Formally proving cryptographic properties of pseudo-random number generators

Katherine Ye ‘16

Advised by Profs Andrew Appel (Princeton)
and Matthew Green (Johns Hopkins)

October 1, 2015

1. Motivation and goal

Most modern cryptosystems rely on random numbers, which they use to generate secrets that need to be known to users and unknown and unpredictable to attackers. Reducing the entropy of a cryptosystem's pseudo-random number generator (PRNG) is an easy way to break the entire cryptosystem. The PRNG will generate output predictable to an attacker, allowing her to guess private keys, yet the bits may still "look" random. Plus, the rest of the cryptosystem will function normally, since PRNGs tend to be single self-contained components. These two factors make PRNGs very attractive targets to attackers.

The attack has indeed happened in practice (through due to a programming error and not malice), and with devastating consequences. Luciano Bello discovered that the random number generator in Debian OpenSSL, a widely-used cryptographic library, was predictable, allowing attackers to easily guess keys. Debian advised all users to regenerate keys, though some high-profile users didn't. For example, compromised SSH keys were used to access Spotify, Yandex, and gov.uk's public repositories on GitHub.

(This introduction is based on the Wikipedia page for "Random number generator attack.")

Despite the importance of PRNGs, surprisingly little work exists on proving them secure, either by proving on paper that certain widely-used PRNGs are secure, or by verifying with computer-checked proofs that implementations of these PRNGs satisfy their specifications.

Our project aims to do both. We aim prove correctness and security of an OpenSSL implementations of two widely-used PRNGs, CTR-DRBG and HMAC-DRBG. We will do this by proving that the implementation satisfies a functional specification (a high-level implementation in code) of the random number generator, which we trust satisfies its paper specification. Then we will prove that the functional specification guarantees the expected cryptographic properties, most notably that a computationally bounded attacker cannot distinguish generated bits from random bits. These properties will "flow" from the functional spec down to the compiled code via a verified compiler, CompCert. The entire verification will be machine-checked by Coq, a trusted proof assistant.

This thesis attacks the cryptographic aspect of the problem. We hope to prove that PRNGs called CTR-DRBG and HMAC-DBG generate bits that are indistinguishable from random, that the PRGs are backtracking-resistant, that they eventually recover from compromises of internal state, and other important properties. We will write functional specifications of these PRGs in the environment of the Coq proof assistant, then prove that they possess the desired properties by using the Foundational Cryptography Framework in Coq.

2. Problem background and related work

Andrew Appel's group has done the most significant related work in the area. Appel (2015) [1] presents a "full formal machine-checked verification of a C program: the OpenSSL implementation of SHA-256." Appel et al. (2015) [2] build on this work to do the same for HMAC, adding a proof of security that relies on the security of SHA. We plan to use the same approach for CTR-DRBG and HMAC-DRBG.

In addition, there exist paper proofs of the security of CTR-DRBG [3] and HMAC-DRBG [4], though they haven't appeared in peer-reviewed venues.

In the area of checking game-based proofs of cryptographic security within a proof assistant, there are two main tools: EasyCrypt and its cousin CertiCrypt (neither of which is foundational), and the Foundational Cryptography Framework. [5]

In the general area of formalizing PRNGs, several crypto papers analyze the security of PRNGs and propose new security properties, e.g. Dodis et al. [6] who propose the "robustness" property and show that the built-in Linux PRNG, /dev/random, is not robust.

There's not much prior work on formal verification of PRNGs in our style. Dörre and Klebanov (2015) [7] focus on verifying that a PRNG uses all its entropy. They perform this logic-based information flow verification using the KeY system for Java, which uses symbolic execution. This only defends against one particular attack (that of "squandering entropy") and does not guarantee functional correctness or other security properties we may care about, such as indistinguishability from randomness and backtracking resistance.

Affeldt (2009) [8] do include game-playing proofs of provable security in Coq. However, they focus on doing so directly on an assembly implementation of a PRNG, not on a high-level functional specification. They also verified their own assembly implementation, not a widely-used existing one. One advantage of this approach is that it avoids mismatches between the functional specification of the C code and the functional specification used for cryptographic proofs.

Our approach is unique because it provides an end-to-end and foundational verification that guarantees both correctness and security. Our stack consists of Coq, the Foundational Cryptography Framework, the Verified Software Toolchain (using separation logic), and CompCert (a verified C compiler). In addition, we verify an existing widely-used PRG implementation in C, making our approach more useful in practice.

[1] http://www.cs.princeton.edu/~appel/papers/verif-sha.pdf
[2] http://www.cs.princeton.edu/~appel/papers/verified-hmac.pdf
[3] https://eprint.iacr.org/2006/379.pdf
[4] http://repo.flib.u-fukui.ac.jp/dspace/bitstream/10098/2126/1/art.pdf
[5] http://arxiv.org/abs/1410.3735
[6] https://eprint.iacr.org/2013/338.pdf
[7] https://formal.iti.kit.edu/~klebanov/pubs/vstte2015.pdf
[8] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.165.6623&rep=rep1&type=pdf

3. Approach

We will first write a functional specification of the random number generator, as well as primitives it depends on, such as AES. Then we will prove its security and other properties using game-based cryptographic proofs in the Foundational Cryptography Framework in Coq.

This approach has worked before on HMAC [2] and we are reasonably confident it will work again.

4. Plan

The first part of the plan is divided into two sections, the math section and the code section.

Math

- read HMAC-DRBG NIST specification
- read HMAC-DRBG proof
- figure out how to translate proof into game-based crypto proofs
- read crypto and probability chapters
- learn about game-based proofs
- write out these game-based crypto proofs on paper

- may need to do original proofs for DRBG properties not covered in the literature

Risky: lots of background knowledge needed, existing math may be difficult to understand, and we may need to do original math.

Code

- read FCF El-Gamal example, figure out how to use FCF
- read FCF paper
- write HMAC-DRBG functional spec
- prove indistinguishability (using paper proofs)
- prove backtracking resistance (using paper proofs)

Risky: FCF may be hard to use and un-maintained. FCF may not have enough library support for PRG proofs, and we will need to spend our time writing libraries.

Then repeat the process with AES and CTR-DRBG. Risky: adds a lot of work, but shows that our process is portable and general.

Then link the entire project together with Steve and Naphat's work (the two other students on the project). Risky: there may be specification gaps or type mismatches. These have happened before with the HMAC project, where we had to prove the equivalence of two specifications, an abstract spec that used bitvectors and a concrete spec that used lists of integers.

5. Evaluation

If we succeed in the goal of verifying everything we want to, then the project has succeeded. If we only succeed partially, how much have we verified? Did we uncover any interesting difficulties?

In general we can ask the following questions to measure our success:

- How automated is the verification? How much human effort was required to do tedious things?
- How many lines of code and of proofs were written?
- Did we contribute original mathematics?
- How many properties were we able to verify, and how important are they?
- How complex is the code verified? (e.g. OpenSSL uses more sophisticated C features than BoringSSL, such as "engines")
- Is our verification actually right? We may find bugs in the code along the way. Also, we should fail to verify broken code.
- What attacks can be definitively ruled out by our verification? What attacks are still possible? (e.g. timing or cache side-channel attacks)

- Are the security and formal verification communities excited about using or building on our work?