Verified Correctness and Security of mbedTLS HMAC–DRBG

Katherine Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew Appel

Princeton / CMU, Johns Hopkins, Princeton / Dropbox, Princeton, Oracle, Princeton
Most modern cryptosystems rely on high-quality randomness.

e.g. RSA generates random big primes that are used to compute a private key.
Pseudorandom number generator

1100101

PRG

0111111011111010010101100110100010010101010010111101110101110001010101000110001110100
Pseudorandom number generator

PRG

1100101

≈

0111111011110110010101100110101100111110111111100110110001100011111101110001

≈

0001011000101011000010110010000000011111111111111111111000111111011101000100011000111111011111101001010110011010001000111101111110101110001010101000110001110100
Pseudorandom number generator

1100101

PRG

01111111011111010010101100110100010001111101111110101110011010101000110001110100

! ≈

0000101110011010110000010110010000011111011111000111110011011101000000010000110111111110100101011001100100100011110111111010111001010101000111001110100
Reducing the entropy of a cryptosystem’s pseudorandom number generator (PRG) is an easy way to break the entire cryptosystem.
Dual-EC-DRBG
Debian OpenSSL PRG

- Removed sources of system entropy → only 32,767 choices (process ID!)
- Predictable SSL/SSH keys (Spotify, Yandex...)
- Can read encrypted traffic, log into remote servers, forge messages

https://www.xkcd.com/424/

We need secure PRGs
But how?

Dilbert By Scott Adams

Tour of Accounting

Over here we have our random number generator.

Nine nine nine nine nine nine.

Are you sure that's random?

That's the problem with randomness: you can never be sure.
Our work

Proved functional correctness and cryptographic security of a widely used implementation of a PRG

- mbedTLS
- HMAC-DRBG

- Verified Software Toolchain
- Foundational Crypto Framework
PRG security property

• Proved that output is indistinguishable from random to a computationally bounded adversary, subject to assumptions

• Typical real/ideal indistinguishability proof in the computational model, using a hybrid argument on number of PRG calls

• Derived a concrete bound on advantage
The Foundational Cryptography Framework, Petcher and Morrisett (POST ‘15)

Program Logics for Certified Compilers, Appel et al. (2014)
Our work

$x \rightarrow y$: $x$ implements $y$

Functional specifications of HMAC-DRBG

transcribe

NIST paper spec of HMAC-DRBG
Our work

\[ x \rightarrow y: \]
\[ x \text{ implements } y \]

Theorems about crypto properties

Functional specifications of HMAC-DRBG

prove with FCF

transcribe

NIST paper spec of HMAC-DRBG
Our work

- Theorems about crypto properties
  - prove with FCF
- Functional specifications of HMAC-DRBG
  - prove with VST

\[ x \rightarrow y: \]
\[ x \text{ implements } y \]

NIST paper spec of HMAC-DRBG

transcribe

mbedtls implementation of HMAC-DRBG
Our work

Theorems about crypto properties

Functional specifications of HMAC-DRBG

NIST paper spec of HMAC-DRBG

mbedTLS implementation of HMAC-DRBG

security!

correctness!

\[ x \rightarrow y : \text{x implements y} \]
Our work

Theorems about crypto properties

security!

Functional specifications of HMAC-DRBG

transcribe

NIST paper spec of HMAC-DRBG

correctness!

mbedTLS HMAC-DRBG

CompCert verified compiler

Verified assembly
Our work

Theorems about crypto properties

Functional specifications of HMAC-DRBG

mbedTLS implementation of HMAC-DRBG

Modular proofs!

NIST paper spec of HMAC-DRBG
Modular proofs

- More theorems
- Theorems about crypto properties
- More theorems

Functional specifications of HMAC-DRBG

- mbedTLS implementation of HMAC-DRBG
- NIST paper spec of HMAC-DRBG
Modular proofs

Functional specifications of HMAC-DRBG

- More theorems
- Theorems about crypto properties
- More theorems

- NIST paper spec of HMAC-DRBG
- mbedTLS implementation of HMAC-DRBG
- Other implementation of HMAC-DRBG
- Other implementation of HMAC-DRBG
Modular proofs

- More theorems
- Theorems about crypto properties
- More theorems

Functional specifications of HMAC-DRBG

- NIST paper spec of HMAC-DRBG
- transcribe

- re-prove

- Other implementation of HMAC-DRBG
- mbedTLS implementation of HMAC-DRBG
- Other implementation of HMAC-DRBG
Modular proofs

- More theorems
- Theorems about crypto properties
- More theorems

Functional specifications of HMAC-DRBG

- NIST paper spec of HMAC-DRBG

- Other implementation of HMAC-DRBG
- mbedTLS implementation of HMAC-DRBG
- Other implementation of HMAC-DRBG
HMAC-DRBG
(keyed-hash message authentication code
deterministic random bit generator)
Generic pseudorandom number generator

Generic pseudorandom number generator

Instantiate
Generate (bits)
Reseed (add entropy)
Update (internal state)

Typical PRG use

User/Adversary:

Instantiate,

Generate 10 blocks,
  Update K and V

Generate 20 blocks,
  Update K and V

Generate 1 block,
  Update K and V,

Generate 10000000 blocks,
  Update K and V,
  RESEED,

Generate 1 block,
  Update K and V,
...

27
k = secret key; v = initialization vector (internal state)

f = hash function (e.g. HMAC)

\[
\text{rand_bits} = f(k, v) || f(k, f(k, v)) || f(k, (f(k, f(k, v)))) \ldots
\]
The outputs are used as inputs

Inner loop of **Generate**

\[ f \quad k \]
Generating bits

$v_{in}$  $n = 3$

$k_{in}$  $f$

Generate  $\text{loop}$  last block

blocks
Generating bits

$v_{in}$  $n = 3$

$k_{in}$  $f$

$k_{out}$  $v_{out}$

blocks

Update

loop
Updating the internal state

\[ \nu_{in} \]

Update

\[ k_{in} \]

\[ f \]

\[ k_{out} \]
Updating the internal state

\[ v_{in} \]

Update

\[ k_{in} \]

\[ f \]

\[ k_{out} \]

\[ v_{out} \]
HMAC–DRBG in use
HMAC-DRBG in use

[3, 1, 2]
Our proof of indistinguishability from random
(HMAC–DRBG security)
Previously, we built a machine-checked proof that HMAC is a PRF, subject to the usual assumptions.

*Verified correctness and security of OpenSSL HMAC, Beringer et al, USENIX Security ’15*
Our work

proof of indistinguishability

Structurally similar to
Security Analysis of DRBG
Using HMAC in NIST SP 800-90,
Hirose (2008)
(though done independently)

x \rightarrow y:
x implements y

Theorems about crypto properties

Functional specifications of HMAC-DRBG

NIST paper spec of HMAC-DRBG

mbedTLS implementation of HMAC-DRBG

transcribe
Combine Generate and Update

[3, 1, 2]
Combine Generate and Update

[3, 1, 2]
Combine Generate and Update

[3, 1, 2]
Real-world and ideal-world hybrids
Real-world and ideal-world hybrids
Real-world and ideal-world games
Real-world and ideal-world games

Prove that it’s hard to tell the difference!
Bridging the real-world and ideal-world definitions with hybrids
Real-world and ideal-world games
Prove that it’s hard to tell the difference!
real-world hybrid
replace PRF with random function

Hybrids
sample all outputs
truly randomly
repeat for each call to the PRG
repeat for each call to the PRG

ideal-world hybrid

Hybrids
Why are the games close?
The hybrid argument
Repeatedly applying the triangle inequality

To distinguishing distributions:

$$\text{Adv}_{D_1,D_2}(A) \leq \text{Adv}_{D_1,H}(A) + \text{Adv}_{H,D_2}(A)$$
For the HMAC-DRBG construction, the main hybrids are:
Distinguishing a pseudorandom function from a random function
the chance of distinguishing between these two hybrids
Recall: HMAC is a pseudorandom function.

What’s a random function again?
Random function as lookup table

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Random function as lookup table

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1" alt="Sample Random Output" /></td>
</tr>
</tbody>
</table>

Sample random output

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image2" alt="Sample Random Output" /></td>
</tr>
</tbody>
</table>
Random function as lookup table

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

look up without resampling
query again

62
Random function as lookup table

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

query again

look up without resampling
Pseudorandom function
(e.g. HMAC)
PRF advantage
Find the fake!
Distinguishing a random function from true randomness
(as used in HMAC-DRBG)
Hybrids

the chance of distinguishing between these two hybrids
Random function vs. true randomness: only noticeable if you call the function on the same input twice

(which we prove is unlikely, as used in HMAC–DRBG)
Random function vs. true randomness: only noticeable if you call the function on the same input twice

(which we prove is unlikely, as used in HMAC–DRBG)
Formalizing the proof of indistinguishability
Method

- Code-based game-playing proofs (Bellare and Rogaway, 2008)
- Programs written in a probabilistic programming language; use Hoare-style logic for relating pairs of programs
- Use “identical until bad” lemma
- Machine-checked proof in FCF and Coq
Proof tree

The real-world and ideal-world games are close
(distinguishable by a PPT A with a particular small probability)

The real-world game is the first hybrid
Move the \( v \)-update in \texttt{Update} to the beginning of the next \texttt{Generate}
Adjacent hybrids \( G_i \) and \( G_{i+1} \) are close
Apply the hybrid argument (induction on \( i \))
The ideal-world game is the last hybrid

Gray denotes program equivalence proofs.
Full proof tree (to give a sense of the structure)

The real-world and ideal-world games are close
(distinguishable by a PPT A with a particular small probability)

The real-world game is the first hybrid
Move the v-update in Update to the beginning of the next Generate
Rewrite G_i to use a PRF oracle on the ith call

Adjacent hybrids G_i and G_{i+1} are close

Apply the hybrid argument (induction on i)

The ideal-world game is the last hybrid
G_i uses a RF oracle on the ith call (denoted G_i').
G_i and G_i' are close.

G_i uses a RB oracle on the ith call (denoted G_i'').
G_i' and G_i'' are close.

Using the RB oracle (G_i'') is equivalent to the next hybrid G_{i+1}.

G_i' and G_i'' are identical until bad.

The bad event is the probability of collisions in the inputs to the oracle, which is small

Use the "fundamental lemma" to bound the difference between G_i' and G_i''.

G_i' and G_i'' are the same if the bad event doesn't happen.

The bad event has the same chance of happening in G_i' and G_i''
Verifying correctness of the mbedTLS C program

For brevity, we discuss HMAC here. See *Verified Correctness and Security of OpenSSL HMAC* (Beringer et al, USENIX Security '15) for details.
Our work

- Theorems about crypto properties
- Functional specifications of HMAC-DRBG
- proof of functional correctness
- mbedTLS implementation of HMAC-DRBG
- NIST paper spec of HMAC-DRBG

\[ x \rightarrow y : x \text{ implements } y \]
Proofs about functions in Coq

Fixpoint map {A B} (f: A->B) (al: list A) :=
  match al with
  | a::r => f a :: map f r
  | nil => nil
end

Fixpoint cat {A} (al bl: list A) :=
  match al with
  | a::r => a :: cat r bl
  | nil => nil
end.

Theorem distr_map_cat:   forall {A B} (f: A->B) (al bl: list A),
  map f (cat al bl) = cat (map f al) (map f bl).


This is a rather trivial theorem. A more interesting one is,
“The HMAC-DRBG algorithm, expressed as a function in Gallina, produces cryptographically strong pseudorandom output.”
Proofs about C programs

```c
struct list catenate (struct list *p, struct list *q) {
    if (p==NULL) return q;
    while (p->tail != NULL) p=p->tail;
    p->tail=q;
    return p;
}
```

```
DECLARE _catenate
WITH p: val, q: val, \sigma_1: list val, \sigma_2: list val
PRE [ _p OF tptr (tstruct _list), _q OF tptr (tstruct _list)]
    PROP() LOCAL (temp _p p; temp _q q) SEP (listrep \sigma_1 p; listrep \sigma_2 q)
POST [ tptr (tstruct _list) ]
    EX v: val, PROP() LOCAL (temp ret_temp v) SEP (listrep (cat \sigma_1 \sigma_2) v).
```

This is a rather trivial theorem. A more interesting one is,
“The mbedTLS implementation of HMAC-DRBG correctly implements the HMAC-DRBG algorithm expressed as a function in Gallina.”
Proofs about C programs

```c
struct list catenate (struct list *p, struct list *q) {
    if (p==NULL) return q;
    while (p->tail != NULL) p=p->tail;
    p->tail=q;
    return p;
}
```

```plaintext
DECLARE _catenate
WITH p: val, q: val,
σ₁: list val, σ₂: list val
PRE [ _p OF tptr (tstruct_list), _q OF tptr (tstruct_list)]
PROP() LOCAL (temp _p p; temp _q q) SEP (listrep σ₁ p; listrep σ₂ q)
POST [ tptr (tstruct_list) ]
EX v: val, PROP() LOCAL (temp ret_temp v) SEP (listrep (cat σ₁ σ₂) v).
```

This is a rather trivial theorem. A more interesting one is,
“The mbedTLS implementation of HMAC-DRBG correctly implements the HMAC-DRBG algorithm expressed as a function in Gallina.”
HMAC function

Definition HmacCore
    IP OP txt (key: list byte): list Z :=
    OUTER OP key (INNER IP key txt).

https://github.com/PrincetonUniversity/VST/blob/8750fdd00c8a3156e5103d2f9924b9de3c6ca7b2/cha/HMAC_functional_prog.v
https://github.com/PrincetonUniversity/VST/blob/8750fdd00c8a3156e5103d2f9924b9de3c6ca7b2/hmacdrbg/HMAC_DRBG_algorithms.v
HMAC-DRBG **Generate** function

Function `HMAC_DRBG_generate_helper_Z`  
(HMAC: list Z -> list Z -> list Z)  
(key v: list Z)(requested_number_of_bytes: Z)  
{measure Z.to_nat requested_number_of_bytes} : (list Z * list Z) :=  
if 0 >=? requested_number_of_bytes then (v, [])  
else  
  let len := 32%nat in  
  let (v, rest) := `HMAC_DRBG_generate_helper_Z HMAC key v`  
    (requested_number_of_bytes - (Z.of_nat len)) in  
  let v := HMAC v key in  
  let temp := v in  
  (v, rest ++ temp).  

https://github.com/PrincetonUniversity/VST/blob/8750fdd00c8a3156e5103d2f9924b9de3c6ca7b2/sha/HMAC_functional_progs.v  
https://github.com/PrincetonUniversity/VST/blob/8750fdd00c8a3156e5103d2f9924b9de3c6ca7b2/hmacdrbg/HMAC_DRBG_algorithms.v
HMAC API in C

unsigned char *HMAC (unsigned char *key, int key_len, unsigned char *d, int n, unsigned char *md);

- Key input
- Message input
- Message-digest output
### API Spec of HMAC

**Declare _HMAC**


**Precondition**

- PROP(writable share shmd;
  - has lengthK (LEN key) (CONT key);
  - has lengthD 512 (LEN msg) (CONT msg))

**Local/global variable bindings**

- LOCAL(temp _md md; temp _key kp; temp _d msgVal;
  - temp _key_len (Vint (Int.repr (LEN key)));
  - temp _n (Vint (Int.repr (LEN msg)));
  - var_K256 (tarray tuint 64) KV)

**Spatial (memory) predicates**

- SEP(data-block Tsh (CONT key) kp;
  - data-block Tsh (CONT msg) msgVal;
  - K-vector KV;
  - memory-block shmd (Int.repr 32) md)

**Postcondition**

- POST [ tvoid ]

**Functional spec**

- SEP(K-vector KV;
  - data-block shmd (HMAC (CONT msg) (CONT key)) md;
  - data-block Tsh (CONT key) kp;
  - data-block Tsh (CONT msg) msgVal)
Our work

First end-to-end formal security-and-correctness verification of a real-world PRG.
Our work

Theorems about crypto properties

Functional specifications of HMAC-DRBG

security!

correctness!

mbedTLS implementation of HMAC-DRBG

transcribe

NIST paper spec of HMAC-DRBG

\[ x \rightarrow y: \]
\[ x \text{ implements } y \]
Modular proofs

More theorems

Theorems about crypto properties

More theorems

Functional specifications of HMAC-DRBG

transcribe

NIST paper spec of HMAC-DRBG

Other implementation of HMAC-DRBG

mbedTLS implementation of HMAC-DRBG

Other implementation of HMAC-DRBG
Open problem

• Security of HMAC-DRBG **Instantiate** relies on HMAC being an entropy extractor

• It is not known whether HMAC is an entropy extractor (the way that HMAC-DRBG uses it)!
Future work

Prove more security properties of PRGs, e.g. backtracking resistance and prediction resistance
Lessons learned

- NIST design decisions: the good (PRF re-key method), the bad (**Instantiate** key with entropy in PRF as input, not key), the ugly (re-key location)
- Verification helps deal with tricky indices and typos in argument
Lessons learned

• Stitch together proofs via machine-checking (see KRACK)

• Formal specifications are useful and necessary!

Key Reinstallation Attacks: Forcing Nonce Reuse in WPA (Vanhoef and Piessens, CCS ’17)
Thanks!

Don’t believe us?
Check out the artifact:

github.com/PrincetonUniversity/VST/tree/master/hmacdrbg
Appendix
HMAC_DRBG Instantiate Process:

1. \( \text{seed} \_\text{material} = \text{entropy}\_\text{input} \| \text{nonce} \| \text{personalization}\_\text{string} \).
2. \( \text{Key} = 0x00\ 00...00 \). Comment: \( \text{outlen} \) bits.
3. \( V = 0x01\ 01...01 \). Comment: \( \text{outlen} \) bits.
   Comment: Update Key and V.
4. \( (\text{Key},\ V) = \text{HMAC}\_\text{DRBG} \_\text{Update} (\text{seed}\_\text{material},\ \text{Key},\ V) \).
5. \( \text{reseed}\_\text{counter} = 1 \).
6. Return \( V,\ \text{Key} \) and \( \text{reseed}\_\text{counter} \) as the initial\_working\_state.

HMAC_DRBG Update Process:

1. \( K = \text{HMAC} (K,\ V \| \ 0x00 \| \text{provided}\_\text{data}) \).
2. \( V = \text{HMAC} (K,\ V) \).
3. If \( \text{provided}\_\text{data} = \text{Null} \), then return \( K \) and \( V \).
4. \( K = \text{HMAC} (K,\ V \| \ 0x01 \| \text{provided}\_\text{data}) \).
5. \( V = \text{HMAC} (K,\ V) \).
6. Return \( K \) and \( V \).
Questions

• What’s the trusted code base?

• What bugs or attacks does your method not prevent? What about side-channels?

• How well does your method scale to larger codebases?

• Is your proof still valid if the underlying mbedTLS code changes?

• Can I apply your method to verify other faster or better DRBGs, like AES-DRBG?

• Would your method have prevented real-world incidents like the Debian OpenSSL fiasco or the Juniper bugs?
Questions

• How would you prove other security properties of DRBGs, like backtracking resistance and prediction resistance?

• How does your proof link with other proofs that might involve HMAC-DRBG, like proving security of TLS?

• What does the bound on your proof mean? Concretely, how long would it take an adversary to break HMAC-DRBG indistinguishability by brute force?

• So your proof means everyone should be using mbedTLS HMAC-DRBG, right?
Questions

• Did your proof involve any new math? How does it differ from Hirose’s proof?

• Does it matter that you assume a nonadaptive adversary?

• How does assuming an ideal Instantiate (without entropy) weaken your proof? What about the additional input?

• Why use the logics of PRHL and Verifiable C?