Intel® Digital Random Number Generator (DRNG)

Software Implementation Guide

Revision 2.0

May 15, 2014

Revision History

Revision	Revision History	Date
0.1	Beta Release Revision	February 14, 2011
0.2	Release Revision	April 24, 2011
0.3	Release Revision	June 9, 2011
1.0	Ivy Bridge Release Revision	May 1, 2012
1.1	Replace code samples for _rdrand16_step, _rdrand32_step, and _rdrand64_step	August 7, 2012
2.0	Broadwell Release Revision, adding information on RDSEED. Revamping code samples, updated performance information, and expanded developer recommendations.	May 15, 2014

Contents

1 Introduction

Intel® Secure Key, code-named Bull Mountain Technology, is the Intel name for the Intel® 64 and IA-32 Architectures instructions RDRAND and RDSEED and the underlying Digital Random Number Generator (DRNG) hardware implementation. Among other things, the DRNG using the RDRAND instruction is useful for generating high-quality keys for cryptographic protocols, and the RSEED instruction is provided for seeding software-based pseudorandom number generators (PRNGs)

This Digital Random Number Generator Software Implementation Guide is intended to provide a complete source of technical information on RDRAND usage, including code examples. This document includes the following sections:

Section 2: Random Number Generator (RNG) Basics and Introduction to the DRNG. This section describes the nature of an RNG and its pseudo- (PRNG) and true- (TRNG) implementation variants, including modern cascade construction RNGs. We then present the DRNG's position within this broader taxonomy.

Section 3: DRNG Overview. In this section, we provide a technical overview of the DRNG, including its component architecture, robustness features, manner of access, performance, and power requirements.

Section 4: RDRAND and RDSEED Instruction Usage. This section provides reference information on the RDRAND and RDSEED instructions and code examples showing its use. This includes platform support verification and suggestions on DRNG-based libraries.

Programmers who already understand the nature of RNGs may refer directly to section 4 for instruction references and code examples. RNG newcomers who need some review of concepts to understand the nature and significance of the DRNG can refer to section 2. Nearly all developers will want to look at section 3, which provides a technical overview of the DRNG.

2 RNG Basics and Introduction to the DRNG

The Digital Random Number Generator, using the RDRAND instruction, is an innovative hardware approach to high-quality, high-performance entropy and random number generation. To understand how it differs from existing RNG solutions, this section details some of the basic concepts underlying random number generation.

2.1 Random Number Generators (RNGs)

An RNG is a utility or device of some type that produces a sequence of numbers on an interval [min, max] such that values appear unpredictable. Stated a little more technically, we are looking for the following characteristics:

- Each new value must be *statistically independent* of the previous value. That is, given a generated sequence of values, a particular value is not more likely to follow after it as the next value in the RNG's random sequence.
- The overall distribution of numbers chosen from the interval is *uniformly distributed.* In other words, all numbers are equally likely and none are more "popular" or appear more frequently within the RNG's output than others.
- The sequence is *unpredictable*. An attacker cannot guess some or all of the values in a generated sequence. Predictability may take the form of *forward* prediction (future values) and backtracking (past values).

Since computing systems are by nature deterministic, producing quality random numbers that have these properties (statistical independence, uniform distribution, and unpredictability) is much more difficult than it might seem. Sampling the seconds value from the system clock, a common approach, may seem random enough, but process scheduling and other system effects may result in some values occurring far more frequently than others. External entropy sources like the time between a user's keystrokes or mouse movements may likewise, upon further analysis, show that values do not distribute evenly across the space of all possible values; some values are more likely to occur than others, and certain values almost never occur in practice.

Beyond these requirements, some other desirable RNG properties include:

- The RNG is fast in returning a value (i.e., low response time) and can service a large number of requests within a short time interval (i.e., highly scalable).
- The RNG is secure against attackers who might observe or change its underlying state in order to predict or influence its output or otherwise interfere with its operation.

2.2 Pseudo-Random Number Generators (PRNGs)

One widely used approach for achieving good RNG statistical behavior is to leverage mathematical modeling in the creation of a Pseudo-Random Number Generator. A PRNG is a deterministic algorithm, typically implemented in software that computes a sequence of numbers that "look" random. A PRNG requires a seed value that is used to initialize the state of the underlying model. Once seeded, it can then generate a sequence of numbers that exhibit good statistical behavior.

PRNGs exhibit periodicity that depends on the size of its internal state model. That is, after generating a long sequence of numbers, all variations in internal state will be exhausted and the sequence of numbers to follow will repeat an earlier sequence. The best PRNG algorithms available today, however, have a period that is so large this weakness can practically be ignored. For example, the Mersenne Twister MT19937 PRNG with 32-bit word length has a periodicity of 2^{19937}-1. (1)

A key characteristic of all PRNGs is that they are *deterministic*. That is, given a particular seed value, the same PRNG will always produce the exact same sequence of "random" numbers. This is because a PRNG is computing the next value based upon a specific internal state and a specific, well-defined algorithm. Thus, while a generated sequence of values exhibit the statistical properties of randomness (independence, uniform distribution), overall behavior of the PRNG is entirely predictable.

In some contexts, the deterministic nature of PRNGs is an advantage. For example, in some simulation and experimental contexts, researchers would like to compare the outcome of different approaches using the same sequence of input data. PRNGs provide a way to generate a long sequence of random data inputs that are repeatable by using the same PRNG, seeded with the same value.

In other contexts, however, this determinism is highly undesirable. Consider a server application that generates random numbers to be used as cryptographic keys in data exchanges with client applications over secure communication channels. An attacker who knew the PRNG in use and also knew the seed value (or the algorithm used to obtain a seed value) would quickly be able to predict each and every key (random number) as it is generated. Even with a sophisticated and unknown seeding algorithm, an attacker who knows (or can guess) the PRNG in use can deduce the state of the PRNG by observing the sequence of output values. After a surprisingly small number of observations (e.g., 624 for Mersenne Twister MT19937), each and every subsequent value can be predicted. For this reason, PRNGs are considered to be cryptographically insecure.

PRNG researchers have worked to solve this problem by creating what are known as Cryptographically Secure PRNGs (CSPRNGs). Various techniques have been invented in this domain, for example, applying a cryptographic hash to a sequence of consecutive integers, using a block cipher to encrypt a sequence of consecutive integers ("counter mode"), and XORing a stream of PRNG-generated numbers with plaintext ("stream cipher"). Such approaches improve the problem of inferring a PRNG and its state by greatly increasing its computational complexity, but the resulting values may or may not exhibit the correct statistical properties (i.e., independence, uniform distribution) needed for a robust random number generator. Furthermore, an attacker could discover any deterministic algorithm by various means (e.g., disassemblers, sophisticated memory attacks, a disgruntled employee). Even more common, attackers may discover or infer PRNG seeding by narrowing its range of possible values or snooping memory in some manner. Once the deterministic algorithm and its seed is known, the attacker may be able to predict each and every random number generated, both past and future.

2.3 True Random Number Generators (TRNGs)

For contexts where the deterministic nature of PRNGs is a problem to be avoided (e.g., gaming and computer security), a better approach is that of True Random Number Generators.

Instead of using a mathematical model to deterministically generate numbers that look random and have the right statistical properties, a TRNG extracts randomness (entropy) from a physical source of some type and then uses it to generate random numbers. The physical source is also referred to as an *entropy source* and can be selected among a wide variety of physical phenomenon naturally available, or made available, to the computing system using the TRNG. For example, one can attempt to use the time between user key strokes or mouse movements as an entropy source. As pointed out earlier, this technique is crude in practice and resulting value sequences generally fail to meet desired statistical properties with rigor. What to use as an entropy source in a TRNG is a key challenge facing TRNG designers.

Beyond statistical rigor, it is also desirable for TRNGs to be fast and scalable (i.e., capable of generating a large number of random numbers within a small time interval). This poses a serious problem for many TRNGs because sampling an entropy source external to the computing system typically requires device I/O and long delay times relative to the processing speeds of today's computer systems. In general, sampling an entropy source in TRNGs is slow compared to the computation required by a PRNG to simply calculate its next random value. For this reason, PRNGs characteristically provide far better performance than TRNGs and are more scalable.

Unlike PRNGs, however, TRNGs are not deterministic. That is, a TRNG need not be seeded, and its selection of random values in any given sequence is highly unpredictable. As such, an attacker cannot use observations of a particular random number sequence to predict subsequent values in an effective way. This property also implies that TRNGs have no periodicity. While repeats in random sequence are possible (albeit unlikely), they cannot be predicted in a manner useful to an attacker.

2.4 Cascade Construction RNGs

A common approach used in modern operating systems (e.g., Linux* (2)) and cryptographic libraries is to take input from an entropy source in order to supply a buffer or pool of entropy (refer to Figure 1). This entropy pool is then used to provide nondeterministic random numbers that periodically seed a cryptographically secure PRNG (CSPRNG). This CSPRNG provides cryptographically secure random numbers that appear truly random and exhibit a well-defined level of computational attack resistance.

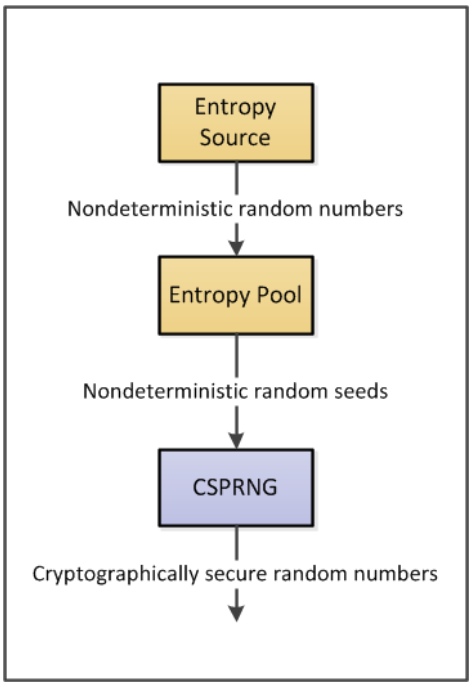

Figure 1. Cascade Construction Random Number Generator

A key advantage of this scheme is performance. It was noted above that sampling an entropy source is typically slow since it often involves device I/O of some type and often additional waiting for a real-time sampling event to transpire. In contrast, CSPRNG computations are fast since they are processor-based and avoid I/O and entropy source delays.This approach offers improved performance: a slow entropy source periodically seeding a fast CSPRNG capable of generating a large number of random values from a single seed.

While this approach would seem ideal, in practice it often falls far short. First, since the implementation is typically in software, it is vulnerable to a broad class of software attacks. For example, considerable state requirements create the potential for memory-based attacks or timing attacks. Second, the approach does not solve the problem of what entropy source to use. Without an external source of some type, entropy quality is likely to be poor. For example, sampling user events (e.g., mouse, keyboard) may be impossible if the system resides in a large data center. Even with an external entropy source, entropy sampling is likely to be slow, making seeding events less frequent than desired.

2.5 Introducing the Digital Random Number Generator (DRNG)

The Digital Random Number Generator (DRNG) is an innovative hardware approach to high-quality, high-performance entropy and random number generation. It is composed of the new Intel 64 Architecture instructions RDRAND and RDSEED and an underlying DRNG hardware implementation.

With respect to the RNG taxonomy discussed above, the DRNG follows the cascade construction RNG model, using a processor resident entropy source to repeatedly seed a hardware-implemented CSPRNG. Unlike software approaches, it includes a high-quality entropy source implementation that can be sampled quickly to repeatedly seed the CSPRNG with high-quality entropy. Furthermore, it represents a self-contained hardware module that is

isolated from software attacks on its internal state. The result is a solution that achieves RNG objectives with considerable robustness: statistical quality (independence, uniform distribution), highly unpredictable random number sequences, high performance, and protection against attack.

This method of digital random number generation is unique in its approach to true random number generation in that it is implemented in the processor's hardware and can be utilized through instructions added to the Intel 64 instruction set. As such, response times are comparable to those of competing PRNG approaches implemented in software. The approach is scalable enough for even demanding applications to use it as an exclusive source of random numbers and not merely a high quality seed for a software-based PRNG. Software running at all privilege levels can access random numbers through the instruction set, bypassing intermediate software stacks, libraries, or operating system handling.

The use of RDRAND and RDSEED leverages a variety of cryptographic standards to ensure the robustness of its implementation and to provide transparency in its manner of operation. These include NIST SP800-90A, B, and C, FIPS-140-2, and ANSI X9.82. Compliance to these standards makes the Digital Random Number Generation a viable solution for highly regulated application domains in government and commerce.

Section 3 describes digital random number generation in detail. Section 4 describes use of RDRAND and RDSEED, the Intel instruction set extensions for using the DRNG.

2.6 Applications for the Digital Random Number Generator

Information security is a key application that utilizes the DRNG. Cryptographic protocols rely on RNGs for generating keys and fresh session values (e.g., a nonce) to prevent replay attacks. In fact, a cryptographic protocol may have considerable robustness but suffer from widespread attack due to weak key generation methods underlying it (e.g., the Debian*/OpenSSL* fiasco (3)). The DRNG can be used to fix this weakness, thus significantly increasing cryptographic robustness.

Closely related are government and industry applications. Due to information sensitivity, many such applications must demonstrate their compliance with security standards like FISMA, HIPPA, PCIAA, etc. RDRAND has been engineered to meet existing security standards like NIST SP800-90, FIPS 140-2, and ANSI X9.82, and thus provides an underlying RNG solution that can be leveraged in demonstrating compliance with information security standards.

Other uses of the DRNG include:

- Communication protocols
- Monte Carlo simulations and scientific computing
- Gaming applications
- Bulk entropy applications like secure disk wiping or document shredding
- Protecting online services against RNG attacks
- Seeding software-based PRNGs of arbitrary width

3 DRNG Overview

In this section, we describe in some detail the components of the DRNG using the RDRAND and RDSEED instructions and their interaction.

3.1 Processor View

Figure 2 provides a high-level schematic of the RDRAND and RDSEED Random Number Generators. As shown, the DRNG appears as a hardware module on the processor. An interconnect bus connects it with each core.

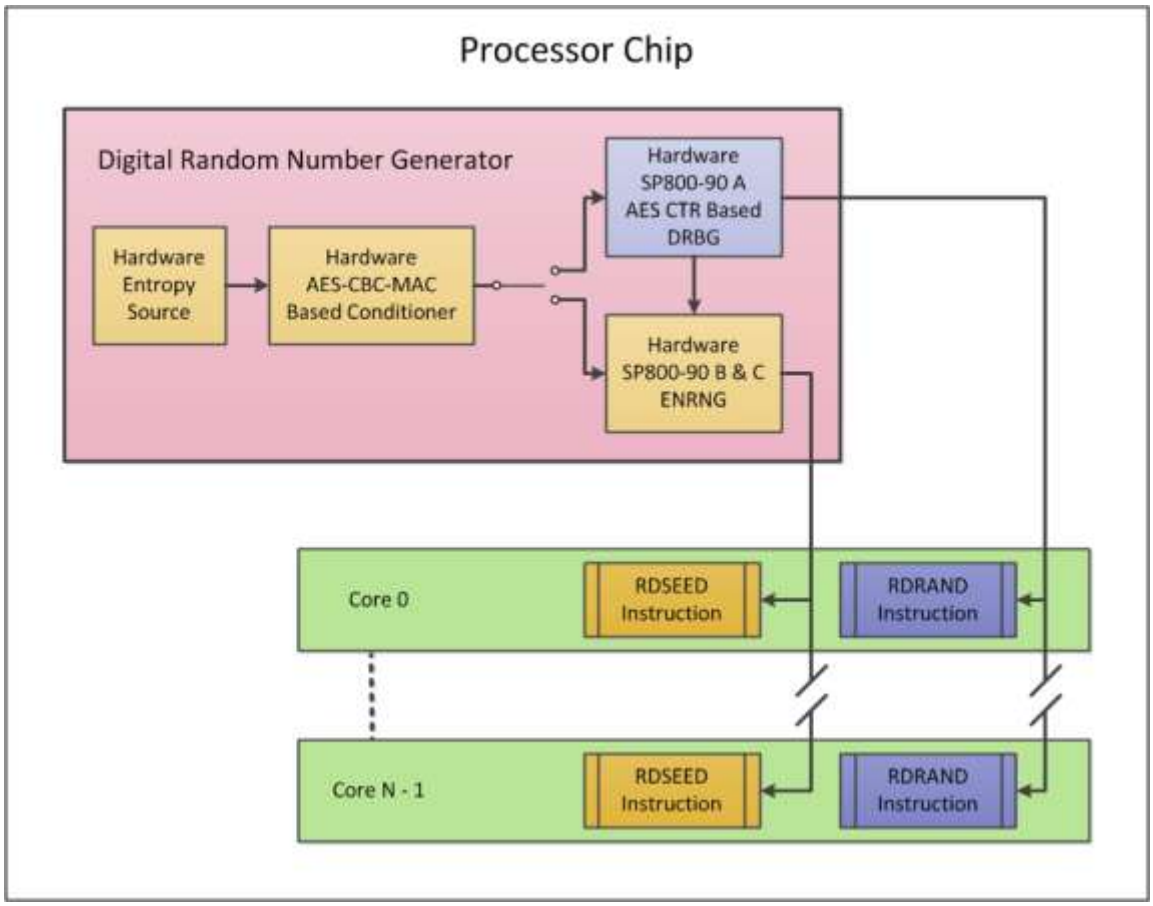

Figure 2. Digital Random Number Generator design

The RDRAND and RDSEED instructions (detailed in section 4) are handled by microcode on each core. This includes an RNG microcode module that handles interactions with the DRNG hardware module on the processor.

3.2 Component Architecture

As shown in Figure 3, the DRNG can be thought of as three logical components forming an asynchronous production pipeline: an entropy source (ES) that produces random bits from a nondeterministic hardware process at around 3 Gbps, a conditioner that uses AES (4) in CBC-MAC (5) mode to distill the entropy into high-quality nondeterministic random numbers, and two parallel outputs:

1. A deterministic random bit generator (DRBG) seeded from the conditioner.
2. An enhanced, nondeterministic random number generator (ENRNG) that provides seeds from the entropy conditioner.

Note that the conditioner *does not* send the same seed values to both the DRBG and the ENRNG. This pathway can be thought of as an alternating switch, with one seed going to the DRGB and the next seed going to the ENRNG. This construction ensures that a software application can *never* obtain the value used to seed the DRBG, nor can it launch a Denial of Service (DoS) attack against the DRBG through repeated executions of the RDSEED instruction.

The conditioner can be equated to the entropy pool in the cascade construction RNG described previously. However, since it is fed by a high-quality, high-speed, continuous stream of entropy that is fed faster than downstream processes can consume, it does not need to maintain an entropy pool. Instead, it is always conditioning fresh entropy independent of past and future entropy.

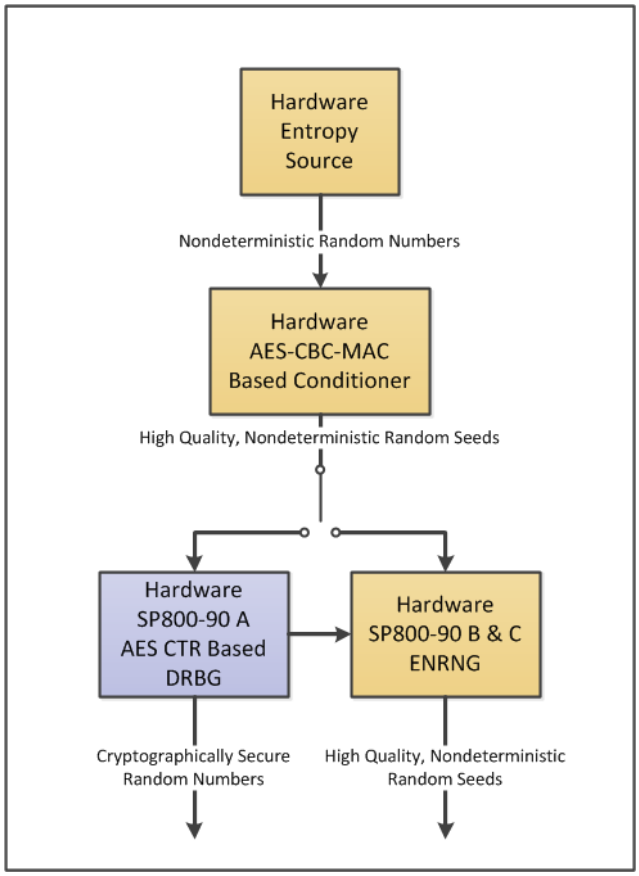

Figure 3. DRNG Component Architecture

The final two stages are:

1. A hardware CSPRNG that is based on AES in CTR mode and is compliant with SP800-90A. In SP800-90A terminology, this is referred to as a DRBG (Deterministic Random Bit Generator), a term used throughout the remainder of this document.

2. An ENRNG (Enhanced Nondeterministic Random Number Generator) that is compliant with SP800-90B and C.

3.2.1 Entropy Source (ES)

The all-digital Entropy Source (ES), also known as a non-deterministic random bit generator (NRBG), provides a serial stream of entropic data in the form of zeroes and ones.

The ES runs asynchronously on a self-timed circuit and uses thermal noise within the silicon to output a random stream of bits at the rate of 3 GHz. The ES needs no dedicated external power supply to run, instead using the same power supply as other core logic. The ES is designed to function properly over a wide range of operating conditions, exceeding the normal operating range of the processor.

Bits from the ES are passed to the conditioner for further processing.

3.2.2 Conditioner

The conditioner takes pairs of 256-bit raw entropy samples generated by the ES and reduces them to a single 256-bit conditioned entropy sample using AES-CBC-MAC. This has the effect of distilling the entropy into more concentrated samples.

AES, Advanced Encryption Standard, is defined in the FIPS-197 Advanced Encryption Standard (4). CBC-MAC, Cipher Block Chaining - Message Authentication Code, is defined in NIST SP 800-38A Recommendation for Block Cipher Modes of Operation (5).

The conditioned entropy is output as a 256-bit value and passed to the next stage in the pipeline to be used as a DRBG seed value.

3.2.3 Deterministic Random Bit Generator (DRBG)

The role of the deterministic random bit generator (DRBG) is to "spread" a conditioned entropy sample into a large set of random values, thus increasing the amount of random numbers available by the hardware module. This is done by employing a standards-compliant DRBG and continuously reseeding it with the conditioned entropy samples.

The DRBG chosen for this function is the CTR_DRBG defined in section 10.2.1 of NIST SP 800-90A (6), using the AES block cipher. Values that are produced fill a FIFO output buffer that is then used in responding to RDRAND requests for random numbers.

The DRBG autonomously decides when it needs to be reseeded to refresh the random number pool in the buffer and is both unpredictable and transparent to the RDRAND caller. An upper bound of 511 128-bit samples will be generated per seed. That is, no more than 511*2=1022 sequential DRNG random numbers will be generated from the same seed value.

3.2.4 Enhanced Nondeterministic Random Number Generator

The role of the enhanced nondeterministic random number generator is to make conditioned entropy samples directly available to software for use as seeds to other software-based DRBGs. Values coming out of the ENRNG have multiplicative brute-force prediction resistance, which means that samples can be concatenated and the brute-force prediction resistance will scale with them. When two 64-bit samples are concatenated together, the resulting 128-bit value will have 128 bits of brute-force prediction resistance ($2^{64} * 2^{64} = 2^{128}$).

This operation can be repeated indefinitely and can be used to easily produce random seeds of arbitrary size. Because of this property, these values can be used to seed a DRBG of any size.

3.2.5 Robustness and Self-Validation

To ensure the DRNG functions with a high degree of reliability and robustness, validation features have been included that operate in an ongoing manner at system startup. These include the DRNG Online Health Tests (OHTs) and Built-In Self Tests (BISTs), respectively. Both are shown in Figure 4.

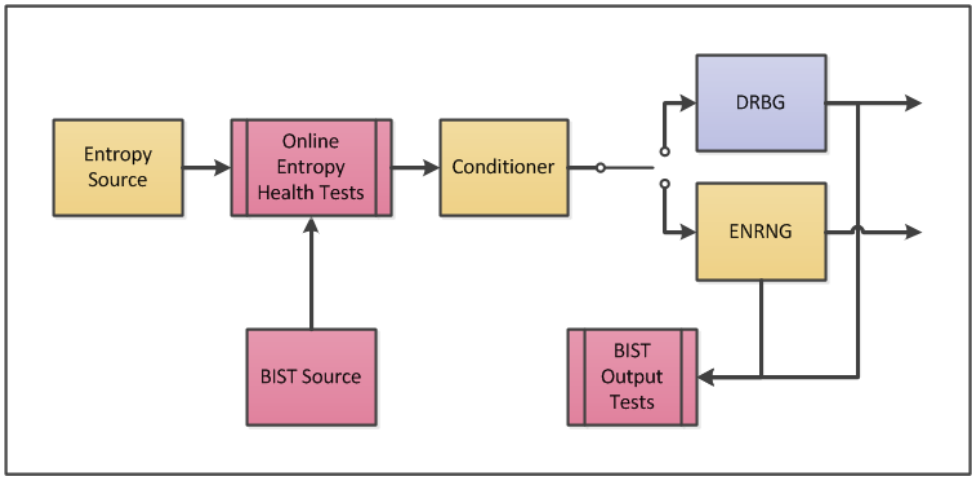

Figure 4. DRNG Self-Validation Components

3.2.6 Online Health Tests (OHTs)

Online Health Tests (OHTs) are designed to measure the quality of entropy generated by the ES using both per sample and sliding window statistical tests in hardware.

Per sample tests compare bit patterns against expected pattern arrival distributions as specified by a mathematical model of the ES. An ES sample that fails this test is marked "unhealthy." Using this distinction, the conditioner can ensure that at least two healthy samples are mixed into each seed. This defends against hardware attacks that might seek to reduce the entropic content of the ES output.

Sliding window tests look at sample health across many samples to verify they remain above a required threshold. The sliding window size is large (65536 bits) and mechanisms ensure that the ES is operating correctly overall before it issues random numbers. In the rare event that the DRNG fails during runtime, it would cease to issue random numbers rather than issue poor quality random numbers.

3.2.7 Built-In Self Tests (BISTs)

Built-In Self Tests (BISTs) are designed to verify the health of the ES prior to making the DRNG available to software. These include Entropy Source Tests (ES-BIST) that are statistical in nature and comprehensive test coverage of all the DRNG's deterministic downstream logic through BIST Known Answer Tests (KAT-BIST).

ES-BIST involves running the DRNG for a probationary period in its normal mode before making the DRNG available to software. This allows the OHTs to examine ES sample health

for a full sliding window (256 samples) before concluding that ES operation is healthy. It also fills the sliding window sample pipeline to ensure the health of subsequent ES samples, seeds the PRNG, and fills the output queue of the DRNG with random numbers.

KAT-BIST tests both OHT and end-to-end correctness using deterministic input and output validation. First, various bit stream samples are input to the OHT, including a number with poor statistical quality. Samples cover a wide range of statistical properties and test whether the OHT logic correctly identifies those that are "unhealthy." During the KAT-BIST phase, deterministic random numbers are output continuously from the end of the pipeline. The BIST Output Test Logic verifies that the expected outputs are received.

If there is a BIST failure during startup, the DRNG will not issue random numbers and will issue a BIST failure notification to the on-processor test circuitry. This BIST logic avoids the need for conventional on-processor test mechanisms (e.g., scan and JTAG) that could undermine the security of the DRNG.

3.3 Instructions

Software access to the DRNG is through the RDRAND and RDSEED instructions, documented in Chapter 3 of (7).

3.3.1 RDRAND

RDRAND retrieves a hardware-generated random value from the SP800-90A compliant DRGB and stores it in the destination register given as an argument to the instruction. The size of the random value (16-, 32-, or 64-bits) is determined by the size of the register given. The carry flag (CF) must be checked to determine whether a random value was available at the time of instruction execution.

Note that RDRAND is available to any system or application software running on the platform. That is, there are no hardware ring requirements that restrict access based on process privilege level. As such, RDRAND may be invoked as part of an operating system or hypervisor system library, a shared software library, or directly by an application.

To determine programmatically whether a given Intel platform supports RDRAND, developers can use the CPUID instruction to examine bit 30 of the ECX register. See Reference (7) for details.

3.3.2 RDSEED

RDSEED retrieves a hardware-generated random seed value from the SP800-90B and C compliant ENRNG and stores it in the destination register given as an argument to the instruction. Like the RDRAND instruction, the size of the random value is determined by the size of the given register, and the carry flag (CF) must be checked to determine whether or not a random seed was available at the time the instruction was executed.

Also like RDRAND, there are no hardware ring requirements that restrict access to RDSEED based on process privilege level.

To determine programmatically whether a given Intel platform supports the RDSEED instruction, developers can use the CPUID instruction to examine bit 18 of the EBX register. See Reference (8) for details.

3.4 DRNG Performance

Designed to be a high performance entropy resource shared between multiple cores/threads, the Digital Random Number Generator represents a new generation of RNG performance.

The DRNG is implemented in hardware as part of the Intel processor. As such, both the entropy source and the DRBG execute at processor clock speeds. Unlike other hardware-based solutions, there is no system I/O required to obtain entropy samples and no off-processor bus latencies to slow entropy transfer or create bottlenecks when multiple requests have been issued.

Random values are delivered directly through instruction level requests (RDRAND and RDSEED). This bypasses both operating system and software library handling of the request. The DRNG is scalable enough to support heavy server application workloads. Within the context of virtualization, the DRNG's stateless design and atomic instruction access mean that RDRAND and RDSEED can be used freely by multiple VMs without the need for hypervisor intervention or resource management.

3.4.1 RDRAND Performance†

In current-generation Intel processors the DRBG runs on a self-timed circuit clocked at 800 MHz and can service a RDRAND transaction (1 Tx) every 8 clocks for a maximum of 100 MTx per second. A transaction can be for a 16-, 32-, or 64-bit RDRAND, and the greatest throughput is achieved with 64-bit RDRANDs, capping the throughput ceiling at 800 MB/sec. These limits are an upper bound on all hardware threads across all cores on the CPU.

Single thread performance is limited by the instruction latencies imposed by the bus infrastructure, which is also impacted in part by clock speed. On real-world systems, a single thread executing RDRAND continuously may see throughputs ranging from 70 to 200 MB/sec, depending on the SPU architecture.

If multiple threads are invoking RDRAND simultaneously, total RDRAND throughput (across all threads) scales approximately linearly with the number of threads until no more hardware threads remain, the bus limits of the processor are reached, or the DRNG interface is fully saturated. Past this point, the maximum throughput is divided equally among the active threads. No threads get starved.

Figure 5 shows the multithreaded RDRAND throughput plotted as a ratio to single thread throughput for six different CPU architectures. The dotted line represents linear scaling. This shows that total RDRAND throughput scales nearly linearly with the number of active threads on the CPU, prior to reaching saturation.

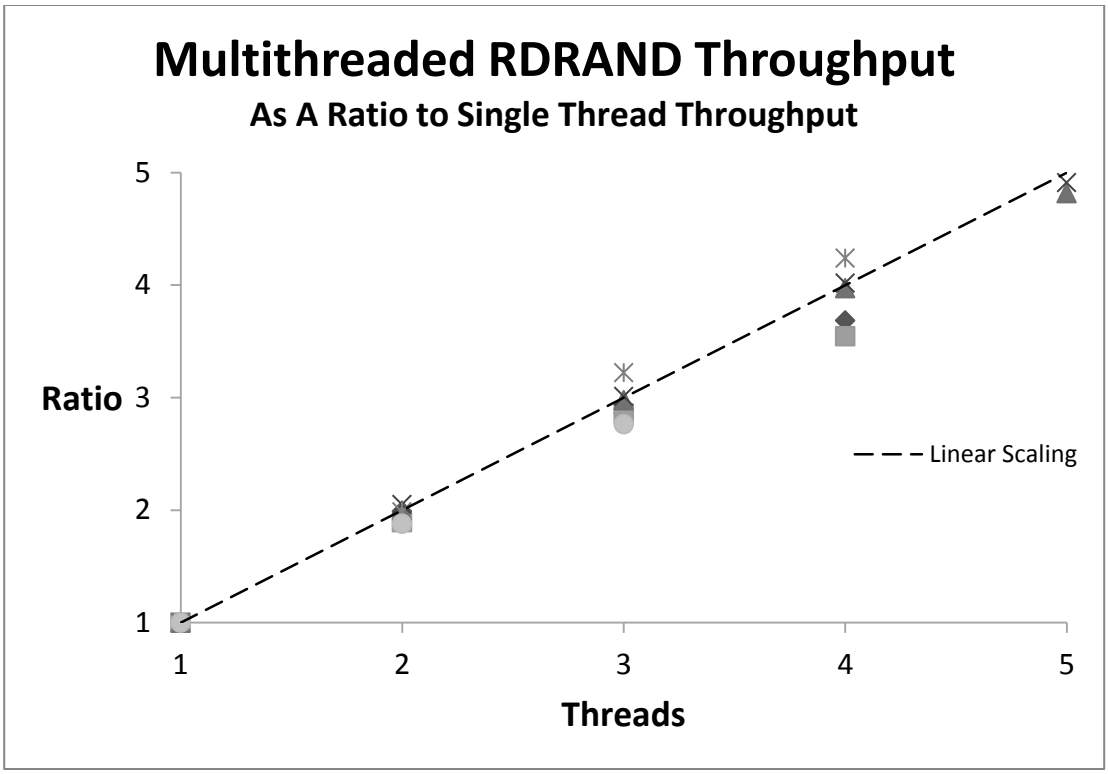

Figure 5. Multithreaded RDRNAD throughput scaling

Figure 6 shows the multithreaded performance of a single system, also as a ratio, up to saturation and beyond. As in Figure 5, total throughput scales nearly linearly until saturation, at which point it reaches a steady state.

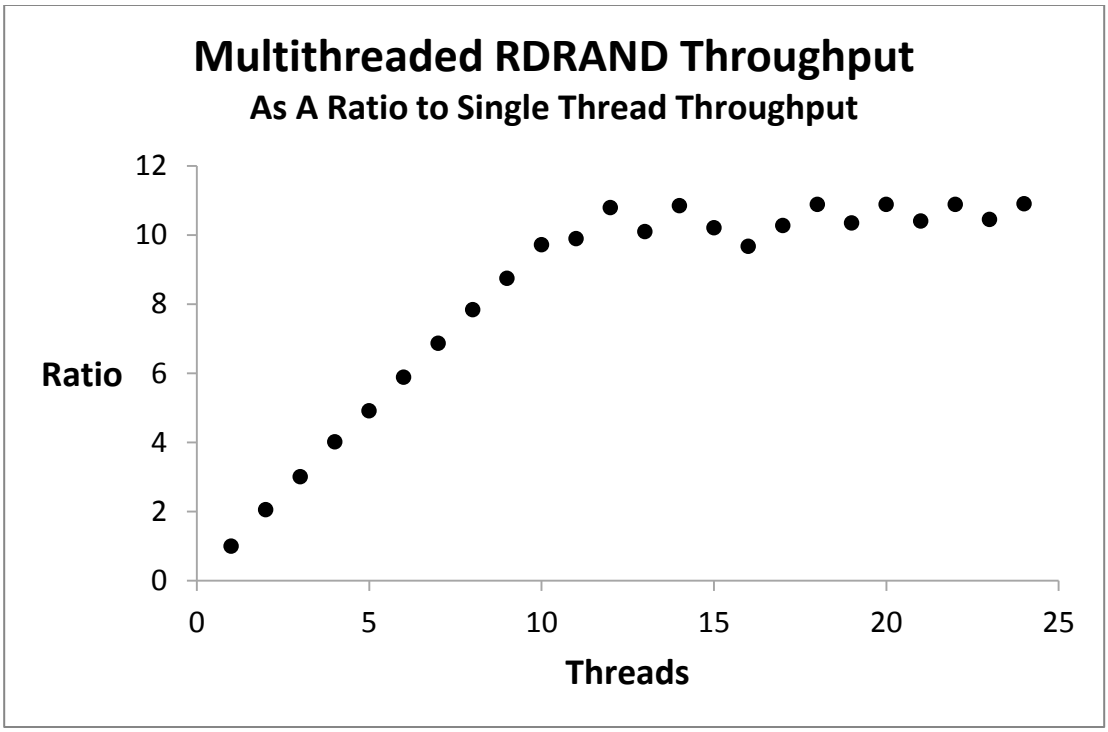

Figure 6. RDRAND throughput past saturation

†Results have been estimated based on internal Intel® analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

3.5 Power Requirements

The DRNG hardware resides on the processor and, therefore, does not need a dedicated power supply to run. Instead, it simply uses the processor's local power supply. As described in section 3.2.1, the hardware is designed to function across a range of process voltage and temperature (PVT) levels, exceeding the normal operating range of the processor.

The DRNG hardware does not impact power management mechanisms and algorithms associated with individual cores. For example, ACPI-based mechanisms for regulating processor performance states (P-states) and processor idle states (C-states) on a per core basis are unaffected.

To save power, the DRNG clock gates itself off when queues are full. This idle-based mechanism results in negligible power requirements whenever entropy computation and post processing are not needed.

4 Instruction Usage

In this section, we provide instruction references for RDRAND and RDSEED and usage examples for programmers. All code examples in this guide are licensed under the new, 3-clause BSD license, making them freely usable within nearly any software context.

For additional details on RDRAND usage and code examples, see Reference (7).

4.1 Determining Processor Support for RDRAND and RDSEED

Before using the RDRAND or RDSEED instructions, an application or library should first determine whether the underlying platform supports the instruction, and hence includes the underlying DRNG feature. This can be done using the CPUID instruction. In general, CPUID is used to return processor identification and feature information stored in the EAX, EBX, ECX, and EDX registers. For detailed information on CPUID, refer to References (7) and (8).

To be specific, support for RDRAND can be determined by examining bit 30 of the ECX register returned by CPUID, and support for RDSEED can be determined by examining bit 31 of the EBX register. As shown in Table 1 (below) and 2-23 in Reference (7), a value of 1 indicates processor support for the instruction while a value of 0 indicates no processor support.

Table 1. Feature information returned in the ECX register

Leaf	Register	Bit	Mnemonic	Description
1	ECX	30	RDRAND	A value of 1 indicates that processor supports the RDRAND instruction
7	EBX	18	RDSEED	A value of 1 indicates that processor supports the RDSEED instruction

The two options for invoking the CPUID instruction within the context of a high-level programming language like C or C++ are with:

- An inline assembly routine
- An assembly routine defined in an independent file.

The advantage of inline assembly is that it is straightforward and easily readable within its source code context. The disadvantage, however, is that conditional code is often needed to handle the possibility of different underlying platforms, which can quickly compromise readability. This guide describes a Linux implementation that should also work on OS X*. Please see the DRNG downloads for Windows* examples.

```
/* These are bits that are OR'd together */

#define DRNG_NO_SUPPORT        0x0     /* For clarity */
#define DRNG_HAS_RDRAND        0x1
#define DRNG_HAS_RDSEED        0x2
```

```
int get_drng_support ()
{
        static int drng_features= -1;

        /* So we don't call cpuid multiple times for
         * the same information */

        if ( drng_features == -1 ) {
                drng_features= DRNG_NO_SUPPORT;

                if ( _is_intel_cpu() ) {
                        cpuid_t info;

                        cpuid(&info, 1, 0);

                        if ( (info.ecx & 0x40000000) == 0x40000000 ) {
                                drng_features|= DRNG_HAS_RDRAND;
                        }

                        cpuid(&info, 7, 0);

                        if ( (info.ebx & 0x40000) == 0x40000 ) {
                                drng_features|= DRNG_HAS_RDSEED;
                        }
                }
        }

        return drng_features;
}
```

Code Example 1 shows the definition of the function get_drng_support for gcc compilation on 64-bit Linux. This is included in the source code module drng.c that is included in the DRNG samples source code download that accompanies this guide.

```
/* These are bits that are OR'd together */

#define DRNG_NO_SUPPORT        0x0     /* For clarity */
#define DRNG_HAS_RDRAND        0x1
#define DRNG_HAS_RDSEED        0x2

int get_drng_support ()
{
        static int drng_features= -1;

        /* So we don't call cpuid multiple times for
         * the same information */

        if ( drng_features == -1 ) {
                drng_features= DRNG_NO_SUPPORT;

                if ( _is_intel_cpu() ) {
                        cpuid_t info;

                        cpuid(&info, 1, 0);
```

```
                        if ( (info.ecx & 0x40000000) == 0x40000000 ) {
                                drng_features|= DRNG_HAS_RDRAND;
                        }

                        cpuid(&info, 7, 0);

                        if ( (info.ebx & 0x40000) == 0x40000 ) {
                                drng_features|= DRNG_HAS_RDSEED;
                        }
                }
        }

        return drng_features;
}
```

Code Example 1. Determining support for RDRAND and RDSEED on 64-bit Linux*

This function first determines if the processor is an Intel CPU by calling into the _is_intel_cpu() function, which is defined in Code Example 2. If it is, the function then checks the feature bits using the CPUID instruction to determine instruction support.

```
typedef struct cpuid_struct {
        unsigned int eax;
        unsigned int ebx;
        unsigned int ecx;
        unsigned int edx;
} cpuid_t;

int _is_intel_cpu ()
{
        static int intel_cpu= -1;
        cpuid_t info;

        if ( intel_cpu == -1 ) {
                cpuid(&info, 0, 0);

                if (
                        memcmp((char *) &info.ebx, "Genu", 4) ||
                        memcmp((char *) &info.edx, "ineI", 4) ||
                        memcmp((char *) &info.ecx, "ntel", 4)
                ) {
                        intel_cpu= 0;
                } else {
                        intel_cpu= 1;
                }
        }

        return intel_cpu;
}

void cpuid (cpuid_t *info, unsigned int leaf, unsigned int subleaf)
{
        asm volatile("cpuid"
        : "=a" (info->eax), "=b" (info->ebx), "=c" (info->ecx), "=d" (info-
>edx)
```

```
        : "a" (leaf), "c" (subleaf)
        );
}
```

Code Example 2. Calling CPUID

The CPUID instruction is run using inline assembly via the cpuid() function. It is declared as "volatile" as a precautionary measure, to prevent the compiler from applying optimizations that might interfere with its execution.

4.2 Using RDRAND to Obtain Random Values

Once support for RDRAND can be verified using CPUID, the RDRAND instruction can be invoked to obtain a 16-, 32-, or 64-bit random integer value. Note that this instruction is available at all privilege levels on the processor, so system software and application software alike may invoke RDRAND freely.

Reference (7) provides a table describing RDRAND instruction usage as follows:

Table 2. RDRAND instruction reference and operand encoding

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C7 /6 RDRAND r16	A	V/V	RDRAND	Read a 16-bit random number and store in the destination register.
0F C7 /6 RDRAND r32	A	V/V	RDRAND	Read a 32-bit random number and store in the destination register.
REX.W + 0F C7 /6 RDRAND r64	A	V/I	RDRAND	Read a 64-bit random number and store in the destination register.

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (w)	NA	NA	NA

Essentially, developers invoke this instruction with a single operand: the destination register where the random value will be stored. Note that this register must be a general purpose register, and the size of the register (16, 32, or 64 bits) will determine the size of the random value returned.

After invoking the RDRAND instruction, the caller must examine the carry flag (CF) to determine whether a random value was available at the time the RDRAND instruction was executed. As Table 3 shows, a value of 1 indicates that a random value was available and placed in the destination register provided in the invocation. A value of 0 indicates that a random value was not available. In current architectures the destination register will also be zeroed as a side effect of this condition.

Note that a destination register value of zero *should not* be used as an indicator of random value availability. The CF is the *sole indicator* of the success or failure of the RDRAND instruction.

Table 3. Carry Flag (CF) outcome semantics.

Carry Flag Value	Outcome
CF = 1	Destination register valid. Non-zero random value available at time of execution. Result placed in register.
CF = 0	Destination register all zeroes. Random value not available at time of execution. May be retried.

4.2.1 Retry Recommendations

It is recommended that applications attempt 10 retries in a tight loop in the unlikely event that the RDRAND instruction does not return a random number. This number is based on a binomial probability argument: given the design margins of the DRNG, the odds of ten

failures in a row are astronomically small and would in fact be an indication of a larger CPU issue.

4.2.2 Simple RDRAND Invocation

The unlikely possibility that a random value may not be available at the time of RDRAND instruction invocation has significant implications for system or application API definition. While many random functions are defined quite simply in the form:

```
unsigned int GetRandom()
```

use of RDRAND requires wrapper functions that appropriately manage the possible outcomes based on the CF flag value.

One handling approach is to simply pass the instruction outcome directly back to the invoking routine. A function signature for such an approach may take the form:

```
int rdrand(unsigned int *therand)
```

Here, the return value of the function acts as a flag indicating to the caller the outcome of the RDRAND instruction invocation. If the return value is 1, the variable passed by reference will be populated with a usable random value. If the return value is 0, the caller understands that the value assigned to the variable is not usable. The advantage of this approach is that it gives the caller the option to decide how to proceed based on the outcome of the call.

Code Example 3 shows this implemented for 16-, 32-, and 64-bit invocations of RDRAND using inline assembly.

```
#define <stdint.h>

int rdrand16_step (uint16_t *rand)
{
        unsigned char ok;

        asm volatile ("rdrand %0; setc %1"
                : "=r" (*rand), "=qm" (ok));

        return (int) ok;
}

int rdrand32_step (uint32_t *rand)
{
        unsigned char ok;

        asm volatile ("rdrand %0; setc %1"
                : "=r" (*rand), "=qm" (ok));

        return (int) ok;
}

int rdrand64_step (uint64_t *rand)
{
        unsigned char ok;
```

```
        asm volatile ("rdrand %0; setc %1"
                : "=r" (*rand), "=qm" (ok));

        return (int) ok;
}
```

Code Example 3. Simple RDRAND invocations for 16-bit, 32-bit, and 64-bit values

4.2.3 RDRAND Retry Loop

An alternate approach if random values are unavailable at the time of RDRAND execution is to use a retry loop. In this approach, an additional argument allows the caller to specify the maximum number of retries before returning a failure value. Once again, the success or failure of the function is indicated by its return value and the actual random value, assuming success, is passed to the caller by a reference variable.

Code Example 4 shows an implementation of RDRAND invocations with a retry loop.

```
int rdrand16_retry (unsigned int retries, uint16_t *rand)
{
        unsigned int count= 0;

        while ( count <= retries ) {
                if ( rdrand16_step(rand) ) {
                        return 1;
                }

                ++count;
        }

        return 0;
}

int rdrand32_retry (unsigned int retries, uint32_t *rand)
{
        unsigned int count= 0;

        while ( count <= retries ) {
                if ( rdrand32_step(rand) ) {
                        return 1;
                }

                ++count;
        }

        return 0;
}

int rdrand64_retry (unsigned int retries, uint64_t *rand)
{
        unsigned int count= 0;

        while ( count <= retries ) {
                if ( rdrand64_step(rand) ) {
                        return 1;
```

```
                }

                ++count;
        }

        return 0;
}
```

Code Example 4. RDRAND invocations with a retry loop

4.2.4 Initializing Data Objects of Arbitrary Size

A common function within RNG libraries is shown below:

```
int rdrand_get_bytes(unsigned int n, unsigned char *dest)
```

In this function, a data object of arbitrary size is initialized with random bytes. The size is specified by the variable n, and the data object is passed in as a pointer to `unsigned char` or `void`.

Implementing this function requires a loop control structure and iterative calls to the `rdrand64_step()` or `rdrand32_step()` functions shown previously. To simplify, let's first consider populating an array of unsigned int with random values in this manner using `rdrand32_step()`.

```
unsigned int rdrand_get_n_uints (unsigned int n, unsigned int *dest)
{
        unsigned int i;
        uint32_t *lptr= (uint32_t *) dest;

        for (i= 0; i< n; ++i, ++dest) {
                if ( ! rdrand32_step(dest) ) {
                        return i;
                }
        }

        return n;
}
```

Code Example 5. Initializing an array of 32-bit integers

The function returns the number of unsigned int values assigned. The caller would check this value against the number requested to determine whether assignment was successful. Other implementations are possible, for example, using a retry loop to handle the unlikely possibility of random number unavailability.

In the next example, we reduce the number of RDRAND calls in half by using `rdrand64_step()` instead of `rdrand32_step()`.

```
unsigned int rdrand_get_n_uints (unsigned int n, unsigned int *dest)
{
        unsigned int i;
```

```
        uint64_t *qptr= (uint64_t *) dest;
        unsigned int total_uints= 0;
        unsigned int qwords= n/2;

        for (i= 0; i< qwords; ++i, ++qptr) {
                if ( rdrand64_retry(RDRAND_RETRIES, qptr) ) {
                        total_uints+= 2;
                } else {
                        return total_uints;
                }
        }

        /* Fill the residual */

        if ( n%2 ) {
                unsigned int *uptr= (unsigned int *) qptr;

                if ( rdrand32_step(uptr) ) {
                        ++total_uints;
                }
        }

        return total_uints;
}
```

Code Example 6. Initializing an object of arbitrary size using RDRAND

Finally, we show how a loop control structure and `rdrand64_step()` can be used to populate a byte array with random values.

```
unsigned int rdrand_get_bytes (unsigned int n, unsigned char *dest)
{
        unsigned char *headstart, *tailstart;
        uint64_t *blockstart;
        unsigned int count, ltail, lhead, lblock;
        uint64_t i, temprand;

        /* Get the address of the first 64-bit aligned block in the
         * destination buffer. */

        headstart= dest;
        if ( ( (uint64_t)headstart % (uint64_t)8 ) == 0 ) {

                blockstart= (uint64_t *)headstart;
                lblock= n;
                lhead= 0;
        } else {
                blockstart= (uint64_t *)
                        ( ((uint64_t)headstart & ~(uint64_t)7) + (uint64_t)8
);

                lblock= n - (8 - (unsigned int) ( (uint64_t)headstart &
(uint64_t)8 ));
```

```
                lhead= (unsigned int) ( (uint64_t)blockstart -
(uint64_t)headstart );
        }

        /* Compute the number of 64-bit blocks and the remaining number
         * of bytes (the tail) */

        ltail= n-lblock-lhead;
        count= lblock/8;        /* The number 64-bit rands needed */

        if ( ltail ) {
                tailstart= (unsigned char *)( (uint64_t) blockstart +
(uint64_t) lblock );
        }

        /* Populate the starting, mis-aligned section (the head) */

        if ( lhead ) {
                if ( ! rdrand64_retry(RDRAND_RETRIES, &temprand) ) {
                        return 0;
                }

                memcpy(headstart, &temprand, lhead);
        }

        /* Populate the central, aligned block */

        for (i= 0; i< count; ++i, ++blockstart) {
                if ( ! rdrand64_retry(RDRAND_RETRIES, blockstart) ) {
                        return i*8+lhead;
                }
        }

        /* Populate the tail */

        if ( ltail ) {
                if ( ! rdrand64_retry(RDRAND_RETRIES, &temprand) ) {
                        return count*8+lhead;
                }

                memcpy(tailstart, &temprand, ltail);
        }

        return n;
}
```

Code Example 7. Initializing an object of arbitrary size using RDRAND

4.2.5 Guaranteeing DBRG Reseeding

As a high performance source of random numbers, the DRNG is both fast and scalable. It is directly usable as a sole source of random values underlying an application or operating system RNG library. Still, some software venders will want to use the DRNG to seed and reseed in an ongoing manner their current software PRNG. Some may feel it necessary, for

standards compliance, to demand an absolute guarantee that values returned by RDRAND reflect independent entropy samples within the DRNG.

As described in section 3.2.3, the DRNG uses a deterministic random bit generator, or DRBG, to "spread" a conditioned entropy sample into a large set of random values, thus increasing the number of random numbers available by the hardware module. The DRBG autonomously decides when it needs to be reseeded, behaving in a way that is unpredictable and transparent to the RDRAND caller. There is an upper bound of 511 samples per seed in the implementation where samples are 128 bits in size and can provide two 64-bit random numbers each. In practice, the DRBG is reseeded frequently, and it is generally the case that reseeding occurs long before the maximum number of samples can be requested by RDRAND.

There are two approaches to structuring RDRAND invocations such that DRBG reseeding can be guaranteed:

- Iteratively execute RDRAND beyond the DRBG upper bound by executing more than 1022 64-bit RDRANDs
- Iteratively execute 32 RDRAND invocations with a 10 us wait period per iteration.

The latter approach has the effect of forcing a reseeding event since the DRBG aggressively reseeds during idle periods.

4.2.6 Generating Seeds from RDRAND

Processors that do not support the RDSEED instruction can leverage the reseeding guarantee of the DRBG to generate random seeds from values obtained via RDRAND.

The program below takes the first approach of guaranteed reseeding—generating 512 128-bit random numbers—and mixes the intermediate values together using the CBC-MAC mode of AES. This method of turning 512 128-bit samples from the DRNG into a 128-bit seed value is sometimes referred to as the “512:1 data reduction” and results in a random value that is fully forward and backward prediction resistant, suitable for seeding a NIST SP800-90 compliant, FIPS 1402-2 certifiable, software DRBG.

This program relies on libgcrypt from the Gnu Project for the encryption routines.

```
#include “drng.h”
#include <stdio.h>
#include <gcrypt.h>
#include <string.h>

#define AES_BLOCK_SIZE 16      /* AES uses 128-bit blocks (16 bytes) */
#define AES_KEY_SIZE   16      /* AES with 128-bit key (AES-128) */
#define RDRAND_SAMPLES  512    /* the DRNG reseeds after generating 511
                                * 128-bit (16-byte) values */
#define BUFFER_SIZE            16*RDRAND_SAMPLES

#define MIN_GCRYPT_VERSION "1.0.0"

int main (int argc, char *argv[])
{
        unsigned char rbuffer[BUFFER_SIZE];
```

```
        unsigned char aes_key[AES_KEY_SIZE];
        unsigned char aes_iv[AES_KEY_SIZE];
        unsigned char seed[16];
        static gcry_cipher_hd_t gcry_cipher_hd;
        gcry_error_t gcry_error;

        if ( ! ( get_drng_support() & DRNG_HAS_RDRAND ) ) {
                fprintf(stderr, "No RDRAND support\n");
                return 1;
        }

        /* Generate a random AES key */

        if ( rdrand_get_bytes(AES_KEY_SIZE, aes_key) < AES_KEY_SIZE ) {
                fprintf(stderr, "Random numbers not available\n");
                return 1;
        }

        /* Generate a random IV */

        if ( rdrand_get_bytes(AES_BLOCK_SIZE, aes_iv) < AES_BLOCK_SIZE ) {
                fprintf(stderr, "Random numbers not available\n");
                return 1;
        }

        /*
         * Fill our buffer with 512 128-bit rdrands. This
         * guarantees that /at least/ one reseed takes place.
         */

        if ( rdrand_get_bytes(BUFFER_SIZE, rbuffer) < BUFFER_SIZE ) {
                fprintf(stderr, "Random numbers not available\n");
                return 1;
        }

        /* Initialize the cryptographic library */

        if (!gcry_check_version(MIN_GCRYPT_VERSION)) {
                fprintf(stderr,
                        "gcry_check_version: have version %s, need version
%s or newer",
                        gcry_check_version(NULL), MIN_GCRYPT_VERSION
                );

                return 1;
        }

        gcry_error= gcry_cipher_open(&gcry_cipher_hd, GCRY_CIPHER_AES128,
                GCRY_CIPHER_MODE_CBC, 0);
        if ( gcry_error ) {
                fprintf(stderr, "gcry_cipher_open: %s",
gcry_strerror(gcry_error));
                return 1;
        }
```

```
        gcry_error= gcry_cipher_setkey(gcry_cipher_hd, aes_key,
AES_KEY_SIZE);
        if ( gcry_error ) {
                fprintf(stderr, "gcry_cipher_setkey: %s",
gcry_strerror(gcry_error));
                gcry_cipher_close(gcry_cipher_hd);
                return 1;
        }

        gcry_error= gcry_cipher_setiv(gcry_cipher_hd, aes_iv,
AES_BLOCK_SIZE);
        if ( gcry_error ) {
                fprintf(stderr, "gcry_cipher_setiv: %s",
gcry_strerror(gcry_error));
                gcry_cipher_close(gcry_cipher_hd);
                return 1;
        }

        /*
         * Do the encryption in-place. This has the nice side effect of
         * erasing the original values.
         */

        gcry_error= gcry_cipher_encrypt(gcry_cipher_hd, rbuffer,
BUFFER_SIZE,
                NULL, 0);
        if ( gcry_error ) {
                fprintf(stderr, "gcry_cipher_encrypt: %s\n",
                        gcry_strerror(gcry_error));
                return 1;
        }

        gcry_cipher_close(gcry_cipher_hd);

        /* The last block of the cipher text is the MAC, and our seed
value. */

        memcpy(seed, &rbuffer[BUFFER_SIZE-16], 16);
```

Code Example 8. Generating random seeds from RDRAND

4.3 Using RDSEED to Obtain Random Seeds

Once support for RDSEED has been verified using CPUID, the RDSEED instruction can be used to obtain a 16-, 32-, or 64-bit random integer value. Again, this instruction is available at all privilege levels on the processor, so system software and application software alike may invoke RDSEED freely.

RDSEED instruction is documented in (9). Usage is as follows:

Table 4. RDSEED instruction reference and operand encoding

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C7 /7 RDSEED r16	A	V/V	RDSEED	Read a 16-bit random number and store in the destination register.
0F C7 /7 RDSEED r32	A	V/V	RDSEED	Read a 32-bit random number and store in the destination register.
REX.W + 0F C7 /7 RDSEED r64	A	V/I	RDSEED	Read a 64-bit random number and store in the destination register.

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (w)	NA	NA	NA

As with RDRAND, developers invoke the RDSEED instruction with the destination register where the random seed will be stored. This register must be a general purpose one whose size determines the size of the random seed that is returned.

After invoking the RDSEED instruction, the caller must examine the carry flag (CF) to determine whether a random seed was available at the time the RDSEED instruction was executed. As shown in Table 5, a value of 1 indicates that a random seed was available and placed in the destination register provided in the invocation. A value of 0 indicates that a random seed was not available. In current architectures the destination register will also be zeroed as a side effect of this condition.

Again, a destination register value of zero *should not* be used as an indicator of random seed availability. The CF is the *sole indicator* of the success or failure of the RDSEED instruction.

Table 5. Carry Flag (CF) outcome semantics

Carry Flag Value	Outcome
CF = 1	Destination register valid. Non-zero random seed available at time of execution. Result placed in register.
CF = 0	Destination register all zeroes. Random seed not available at time of execution. May be retried.

4.3.1 Retry Recommendations

Unlike the RDRAND instruction, the seed values come directly from the entropy conditioner, and it is possible for callers to invoke RDSEED faster than those values are generated. This means that applications must be designed robustly and be prepared for calls to RDSEED to fail because seeds are not available (CF=0).

If only one thread is calling RDSEED infrequently, it is very unlikely that a random seed will not be available. Only during periods of heavy demand, such as when one thread is calling RDSEED in rapid succession or multiple threads are calling RDSEED simultaneously, are

underflows likely to occur. Because the RDSEED instruction does not have a fairness mechanism built into it, however, there are no guarantees as to how often a thread should retry the instruction, or how many retries might be needed, in order to obtain a random seed. In practice, this depends on the number of hardware threads on the CPU and how aggressively they are calling RDSEED.

Since there is no simple procedure for retrying the instruction to obtain a random seed, follow these basic guidelines.

4.3.1.1 Synchronous applications

If the application is not latency-sensitive, then it can simply retry the RDSEED instruction indefinitely, though it is recommended that a PAUSE instruction be placed in the retry loop. In the worst-case scenario, where multiple threads are invoking RDSEED continually, the delays can be long, but the longer the delay, the more likely (with an exponentially increasing probability) that the instruction will return a result.

If the application is latency-sensitive, then applications should either sleep or fall back to generating seed values from RDRAND.

4.3.1.2 Asynchronous applications

The application should be prepared to give up on RDSEED after a small number of retries, where "small" is somewhere between 1 and 100, depending on the application's sensitivity to delays. As with synchronous applications, it is recommended that a PAUSE instruction be inserted into the retry loop.

Applications needing a more aggressive approach can alternate between RDSEED and RDRAND, pulling seeds from RDSEED as they are available and filling a RDRAND buffer for future 512:1 reduction when they are not.

4.3.2 Simple RDSEED Invocation

Code Example 39 shows inline assembly implementations for 16-, 32-, and 64-bit invocations of RDSEED.

```
int rdseed16_step (uint16_t *seed)
{
        unsigned char ok;

        asm volatile ("rdseed %0; setc %1"
                : "=r" (*seed), "=qm" (ok));

        return (int) ok;
}

int rdseed32_step (uint32_t *seed)
{
        unsigned char ok;

        asm volatile ("rdseed %0; setc %1"
                : "=r" (*seed), "=qm" (ok));

        return (int) ok;
```

```
}

int rdseed64_step (uint64_t *seed)
{
        unsigned char ok;

        asm volatile ("rdseed %0; setc %1"
                : "=r" (*seed), "=qm" (ok));

        return (int) ok;
}
```

Code Example 9. Simple RDSEED invocations for 16-bit, 32-bit, and 64-bit values

5 Summary

Intel Data Protection Technology with Secure Key represents a new class of random number generator. It combines a high-quality entropy source with a CSPRNG into a robust, self-contained hardware module that is isolated from software attacks. The resulting random numbers offer excellent statistical qualities, highly unpredictable random sequences, and high performance. Accessible via two simple instructions, RDRAND and RDSEED, the random number generator is also very easy to use. Random numbers are available to software running at all privilege levels, and requires no special libraries or operating system handling.

References

1. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*. **Nishimura, Makoto Matsumoto and Takuji.** 1, January 1998, ACM Transactions on Modeling and Computer Simulation, Vol. 8.

2. **Z. Gutterman, B. Pinkas, and T. Reinman.** Analysis of the Linux Random Number Generator. [Online] March 2006. http://software.intel.com/sites/default/files/m/6/0/9/gpr06.pdf.

3. CVE-2008-0166. *Common Vulnerabilities and Exposures.* [Online] January 9, 2008. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166.

4. Specification for the Advanced Encryption Standard (AES). [Online] November 26, 2001. http://software.intel.com/sites/default/files/m/4/d/d/fips-197.pdf.

5. Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode. [Online] October 2010. http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist_sp800-38A.pdf.

6. Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised). [Online] January 2012. http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf.

7. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z. [Online] http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

8. Intel® Processor Identification and the CPUID Instruction. [Online] April 2012. http://www.intel.com/content/www/us/en/processors/processor-identification-cpuid-instruction-note.html.

9. Intel® Architecture Instruction Set Extensions Programming Reference. [Online] https://software.intel.com/en-us/intel-isa-extensions.

Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others