighter limits on number of outputs for TDEA*

Block Cipher DRBGs: General Security Comments

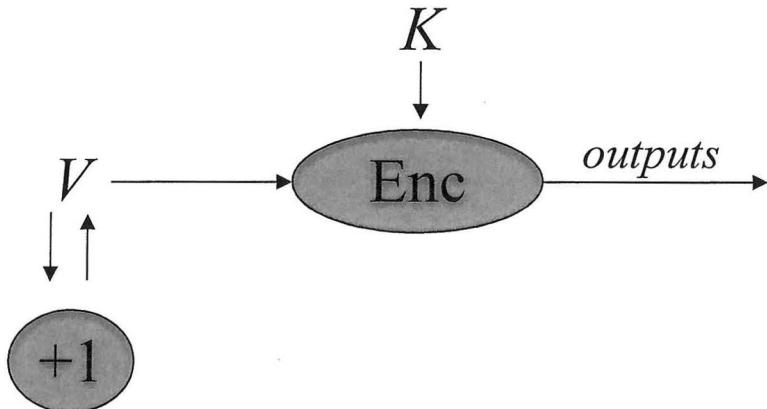
- DRBG security always relates cleanly to block cipher security
- Distinguishing DRBG outputs from random

means

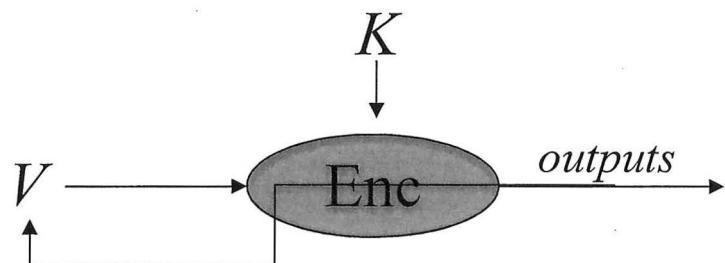
Distinguishing block cipher from random permutation

- Block size is very important, choice of OFB/CTR much less so.

Counter and OFB DRBGs



Counter-mode DRBG



OFB-mode DRBG

Both DRBGs share some properties:

- One encryption per *blocksize* bit output
- Cipher is used only in forward direction
- Rekey after each Generate request
- Simple relation between DRBG security and cipher security

Block Cipher DRBGs: Security of Generate Outputs

- Both DRBGs have straightforward reduction to security of block cipher for one Generate call
- New key generated from same mechanism to satisfy next call
 - If attacker given key, can distinguish from random, can break DRBG
- Permutation/Function difference is relevant
 - TDEA's 64-bit block causes some problems
 - AES' 128-bit block is easier to work with

Distinguishing DRBG Outputs

- Generate output: no blocks repeat
 - Can't happen for CTR
 - Won't happen for OFB (if so, disaster!)
- Ideal random sequence expects some chance of repeats:
 - In 2^{28} 128-bit output blocks, prob. about 2^{-73} .
Given 2^{32} such output sequences, about 2^{-41} .
 - In 2^{13} 64-bit output blocks, prob. about 2^{-39}
In 2^{16} such requests, prob. about 2^{-23} .

But this is less than 2^{64} bound on innocent operations used elsewhere!

Block Cipher DRBGs: Updating State

- New state (K, V) generated as follows:

update (*seed*):

$T = \text{DRBG run to generate } \text{keysize} + \text{blocksize bits}$

$T = T \oplus \text{seed}$

$(K, V) = T$

- Assumes *seed* is *keysize + blocksize bits*
- *When seed comes from freeform input, DRBG uses bc_df to derive random-looking input of right size.*

Block Cipher DRBGs: Backtracking Resistance

- Consider attacker who learns (K, V) , and wants to know previous K .
 - $(K, V) = \text{known value XOR DRBG outputs from old } K$
 - If attacker can recover old K , can break DRBG
- New K, V selected almost at random:
 - Attacker knows no block of K, V can be same as block seen in output sequence
 - *This is never relevant*

Block Cipher DRBGs: Derivation Functions and Conditioned Entropy Sources

- Block cipher DRBGs support two kinds of input:
 - Freeform input—process with block cipher derivation function.
 - Conditioned entropy input—use directly
- Block cipher derivation function is expensive and complicated
 - When gate count or code size is an issue, nice to be able to avoid using it!

Block Cipher DRBGs: Instantiation and Recovery from Compromise

- *Instantiate sets (K, V) to constants and calls Reseed.*
- Suppose attacker knows (K, V) , not *seed input to update function*.
 - $(K, V) = \text{known values XOR seed}$
- Note that seed is either
 - Conditioned entropy source output (random)
 - bc_df output (pseudorandom when input unguessable)
- In either case, attacker knows nothing of (K, V) after update function.

Block Cipher DRBGs: Chosen Input Attacks

- Consider update function (K, V) not known to attacker; input *seed chosen by attacker*.
- *New (K, V) is DRBG output XOR seed*
- *Attacker who can't break DRBG can't even distinguish new (K, V) from random*

Block Cipher DRBGs: Wrapup

- CTR vs OFB: No practical security difference
 - Both included for implementor convenience
 - Likely reuse of code/hardware from other chaining modes or protocols
- AES vs TDEA: Block size is a big deal!
 - TDEA has distinguishers for large output sequences from many different Generate requests
 - Probably not practically relevant
 - AES's larger block size is a win

Symmetric DRBGs Wrapup: How Do I Choose a DRBG?

- Implementation complexity / gate count
 - Reuse existing components
- Performance requirements
 - Overhead per Generate call
 - Work per bit of output
 - Parallelism in Hash_DRBG and CTR_DRBG
- Security assumptions
 - Based on block cipher strength
 - Based on various assumptions on hash function

Symmetric DRBGs Wrapup: Open Issues

- Current designs assume large outputs per Generate request
 - Should we tune these to smaller Generate outputs, larger numbers of Generate calls per reseed?
 - Biggest impact with TDEA-OFB/TDEA-CTR:
 - Limit Generate to 256 output bytes, and we can allow 2^{32} Generate calls!
- Do we always need backtracking resistance?
 - DSA/ECDSA?
- Should we assume *outlen bit security in hash based DRBGs, or outlen/2 bit security?*