Seeding Random Number Generators

Jesse Walker
Intel Corporation
Intel Labs
Circuits and System Research
Security Research Lab

Agenda

- The problem
- RNG Seeding Requirements
- Example 1: Intel's new hardware RNG
- Example 2: Fixing low-entropy key generation

The Problem

- In February 2012 Arjlen Lenstra et al posted a paper titled "Ron was Wrong, Whit was Right"
- Reported that the moduli of thousands of RSA keys on deployed systems share prime factors and hence provide no security
  - If $N = pq$ and $N' = pq'$ are two RSA moduli, then $\gcd(N,N') = p$ and we can trivially find q and q'
- Review of an affected product:
  - Uses OpenSSL to generate its RSA keys
  - As far as we know, the OpenSSL RNG is competent
  - As far as we know, the OpenSSL prime number generator is competent
  - The problem must be somewhere else?

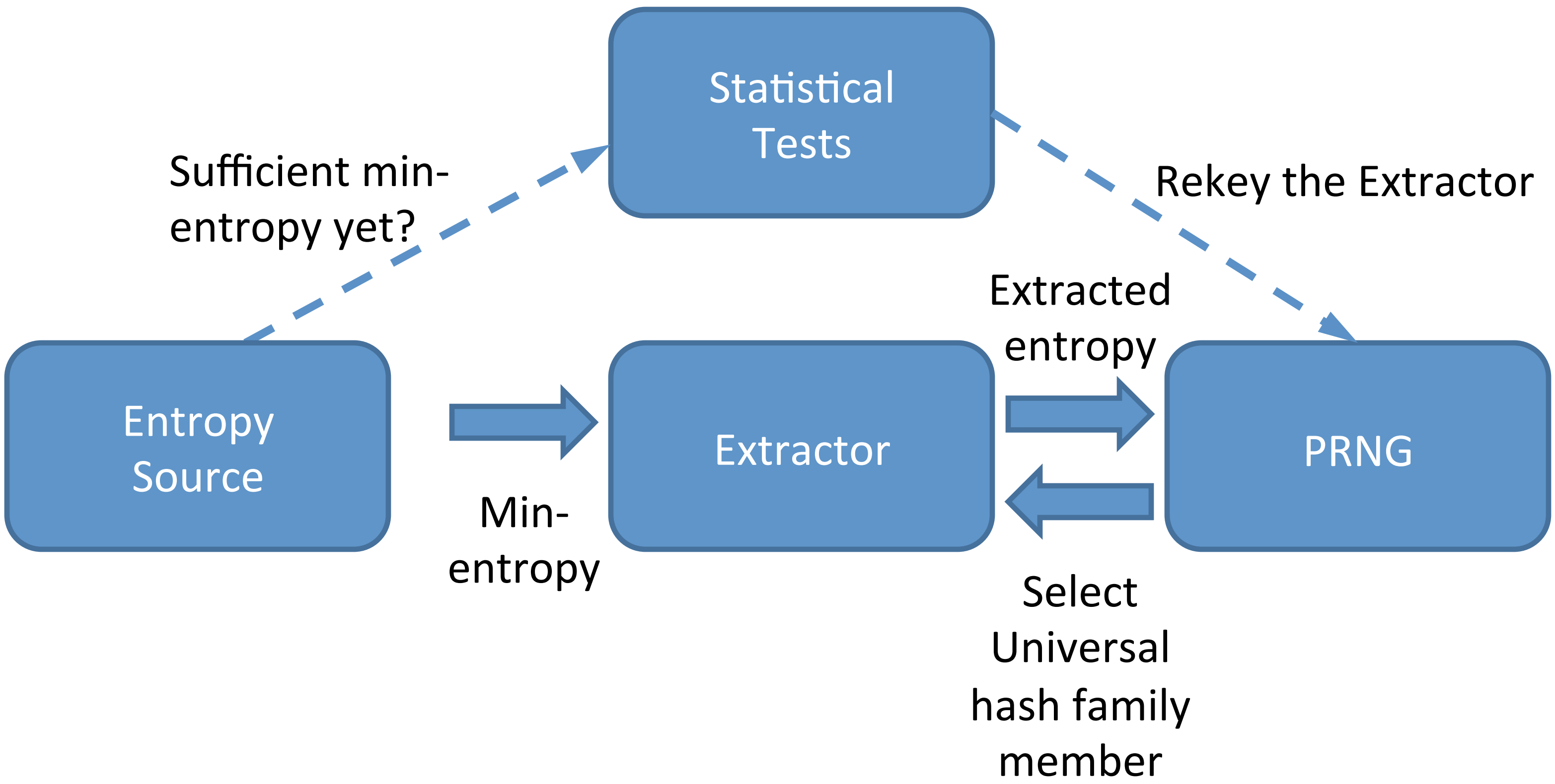
Key Generation Procedure

- RSA key is generated when the product is manufactured
- The RSA key is generated using the following procedure:
  - Step 1: Set the system clock to Midnight, January 1, 1970
  - Step 2: Then use time to seed the OS RNG
  - Step 3: Use the OS RNG to generate 32 words
  - Step 4: Use the 32 words to seed the OpenSSL RNG
  - Step 5: Run the OpenSSL RSA key generation function
- Oops
- This is a common idiom; can we do better?

Requirements

- Requirement 1. An RNG must be seeded from an ignorance source
  - Ignorance source is usually called an entropy source
  - Ignorance source means we do not know the internal state of the source
- Requirement 2. Our ignorance of the source must be necessary
  - This means the source has a well-defined min-entropy, and our ignorance of some of its state is necessary
- Requirement 3. It must be unlikely an adversary can make the same measurements of the source we make
  - Because the seed is supposed to be a secret
- Requirement 4. A seed must be extracted from the source
  - Because the samples from the source will never be ideal
- Requirement 5. The ignorance source must be simple enough to be modeled and validated
  - Because otherwise we don't know when it is doing something useful

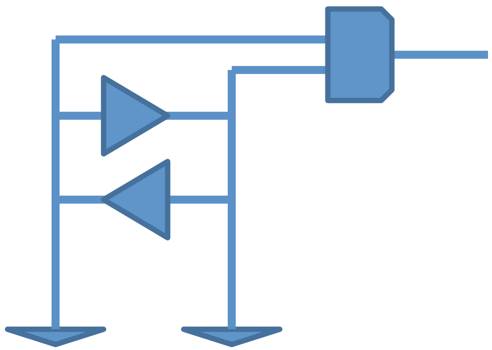
Conceptual Framework



Entropy Source for Intel's HW RNG

- Our extractor is AES-CBC-MAC
  - In a Crypto 2004 Dodis et al showed CBC-MAC of b block strings is a $(1+\eta)/2^n$ -universal hash family, where $\eta = O(b^3/2^{2n})$ and n is the block size
- By the Leftover Hash Lemma, with this universal hash family we need 382 bits = 3 AES blocks of min-entropy from our sample to produce a full entropy output with AES-CBC-MAC
- This should fulfill requirement 4

Entropy Source



The entropy source in Intel's new RNG is latch built from a pair of cross-coupled inverters

- — Circuit assumes two stable (0/1) and one unstable state (meta-stable)
- — Circuit powered on in the meta-stable state
- — Circuit held in meta-stable state until Johnson thermal noise resolves circuit's value to 0 or 1
- — After the circuit resolves and outputs one bit value, power it off
- — Repeat at machine clock rate

This should fulfill requirement 1

Entropy Source Models

- This entropy source circuit is simple enough to model faithfully
- We created several models that can be used to validate the hardware
  - All the models can be configured with thermal and electrical characteristics of the circuit design
- The most explanatory model is an Ornstein-Uhlenbeck process
  - Over time an Ornstein-Uhlenback process tends to drift toward its long-term mean
  - A digital latch tends to resolve to its previous state, so our circuit slightly biases the next output slightly
- The statistical tests continuous validate the entropy source conforms to its model
- This should fulfill requirements 2 and 5

Fixing reseed (time (0))?

- Measure a source of ignorance
  - e.g., the latency of a system call

```
unsigned before, after, entropy;
before = read_TSC ();
usleep(0);
after = read_TSC ();
entropy = (after – before) & 0x0ff;
Repeat until sufficient entropy harvested
```

- Intuitively the scheduler should be a source of ignorance
  - It asks the scheduler to run any other ready thread and return immediately if there is no other ready thread
  - We don't know which thread will be ready next, nor how long it will run
  - Seems to satisfy requirement 1

Analysis

- The proposed source only works for applications
  - Any kernel-space code can potentially read the source's state
  - Can be attacked by infecting libc, which hopefully is not a problem during manufacturing – requirement 3 satisfied
- Empirical measurement on multiple instances of the target hardware showed that the source can be modeled as an AR(1) process
  - The computed min-entropy of 4.9 bits/byte appears consistent with empirical measurement on the target platform
  - Empirically it seems to satisfy requirement 2
- We got lucky – the process isn't even stationary with the same OS on other hardware
  - This software entropy source is not portable across platforms without extensive rework

Summary

- A theory and consensus examples for seeding PRNGs is an urgent need
- Entropy sources are sources of ignorance
  - Our knowledge of their state is necessarily limited
- Real entropy sources ***at best*** produce min-entropy, not entropy, so we have to extract
- To be useful entropy sources must have a well-defined and validatable min-entropy
  - Because the Left-over Hash Lemma is our best tool to convert min-entropy into entropy
- Entropy sources must be easy to analyze
  - Or else we cannot validate they deliver min-entropy

Feedback?