Analysis of Intel's Ivy Bridge Digital Random Number Generator

Prepared for Intel by

Mike Hamburg
Paul Kocher
Mark E. Marson

Cryptography Research, Inc.
575 Market St., 11th Floor
San Francisco, CA 94105
(415) 397-0123

March 12, 2012

CRYPTOGRAPHY™
RESEARCH
a division of Rambus

Contents

1 Introduction

Good cryptography requires good random numbers. This paper evaluates Intel's hardware-based digital random number generator (RNG) for use in cryptographic applications.

Almost all cryptographic protocols require the generation and use of secret values that must be unknown to attackers. For example, random number generators are required to generate public/private keypairs for asymmetric (public key) algorithms including RSA, DSA, and Diffie-Hellman. Keys for symmetric and hybrid cryptosystems are also generated randomly. RNGs are used to create challenges, nonces (salts), padding bytes, and blinding values.

Because security protocols rely on the unpredictability of the keys they use, random number generators for cryptographic applications must meet stringent requirements. The most important property is that attackers, including those who know the RNG design, must not be able to make any useful predictions about the RNG outputs. In particular, the apparent entropy of the RNG output should be as close as possible to the bit length.

1.1 Entropy

Entropy is a measurement of how random a particular process is. While there are several ways to measure entropy, in this paper we will primarily use Shannon entropy [1]

$$H_1 = -\sum_{i=1}^{n} p_i \log_2 p_i$$

and min-entropy [2]

$$H_\infty = \min_{i=1}^{n}(-\log_2 p_i).$$

In the above formulas, p_i is the probability of the process being in the ith of n possible states, or returning the ith of n possible outputs. By using a base-2 logarithm we are measuring the entropy in bits. Shannon entropy measures the average amount of information required to describe the state, whereas min-entropy measures the probability that an attacker can guess the state with a single guess. The min-entropy of a process is always less than or equal to its Shannon entropy.

In the case of a random number generator that produces a k-bit binary result, p_i is the probability that an output will equal i, where $0 \le i < 2^k$. Thus, for a perfect random number generator, $p_i = 2^{-k}$. In this case the Shannon entropy and min-entropy of the output are both equal to k bits, and all possible outcomes are equally likely. The information present in the output cannot, on average, be represented in a sequence shorter than k bits, and an attacker cannot guess the output with probability greater than 2^{-k}.

An RNG for cryptographic applications should appear to computationally-bounded adversaries to be close as possible to a perfect RNG. For this review, we analyze whether there is any feasible way to distinguish the Intel RNG from a perfect RNG.

1.2 Deterministic random bit generators

Most "random" number sources actually utilize a deterministic random bit generator (DRBG). DRBGs use deterministic processes to generate a series of outputs from an initial seed state. Because the output is purely a function of the seed data, the actual entropy of the output can never exceed the entropy of the seed. It can, however, be computationally infeasible to distinguish a well-seeded DRBG from a perfect RNG.

For example, consider a DRBG seeded with 256 bits of entropy that produces more than 256 bits of output. An attacker who successfully guessed the seed data could predict the entire DRBG output. Guessing a 256-bit seed value is computationally infeasible, however, so such a DRBG can be appropriate for cryptographic applications even though its outputs are not truly random.

Cryptographic applications often demand extremely high quality output, necessitating great care in the development, testing, and selection of DRBG algorithms. NIST has published SP 800-90A [2], which specifies several DRBG constructions. The publication includes recommendations for instantiating, using, and reseeding DRBGs.

DRBGs also require random seeds. A deterministic process cannot create randomness, so ultimately a nondeterministic (aka "true") random number generator is still required for seeding these constructions.

1.3 The need for nondeterministic random bit generators

A nondeterministic random bit generator uses a nondeterministic source to produce randomness. Most operate by measuring unpredictable natural processes, such as thermal (resistance or shot) noise, atmospheric noise, or nuclear decay. The entropy, trustworthiness, and performance all depend on the underlying entropy source.

A DRBG by itself will be insecure without an entropy source for seeding. Seeding requires a source of true randomness, since it is impossible to create true randomness from within a deterministic system.

On computers without a hardware entropy source, programmers typically try to obtain entropy for seed data using existing peripherals. Modern UNIX and Windows OS's have OS-level RNGs based on the timing of kernel IO events. Unfortunately, the quality of the entropy collected depends upon the system's configuration and hardware. For example, the entropy available from embedded devices without hard drives or keyboards may be insufficient. Similarly, an operation that is secure on a busy test network may become insecure when moved to a high-security, low-traffic environment.

Even when it is possible for applications to produce their own secure random data, many do not. Reviews by Cryptography Research frequently identify weaknesses in random number generation. Bruce Schneier writes, "Good random-number generators are hard to design, because their security often depends on the particulars of the hardware and software. Many products we examine use bad ones." [3]

For example, Luciano Bello discovered a serious flaw in the DRBG that shipped with the OpenSSL cryptography library on Debian and Ubuntu Linux systems from September 2006 to May 2008 [4]. All OpenSSL keys generated by the affected systems were compromised, including server certificates, SSH login keys and email signing/encryption keys. More recently, in 2012 a study showed that an unexpectedly large number of RSA moduli share common prime factors, which can easily be computed using the GCD algorithm. One of the most likely causes is poor random number generation processes [5].

The need for strong randomness is not limited to key generation. For example, the popular DSA and ECDSA digital signature standards require a random value when each signature is produced. Even very slight biases in the RNG used to produce this value can lead to exploitable cryptographic weaknesses. Bleichenbacher discovered that the nonce generation method defined in FIPS 186 was slightly biased, and this bias could be used to mount a cryptanalytic attack against DSA and ECDSA [6].

Although RNG problems are common, flaws are often missed because there is no computationally-bounded test that can analyze an RNG's output and authoritatively confirm that the output is random. In addition, cryptographic software libraries often leave it to application developers to supply their own seed material, but programmers may lack the experience to do this effectively.

In other cases, system designers are faced with a trade-off between security and convenience. For example, to avoid having to collect fresh seed data each time the program loads, many software applications store their seed material on the hard drive where there can be a risk of compromise. The best solution to these challenges is for the hardware to provide a well-designed, efficient, and easy-to-use hardware entropy source.

2 Architecture

Intel provided CRI with detailed documentation on the RNG, as well as access to its principal developers. This section describes the overall architecture and the main components of the system.

2.1 System overview

A block diagram of the major components of the Intel Ivy Bridge RNG is shown in Figure 1 below.

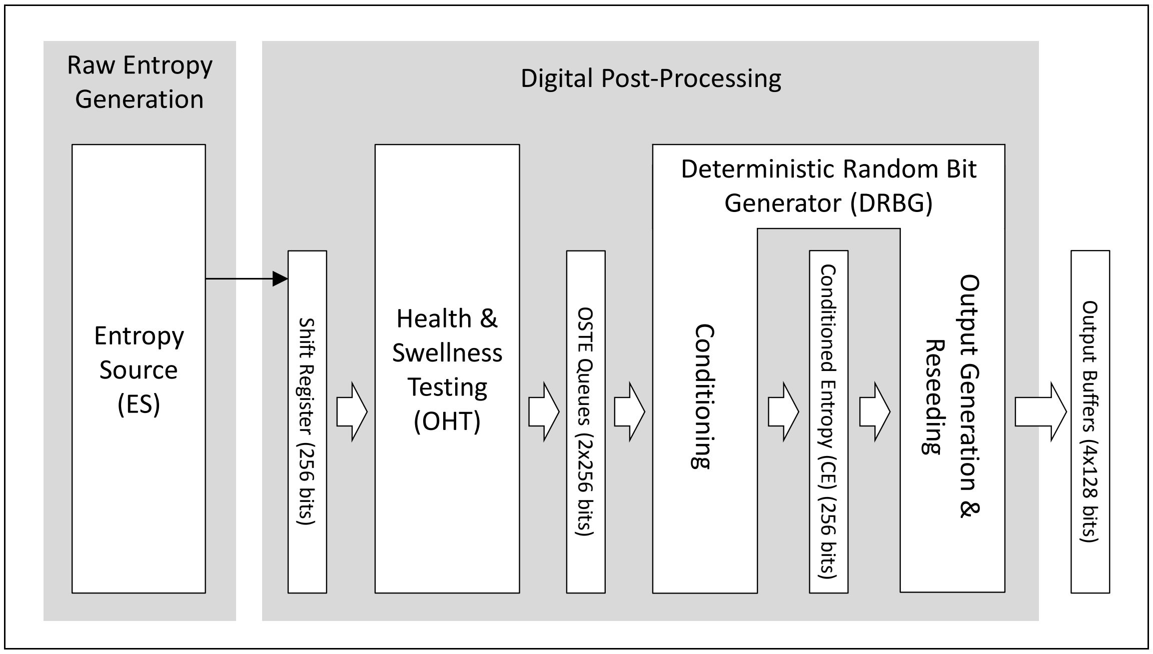
Raw Entropy Generation
Digital Post-Processing
Entropy Source (ES)
Shift Register (256 bits)
Health & Swellness Testing (OHT)
OSTE Queues (2x256 bits)
Deterministic Random Bit Generator (DRBG)
Conditioning
Conditioned Entropy (CE) (256 bits)
Output Generation & Reseeding
Output Buffers (4x128 bits)

Figure 1: Block diagram of the Intel RNG (adapted from [7])

Most modern RNGs, including the Intel Ivy Bridge design, consist of an entropy source (ES) followed by digital post-processing logic. Raw output from entropy sources generally contains detectable biases and other artifacts that distinguish the output from random binary data. The purpose of the post-processing logic is to convert this raw output into lower-bitrate, but higher-quality, random data.

The Intel RNG's post-processing logic is relatively sophisticated. Like many software-based RNGs, the post-processing uses strong cryptography to prevent deficiencies in the entropy source from leading to exploitable weaknesses. In particular, the RNG maintains an entropy pool which is seeded using a relatively large amount of data from the ES. Even if the ES is severely degraded, the final output will remain of high quality and cryptographically strong and should appear indistinguishable from true random by computationally-bounded adversaries (despite being nonrandom from an information theoretic perspective).

One drawback of using post-processing is that defects in the entropy source become more difficult to observe. As a result, users of the RNG have a more difficult time assessing the quality of the underlying entropy source, and some catastrophic failure modes can actually become difficult to detect. The Intel Ivy Bridge designers have employed several strategies that help mitigate these concerns, including the incorporation of logic to monitor the health of the entropy source. In addition, while raw access to entropy source output is not available on production parts, test parts can provide direct access to entropy source outputs.

The Intel RNG operates as follows:

1. The entropy source (ES) is a self-clocking circuit which operates asynchronously and generates random bits at a high rate (about 3 GHz).

2. Random bits generated by the ES are combined, sampled by the synchronous logic, and grouped into 256-bit blocks in a shift register.
3. Basic statistical tests are performed by the online health test logic (OHT) on each 256-bit block to check for potential failure modes of the ES.
4. The 256-bit blocks in the online self-tested entropy (OSTE) queue are cryptographically processed into a 256-bit conditioned entropy pool by the conditioning logic.
5. The conditioned entropy pool is used to reseed the DRBG.
6. The DRBG generates the final bits output by the RNG.

The rest of this section describes in detail each of the components, and how they operate and interact.

2.2 Entropy source

The entropy source (ES) at the heart of the Intel RNG is a self-oscillating digital circuit with feedback, shown in Figure 2 below.

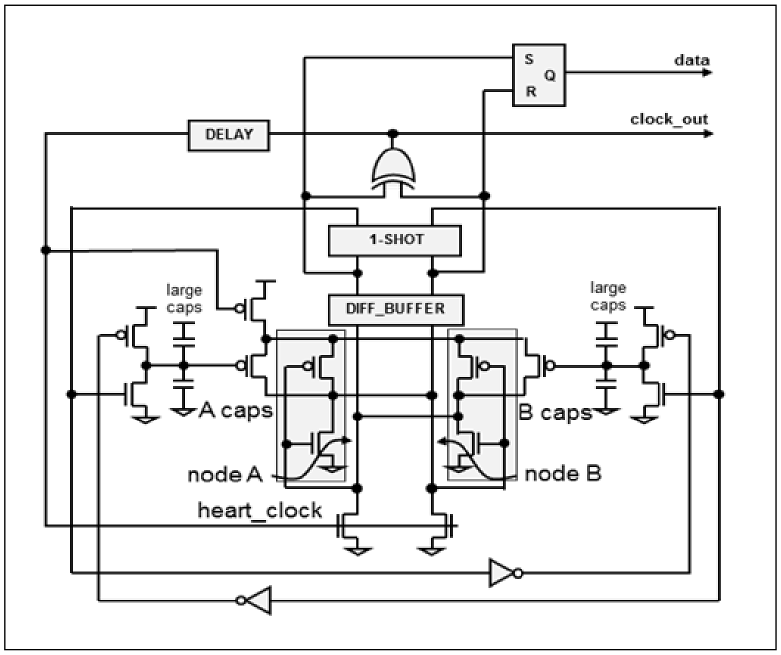

Figure 2: Entropy source for the Intel RNG (from [8])

The ES is a dual differential jamb latch with feedback. It is a latch formed by two cross-coupled inverters (nodes A and B). The circuit is self-clocking (heart_clock), and designed such that when the clock is running, the circuit enters a metastable state. The circuit then resolves to one of two possible states, determined randomly by thermal noise in the system. The settling of the circuit is biased by the differential in the charges on the capacitors (A caps and B caps). The state to which the latch resolves is the random bit of output of the ES.

The circuit is also designed with feedback to seek out its metastable region. Based on how the latch resolves, a fixed amount of charge is drained from one capacitor and added to the other. The goal is to have the latch oscillate around the metastable region, using the last output to determine the charge changes to the capacitors. At normal process, voltage and temperature (PVT) conditions, the ES runs at about 3 GHz. See [8] for a detailed description of the ES circuit.

Intel has developed a theoretical mathematical model for the ES, described in [8] and [9]. We analyze this model theoretically and empirically in Sections 3 and 4, respectively.

The rest of the RNG is clocked at 800 MHz. The ES generates random bits at a rate of about 3 GHz, and they need to be transferred across to the synchronous region of the circuit.

The RNG provides an option to accumulate samples from the ES with a running XOR circuit. This XOR sum of all previous ES bits is stored in a single-bit buffer. When an ES output arrives, it is XORed with the current value of the buffer, and the result is written back to the buffer. The buffer is then sampled by the synchronous logic at 800 MHz. Alternatively, the RNG can be configured to overwrite the buffer with each new ES output. In this case, some ES outputs will not be sampled before the buffer is overwritten, and thus will not be used.

Future versions of the RNG will use a different synchronization logic. The ES output will be deserialized, and then sampled in parallel into the synchronous region, thereby preserving all the ES samples for post-processing.

2.3 Health and "swellness" tests

Once the data is sampled into the synchronous region, it is passed serially to the on-line health test unit in a sliding window, and from there into the 2-deep, 256-bit-wide Online Self Tested Entropy (OSTE) FIFO buffer.

The health check unit evaluates the health of each 256-bit sample. It counts how many times each of six different bit patterns appears in a sample. The sample is deemed "healthy" if and only if the number of times each pattern appears falls within certain bounds. The bit patterns and the bounds for each are shown in Table 1 below.

Bit pattern	Allowable number of occurrences per 256-bit sample
1	$109 < n < 165$
01	$46 < n < 84$
010	$8 < n < 58$
0110	$2 < n < 35$
101	$8 < n < 58$
1001	$2 < n < 35$

Table 1: Health bounds for 256-bit samples

The bounds were determined empirically by Intel, but are close to what one would expect to see for truly random data. The probability that a random sample from a uniform distribution fails the health checks is about 1%.

The health checks are not intended as a comprehensive measure of entropy. Instead, they intended to check if the entropy source is badly broken and stuck outputting simple repeating patterns such as all zeros, all ones, or alternating zeros and ones.

The heath test unit tracks the health status of the most recent 256 256-bit samples. The ES is considered to be "swell" if and only if at least 128 of the most recent 256 samples are healthy. All samples are consumed by DRBG and used to condition the entropy pool, whether or not they are healthy (Section 2.4.1). However, samples are not counted as contributing fresh entropy to the entropy pool unless they are healthy and the ES is already in the "swell" state.

Once the first 256-bit OSTE buffer is full, the bits are shifted in parallel to a second 256-bit OSTE buffer. The bits in the second OSTE buffer are then passed to the Deterministic Random Bit Generator (DRBG), and are processed as described in Section 2.4 below.

2.4 Deterministic random bit generator

The DRBG accumulates the ES samples into a conditioned entropy pool. It then uses the conditioned entropy pool to reseed itself and generate the final random output. Details of the process are shown below.

2.4.1 Conditioning

The DRBG has a 256-bit buffer containing a pool of "conditioned entropy" (CE[255:0]), which is used to reseed the DRBG. The conditioned entropy pool is updated using the data values drawn from the second OSTE buffer (OSTE[255:0]). The lower and upper halves of the conditioned entropy pool are updated independently.

First, the lower half of the entropy pool (CE[127:0]) is updated by processing the current 256 bits in OSTE buffer with AES CBC-MAC mode as defined in [10], using a 128-bit non-secret fixed key K' that is identical in all chips. The pseudocode for this update process is shown below.

1. Temp[127:0] = AES(K', CE[127:0])
2. Temp[127:0] = AES(K', OSTE[127:0] XOR Temp[127:0])
3. CE[127:0] = AES(K', OSTE[255:128] XOR Temp).

This process is repeated (with new data in the OSTE buffer each time) if the OSTE buffer is not healthy, or if the ES is not in the "swell" state.

The upper half of the entropy pool, CE[255:128], is then updated using the same process, using fresh values from OSTE[255:0].

1. Temp[127:0] = AES(K',CE[255:128])
2. Temp[127:0] = AES(K',OSTE[127:0] XOR Temp[127:0])
3. CE[255:128] = AES(K',OSTE[255:128] XOR Temp).

As with the lower half of the entropy pool, the update process is repeated until a healthy sample is processed while the ES is "swell".

The CE update process is shown in Figure 3 below.

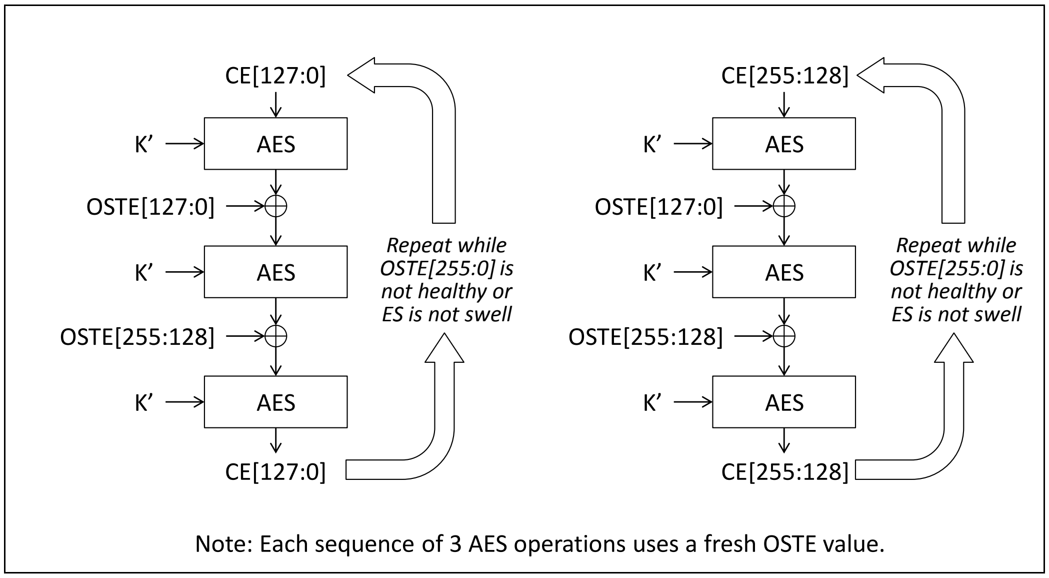
CE[127:0]
K'
AES
OSTE[127:0]
K'
AES
OSTE[255:128]
K'
AES
CE[127:0]
Repeat while OSTE[255:0] is not healthy or ES is not swell
CE[255:128]
K'
AES
OSTE[127:0]
K'
AES
OSTE[255:128]
K'
AES
CE[255:128]
Repeat while OSTE[255:0] is not healthy or ES is not swell
Note: Each sequence of 3 AES operations uses a fresh OSTE value.

Figure 3: Process for updating CE[127:0] and CE[255:128]

The CE update processes combines the prior state of each half of CE with 256 bits from OSTE (or more if a half is updated for health or swellness reasons). Each update process is designed to fully randomize half of CE, even if the OSTE bits are partially random. For example, if either OSTE[127:0] or OSTE[255:128] is random and independent, the updated value for the half of CE will be fully random. More generally, we believe that the AES operation will reliably produce a random result if the OSTE's 256 bits contain at least 128 bits of entropy.

Thus, this stage should be effective at producing high quality output if there are at least 0.5 bits of entropy per bit sampled from the ES, excluding bits produced in blocks that are not healthy or while the ES is not swell. If the entropy from ES is not above 0.5 bits per bit sampled, the output of this stage may cease to be fully random.

2.4.2 Reseeding

The reseeding process must be completed before any random output can be produced, and is additionally performed frequently during normal operation. In particular, the DRBG requires reseeding after it produces 512 128-bit outputs (65536 bits total). However, under normal operations, the DRBG will reseed much more often.

Before the reseeding process can begin, the conditioned entropy pool must have been updated successfully. The lower half (CE[127:0]) is then used to reseed the DRBG AES key K[127:0]. (Note that K is different from the conditioning key K'). The upper half, CE[255:128] is used to reseed the counter (V[127:0]). The reseeding of the key and counter proceed are defined in [2]. The pseudocode for the process is shown below.

Let K denote the key for the DRBG,V the 128-bit counter, and C the number of outputs since the last reseeding. All three are initialized to zero at reset. Then the DRBG reseeds as follows.

1. V[15:0] = (V[15:0] + 1) mod 65536
2. Temp = AES (K,V)
3. V[15:0] = (V[15:0] + 1) mod 65536
4. V = CE[255:128] XOR AES(K,V)
5. K = CE[127:0] XOR Temp
6. C = 0.

The DRBG initially reseeds upon reset during BIST (see Section 2.5). It also requires reseeding after generating a maximum of 512 128-bit outputs (65536 bits total). However, in the design reseeding is given high priority and the DRBG will reseed much more often than that. Intel's simulations suggest that as long as the ES is healthy, the generator will reseed within 22 128-bit outputs even under heavy load. Under light or moderate load, it will reseed before every 128-bit output.

2.4.3 Generation

Once the AES engine has been reseeded, the DRBG is ready to generate random data. It uses the counter mode CTR_DRBG construction as defined in [2], with AES-128 as the block cipher. It first fills four 128-bit output buffers with data, and then generates additional outputs as needed. The generation process is shown below.

```
V[15:0] = (V[15:0] + 1) modulo 65536
C = C+ 1
Output = AES(K,V)
If (update needed)
{
        V[15:0] = (V[15:0] + 1) modulo 65536
        Temp = AES(K,V)
        V[15:0] = (V[15:0] + 1) modulo 65536
        V = AES(K,V)
        K = Temp
}
```

The DRBG will update rather than reseed if the output count C is less than 512, and the conditions for reseeding are not met.

2.5 Built-In Self-Test (BIST)

Upon reset, the DRBG first performs a built-in self-test (BIST) to verify that it is operating properly, and to initialize the DRBG. BIST is conducted in two phases. The first phase tests the health check and DRBG logic. The second phase tests the entropy source (ES) and initializes the DRBG.

In Phase 0, the entropy source is disconnected and a linear feedback shift register (LFSR) generates a deterministic sequence of bits to test the heath check and DRBG circuits. A 32-bit CRC is computed on the output of the DRBG and compared against a hardwired expected value.

In Phase 1, the ES is reconnected and used to generate a sequence of 256 samples, each sample consisting of 256 bits. The samples are used to condition the 256-bit entropy pool. First, the lower half of the entropy pool is conditioned, until 128 healthy samples have been processed. At this point the ES is considered to be swell and the lower half of the conditioned entropy pool is marked as available. The upper half of the entropy pool is then conditioned, and is marked as available as soon as one sample has been processed while the ES is swell. Once both halves of the entropy pool are conditioned, the DRBG is reseeded, replacing the initial all-zero values for the key K and counter C. Finally, the DRBG generates four 128-bit samples, thereby filling the output buffer.

The RNG must pass both phases of BIST to be considered as working correctly. At the end of a successful BIST, the DRBG is ready for normal operation. If BIST fails, the RNG will produce no data. In this case, the instruction which reads from the RNG (RDRAND) will return all zeros and clear the carry flag, indicating there is a problem with the RNG, and its output must not be used.

2.6 Normal operation

In normal operation, the RNG operates autonomously. Upon reset, it performs a full BIST, conditions the output from the ES, reseeds the DRBG, and then fills the four 128-bit output buffers. Debug and test ports are disabled, and the only data exported by the RNG are random bits generated by the DRBG and passed to the output buffer.

Each RDRAND call to the RNG returns 64 bits from the output buffer, which the RNG refills as necessary. In addition, RDRAND uses the carry flag to indicate its status. This flag is set to one by RDRAND if the RNG is behaving properly. If the RNG detects a problem, such as a BIST failure, it will still service every request, but return all zeros for output. Given such a zero return value, RDRAND will clear the carry flag after each request. Users should check the carry flag after each RDRAND call to the RNG, and should not make any assumptions about the randomness of output if the carry flag is cleared.

Modeling and tests by Intel show that the RNG should be able to service all requests even under a heavy load. At an 800 MHz clock rate, the RNG can deliver post-processed random data at a sustained rate of 800 MBytes/sec. In particular, it should not be possible for a malicious process to starve another process.

The RNG supports eight different operational modes. However, most of them are designed to support testing and debugging, and are disabled on production parts. On deployed systems, the RNG will be locked in normal mode. This report is only concerned with the behavior of the system in normal mode.

In additional to the operational modes, the RNG supports a FIPS mode, which can be enabled and disabled independently of the operational modes. FIPS mode sets additional restrictions on how the RNG operates and can be configured, and is intended to facilitate

FIPS-140 certification. In first generation parts, FIPS mode and the XOR circuit will be disabled. Later parts will have FIPS mode enabled. CRI does not believe that these differences in configuration materially impact the security of the RNG. (See Section 3.2.2 for details.)

3 Theoretical Analysis

3.1 Entropy source

The entropy source is the most critical component of the system, and therefore we analyzed it in the most depth.

Our model of the entropy source roughly follows [9]. We model the state of the system as the difference in charge between the two capacitors, plus the previous output bit. This output bit allows us to model some causes of serial correlation. The model's parameters include:

- The distribution and amount of thermal noise affecting the output
- The amount of charge added or removed from the capacitors when outputting a 0 or 1, which we call the "right step size" and "left step size" respectively
- The distribution and amount of thermal noise affecting the step sizes
- The difference in charge on the capacitors when the system starts up.

The procedure to generate a new bit is as follows:

```
1. Let bias = (difference in charge on capacitors)
2. Let serialAdj = (if previous output bit was 1 then 1 else −1) × (serial coefficient)
3. If bias + serialAdj + (random thermal noise) > 0 then:
    a. Next output bit = 1
    b. Reduce (difference in charge on capacitors) by (left step size) + noise
4. Else:
    a. Next output bit = 0
    b. Increase (difference in charge on capacitors) by (right step size) + noise.
```

In an ideal system, we would have:

- Gaussian thermal noise with standard deviation 1 unit
- Steps in either direction are always 0.1 unit, with no noise
- Serial coefficient = 0
- Starting state with no charge on the capacitors.

However, we modeled non-ideal conditions as well, including:

- Non-Gaussian thermal noise
- Step sizes larger or smaller than 0.1 unit
- Steps which are different sizes in each direction
- Noise on the step size
- Positive or negative serial coefficient
- Starting with some charge on the capacitors.

By quantizing the charge difference and limiting it to a few standard deviations, we turned the above model into a Markov process suitable for mathematical analysis. In each case, we used this process to estimate the long-term Shannon entropy, the min-entropy over 32 bits, and local statistics such as autocorrelation and bias. We compared the results to measurements of an Intel tool which simulates the entropy source, and found them in agreement.

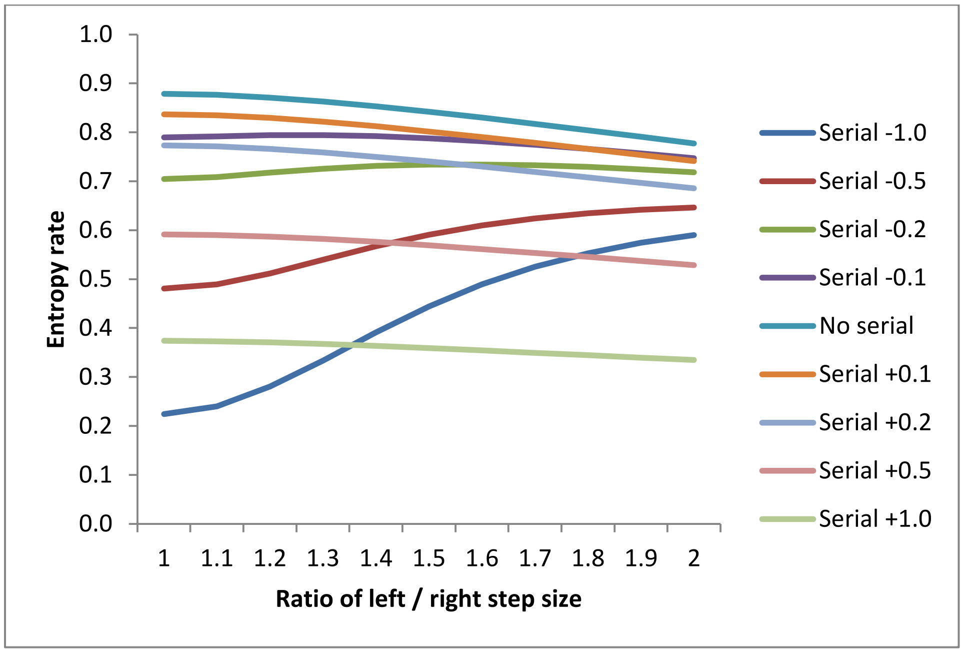

Figure 4: Effect of bias and serial coefficient on min-entropy, mean step size = 0.2

Figure 4 shows an interesting result of this modeling: if the serial coefficient is positive or zero, then bias in the step size will decrease entropy. However, if it is negative, then bias will break the pattern of oscillation, which may increase entropy. This graph shows extreme cases. The real parts we analyzed had serial coefficients under 0.1.

3.2 System analysis

3.2.1 Failure modes

The entropy source is the most sensitive part in the RNG, and also the most difficult to test. Hence, we will first consider its possible failure modes. Failure modes considered include:

- The ES always shows single-bit bias, serial correlation and other small deviations from perfect randomness. If these biases are severe, they may reduce the entropy rate of the ES below acceptable levels.
- The ES might take a long time to warm up, and during this time could output mostly 0s or mostly 1s until it settles on the metastable region.
- The ES might become "stuck", always outputting 0 or always outputting 1.
- The ES might oscillate between 0 and 1, or in some other short pattern.

- The ES might be mostly stuck in one of the preceding patterns, but occasionally deviate from it.
- The ES might be influenced by an external circuit (for example, a nearby bus or the chip's power supply) in a way that is predictable or exploitable by an attacker.

The RNG has a generous safety margin, so these failures will only impact security if they are severe. Of the possible failures above, most should be detected reliably by the health and swellness checks. The most complex issues involve brief externally-induced transitory losses of entropy, but the combination over-collection of entropy and the RNG's use of cryptography should mitigate any such unexpected issues. Also, while such failures can cause the design to behave briefly as a cryptographically-strong deterministic RNG, this should not result in any loss of security.

3.2.2 Health checks

The goal of the health checks is to reject ES outputs with little actual entropy. Since entropy is impossible to measure, instead the design is intended to catch single points of failure in the ES such as the failure modes listed above. If they were run on direct ES outputs, the health checks would do this quite well. In fact, no repeating pattern with a period shorter than 12 bits can pass the health checks.

However, the health checks are performed after the optional XOR filter and synchronization logic. The ratio of the frequencies between the self-clocking ES and the synchronous region is not an exact integer, and will drift over time. Hence the number of ES samples that are included in each sample crossing the clock boundary will vary.

For example, if the ES is "stuck at 1", then the output of the XOR filter will toggle between 0 and 1. The sampled output will oscillate in a more complicated pattern, depending on how many ES outputs were accumulated in each sample. If the frequency ratio is constant and near 3.3, then the sampled output fails health checks by a margin of 5 samples. If the frequency ratio varies slightly, or the ES is only mostly stuck at 1, then the part may pass the health checks despite having little entropy. In our experiments, many samples pass the health checks even if the ES is 96% stuck at 1. Such a failure would go undetected, and would bring the system outside its design margins. Since production parts cannot examine the ES's raw output, software would not be able to detect this failure either.

Still, after the XOR filter and clock domain crossing, the samples in this failure mode have a Shannon entropy rate of nearly 0.4, with min-entropy only slightly lower. While this is less than the design margin of 0.5, the system's conservative initialization allows it to come up securely with min-entropy rate of 0.004, two orders of magnitude less than this failure allows.

The first generation of the RNG does not use the XOR filter. In this case, ES samples will go directly through to the health checks. However, the output of the ES is still under-sampled when crossing the clock boundary. That is, some of its output bits will be used, and others dropped. But the health checks are more effective without the XOR filter.

Without the XOR filter, a failure such as "mostly stuck at 1" will certainly be caught. Instead, the ES would need to oscillate at a slightly inconsistent rate to have an undetected failure. This failure mode seems less likely, though it might happen if the feedback circuit's step size were somehow far too large. However, without the XOR filter, the health checks will not be forgiving of bias in the entropy source. Any part which is biased by more than 57% ones to 43% zeros (or vice versa) is likely to fail BIST.

These concerns can be resolved by having the health checks operate on all the ES output bits directly. In future versions of the RNG, the ES output will be deserialized, and then sampled in parallel into the synchronous region. This newer logic will therefore provide most or all of the raw ES output to the health checks, and avoid this issue.

3.2.3 Swellness check

The swellness check serves three main purposes.

- It causes the first 129 healthy samples from the ES– more than 32 kilobits – to be conditioned into the DRBG's key during BIST. Thus, it will saturate its 128-bit entropy pool even if those samples have a min-entropy rate as low as 0.004.
- It prevents the RNG from passing BIST unless at least 129 of the first 256 samples are healthy.
- It prevents the system from remaining mostly unhealthy for too long.

Swellness also protects the reseed logic, but only in the long term. Reseeds happen every few blocks, but if users are not consuming much entropy, then the time between reseeds may be long. During this time, the ES's capacitors might discharge, and when the ES is turned back on, it might generate poor data. If most of this data fails the health checks, then the swellness check will eventually fail, so that more healthy samples are required. Once this occurs, the ES will need to warm up enough to produce mostly healthy samples. In this (entirely hypothetical) case, some reseeds will be stronger than others. Intel has told us that worst case simulations suggest that only the first 256 bits could be affected, and the warm-up effect cannot be detected in real silicon.

Reseeding preserves the old seed's entropy. Therefore, if the initial seed is strong, poor reseeds will not weaken it. The DRBG reseeds much more often than NIST SP 800-90A requires, and some weak reseeds are not a concern so long as strong ones happen occasionally.

3.2.4 Conditioning data for seeding/reseeding the DRBG

Entropy conditioning is done via two independent AES-CBC-MAC chains, one for the generator's key and one for its counter. AES-CBC-MAC should be suitable as an entropy extractor, and allows reuse of the module's AES hardware. Importantly, the conditioner accumulates at least 129 healthy samples (33,024 bits) for the DRBG's key during BIST, so even if the entropy rate is low, the generator will be in a secure state before it returns any data. We see no problems with this conditioner.

Under moderate load, the generator will reseed before each 128-bit output, so that the output is information-theoretically random if the entropy rate of healthy samples is at least 0.25. Under heavy load, if the DRBG's state were somehow compromised, the conditioning logic would restore it to a strong state if the ES achieves its designed entropy rate of at least 0.5.

3.2.5 Post-processing the data with the DRBG

The DRBG is based on AES in counter mode, per the NIST SP 800-90A recommendations. It is a theoretically sound, conservative design. Under moderate load, its output should be information-theoretically random. Under heavy load, it should provide security equivalent to 128-bit AES, even against an attacker who can see some of its outputs and, after a good reseed, force the ES to output nonrandom, known values.

3.2.6 Clock gating

The RNG supports clock gating to reduce power consumption. If no application requests entropy for a short time, the RNG will freeze its clock and stop the ES. An area of concern with this approach is that the charge on capacitors may dissipate when the ES is not operating, which might affect the quality of the entropy output by the ES when restarted. As stated in Section 3.2.3, simulations and tests by Intel suggest that this is not an issue, as the ES resumes normal operation quickly. In addition, there should be sufficient entropy in the DRBG from the initial seeding during BIST.

4 Empirical Tests

We did not have access to Ivy Bridge parts, so Intel provided us with testing data from pre-production chips. These chips allow access to the raw ES output, a capability which is disabled in production chips. Even so, in normal operation some data is lost crossing the clock boundary. For easier analysis, Intel performed extensive testing with the ES clock synchronized to the system clock, so that all the output could be collected. We also received data from test chips with special hardware to collect the entire ES output, and from chips running in the normal operational mode with the XOR filter and clock-domain-crossing logic in place.

We analyzed all the data files that Intel provided, but in this report we will focus on the data collected with the ES clock synchronized, because this data was collected from a wide variety of chips and shows the most interesting features.

4.1 Basic statistical tests

We ran a number of statistical tests on the entropy source data that Intel provided us. For the most basic test, we measured the bias and serial correlations of the data. These measurements are summarized in Figure 5.

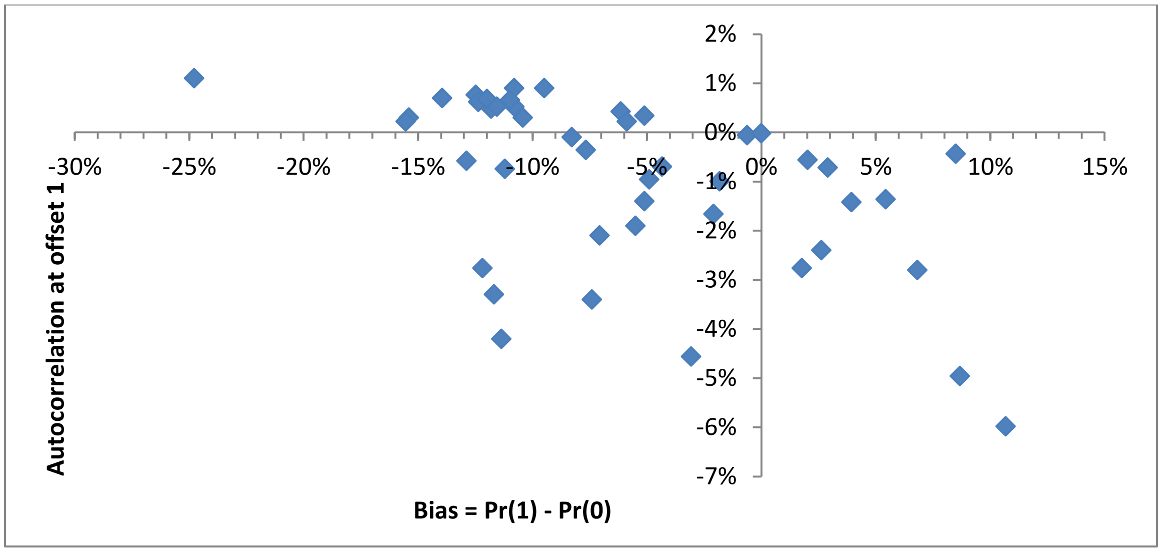

Figure 5: Bias and autocorrelation in the data

Figure 5 shows the bias and serial correlation in the data we received. Each diamond represents data from a single chip. Serial correlations in this data are relatively small, at most a few percent. Single-bit bias is a bigger problem, with 12% typical and an outlier at almost 25%.

The data had serial correlations at higher offsets as well. A small anti-correlation (1-2%) is expected due to the negative feedback circuit. Instead, we saw varying positive and negative autocorrelations at longer offsets, all on the order of 1%. These autocorrelations are symptomatic of a faint "ringing" within the system. In order to investigate this ringing behavior, we looked at the Fourier transforms of the data.

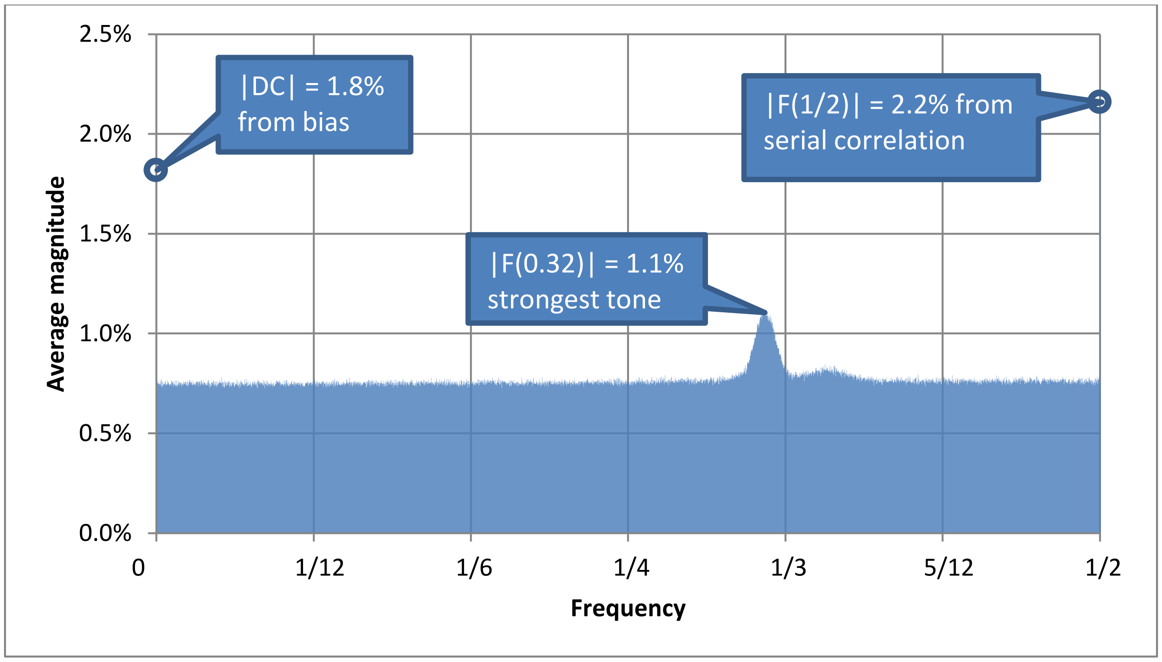

Figure 6: Fourier transform of data from one part, showing ringing

Figure 6 shows the Fourier transform of one particular data file. This file shows a small bias, an offset-1 autocorrelation, and a noticeable tone near 1/3. This tone is symptomatic of "ringing" behavior with period slightly over 3 bits.

Compared to the other parts, this one shows a smaller bias and a stronger tone than most. It is otherwise representative. Most parts show a flat spectrum with a few faint tones, plus a single-bit bias and a small serial correlation.

4.2 Entropy, bit prediction, and Markov modeling

We constructed Markov models of each sample file that Intel provided. The state of our model comprised the last 12 bits of output; with a larger state, errors crept in due to having too few samples in each state. From this model, we used the Markov-Rényi [11] algorithm to estimate the worst-case min-entropy. This algorithm is much more pessimistic than sampling the min-entropy directly. In particular, it assumes that an adversary can wait until the system is in a state which produces low entropy, which is not actually possible.

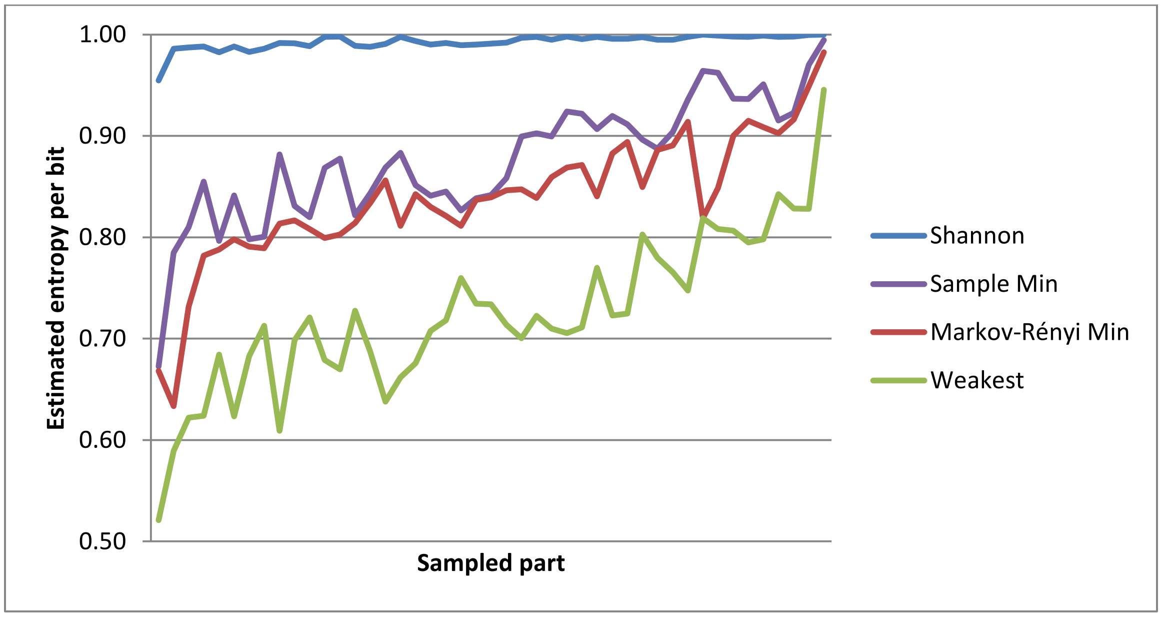

Figure 7: Entropy measurements on 45 parts

Figure 7 summarizes the results of these entropy measurements. The horizontal axis is the part which was sampled, sorted by the mean of the entropy measurements. The vertical axis is the entropy per bit according to the indicated metrics.

The top line shows the Shannon entropy, computed over the bytes of the entropy source's output. This measurement shows how much entropy the entropy source's outputs have on average. These results are clearly very good.

The second line shows the sampled min-entropy, measured on 13-bit samples (our Markov model's state, plus its output). This is a relatively accurate model of the difficulty of guessing the output of the generator.

The third line shows the Markov-Rényi min-entropy, modeling the next bit of output using the previous 12 bits. This is a more conservative model of the difficulty of guessing the output of the generator.

The lowest line shows the entropy of the weakest states of the Markov model. This line is much lower than the Markov-Rényi line because the generator does not stay in these weak states for long. Therefore, they do not pose a threat to the security of the RNG.

Three major effects are visible in this chart.

- Some of the parts generate lower-quality entropy due to correlation and bias. This has a much more dramatic effect on the min-entropy than on the Shannon entropy. Still, even the most pessimistic estimates of the worst parts are higher than the 50% threshold discussed in Section 2.4.1.
- The data for these CPUs was collected by placing them in a testing machine, rather than by issuing debugging commands from the CPU itself. As a result, some runs (in particular, the spike near the right side of the plot) show artifacts where the testing machine began reading before the ES turned on. After discussing these artifacts with Intel, we believe that they cannot happen during operation.
- The ringing behavior of some parts means that after a certain 2- or 3-bit pattern has occurred, it is slightly more likely to occur again. This ringing behavior is faint, but even so it can reduce the Markov-Rényi entropy estimate by up to 10%.

4.3 Charge tracking

We attempted to track the charge on the capacitors, in part to predict the output of the circuit and in part to verify that our model was correct. We did not expect our predictions to be very accurate, because even if the system ideally matched our model, it would have high entropy. But we did expect the estimated charge on the capacitors to correlate to the output.

We saw the expected correlation (and lack of predictability) in the data collected from a running Ivy Bridge CPU, but not in the data collected from an external tester. We suspect that the link between the CPU and the tester is not fast enough to transmit every sample, thwarting our attempts to track the capacitor charge on these parts.

4.4 Randomness tests

We tested the final, post-processed outputs of the RNG with the NIST SP 800-22 statistical test suite [12] in order to make sure that there are no glaring flaws in the generator. As expected, the outputs easily passed the entire test suite.

5 Conclusions

Overall, the Ivy Bridge RNG is a robust design with a large margin of safety that ensures good random data is generated even if the ES is not operating as well as predicted.

The ES is an interesting design based on the random resolution of a circuit designed to seek out its metastable state. Intel has modeled and tested the ES extensively and believes that within a wide range of conditions, including typical PVT variations, the ES generates at least 0.5 bits of entropy per sample. Our modeling and testing agree with this assessment.

The health and swellness tests are generally well designed and should identify badly broken entropy sources. They are most effective when performed directly on ES output, and the optional XOR filtering and clock boundary crossing logic weakens them. However, the large number of ES samples mixed into the DRBG AES key during BIST and reseeding should compensate for an ES which is generating data with even very low entropy.

The DRBG construction is sound. We found no issues with the entropy conditioning, reseeding, and random data generation logic.

Because the Ivy Bridge RNG is implemented as an instruction in the CPU, it is much simpler to use than other hardware-based RNGs and avoids the need for additional software layers that could introduce bugs. Applications do need to perform some simple checks, however, notably testing the carry flag to detect failures and testing to ensure safe operation if run on a CPU without an integrated RNG. In addition, developers should be aware that the RNG instruction can be virtualized, and could be intercepted to deliver nonrandom data to applications. Of course, a malicious hypervisor can ruin applications' security in numerous other, simpler ways.

In conclusion, we believe the Ivy Bridge RNG is well designed, with a wide margin of safety, and the output is appropriate to use directly for cryptographic keys, secret nonces, and other sensitive values. However, the most prudent approach is always to combine any other available entropy sources to avoid having a single point of failure. For OS implementations that maintain an entropy pool, we recommend the frequent incorporation of RNG outputs as an additional input into the OS entropy pool. The exceptional performance of the Intel design also enables direct mixing of data from the Ivy Bridge RNG outputs with output delivered from other RNGs. In all cases, users should check the carry flag after each call to the RNG to verify that it is working properly and the random data received is valid.

6 Bibliography

[1] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal,* vol. 27, pp. 379–423, 623-656, 1948.

[2] E. Barker and J. Kelsey, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators,* NIST Special Publication 800-90A, January 2012.

[3] B. Schneier, *Security Pitfalls in Cryptography,* Counterpane Systems, 1998.

[4] "DSA-1571-1- openssl -- predictable random number generator," Debian, 13 May 2008. [Online]. Available: http://www.debian.org/security/2008/dsa-1571. [Accessed 1 February 2012].

[5] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung and C. Wachter, "Ron was wrong, Whit is right," *IACR eprint archive,* vol. 064, 2012.

[6] D. Bleichenbacher, *On the generation of one-time keys in DL signature schemes,* IEEE P1363 Working Group Meeting, November 2000.

[7] D. J. Johnston, "Mircoarchitecture Specification (MAS) for PP-DRNG," Intel Corporation (unpublished), V1.4, 2009.

[8] C. E. Dike, "3 Gbps Binary RNG Entropy Source," Intel Corporation (unpublished), 2011.

[9] C. E. Dike and S. Gueron, "Digital Symmetric Random Number Generator Mathematics," Intel Corporation (unpublished), 2009.

[10] M. Dworkin, "Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality," NIST Special Publication 800-38C, May 2004.

[11] Z. Rached, F. Alajaji and L. Campbell, "Rényi's Entropy Rate For Discrete Markov Sources," 1999.

[12] NIST, "NIST Special Publication 800-22rev1a," 11 August 2010. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html. [Accessed 2 February 2012].