Verified Correctness and Security of mbedTLS HMAC-DRBG

Katherine Q. Ye
Princeton U., Carnegie Mellon U.

Matthew Green
Johns Hopkins University

Naphat Sanguansin
Princeton University

Lennart Beringer
Princeton University

Adam Petcher
Oracle

Andrew W. Appel
Princeton University

ABSTRACT

We have formalized the functional specification of HMAC-DRBG (NIST 800-90A), and we have proved its cryptographic security—that its output is pseudorandom—using a hybrid game-based proof. We have also proved that the mbedTLS implementation (C program) correctly implements this functional specification. That proof composes with an existing C compiler correctness proof to guarantee, end-to-end, that the machine language program gives strong pseudorandomness. All proofs (hybrid games, C program verification, compiler, and their composition) are machine-checked in the Coq proof assistant. Our proofs are modular: the hybrid game proof holds on any implementation of HMAC-DRBG that satisfies our functional specification. Therefore, our functional specification can serve as a high-assurance reference.

1 INTRODUCTION

Cryptographic systems require large amounts of randomness to generate keys, nonces and initialization vectors. Because many computers lack the large amount of high-quality physical randomness needed to generate these values, most cryptographic devices rely on pseudorandom generators (also known as *deterministic random bit generators* or DRBGs) to "stretch" small amounts of true randomness into large amounts of pseudorandom output.[1]

Pseudorandom generators are crucial to security. Compromising a generator (for example, by selecting malicious algorithm constants, or exfiltrating generator state) can harm the security of nearly any cryptosystem built on top of it. The harm can be catastrophic: for example, an adversary who can predict future outputs of the generator may be able to predict private keys that will be generated, or recover long term keys used as input to the protocol execution.

Moreover, it may be impossible to test that a DRBG has been compromised if one is limited to black-box testing. Current validation standards [26, 30, 32] primarily resort to statistical tests and test vectors, neither of which guarantee that the output is pseudorandom. One can even construct backdoored PRGs that cannot be detected by black-box testing [18].

Despite the importance of DRBGs, their development has not received the scrutiny it deserves. Many catastrophic flaws in DRBGs have been uncovered at both the design level and the implementation level. Some bugs arise from simple programming mistakes. The Debian DRBG, though open-source, was broken for two years because a line of code was erroneously removed, weakening the DRBG's seeding process [29]. Several recent projects have identified and factored large numbers of weak RSA keys produced due to improper generator seeding [12, 23, 28]. A bug in entropy use in the Android DRBG resulted in the theft of $5,700 of Bitcoin [22].

While those flaws were accidental, some are malicious. Adversaries intentionally target DRBGs because breaking a DRBG is an easy way to break the larger cryptosystem. The most notorious example is the NSA's alleged backdooring of the Dual EC DRBG standard [6, 13, 33, 36]. In the Dual EC design, the malicious choice of a single algorithm parameter (an elliptic curve point Q) allows for full state recovery of the DRBG given only a small amount of raw generator output. This enables the passive decryption of protocols such as TLS and IPSEC/IKE [15, 16, 36]. From 2012 to 2015 this backdoor was exploited by unauthorized parties to insert a passive decryption backdoor into Juniper NetScreen VPN devices [15, 39]. Remarkably, a related DRBG flaw in Fortinet VPNs created a similar vulnerability in those devices during the same time period [25].

A key weakness in the deployment of DRBGs is that current government standards both encourage specific designs and lack rigor. The FIPS validation process (required by the U.S. government for certain types of cryptographic device) mandates the use of an "approved" NIST DRBG in every cryptographic module. As a consequence, a small number of DRBG designs have become ubiquitous throughout the industry. These algorithms lack formal security proofs, and their design processes were neither open nor rigorous—in fact, the designs were found to include errors. Even worse, it is easy to implement certain of these DRBGs such that the output of the generator is predictable (given detailed knowledge of the implementation), and yet without this knowledge *the output of the generator is computationally indistinguishable from random* [18]. Unfortunately, the existing formal validation processes for verifying the correctness of DRBG implementations are weak and routinely ignore entire classes of flaws. Given that the widespread deployment of a weak DRBG can undermine the security of entire computer networks, we need a better way to validate these critical systems.

[1] A note on terminology: we use "entropy" loosely to denote randomness that is not predictable by an adversary. We use "sampled uniformly at random" and "ideally random" interchangeably. We use PRG, the acronym for "pseudo-random generator," to refer to the abstract cryptographic concept, whereas we use DRBG, the acronym for "deterministic random bit generator," to denote the specifications and implementations of PRGs. Instead of DRBG, some papers use "PRNG," the acronym for "pseudo-random number generator." The terms are synonymous.

CCS '17, Dallas, TX, USA

DOI: 10.1145/3133956.3133974

1.1 Contributions

DRBGs have the special property that testing, even sophisticated statistical fuzz-testing, cannot assure the security of an implementation. Even at the level of a pure cryptographic protocol, proving the security of a DRBG construction can be quite challenging. (Hybrid-game-based proofs are the only known technique, but for nontrivial DRBGs those proofs have so many technical steps that it is hard to trust them.) Therefore a new paradigm is required, in which the functional model of a DRBG is proved *with a machine-checked proof* to satisfy the appropriate PRF properties, and the C language implementation (*and* its compilation to machine language) is proved *with a machine-checked proof* to correctly implement the functional model, and these proofs are linked together in the same proof-checker.

We present machine-checked proofs, in Coq, of many components, connected and checked at their specification interfaces so that we get a truly end-to-end result: *Verson 2.1.1 of the mbedTLS HMAC-DRBG correctly implements the NIST 800-90A standard, and HMAC-DRBG* ***Generate*** *and* ***Update*** *as described in that same NIST 800-90A standard indeed produces pseudorandom output, subject to the standard assumptions[2] about SHA-2, as well as certain assumptions about the adversary and the HMAC-DRBG instantiation that we state formally and explicitly. We have not proved the security of* ***Instantiate*** *(see §7) and* ***Reseed****.*

To prove this theorem, we took the following steps.

(1) We constructed a proof that the version of HMAC-DRBG described in the standard NIST SP 800-90A generates pseudorandom output, assuming that HMAC is a pseudorandom function and that the DRBG's state is instantiated with ideal randomness. Our proof is similar to that of Hirose [24], but more clearly and explicitly structured. By "pseudorandom," we mean that a nonadaptive[3] probabilistic polynomial-time adversary that requests the maximum amount of output from HMAC-DRBG cannot distinguish its output from uniformly random output with nonnegligible probability.
(2) We mechanized the proof in (1), formalizing the cryptographic algorithm *("crypto spec")* and the security theorem in Coq, instantiating HMAC as HMAC/SHA-256, and assuming that the DRBG's initial state is ideally random. Our crypto spec was written with respect to the NIST standard SP 800-90A [6] which defines HMAC-DRBG. It was written in the probabilistic programming language provided by the Foundational Cryptography Framework [35], which is itself embedded in the Coq proof assistant.
(3) We proved that the mbedTLS implementation of HMAC-DRBG (written in C) correctly implements a functional specification, also derived from SP 800-90A, written as a pure function in Coq. We performed the verification using the Verified Software Toolchain (VST) framework [5], which is also embedded in the Coq proof assistant.
(4) We proved that our two functional specs for HMAC-DRBG (both derived from SP 800-90A) are equivalent, by induction in Coq. This connects the C-program correctness proof to the cryptographic pseudorandomness proof.
(5) Beringer *et al.* [11] and Appel [4] proved that an OpenSSL implementation of HMAC with SHA-256 implements the FIPS-198 and FIPS-180 functional specs (respectively), and those in turn implement a PRF with bounded attacker advantage, subject to some standard (unproven but uncontroversial) assumptions about the security of SHA-256's compression function.

Composing these proofs (which we do in the Coq proof assistant without any gaps at the interfaces) yields our main result. This modular approach allows one to incrementally verify even complex cryptographic protocol implementations in real languages.

The trusted base. Appel [4, §12] discusses at length the trusted code base of this approach. In summary: the mbedTLS DRBG program need not be trusted, as it is proved to implement the spec; the compiler need not be trusted, as it is proved correct; the NIST spec (or our formalization of it) need not be trusted, because it is proved to be a PRF. FCF need not be trusted, because it is proved sound in Coq (only the *definitions* used in the statement of the PRF theorem need be trusted, because if you prove the wrong theorem it's not so useful). VST need not be trusted, because it is proved sound w.r.t. the operational semantics of CompCert C. The CompCert verified optimizing C compiler need not be trusted, because it is proved correct w.r.t. that same operational semantics and the op. sem. of the assembly language. The op. sem. specification of CompCert's source language (C) need not be trusted, because it's just an internal interface of the proof (between VST and CompCert). What must be trusted are: the specification of the assembly language ISA (as an operational-semantic statement in Coq); the statement of the main theorem and the definitions it relies on (one must make sure that the right property is proved); and the Coq kernel (proof checker), along with the OCaml compiler that compiled Coq, and the OCaml compiler that compiled the OCaml compiler (it's turtles all the way down [38]). One could also compile mbedTLS with gcc or clang, and in that case the compiler would be part of the TCB.

Our DRBG proof is for mbedTLS, and our HMAC proof is for OpenSSL, so technically our end-to-end proof is for an implementation with those two components stapled together.

We prove "only" functional correctness, which has useful corollaries of safety and direct information flow: no buffer overruns, no reading from or writing to unspecified locations. We do not address side channels—which are not a functional-correctness property—but because we verify standard C-language implementations, any other analysis tools for side channels in C programs can be applied. Closer to machine code, our approach is compatible with Barthe *et al.*'s methodology of combining functional and implementational correctness proofs with formal proofs of leakage security, based on a noninterference analysis for CompCert's low-level intermediate language Mach [7].

Coq proofs. For the functional specification, cryptographic specification, proof of functional correctness, and security proof, see *github.com/PrincetonUniversity/VST/tree/master/hmacdrbg*. The

[2] The standard PRF assumptions about the security of SHA-2 are stated by Beringer *et al.*[11, §4.2]. No one knows how to prove these, but it is not controversial to assume them. See also §5 of our paper.

[3] In many typical applications of DRBGs, such as TLS, adversaries are nonadaptive because they are not given control over parameters such as the number of requests to the DRBG or the block length.

README file in that directory explains how the Coq files correspond to sections of this paper.

1.2 Prior work

Hirose [24] proves the pseudorandomness of the full HMAC-DRBG on paper, also with respect to a nonadaptive adversary. However, the proof is not linked to any implementation of HMAC-DRBG. Moreover, though it is well-written, the six-page Hirose proof is challenging to read and verify. For example, Hirose justifies his arguments informally; several times, he states that an important implication is "easy to see" or that it "directly follows from the preceding lemmas." Hirose also asserts that HMAC-DRBG is secure when the user provides an optional additional input on initialization; however, he only states the relevant lemmas without proving them. We do not prove any properties relating to additional input.

To perform proofs more readably and rigorously on complex constructions such as DRBG, we harness the power of mechanization. We mechanize our proof in the Coq proof assistant, using the embedded Foundational Cryptography Framework (FCF). In contrast to working on paper, mechanizing a proof requires that all lemmas and relationships between lemmas be precisely stated. Additionally, FCF encourages proofs to be done in the game-playing style, which makes the intermediate definitions more explicit (because they must all be written as code) and the proof as a whole easier to read and verify. See §4.4 for an outline of the structure of our proof.

More specifically, the Coq proof assistant assists the user by providing an interactive development environment and user-programmable automation to construct a formal syntactic proof. Then, the Coq kernel checks that this term proves the stated theorem. Coq has been used to prove mathematical theorems, cryptographic theorems, C program correctness, and C compiler correctness. The latter three are relevant to our work.

Affeldt et al. [1] present an end-to-end machine-checked (in Coq) proof of the Blum-Blum-Shub PRG, proving cryptographic security of the algorithm and correctness of the implementation. They mechanize an existing, well-known proof of BBS's security. Their cryptographic games are linked directly to their own assembly code, whereas our games apply to a high-level specification of the DRBG in question, and connect to a widely used C program. BBS is not used in practice because it is quite slow [37].

Appel [4] proved in Coq that OpenSSL SHA-256 correctly implements the FIPS-180 specification. Beringer *et al.* [11] proved in Coq the PRF security of the FIPS-198 specification of HMAC/SHA-256, and that the OpenSSL implementation of HMAC correctly implements FIPS-198.

Petcher [34] applies mechanization to demonstrate a *negative* example, rather than a positive one. He demonstrates how the process of mechanizing the proof of security of Dual-EC-DRBG naturally results in an unprovable theorem, which reveals the weakness in the DRBG's design. Our mechanized proof of HMAC-DRBG shows which parts have no flaws (*Generate* and *Update*), and which are problematic (*Instantiate*).

Inspired by the Android bug described earlier, the Entroscope tool [20] uses information flow and static analysis (with bounded model checking) to detect entropy loss in the seeding of a PRNG. Entroscope has been successfully applied to find a bug in the Libgcrypt PRNG, and would have detected the Debian incident [29].

Higher-level protocols such as TLS have been proved secure with machine-checked proofs [14], although without proving security of primitives such as DRBGs or HMAC. Our work complements such proofs.

No prior work has proved full cryptographic security and correctness of a DRBG that links all of its components: a specification on paper, a proof of security on paper, a widely used implementation, and its correct compilation to assembly code. The DRBG flaws listed earlier demonstrate that a DRBG may contain flaws anywhere (and everywhere!) in that stack. Thus, it is imperative to verify DRBGs, and cryptographic primitives in general, in a manner that links all of their components.

2 DETERMINISTIC RANDOM BIT GENERATORS

In 2007 the U.S. National Institute of Standards and Technology (NIST) released Special Publication 800-90 describing several pseudorandom generator constructions, replaced in 2012 by 800-90A [6]. Termed *deterministic random bit generators* in NIST's parlance, these generators have been widely adopted in many commercial products, including many operating systems and TLS libraries. This adoption is largely because SP800-90A generators are required in modules validated by the U.S. government's Cryptographic Module Validation Program [32].

The NIST specification describes a specific algorithmic interface as well as an informal list of security goals. We present this below.

Definition 2.1. A DRBG is a tuple of algorithms (**Instantiate**, **Update**, **Reseed**, **Generate**) with the following interface.

Instantiate(*entropy*, *nonce*). On input of an initial entropy string and a personalization nonce, outputs the initial state $\langle k, v\rangle$.

Update(*data*, $\langle k, v\rangle$). On input of (optional) *data* string (for seeding) and a generator state $\langle k, v\rangle$, outputs a new state $\langle k', v'\rangle$.

Reseed($\langle k, v\rangle$, *entropy*). On input of the current generator state and an entropy string, output a new state $\langle k', v'\rangle$.

Generate($\langle k, v\rangle$, n). On input of the current generator state and a number of blocks n, output a string of pseudorandom output as well as a new state $\langle k, v\rangle$.

The NIST standard informally describes several security properties that a DRBG must possess. These include the standard notion of pseudorandomness, as well as several auxiliary properties. In this work we focus primarily on the pseudorandomness property.

Definition 2.2 (Pseudorandomness of the DRBG). Let Π = (**Instantiate**, **Update**, **Reseed**, **Generate**) be a DRBG, let c denote the length of the internal state vector, and let the random variable $\mathsf{PR}_b(\Pi, A, \lambda)$ where $b \in \{0,1\}$, and $\lambda \in \mathbb{N}$ denote the result of the following probabilistic experiment:

$$\begin{aligned}\mathsf{PR}_b(\Pi, A, \lambda) \quad := \quad & E \leftarrow \{0,1\}^\lambda \\ & (k, v) \leftarrow \textbf{Instantiate}(E, \perp) \\ & B \leftarrow A^{O_{\mathsf{Generate}}(b,\cdot)}(\lambda) \\ & \text{Output } B\end{aligned}$$

where $O_{\mathsf{Generate}}(b, n)$ defines an oracle that is initialized with initial state (k, v) and if $b = 0$ runs the **Generate** algorithm using the

internal state and then updates the internal state. The security parameter λ is the number of entropy bits seeding the DRBG. If $b = 1$, the oracle samples a uniformly random string of length nc (where c is the block length) and returns this to A. A DRBG scheme Π is pseudorandom if $\forall$ p.p.t. algorithms A the following two ensembles are computationally indistinguishable:

$$\left\{\mathrm{PR}_0(\Pi, A, \lambda)\right\}_\lambda \overset{c}{\approx} \left\{\mathrm{PR}_1(\Pi, A, \lambda)\right\}_\lambda$$

In this work we will focus primarily on the **Generate** and **Update** algorithms, while leaving analysis of the **Instantiate** algorithm for future analysis (we discuss the reasons in Section 7). Thus in the remainder of this work we will employ the simplifying assumption that the initial state (k, v) is uniformly random.

2.1 HMAC-DRBG

HMAC-DRBG is one of the four pseudorandom generator constructions formalized in NIST SP 800-90A. It works by iterating HMAC, a keyed-hash message authentication function that can be instantiated using a cryptographic hash function such as SHA. HMAC is useful in constructing a PRG. When instantiated with an efficient cryptographic hash function it is highly efficient. Bellare [9, §5] shows that HMAC is a *pseudorandom function* if certain assumptions hold on the hash functions's internal compression function.

HMAC-DRBG has an internal state consisting of two pieces of administrative information, which are constant across calls, and two pieces of secret information, which are updated during each call to the PRG. The administrative component of the state contains the security strength of the PRG's instantiation, for which the *Instantiate* function obtains the appropriate amount of entropy for that security strength, and a flag that indicates whether this instantiation requires prediction resistance.

The secret component of the internal state is the working state (k, v), where k is a secret key of length c for the internal PRF (e.g., HMAC). Each time the PRG generates a string of pseudorandom bits (by calling the **Generate** function), k is updated at least once. The v component holds the newest "block" of pseudorandom bits that the PRG has generated. The length of v is the length of HMAC's output, which is fixed (e.g., 256 bits).

Here we will rephrase the NIST standard in a mostly functional style that closely mirrors our specification in Coq. Let f denote HMAC and f_k denote HMAC partially applied with the key k. Let $||$ denote concatenation.

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

Instantiate initializes (k, v) and *reseed_counter*. For simplicity, we omit the *additional_input* and *personalization_string* parameters from *Instantiate* the "crypto spec" that we present here; these optional parameters allow for the provision of additional entropy to the instantiate call. Our proof of the mbedTLS C code (see §6) does handle these parameters. The omitted parameters are generally used for fork-safety, so therefore we have not proved fork-safety.

Update$(data, k, v)$:=

$k_1 \leftarrow f_k(v||0x00||data)$
$v_1 \leftarrow f_{k_1}(v)$
if ($data$ = nil)
then return (k_1, v_1)
else $k_2 \leftarrow f_{k_1}(v_1||0x01||data)$
$v_2 \leftarrow f_{k_2}(v_1)$
return (k_2, v_2)

Instantiate$(entropy, nonce)$:=

$seed \leftarrow entropy||nonce$
$k \leftarrow 0x00\ldots00$
$v \leftarrow 0x01\ldots01$
$(k_1, v_1) \leftarrow$ **Update**$(seed, k, v)$
$reseed\_counter \leftarrow 1$
return $(k_1, v_1, reseed\_counter)$

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

Generate(k, v, n) :=

(* We don't model the reseed test.
if $reseed\_counter > reseed\_interval$ then …reseed …
else *)
$temp_0 \leftarrow nil$
$v_0 \leftarrow v$
$i \leftarrow 0$
while length$(temp_i) < n$
do $v_{i+1} \leftarrow f_k(v_i)$
$temp_{i+1} \leftarrow temp_i||v_{i+1}$
$i \leftarrow i + 1$
$returned\_bits \leftarrow$ leftmost n of $temp_i$
$(k_1, v_{i+1}) \leftarrow$ **Update**(k, v_i)
$reseed\_counter_1 \leftarrow reseed\_counter + 1$
return $(bits, k_1, v_{i+1}, reseed\_counter_1)$

Reseed$(k, v, entropy)$:=

$(k_1, v_1) \leftarrow Update(seed, k, v)$
$reseed\_counter \leftarrow 1$
return $(k, v, reseed\_counter)$

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

3 OVERVIEW OF THE PROOF

Here we explain the techniques used in our proof. Section 4 presents the mechanized proof in detail.

3.1 Key result: HMAC-DRBG is pseudorandom

We prove the pseudorandomness property of HMAC-DRBG. As with traditional proofs, we show the computational indistinguishability of two main "experiments." In the first (or "real") experiment, which roughly corresponds to the experiment PR_0 in §2, the adversary interacts with an oracle whose generator uses HMAC-DRBG. In the second (or "ideal") experiment, the adversary interacts with an oracle that produces uniformly random output. This corresponds to PR_1. Our goal is to connect these experiments via a series of intermediate "hybrid" experiments, which we will use to show that the real and ideal experiments are computationally indistinguishable by any probabilistic polynomial time (p.p.t.) adversary.

The real experiment (G_{real}). In this experiment, a p.p.t. adversary A makes a sequence of calls to an oracle that implements the **Generate** function of DRBG. At the start of the experiment, the generator is seeded with a uniformly random generator state $\langle k, v\rangle$, and at each call the adversary requests a fixed number n of blocks of generator output, where n is chosen in advance. The oracle returns the requested output, then updates its internal state as required by the **Generate** algorithm. At the conclusion of the experiment, the adversary produces an output bit b.

The ideal experiment (G_{ideal}). The adversary A interacts with an oracle according to the same interface as in the previous experiment. However, this oracle simply returns a uniformly random string of length n blocks. The oracle maintains no internal state. At the conclusion of the experiment, the adversary produces an output bit b.

Stated informally, the probability that A outputs 1 when interacting with G_{real}, minus the probability that the adversary outputs 1 when interacting with G_{ideal}, is a measure of how successful A is in distinguishing the pseudorandom generator from a random generator. Our goal is to bound this quantity to be at most a negligible function of the generator's security parameter (see §5).

Our proof connects the *real* and *ideal* experiments by defining a series of intermediate, or hybrid, experiments. These hybrid experiments comprise a transition between the *real* and *ideal* experiments. We show that for each successive pair of hybrids—beginning with the real experiment and ending with the ideal experiment—the distribution of the adversary's output must be at most negligibly different between the two experiments in the pair. Having performed this task for each pair of experiments leading from *real* to *ideal*, we then apply a standard hybrid argument to prove that if all intermediate hybrids satisfy our condition, then the real and ideal experiments themselves must also be computationally indistinguishable. It remains now to outline our hybrids.

Overview of the hybrids. Recall that the HMAC-DRBG generation procedure consists of an iterated construction with two main stages. Given a uniformly random internal state (k, v_0) the first phase of generation operates as follows: a call to the **Generate** function produces a sequence of output blocks which are calculated by applying the HMAC function $f_k(v_0)$ to generate a block of output v_1, and then applying $f_k(v_1)$ to obtain v_2 and so on until n blocks have been generated. The second phase of **Generate** updates the internal state of the generator to produce a new state (k', v'_0) which will be used in the subsequent call to **Generate**.

Our proof defines a set of hybrids that shows that *at each call to **Generate***, given integer n (whose size is polynomial in λ, the number of entropy bits) and a uniformly random initial state (k, v_0), the output of **Generate**(k, v, n) is indistinguishable from a random string. Using these hybrids, we show that the PRG, over all calls to **Generate**, is secure for any fixed number of calls *numCalls* made by the adversary.

We define the following sequence of hybrids:

G_{real}. This is the real experiment in which all calls are answered using the HMAC-DRBG **Generate** algorithm.

$G_{1\ldots numCalls}$. For each hybrid indexed by $i \in [1, \ldots, numCalls]$, we modify the experiment such that the first i calls are replaced with a function that simply returns ideally random bits and a new uniformly random state (k', v'_0) when queried on any input. Internally this is achieved using two additional hybrids:

(1) **Internal hybrid 1.** In this hybrid, the first $i-1$ oracle calls produce random output. In call i we modify the output to sample a random function r and use r in place of the PRF f to compute the **Generate** algorithm.

(2) **Internal hybrid 2.** Identical to the previous hybrid, except that we substitute each output of the random function with a string of uniformly random bits.

G_{ideal}. This is the ideal experiment.

In the first hybrid, we replace the HMAC function f with a random function r. We are able to show that this replacement does not affect A's output if f is pseudorandom, as this would imply the existence of a distinguisher for the pseudorandom function, a fact which would directly contradict our assumption. With this replacement complete, hybrid (2) requires us to make specific arguments about the input to the random function r, namely that the inputs to the random function do not collide. Thus, the core of our proof is a sequence of theorems arguing that, from the perspective of A, the changes we make at each of the intermediate hybrids described above induces only a negligible difference in A's output. We conclude then by making a standard hybrid argument that bounds the overall distance of G_{real} and G_{ideal} to be negligible in λ. We calculate a concrete bound on A's ability to distinguish between G_{real} and G_{real}.

4 MECHANIZED SECURITY PROOF

In this section, we describe our machine-checked proof of HMAC-DRBG's pseudorandomness. All definitions and theorem statements are taken straight from the Coq proofs and may be looked up by name in the file. See the end of §1.1 for the link to our repository.

4.1 Foundational Cryptography Framework

Our proof uses the Foundational Cryptography Framework [35], a Coq library for reasoning about the security of cryptographic schemes. FCF provides a probabilistic programming language for

describing cryptographic constructions, security definitions, and assumed-hard problems. Probabilistic programs are described using Gallina, the purely functional programming language of Coq, extended with a computational monad that adds uniform sampling on bit vectors. In this language, `{0,1}^n` describes uniform sampling on bit vectors of length n, and monadic arrows (e.g. `<-$`) sequence the operations. `Comp A` is the type of probabilistic computations that return values of type `A`, and a denotational semantics relates every `Comp A` with a probability distribution on `A`.

FCF provides a theory of probability distributions, a program logic, and a library of reusable arguments and tactics that can be used to complete proofs. All of these items are based on theory derived from the semantics of probabilistic programs, and the result is a versatile proof framework with a small foundation. For example, FCF includes a reusable hybrid argument on lists that can be used in proofs like one described in this paper. FCF also provides a standard type and associated theory for an adversary that is allowed to interact with an oracle (called `OracleComp`).

4.2 Cryptographic specification

At a high level, any DRBG in the NIST specification takes as input an initial state (k, v) and a list of natural numbers, where each number is the number of blocks (bitvectors) requested for that call, e.g. $[1, 3, 2]$. It returns a list of lists of bitvectors with the correct number of generated blocks in each list, e.g. queried with $[1, 3, 2]$, our model of the DRBG would return $[[b_1], [b_2, b_3, b_4], [b_5, b_6]]$, where each b_i's size is the output size of the PRF used in the DRBG. Because we model a nonadaptive adversary, there is only this one round of query-response.

DRBG algorithm definitions. We begin by defining the abstract PRG used in our mechanized proof of pseudorandomness. Let f be an arbitrary PRF that takes a key bitvector, an input list, and returns an output bitvector. η is the output size of the PRF, *Bvector* η is the type of bitvectors of size η (also called "blocks"), and *KV* is a type synonym for the PRG's internal state, which is a tuple consisting of two blocks (k, v).

```
Variable η : nat.
Hypothesis nonzero_eta : η > 0.
Variable f : Bvector η -> list bool -> Bvector η.
Definition KV : Set := Bvector η * Bvector η.
```

The function *Instantiate* instantiates the PRG's internal state (k, v) with ideal randomness. This idealized **Instantiate** function does not reflect the NIST specification; we discuss this in Section 7.

```
Definition Instantiate : Comp KV :=
  k <-$ {0,1}^η;
  v <-$ {0,1}^η;
  ret (k, v).
```

The function *Gen_loop* corresponds to the main loop of HMAC-DRBG's *Generate* function. Given an internal state (k, v) and a number of requested blocks n, it returns a list of generated blocks, along with the last generated block, which becomes the new v in the internal state.

```
Fixpoint Gen_loop (k : Bvector η) (v : Bvector η)
          (n : nat) : list (Bvector η) * Bvector η :=
  match n with
  | O => (nil, v)
  | S n' => let v' := f k (to_list v) in
            let (bits, v'') := Gen_loop k v' n' in
            (v' :: bits, v'')
  end.
```

NIST specifies that at the end of a call to *Generate*, *Update* is immediately called to refresh the internal state. In our Coq definition, we have inlined *Update* into *Generate*:

```
Definition Generate (state : KV) (n : nat) :
                  Comp (list (Bvector η) * KV) :=
  [k, v] <-2 state;
  [bits, v'] <-2 Gen_loop k v n;
  k' <- f k (to_list v' ++ zeroes);
  v'' <- f k' (to_list v');
  ret (bits, (k', v'')).
```

The NIST spec allows n to be any number of bits (up to a specified maximum), by discarding extra generated bits up to a multiple of the HMAC block size. Our security theorem measures n in blocks, not bits; it would be straightforward to extend the theorem to account for discarding surplus bits.
Generate and *Instantiate* comprise our definition of the PRG.

The adversary A is given the output of the PRG, which is a list of lists of blocks. It may perform a probabilistic computation on that input, and it returns a boolean guess attempting to distinguish the PRG's output from ideally random output. A is nonadaptive; it is given the output all at once and cannot further query the PRG.

```
Variable A : list (list (Bvector η)) -> Comp bool.
```

Assumptions. We assume that there is a nonzero number of calls *numCalls* to *Generate*. We assume that on each call, the adversary requests the same nonzero number of blocks *blocksPerCall*. This request list is called *requestList*.

```
Variable blocksPerCall : nat.
Variable numCalls : nat.
Definition requestList : list nat :=
              replicate numCalls blocksPerCall.
Hypothesis H_numCalls : numCalls > 0.
Hypothesis H_blocksPerCall : blocksPerCall > 0.
```

Using the definitions of the PRG's functions and of the adversary, we define the real-world game G_{real} that simulates use of the PRG. First, the internal state of the PRG is instantiated. Then the *Generate* function is iterated over the request list using the function *oracleMap*, which tracks the PRG's internal state (k, v) and updates it after each iteration. The resulting list of lists of blocks is passed to the adversary, which returns a boolean guess.

```
Definition G_real : Comp bool :=
  [k, v] <-$2 Instantiate;
  [bits, _] <-$2 oracleMap Generate (k, v) requestList;
  A bits.
```

The ideal-world game G_{ideal} simply generates a list of lists of ideally random blocks, where each sublist is the same length as

the corresponding one in the output generated by the PRG. Here, compMap maps a probabilistic function over a list.

```
Definition G_ideal : Comp bool :=
  bits <-$ compMap Generate_rb requestList;
  A bits.
```

4.3 Formal statement of theorem

Our main theorem states that the probability that an adversary can distinguish between the real-world game G_{real} and the ideal-world game G_{ideal} is bounded by a negligible quantity.

```
Theorem G_real_ideal_close :
  | Pr[G_real] - Pr[G_ideal] | ≤
      numCalls * (PRF_Advantage_Max + Pr_collisions).
```

This quantity is the number of calls *numCalls* to the PRG multiplied by a constant upper bound on the difference between adjacent hybrids, which is the sum of (1) an upper bound on the PRF advantage for HMAC, and (2) an upper bound on the probability of collisions in a list of length *blocksPerCall* of *Bvector* η that are each sampled uniformly at random.

For quantity (1), the PRF advantage of a function f is informally defined to be the maximum probability that any p.p.t. adversary can distinguish f from a PRF. Since a PRF's advantage is defined over all p.p.t. PRF adversaries, we construct a PRF adversary that takes an oracle as input (denoted by *OracleComp* in its type), runs the PRG using that oracle, and returns the PRG adversary A's guess. Technically, *PRF_Adversary* defines a family of adversaries parametrized by i, the index of the hybrid. There is actually a different PRF advantage for each hybrid i.

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

The rationale behind this definition of *PRF_Adversary* will be given in the discussion of lemma *Gi_prf_rf_close* in Section 4.4.

Now we can use the definition of *PRF_Adversary*, along with the PRF f, to define *PRF_Advantage_Game i*, which is the PRF advantage for hybrid i (again discussed further in *Gi_prf_rf_close*).

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

Finally, we define the upper bound on f's PRF advantage to be the maximum (taken over i) of the PRF advantages of hybrid i.

```
Definition PRF_Advantage_Max :=
 PRF_Advantage_Game (argMax PRF_Advantage_Game numCalls).
```

If f is instantiated with HMAC, then *PRF_Advantage_Max* can be instantiated with the bound on the PRF advantage of HMAC derived in [11], modulo some technical details of instantiating the *PRF_Adversary* between the two definitions.

Quantity (2) in the theorem statement is the upper bound on the probability of collisions in a list of length *blocksPerCall* of *Bvector* η that are each sampled uniformly at random.

```
Definition Pr_collisions := (1 + blocksPerCall)^2 / 2^η.
```

This arises from bounding the probability of the "bad event" in intermediate hybrids, the probability of collisions in the inputs to a random function as used in that intermediate hybrid.

4.4 Proof that HMAC-DRBG is pseudorandom

We formulate the real-world and ideal-world outputs as games involving interactions with the adversary A, and use the code-based game-playing technique [10] to bound the difference in probability that the adversary returns *True*. We call this "bounding the distance between the two games." We use "equivalent" to mean that two games correspond to the same probability distribution. We abbreviate "random function" as "RF" and "ideal randomness" as "RB," for "random bits." The output block-size of the PRF is denoted by η.

Now we outline the proof of the main result *G_real_ideal_close*. Here is the proof script:

```
Theorem G_real_ideal_close :
  | Pr[G_real] - Pr[G_ideal] | ≤
      numCalls * (PRF_Advantage_Max + Pr_collisions).
Proof.
  rewrite Generate_move_v_update.
  rewrite G_real_is_first_hybrid.
  rewrite G_ideal_is_last_hybrid.
  specialize (distance_le_prod_f (fun i => Pr[Gi_prg i])
       Gi_adjacent_hybrids_close numCalls).
  intuition.
Qed.
```

The proof is quite short, since it consists of applying five lemmas: rewriting G_{real} into the nicer form defined by *G1_prg*, then rewriting the ideal-world and real-world games as hybrids, then applying the theorem *Gi_adjacent_hybrids_close* that bounds the difference betweeen any two adjacent hybrids by *Gi_Gi_plus_1_bound*. Most of the work is done in *Gi_adjacent_hybrids_close*.

Our proof is structured according to this outline, and we will present the lemmas in this order:

- G_real_ideal_close
  1. Generate_move_v_update
  2. G_real_is_first_hybrid
  3. G_ideal_is_last_hybrid
  4. Gi_adjacent_hybrids_close
     a. Gi_prog_equiv_prf_oracle
     b. Gi_replace_prf_with_rf_oracle
     c. Gi_replace_rf_with_rb_oracle
        i. Gi_prog_equiv_rb_oracle
        ii. Gi_rb_rf_identical_until_bad
            (1) fcf_fundamental_lemma
            (2) Gi_rb_rf_return_bad_same
            (3) Gi_rb_rf_no_bad_same
        iii. Gi_Pr_bad_event_collisions
  5. hybrid_argument

We start by proving a lemma to rewrite G_{real} into a nicer form defined by *G1_prg*.

```
Lemma Generate_move_v_update :
  Pr[G_real] == Pr[G1_prg].
```

The way NIST's *Generate* function updates v makes it difficult to write neat hybrids. Specifically, the PRF is re-keyed on the second

line of *Generate*; then, on the next line, in order to update v, *Generate* queries the *new* PRF. Therefore, if we were to replace f_k with a PRF oracle, it would have to span the last line of *Generate* and the first two lines of the next *Generate* call, but not include the second call's v-update. This would be messy to reason about. We solve this problem by moving each v-update to the beginning of the next call of *Generate*, then prove that the outputs are still identically distributed. Then, after the PRF is re-keyed, we do *not* further query it in this call of *Generate_v*.

In the intermediate game *G1_prg*, we move each terminal v-update to the beginning of the next call to *Generate* by splitting the *Generate* function into *Generate_noV* and *Generate_v*.

The function *Generate_noV* is used for the first call. It differs from *Generate* because it does not start by updating the v.

```
Definition Generate_noV (state : KV) (n : nat) :
              Comp (list (Bvector η) * KV) :=
  [k, v] <-2 state;
  [bits, v'] <-2 Gen_loop k v n;
  k' <- f k (to_list v' ++ zeroes);
  ret (bits, (k', v')).
```

The second version, *Generate_v*, starts by updating v and does not update v again. Then, as in G_{real}, the resulting list of lists of blocks is passed to the adversary, which returns a boolean guess.

```
Definition Generate_v (state : KV) (n : nat) :
          Comp (list (Bvector η) * KV) :=
  [k, v] <-2 state;
  v' <- f k (to_list v);     (* new *)
  [bits, v''] <-2 Gen_loop k v' n;
  k' <- f k (to_list v'' ++ zeroes);
  ret (bits, (k', v'')).
```

This is the revised real-world game that uses the two split-up *Generate* versions defined above.

```
Definition G1_prg : Comp bool :=
  [k, v] <-$2 Instantiate;
  [head_bits, state'] <-$2 Generate_noV (k, v)
                           blocksPerCall;
  [tail_bits, _] <-$2 oracleMap Generate_v state'
                           (tail requestList);
  A (head_bits :: tail_bits).
```

To prove the real-world game G_{real} equivalent to the new version *G1_prg*, we prove that the pseudorandom output produced by this sequence of calls

$$[\textit{Generate}, \textit{Generate}, \ldots]$$

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

$$[\textit{Generate\_noV}, \textit{Generate\_v}, \textit{Generate\_v}, \ldots].$$

Using *G1_prg*, to set up the hybrid argument, we rewrite the two "endpoint" games as hybrids. We first prove (by a straightforward induction) that the real-world game is equal to the first hybrid.

```
Lemma G_real_is_first_hybrid :
  Pr[G1_prg] == Pr[Gi_prg 0].
```

Next: the ideal-world game is equal to the last hybrid, also proved by a straightforward induction.

```
Lemma G_ideal_is_last_hybrid :
  Pr[G_ideal] == Pr[Gi_prg numCalls].
```

Bounding the difference between adjacent hybrids. As described in Section 4.3, the difference between any two adjacent hybrids is defined to be the sum of the maximum PRF advantage over the PRF advantage for each hybrid, plus an upper bound on the probability of collisions in a list of length *blocksPerCall* of uniformly randomly sampled blocks of length η each.

```
Theorem Gi_adjacent_hybrids_close : forall (n : nat),
  | Pr[Gi_prg n] - Pr[Gi_prg (1+n)] |
   ≤ PRF_Advantage_Max + Pr_collisions.
```

The game *Gi_prg i* defines the ith hybrid game, which replaces the output of the PRF with ideal randomness for any call before the ith call.

```
Definition Gi_prg (i : nat) : Comp bool :=
  [k, v] <-$2 Instantiate;
  [bits, _] <-$2 oracleMap (choose_Generate i)
                   (0, (k, v)) requestList;
  A bits.
```

To do this, *Gi_prg i* uses an oracle *choose_Generate i* that tracks the number of calls *callsSoFar* (in addition to the PRG's internal state (k, v)) and returns the appropriate version of the PRG for that call. *choose_Generate i* returns *Generate_rb_intermediate* if the call index is less than i, and *Generate_noV* or *Generate_v* as appropriate otherwise.

```
Definition choose_Generate (i : nat) (sn: nat * KV)
  (n: nat) : Comp (list (Bvector η) * (nat * KV)) :=
  [callsSoFar, state] <-2 sn;
  let Gen := if lt_dec callsSoFar i
             then Generate_rb_intermediate
             else if beq_nat callsSoFar 0 then Generate_noV
             else Generate_v in
  [bits, state'] <-$2 Gen state n;
  ret (bits, (1+callsSoFar, state')).
```

The two quantities that compose the bound between any two adjacent hybrids *Gi_prg i* and *Gi_prg* $(i + 1)$ are defined as follows (see also Section 4.3).

```
Definition PRF_Advantage_Max :=
 PRF_Advantage_Game (argMax PRF_Advantage_Game numCalls).
Definition Pr_collisions := (1 + blocksPerCall)^2 / 2^η.
```

To bound the distance between adjacent hybrids *Gi_prg i* and *Gi_prg* $(i + 1)$, we derive the two quantities in the bound in two steps. *PRF_Advantage_Max* is derived in lemma *Gi_prf_rf_close*, and *Pr_collisions* in lemma *Gi_rf_rb_close*.

Before we can derive the bound, we first need to transform our normal hybrid definition *Gi_prg i* into the equivalent oracle-using definition *Gi_prf i*.

```
Lemma Gi_prog_equiv_prf_oracle : forall (i : nat),
  Pr[Gi_prg i] == Pr[Gi_prf i].
```

The only difference between the two definitions is that *Gi_prf i* is written in the form of an oracle computation that uses a provided oracle on the ith call. In the new game, the oracle we provide is

the PRF, defined in the code as *f_oracle f k*. The old game *Gi_prg i* already uses the PRF on the *i*th call by definition.

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

The computation *PRF_Adversary* is a constructed adversary against the PRF that rewrites our PRG execution, previously defined using *oracleMap*, in terms of *oracleCompMap*, which is a version of *oracleMap* that replaces the PRF on the *i*th call of hybrid *i* with a provided oracle. The provided oracle is the PRF, *f_oracle f k*.

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

The technical lemma *Gi_normal_prf_eq* allows us to provide different oracles to the new oracle-taking form of the hybrid to use on its *i*th call, which we use to define the two intermediate hybrids after *Gi_prg i*.

We derive the first quantity used in the bound, *PRF_Advantage_Max*.

```
Lemma Gi_replace_prf_with_rf_oracle : forall (i : nat),
    i ≤ numCalls ->
| Pr[Gi_prf i] - Pr[Gi_rf i] | ≤ PRF_Advantage_Max.
```

Recall that we have just rewritten the *i*th hybrid *Gi_prg i* as *Gi_prf*, defined above, which uses the PRF oracle *f_oracle f k*. In *Gi_rf*, we replace the PRF oracle with a random function oracle *randomFunc*.

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

A random function is simply a function that, queried with any new input, returns an ideally random *Bvector η* and caches the result. If it is queried with a previous input, it returns the previously-sampled bit-vector. By the definition of PRF advantage, for any adversary against the PRF, the probability that the adversary can distinguish between the PRF and a random function is *PRF_Advantage_Game*(*i*), which is upper-bounded by the maximum taken over *i* (over all hybrids).

Having bounded the distance between *Gi_prg i* and the intermediate hybrid *Gi_rf i*, we proceed to derive the second quantity in the bound: the distance between *Gi_rf i* and the next normal hybrid *Gi_prg* $(i+1)$ is *Pr_collisions*.

```
Lemma Gi_replace_rf_with_rb_oracle : forall (i : nat),
  | Pr[Gi_rf i] - Pr[Gi_prg (1+i)] | ≤ Pr_collisions.
```

Similarly to lemma *Gi_prf_rf_close*, we must first rewrite the leftmost hybrid *Gi_prg* $(i+1)$ in the form of the oracle-using hybrid *Gi_rb* that uses an oracle *rb_oracle* on its *i*th call.

```
Lemma Gi_prog_equiv_rb_oracle : forall (i : nat),
    Pr[Gi_prg (1+i)] == Pr[Gi_rb i].

Definition Gi_rb (i : nat) : Comp bool :=
  [b, state] <-$2 PRF_Adversary i rb_oracle nil;
  let rbInputs := fst (split state) in ret b.
```

This new oracle *rb_oracle* returns an ideally random *Bvector η* for every query, not caching any of its queried inputs.

```
 Definition rb_oracle (state : list (Blist * Bvector η))
                                      (input : Blist) :=
  output <-$ ({0,1}^eta);
  ret (output, (input, output) :: state).
```

We prove this rewrite, *Gi_normal_rb_eq*, via another induction. We use this rewrite to bound the difference between the two oracle-using hybrids *Gi_rf i* and *Gi_rb i*. The two games differ in only one aspect: on the *i*th call, *Gi_rf i* replaces the PRF with a random function oracle, whereas *Gi_rb i* replaces the PRF with an ideally random oracle. These two oracles themselves can be distinguished by an adversary in only one way: if the random function is queried with an input that has been previously queried, then it becomes deterministic and returns the same output as before, whereas the ideally random oracle will still return ideally random output.

Therefore, the distance between *Gi_rf i* and *Gi_rb i* ought to be bounded by the probability that there are duplicates in the inputs to the random function, which we refer to as the probability of the *bad event* occurring. The probability that the adversary can distinguish between *Gi_rf i* and *Gi_rb i* should then be upper-bounded by the probability of the bad event.

We formalize this intuitive argument by applying the *fundamental lemma of game-playing* [10], which states that the difference between two games that are "identical until bad" is upper-bounded by the probability of the "bad event" happening. Two games are "identical until bad" if they are syntactically identical except for statements that follow the setting of a flag *bad* to *true*.

We state the bound on the difference between the game in a form following that of the fundamental lemma:

```
Lemma Gi_rb_rf_identical_until_bad : forall (i : nat),
  | Pr[x <-$ Gi_rf_dups_bad i; ret fst x]
      - Pr[x <-$ Gi_rb_bad i; ret fst x] |
  ≤ Pr[x <-$ Gi_rb_bad i; ret snd x].
```

The new games *Gi_rf_dups_bad* and *Gi_rb_bad* are just versions of the original games rewritten to return a tuple, where the first element is the usual output of *Gi_rf* or *Gi_rb* and the second event is a boolean that indicates whether the bad event has occurred. To apply the fundamental lemma to prove *Gi_rb_rf_identical_until_bad*, we first prove that the games *Gi_rf_dups_bad* and *Gi_rb_bad* are indeed identical until bad. First we prove that the bad event has the same probability of happening in each game:

```
Lemma Gi_rb_rf_return_bad_same : forall (i : nat),
    Pr  [x <-$ Gi_rb_bad i; ret snd x ] ==
    Pr  [x <-$ Gi_rf_dups_bad i; ret snd x ].
```

This is clearly true (but don't take our word for it: ask Coq!). Second, we prove that if the bad event does not happen, the two games have identically distributed outputs (evalDist takes a Comp and produces the corresponding probability mass function).

```
Lemma Gi_rb_rf_no_bad_same : forall (i: nat) (a: bool),
     evalDist (Gi_rb_bad i) (a, false) ==
     evalDist (Gi_rf_dups_bad i) (a, false).
```

This, too, is clearly true. Thus, we can apply the fundamental lemma to derive the upper bound:

```
Pr[x <-$ Gi_rb_bad i; ret snd x].
```

That concludes the identical-until-bad argument, which bounded the difference between the two games by the probability of the bad event. We finish the proof of the main result by calculating an upper bound on the probability of the bad event, which is, again, a collision in the inputs to the RF.

```
Theorem Gi_Pr_bad_event_collisions:
  forall (i:nat) (v: Bvector η),
  Pr [x <-$ Gi_rb_bad i; ret snd x ] ≤ Pr_collisions.
```

On the ith call to the PRG, the provided oracle (here the random function) is queried once with a constant v and then *blocksPerCall* times after that, where each RF input is actually a previous output from the RF due to the design of HMAC-DRBG's *Gen_loop*. If an RF output is not a duplicate, it is ideally random. (To derive a more general bound, *blocksPerCall* can be generalized to the ith element in the request list. The rest of the proof is unchanged.)

Therefore, the probability of collisions in the inputs to the RF simplifies to the probability of collisions in a list of length *blocksPerCall*+1 of elements *Bvector* η that are each sampled uniformly at random. By the union bound, this is less than $(1 + blocksPerCall)^2/2^\eta$.

We have bounded the distance between adjacent hybrids with *Gi_adjacent_hybrids_close*, so we conclude the whole proof by applying the hybrid argument (which is essentially the triangle inequality, repeatedly applied) with that lemma to bound the distance between the first and last hybrid.

```
Theorem hybrid_argument :
      forall (G : nat -> Rat) (bound : Rat),
      (forall i : nat, | G i - G (1+i) | ≤ bound) ->
      forall n : nat, | G 0 - G n | ≤ n * bound
```

This generic theorem states that if the distance between adjacent games $G\ i$ and $G\ (i+1)$ is bounded by the rational number *bound*, the difference between the first hybrid $G\ 0$ and the last hybrid $G\ n$ is bounded by $n \cdot bound$.

4.5 Coq-specific proofs

Working in Coq requires doing many proofs of program and game equivalence. These program equivalence proofs are generally elided, by choice or by accident, in paper proofs. Two such proofs are `Gi_prog_equiv_prf_oracle` and `Gi_prog_equiv_rb_oracle`. Because using oracles changes the types of all computations that use that computation, the types of *Gen_loop*, *Generate_v*, *choose_Generate*, and *oracleMap* all change from *Comp* to *OracleComp*, and we write new functions with the new type and prove them each equivalent. Proving the two lemmas requires doing complex inductions over both the list of blocks provided by the adversary and over each number of blocks in a call.

Semiformal crypto proofs (e.g., [24]) often leave the burden of checking for semantic equivalence on the reader. An unintentional error in this part of a crypto proof is unlikely, but an intentional error is possible, and it may be subtle enough that a standards body would not notice it.

4.6 Efficiency of adversary

We do not formally prove the efficiency of the constructed adversaries in the proof. It is necessary to inspect the family of constructed adversaries (that is, Definition PRF_Adversary in §3.1) in order to establish that it is probabilistic polynomial time; which clearly it is (assuming the adversary against the PRG is p.p.t.). While this omission increases the amount of code that must be inspected, it also allows the result to be interpreted in any model of computation and cost.

5 CONCRETE SECURITY ESTIMATE

The distance between G_{real} and G_{ideal} is

$$numCalls \cdot (PRF\_Advantage\_Max + (1 + blocksPerCall)^2/2^\eta).$$

What does that mean concretely about an adversary's ability to predict the output of HMAC-DRBG? Let us assume the adversary runs in time and space at most 2^t.

Beringer *et al.* proved that the PRF advantage for HMAC is the sum of the advantages associated with these three assumptions: (1) SHA-256 is weakly collision-resistant, (2) the SHA-256 compression function is a PRF, and (3) the dual family of the SHA256 compression function is a PRF against a (very) restricted related-key attack.

(1) If the best method for finding collisions is a birthday attack, a collision will be found with probability at most 2^{2t-256}. (2) If the best method for distinguishing SHA-256's compression from a random function is an exhaustive search on keys, the probability of success is at most 2^{t-256}. (3) If, in addition, we consider that the constructed adversary in the proof gains little or no additional advantage from the related key oracle, then the probability of success is at most 2^{t-256} [9].

From these assumptions, we can conclude that the adversary's advantage in distinguishing HMAC-SHA256 from a random function is at most $2^{2t-256} + 2^{t-255}$. For $t = 78$, the advantage is at most $2^{-100} + 2^{-177}$.

Assuming there are 2^{48} calls to the DRBG with block length $\eta = 128$ and $blocksPerCall = 10$, then the distance is about $2^{48} \cdot (2^{-100} + 2^{-177} + (1+10)^2/2^{128}$, or about 2^{-52}. That is, the adversary with a thousand million terabytes ($< 2^{78}$ bits), and a million 1-gigahertz processors running for a year ($< 2^{78}$ cycles) has a 2^{-52} chance of guessing better than random.

It also means that block length 128 is better than 64, but increases beyond that are not particularly useful.

Warning: These calculations are based on unproved assumptions (1)(2)(3) about SHA-256 and the best ways to crack it. The assumptions may not be true, and there may be better ways to crack SHA-256.

6 MBEDTLS IMPLEMENTATION IS CORRECT

6.1 Functional specification program

We verify the mbedTLS implementation of HMAC-DRBG with respect to a functional program in Coq, with functions **mbedtls_generate**, …, **mbedtls_reseed**. Where the Coq functions **Generate**, etc. described in Section 4.2 operate over *Bvector* η, these functions operate over finite lists of sequences of integers (type `list Z` in Coq), and over abstract DRBG states of type

```
Inductive hmac256drbgabs :=
  HMAC256DRBGabs: forall (key V: list Z)
       (reseedCounter entropyLen: Z)
       (predictionResistance: bool)
       (reseedInterval: Z), hmac256drbgabs.
```

The functions' definitions are structured in accordance with NIST 800-90A, specialising generic DRBG functions to the functional program for HMAC-SHA256 from Beringer *et al.* [11]. The detailed definitions are available in our source code.

There is a straightforward relation $\sim$ between states (k, v) from Section 4 and states (`HMAC256DRBGabs k v 0 0 false` 2^{48}) of our **mbedtls** functional specification.

Based on $\sim$, we formulate an equivalence theorem between the two specifications of the *Generate* function, under certain conditions. For example, mbedTLS measures `out_len` in bytes, up to 1024, while our crypto spec measures n in 256-bit blocks; the reseed counter must be less than the reseed interval; the *predictionResistance* flag must be false, and so on.

Theorem 6.1. *The functions* ***mbedtls_generate****, etc. are equivalent to* ***Generate****, etc., for* $\sim$ *similar states.*

Proof. By induction on iterations of the *Generate* loop. □

Proving the C program correct w.r.t. the functional program, then proving Theorem 6.1, is simpler than directly proving the C program implements the crypto spec. There are other advantages too: **mbedtls_generate** is directly executable, though about a million times slower than the C program. Successfully validating our functional program against all 240 noReseed-noPredictionResistance test vectors from NIST [31] takes about 30 mins.

6.2 Verification of C code

The Verified Software Toolchain's program logic, called *Verifiable C*, is a higher-order impredicative separation logic [5]. That is, it is a Hoare logic that can handle advanced features such as pointer data structures, function-pointers, quantification over predicates, recursive predicates, and so on. Its judgments take the shape of a Hoare triple $\{pre(z)\}c\{post(z)\}$, where assertions $pre(.)$ and $post(.)$ are predicates on states. Variable z is implicitly universally quantified and is used to express functional correctness properties in which postconditions refer to (some aspects of) preconditions. Our proof automation operates over assertions of the form

PROP P LOCAL L SEP R

where P is a purely propositional term, L describes the content of local and global variables, and R describes the content of the heap, as a list of formulae that are implicitly interpreted to refer to non-overlapping memory regions.

Figure 1 shows a specification of `mbedtls_hmac_drbg_random`. It means, suppose the function is called with argument `p_rng` holding a concrete representation of some abstract DRBG state I, and `output` pointing to a (separate) memory region of length $0 \le n \le 1024$, specified in `out_len`. Suppose further that requesting n bytes from I and entropy stream s succeeds according to the *functional* specification program `mbedtls_generate`, yielding `(bytes,ss, F)`. Then, executing the body of the C function is *safe*: it will not perform

```
Definition drbg_random_abs_spec :=
DECLARE _mbedtls_hmac_drbg_random
 WITH output: val, n: Z, ctx: val,
      I, F: hmac256drbgabs, kv: val,
      s, ss: ENTROPY.stream, bytes: list Z
 PRE [_p_rng OF tptr tvoid, _output OF tptr tuchar,
      _out_len OF tuint ]
  PROP (0 <= n <= 1024;
        mbedtls_generate s I n = Some (bytes, ss, F)
  LOCAL (temp _p_rng ctx; temp _output output;
         temp _out_len (Vint (Int.repr n));
         gvar sha._K256 kv)
  SEP (data_at_ Tsh (tarray tuchar n) output;
       AREP kv I ctx; Stream s)
 POST [ tint ]
  PROP ()
  LOCAL (temp ret_temp (Vint Int.zero))
  SEP (data_at Tsh (tarray tuchar n)
                 (map Vint (map Int.repr bytes)) output;
       AREP kv F ctx; Stream ss).
```

Figure 1: VST specification of `mbedtls_hmac_drbg_random`

memory loads or stores outside the region specified by the precondition's SEP clause, the function's stack frame, and potentially locally allocated memory; it will not experience division-by-zero or other runtime errors. If terminating, the function will yield return value 0, and a state in which `ctx` indeed holds abstract state F, `output` indeed holds `bytes`, and the entropy stream is updated as specified.

The use of representation predicate `AREP` to relate an abstract DRBG state to a concrete C data structure laid out in memory is typical in separation logic, and emphasizes that client code should only operate on DRBG contexts using the interface functions. Abstract states I and F are elements of the type `hmac256drbgabs` defined above. The definition of `AREP` includes numeric invariants asserting that, e.g., `reseedCounter` is bounded between 0 and `Int.max_signed`, but is only unfolded in the proofs of the function bodies.

The code of `mbedtls_hmac_drbg_random` invokes the more general function `mbedtls_hmac_drbg_random_with_add`, our specification of which indeed supports the supply of additional input, respects the prediction resistance flag and reseeds the DRBG if necessary. The specification shown in Figure 1 restricts input arguments so that certain error cases are avoided; again, our formalization also includes more general specifications.

Our C implementation is based on mbedTLS `hmac_drbg.c`, specialized to HMAC-SHA256 by instantiating C preprocessor macros. We make minor local edits to satisfy Verifiable C's syntactic restrictions: isolating memory loads/stores into separate assignment statements. The program uses a `volatile` keyword to prevent the C compiler from deleting the clearing of an array after its last use; reasoning about `volatile` is not supported in Verifiable C, so we had to remove it. Calls to HMAC are rerouted to the OpenSSL implementation [11].

To formally prove that the C code satisfies the specification, we use the VST-Floyd proof automation system: a collection of Coq

tactics, lemmas, and auxiliary definitions that perform forward symbolic execution with substantial interactive help from the user, who supplies function specifications, loop invariants, and proofs of application-specific transformations using Coq's built-in proof commands. Building on the verification of `HMAC_SHA256`, we have specified and proved all core functions in `hmac_drbg.c`, including the flag update functions.

7 LESSONS LEARNED

Three NIST design decisions made the proof harder or easier.

The good. NIST re-keys the PRF with a length-extended input. This simplifies the component of the proof that bounds the probability of collisions in inputs to the random function. Because HMAC can take inputs of any length, and all previous inputs for HMAC with that key were of fixed length (since HMAC has a fixed output), it is easy to prove that the length-extended input cannot be a duplicate of any previous input to the random function.

The bad. In *Instantiate*, NIST begins the initialization of the state (k, v) as follows:

$k \leftarrow HMAC_{c_1}(c_2||\texttt{0x00}||entropy\_input)$
$v \leftarrow HMAC_k(c_2)$

If we assume that *entropy_input* is ideally random, then it would be better to incorporate it into the first argument (c_1) of HMAC, rather than the second, because the latter forces us to assume or prove that HMAC is an entropy extractor.

If *entropy_input* is *not* ideally random (as would likely be the case in a real system), then we will indeed need an entropy extractor. HMAC seems to be as good a candidate as any. However, existing proofs of HMAC as a randomness extractor [19, 21, 27] are only for keys c_1 that are random, so they may not apply here because HMAC-DRBG's $c_1 = 0$. So we have no security proof of *Instantiate*. Instead, we assume that the (k, v) produced by *Instantiate* is ideally random.

The ugly. It is hard to reason about NIST's decision to update the PRG's internal state vector v immediately after re-keying the PRF in the same call to *Generate*. This decision makes it difficult to define hybrids that replace a PRF on each call, since the PRF changes and is used before a call is over. We solve this problem by moving each v-update to the beginning of the next call and proving that the new sequence of programs is indistinguishable to the adversary, and we continue the proof with the rewritten *Generate*.

How verification helped. The process of mechanizing our proof and working in Coq and FCF enabled us to state definitions, functions, and games in a language that was expressive, precise, and executable. This benefited our proof in two main ways.

First, working in this environment prevented us from making subtle mistakes in dealing with a tricky hybrid argument. For example, for every lemma bounding the difference between adjacent hybrids H_i and H_{i+1}, Coq requires the lemma to be parametrized by i. This helped early in the proof; it forced us to realize that there were multiple PRF adversaries: one for each hybrid H_i.

Additionally, Coq prevented us from making suble off-by-one errors in the hybrid argument. It's easy to mix up the total number of hybrids for a list of blocks of n elements—is it n or $n + 1$ hybrids? It's also easy to mix up whether the oracle is replaced in the ith call or the $i + 1$th call. If the numbering is wrong, the relevant lemmas will become unprovable.

Second, working in this environment allowed us to "execute" computations in lemmas and test if they held true on small examples, which helped us spot typos in the proof. For example, in *Generate_v*, we had typed ret $(bits, (k', v'))$ instead of ret $(bits, (k', v''))$, which made one lemma unprovable until we fixed it. Similar bugs have been found in real-world code because a typo or mistake rendered the code unverifiable.

The trusted base. As explained in §1, a key part of the trusted base of any machine-checked proof is *the statement of the theorem and assumptions.* Our theorem is stated as 200 lines of Coq, (pseudorandom properties of HMAC-DRBG), on top of about 50 lines of supporting FCF definitions defining the notions of pseudorandomness, etc. We rely on assumptions related to collision-resistance of SHA [11, §4.2], which means that the 169-line definition of SHA in Coq is also part of the trusted base.

On the other hand, the definitions of the HMAC function, the HMAC-DRBG function, and the specifications of internal APIs need not be trusted, because we prove an end-to-end theorem that, regardless of what these functions actually are, they are correctly implemented by the C code and they correctly provide a PRF.

Proof effort. The proof of cryptographic security spans about 4500 lines of Coq code—which need not be trusted because it is checked by machine, but still it has a pleasantly clear structure as explained in §4.4. Developing and mechanizing this proof took about 5 person-months. The C program-correctness proof is about 10,200 lines of Coq, and took 1 person-month by a new user of Verifiable C. Theorem 6.1 (about 700 lines), along with the connection to the HMAC proofs and other configuration engineering (included in the 10,200 lines) took about 1 person-month. Coq checks all the proofs in about 16.5 minutes (running Coq 8.6 on a 2.9 GHz Intel Core i7 on a single core; faster on multicore).

8 FUTURE WORK

In this work we focused primarily on the pseudorandomness property of the DRBG. In SP 800-90A, NIST proposes a variety of additional properties, including "backtracking resistance" and "prediction resistance." The former property ensures that a compromise of generator state should not affect the security of output generated *prior* to the compromise, while the latter property addresses post-compromise security (under the assumption of a continually reseeded generator). Extending our proof to backtracking resistance should be straightforward. A proof that HMAC-DRBG is prediction-resistant would require a careful specification of the rate at which new entropy must be provided.

Our security proof leaves out some optional parameters such as the personalization nonce and additional input. We hope to address these in the future.

While we have addressed only a single NIST DRBG construction, future work may prove the security of other widely-used SP 800-90A constructions such as CTR-DRBG and HASH-DRBG (based on block ciphers and hash functions respectively). These generators have similar constructions to HMAC-DRBG, and thus it is likely that our work may be adapted with reasonable modifications.

HMAC-DRBG's Instantiate function uses HMAC with fixed non-random key as a randomness extractor; this is not known to be sound (see §7).

Concerning implementation correctness, we would like to extend our framework by integrating provable protection against side-channel attacks [3, 7], including protection against timing attacks that recently affected implementations of MEE-CBC [2].

We hope this work will lead to formal verification efforts on other widely used cryptographic software packages. Ideally, this type of verification could replace or supplement the use of test vectors, which are currently used by organizations such as the U.S. Cryptographic Module Validation Program to validate DRBG correctness.

9 CONCLUSION

We have created the first end-to-end formal security-and-correctness verification of a real-world random number generator, and the first machine-checked proofs of HMAC-DRBG security and of a correct implementation of HMAC-DRBG. We can say with very high confidence that HMAC-DRBG and its mbedTLS implementation are secure in their basic functions against a nonadaptive adversary. There are no bugs (including buffer overruns) and no backdoors, purposeful or otherwise. We have not proved security of the *Instantiate* algorithm (though we have proved that mbedTLS implements it correctly). Our proof is modular; thus, it is portable to other implementations of HMAC-DRBG. Our proof should be extensible to other features and properties of HMAC-DRBG, with limitations as discussed in §8.

Machine-checked proofs of cryptographic security should be a required part of the NIST standardization process. This would likely have detected serious generator flaws such as those that occurred in Juniper NetScreen devices [15]. And though HMAC-DRBG is a good DRBG, proving the security of the construction at NIST standardization time would likely have resulted in a more elegant construction, and would have allowed NIST to clearly specify the security assumptions on HMAC required by **Instantiate**.

Finally, implementations of cryptographic standards should come with machine-checked proofs of correctness. This would have prevented the HeartBleed bug, and probably prevented the Debian/OpenSSL fiasco (see the appendix).

Acknowledgments. This research was supported in part by DARPA agreement FA8750-12-2-0293 and by NSF Grant CCF-1521602. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] Reynald Affeldt, David Nowak, and Kiyoshi Yamada. 2012. Certifying assembly with formal security proofs: The case of BBS. *Science of Computer Programming* 77, 10 (2012), 1058–1074.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2015. Verifiable side-channel security of cryptographic implementations: constant-time MEE-CBC. *IACR Cryptology ePrint Archive* 2015 (2015), 1241. http://eprint.iacr.org/2015/1241

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium, USENIX Security 16*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 53–70. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[4] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. on Programming Languages and Systems* 37, 2 (April 2015), 7:1–7:31.

[5] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers.* Cambridge University Press, New York.

[6] Elaine Barker and John Kelsey. 2012. *Special Publication 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* Technical Report 800-90A. National Institute of Standards and Technology. http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf

[7] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1267–1279. DOI: https://doi.org/10.1145/2660267.2660283

[8] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. 2013. Stealthy Dopant-level Hardware Trojans. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, Berlin, 197–214.

[9] Mihir Bellare. 2006. New proofs for NMAC and HMAC: Security without collision-resistance. In *Annual International Cryptology Conference*. Springer, 602–619.

[10] Mihir Bellare and Phillip Rogaway. 2006. The security of triple encryption and a framework for code-based game-playing proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 409–426.

[11] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *24th USENIX Security Symposium*. USENIX Assocation, 207–221.

[12] Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko van Someren. 2013. Factoring RSA Keys from Certified Smart Cards: Coppersmith in the Wild. In *ASIACRYPT 2013: 19th International Conference on the Theory and Application of Cryptology and Information Security, Proceedings Part II*. Springer, Berlin, 341–360. DOI: https://doi.org/10.1007/978-3-642-42045-0_18

[13] Daniel J. Bernstein, Tanja Lange, and Ruben Niederhagen. 2016. Dual EC: A Standardized Back Door. In *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday (Lecture Notes in Computer Science)*, Peter Y. A. Ryan, David Naccache, and Jean-Jacques Quisquater (Eds.), Vol. 9100. Springer, 256–281. DOI: https://doi.org/10.1007/978-3-662-49301-4_17

[14] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 2013. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 445–459.

[15] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. 2016. A Systematic Analysis of the Juniper Dual EC Incident. In *CCS '16: 23rd ACM Conference on Computer and Communications Security*. ACM, New York, NY, USA, 468–479. DOI: https://doi.org/10.1145/2976749.2978395

[16] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. 2014. On the Practical Exploitability of Dual EC in TLS Implementations. In *Usenix Security '14*. USENIX Association, San Diego, CA, 319–335. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/checkoway

[17] Russ Cox. 2008. Lessons from the Debian/OpenSSL Fiasco. (21 May 2008). https://research.swtch.com/openssl

[18] Yevgeniy Dodis, Chaya Ganesh, Alexander Golovnev, Ari Juels, and Thomas Ristenpart. 2015. A formal treatment of backdoored pseudorandom generators. In *EUROCRYPT (1)*. 101–126.

[19] Yevgeniy Dodis, Rosario Gennaro, Johan Håstad, Hugo Krawczyk, and Tal Rabin. 2004. Randomness extraction and key derivation using the CBC, cascade and HMAC modes. In *Annual International Cryptology Conference*. Springer, 494–510.

[20] Felix Dörre and Vladimir Klebanov. 2016. Practical Detection of Entropy Loss in Pseudo-Random Number Generators. In *CCS'16: 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, 678–689.

[21] Pierre-Alain Fouque, David Pointcheval, and Sébastien Zimmer. 2008. HMAC is a randomness extractor and applications to TLS. In *ACM symposium on Information,*

Computer and Communications Security. ACM, 21–32.
[22] Dan Goodin. 2013. Google confirms critical Android crypto flaw used in $5,700 Bitcoin heist. *Ars Technica* (14 Aug. 2013). https://arstechnica.com/security/2013/08/google-confirms-critical-android-crypto-flaw-used-in-5700-bitcoin-heist/
[23] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2012. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *21st USENIX Security Symposium*. USENIX Association, 205–220.
[24] Shoichi Hirose. 2008. Security analysis of DRBG using HMAC in NIST SP 800-90. In *International Workshop on Information Security Applications*. Springer, 278–291.
[25] Fortinet Inc. 2016. CVE-2016-8492. Available at https://fortiguard.com/advisory/FG-IR-16-067|. (2016).
[26] ISO. 2012. ISO 19790:2012: Security requirements for cryptographic modules. Available at https://www.iso.org/standard/52906.html. (August 2012).
[27] Hugo Krawczyk. 2010. Cryptographic extraction and key derivation: The HKDF scheme. In *Annual Cryptology Conference*. Springer, 631–648.
[28] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. 2012. Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064. (2012). http://eprint.iacr.org/2012/064.
[29] H. D. Moore. 2008. Debian OpenSSL Flaw. Available at https://hdm.io/tools/debian-openssl/. (2008).
[30] National Institute of Standards and Technology. 2014. NIST RNG Cryptographic Toolkit. Available at http://csrc.nist.gov/groups/ST/toolkit/rng/. (July 2014).
[31] National Institute of Standards and Technology. 2017. CAVP Testing: Random Number Generators. (2017). http://csrc.nist.gov/groups/STM/cavp/random-number-generation.html.
[32] National Institute of Standards and Technology (NIST). 2001. FIPS 140-2: Security Requirements for Cryptographic Modules. Available at http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf. (May 2001).
[33] Nicole Perlroth, Jeff Larson, and Scott Shane. 2013. N.S.A. Able to Foil Basic Safeguards of Privacy on Web. *The New York Times* (6 Sept. 2013). http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html
[34] Adam Petcher. 2015. *A Foundational Proof Framework for Cryptography*. Ph.D. Dissertation. Harvard University.
[35] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *4th International Conference on Principles of Security and Trust (POST)*. Springer LNCS 9036, Berlin, 53–72.
[36] Dan Shumow and Niels Ferguson. 2007. On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng. CRYPTO 2007 Rump Session. (Aug. 2007). http://rump2007.cr.yp.to/15-shumow.pdf
[37] Andrey Sidorenko and Berry Schoenmakers. 2005. Concrete Security of the Blum-Blum-Shub Pseudorandom Generator. In *Cryptography and Coding: 10th IMA International Conference*. Springer LNCS 3796, Berlin, 355–375.
[38] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* 27, 8 (1984), 761–763.
[39] Kim Zetter. 2016. New Discovery Around Juniper Backdoor Raises More Questions About the Company. *WIRED.com* (Jan. 2016). https://www.wired.com/2016/01/new-discovery-around-juniper-backdoor-raises-more-questions-about-the-company/

APPENDIX: DEBIAN/OPENSSL FIASCO

In 2006, in order to eliminate a Purify warning, a developer at Debian removed a line of code from the randomness-initializer of OpenSSL as shipped with Debian [29]. This reduced the amount of true entropy for initializing SSH keys to approximately 15 bits, or (in some realistic scenarios) 0 bits.

Cox [17] has an excellent analysis of the technical problem *and the social processes* that led to this failure. We have no library in Coq for proving things about social processes, but we can say:

(1) The all-too-clever C code in the OpenSSL RAND_poll and RAND_add tolerates the use of a partially uninitialized array. Cox explains why (a) it's not strictly speaking *wrong* to do this and (b) it's still a terrible idea. The *Verifiable C* proof system would reject this program.
(2) The Debian developer asked about the safety of this change on the official OpenSSL developer's mailing list, and got no response. Verifiable C cannot help with that.
(3) Presumably a (hypothetical) verified implementation distributed by OpenSSL would be marked as such (in comments) and distributed with its open-source proof, and not so casually modified by the Debian developer.
(4) Suppose there had been a different insufficient-entropy bug, one not involving uninitialized variables. In such a case, it would not be automatically rejected by Verifiable C. Instead, either (a) the C code could not be proved to implement the functional model, or (b) the functional model could not be proved to provide sufficient entropy at initialization time. In either case, the bug would be detected.
(5) Suppose the functional model and C code were both proved correct; the correct program asks an external device for n bits of "true" entropy, and instead gets n bits with almost no entropy [8]. Our proofs cannot help here.

In summary: Yes, proofs would probably have prevented the Debian/OpenSSL fiasco.