

Intel® Digital Random Number Generator (DRNG)

Software Implementation Guide

Revision 1.1
August 7, 2012



Revision History

Revision	Revision History	Date
0.1	Beta Release Revision	Feb 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	Replaced Code Samples for <code>_rdrand16_step</code> , <code>_rdrand32_step</code> , and <code>_rdrand64_step</code>	August 7, 2012

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 Intel® Active Management Technology 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 by visiting Intel's Web Site.

Intel® Active Management Technology requires the computer system to have an Intel(R) AMT-enabled chipset, network hardware and software, as well as connection with a power source and a corporate network connection. Setup requires configuration by the purchaser and may require scripting with the management console or further integration into existing security frameworks to enable certain functionality. It may also require modifications of implementation of new business processes. With regard to notebooks, Intel AMT may not be available or certain capabilities may be limited over a host OS-based VPN or when connecting wirelessly, on battery power, sleeping, hibernating or powered off. For more information, see www.intel.com/technology/platform-technology/intel-amt/

Throughout this document Intel ME refers to Intel® Management Engine and Intel® AMT refers to Intel® Active Management Technology.

Requires a system with a 64-bit enabled processor, chipset, BIOS and software. Performance will vary depending on the specific hardware and software you use. Consult your PC manufacturer for more information. For more information, visit <http://www.intel.com/info/em64t>

Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, virtual machine monitor (VMM). Functionality, performance or other benefits will vary depending on hardware and software configurations. Software applications may not be compatible with all operating systems. Consult your PC manufacturer. For more information, visit <http://www.intel.com/go/virtualization>

Intel, the Intel logo, Intel® AMT, Intel vPro, Centrino, Centrino Inside, and vPro Inside are trademarks or registered trademarks of Intel Corporation in the United States and other countries.

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

Copyright © 2011 Intel Corporation. All rights reserved.

Contents

Revision History	3
1 Introduction	5
2 RNG Basics and Introduction to the DRNG.....	6
2.1 Random Number Generators (RNGs).....	6
2.2 Pseudo-Random Number Generators (PRNGs).....	6
2.3 True Random Number Generators (TRNGs).....	8
2.4 Cascade Construction RNGs	8
2.5 Introducing the Digital Random Number Generator (DRNG)	9
2.6 Applications for the Digital Random Number Generator	10
3 DRNG Overview	11
3.1 Processor Chip View.....	11
3.2 Component Architecture	12
3.2.1 Entropy Source (ES)	12
3.2.2 Conditioner.....	13
3.2.3 Deterministic Random Bit Generator (DRBG).....	13
3.3 Robustness and Self-Validation.....	13
3.3.1 Online Health Tests (OHTs).....	16
3.3.2 Built-In Self Tests (BISTs)	16
3.4 RDRAND.....	17
3.5 Performance	17
3.6 Power Requirements	19
4 RDRAND Instruction Usage	20
4.1 Using RDRAND to Obtain Random Values.....	23
4.1.1 Simple RDRAND Invocation.....	24
4.1.2 RDRAND Retry Loop.....	25
4.2 Initializing Data Objects of Arbitrary Size.....	26
4.3 Library API Recommendations.....	29
4.4 Guaranteeing DRBG Reseeding.....	30
5 References	33
Figures	
Figure 1: Cascade Construction Random Number Generator.....	9
Figure 2: Digital Random Number Generation Design.....	11
Figure 3: DRNG Component Architecture.....	12
Figure 4: DRNG Self-Validation Components	14

1 Introduction

Intel® Secure Key, previously code-named Bull Mountain Technology, is the Intel name for the Intel® 64 and IA-32 Architectures instruction RDRAND and its 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.

This *Digital Random Number Generator Software Implementation Guide* is intended to provide a complete source of technical information on RDRAND usage, including code examples. Included in this document are the following additional 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 Instruction Usage. This section provides reference information on the RDRAND instruction and code examples showing its use. This includes RDRAND platform support verification and suggestions on DRNG-based libraries.

This document is designed to serve a variety of readers. Programmers who already understand the nature of RNGs may refer directly to section 4 for RDRAND instruction reference 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 users 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, we discuss in this section 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 to 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.

For 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, any deterministic algorithm is subject to discovery by an attacker through a wide variety of 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, whatever it is, then 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.

Rather than 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. The problem of 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. 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.

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. Combined, the approach offers improved performance over TRNGs: a slow entropy source periodically seeding a fast CSPRNG capable of generating a large number of random values from a single seed.

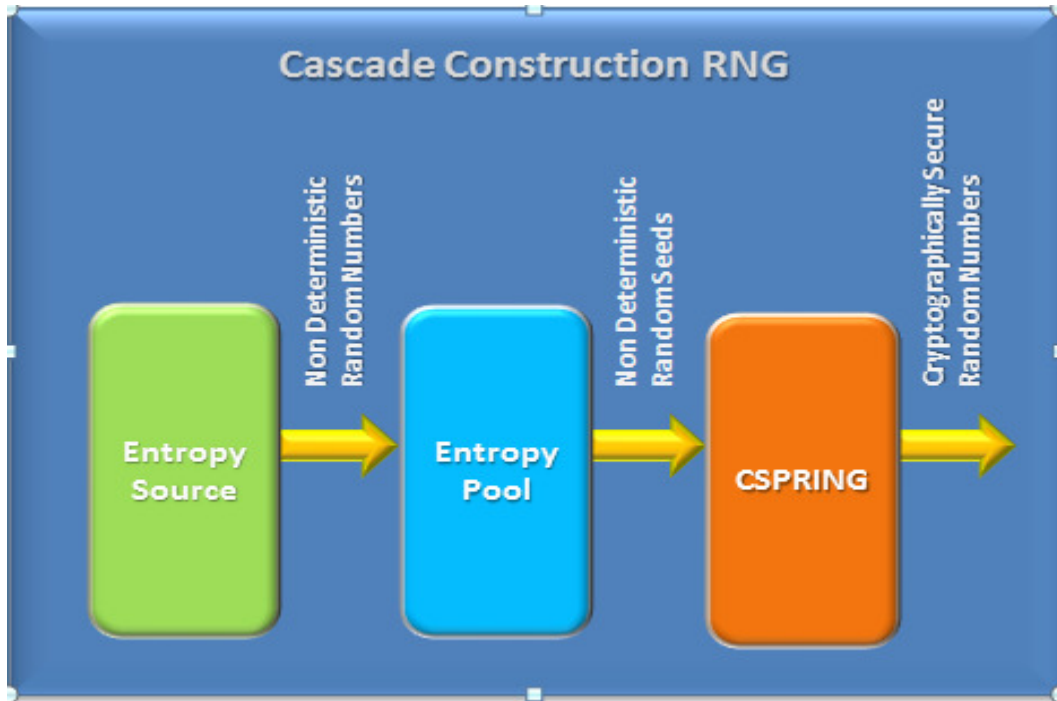


Figure 1: Cascade Construction Random Number Generator

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 a new Intel 64 Architecture instruction, RDRAND, 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 hardware on the processor chip itself and can be utilized through a new instruction 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 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 SP800-90, FIPS-140-2, and ANSI X9.82. Compliance to these standards makes Digital Random Number Generation with RDRAND 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, an Intel 64 instruction set extension 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., 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

3 DRNG Overview

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

3.1 Processor Chip View

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

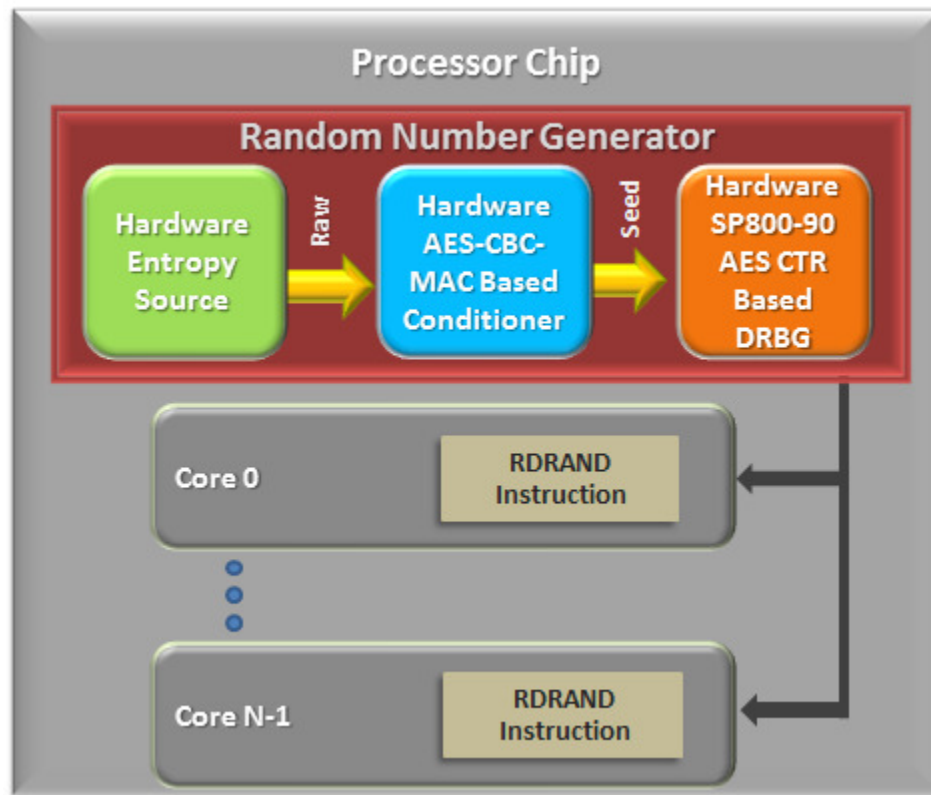


Figure 2: Digital Random Number Generator using RDRAND Design

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

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[6] mode to distill the entropy into high-quality nondeterministic random numbers, and a deterministic random bit generator (DRBG) that is seeded from the conditioner.

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 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.

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

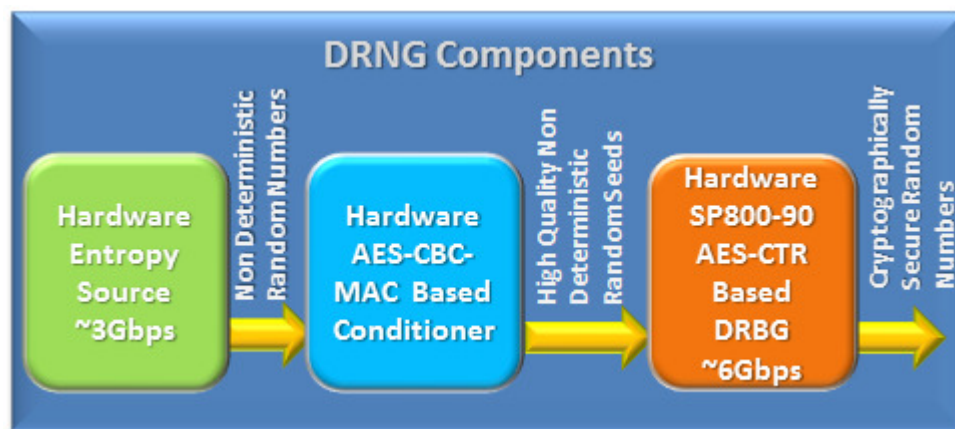


Figure 3: DRNG Component Architecture

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 zeros 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 [6].

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 800-90 [5], 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.

3.3 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 and at system startup time. These include the DRNG Online Health Tests (OHTs) and Built-In Self Tests (BISTs), respectively. Both are shown in Figure 4.

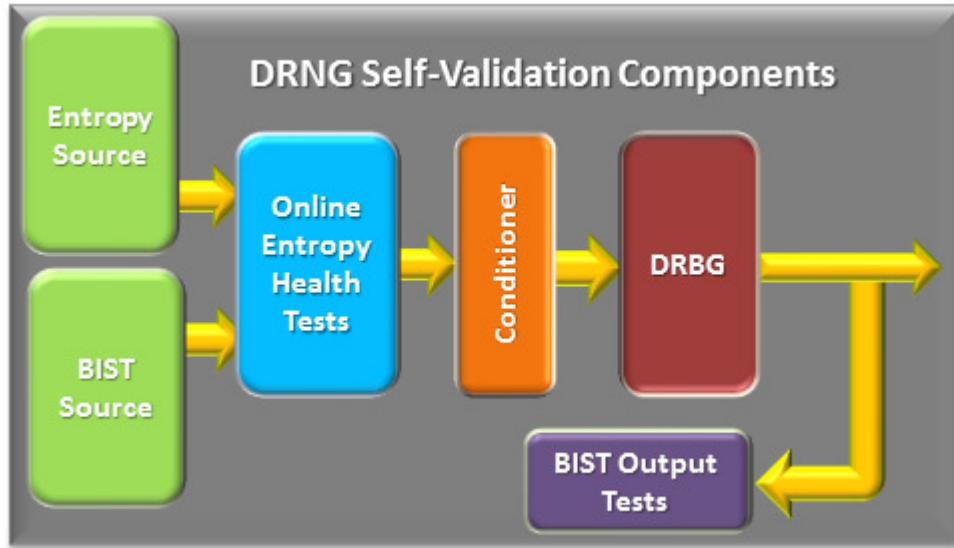


Figure 4: DRNG Self-Validation Components

3.3.1 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 overall ensure that ES is operating correctly before it will issue 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.

3.3.2 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 refuse to issue random numbers and will issue a BIST failure notification to the on-chip test circuitry. This BIST logic avoids the need for conventional on-chip test mechanisms (e.g., scan and JTAG) that could undermine the security of the DRNG.

3.4 RDRAND

Software access to the DRNG is through a new instruction, RDRAND, and is documented in Section 7 of the [Intel® 64 and IA-32 Architectures Software Developer's Manual](#).

RDRAND retrieves a hardware generated random value from the DRNG 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, the user can use the CPUID instruction to examine bit 30 of the ECX register. See Reference [7] for details.

3.5 Performance

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

The DRNG is implemented in hardware as part of the processor chip. 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-chip bus latencies to slow entropy transfer or create bottlenecks when multiple requests have been issued.

Random values are delivered directly through an instruction level request (RDRAND). 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 means that RDRAND can be used freely by multiple VMs without the need for hypervisor intervention or resource management.

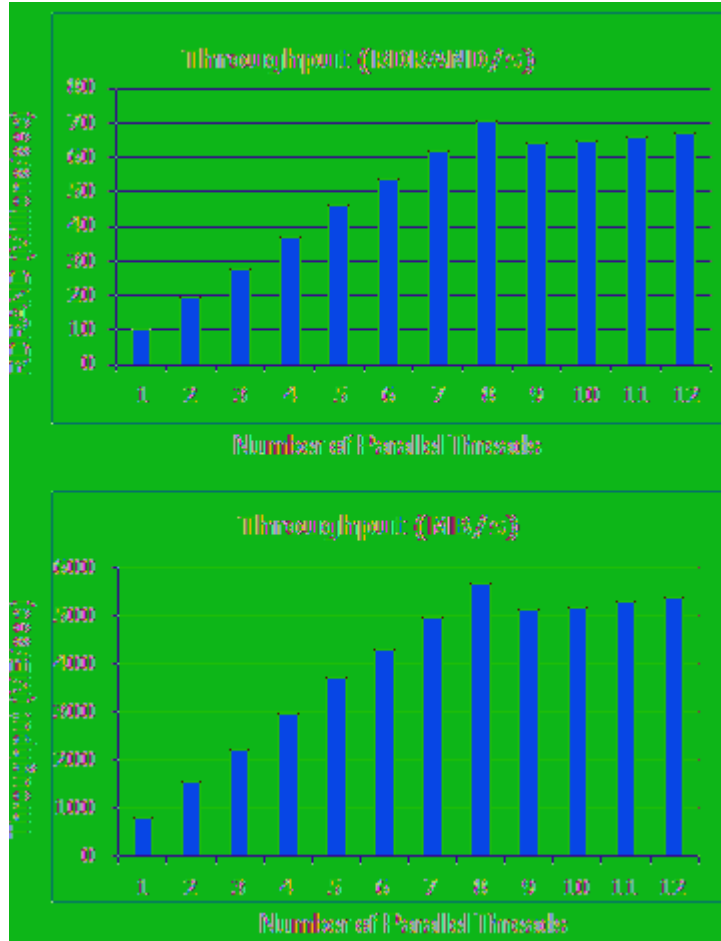
Below are some charts showing some preliminary data from a pre-production sample on a system with a 3rd generation Intel® Core™ family processor, code-named Ivy Bridge:

Disclaimer: Data taken from an early engineering sample board with a 3rd generation Intel Core family processor, code-named Ivy Bridge, quad core, 4 GB memory, hyper-threading enabled. Software: LINUX* Fedora 14, GCC version 4.6.0 (experimental) with RDRAND support, test uses p-threads kernel API.

Measured Throughput:

- Up to 70 million RDRAND invocations per second
- 500+ million bytes of random data per second
- Throughput ceiling is insensitive to the number of contending parallel threads

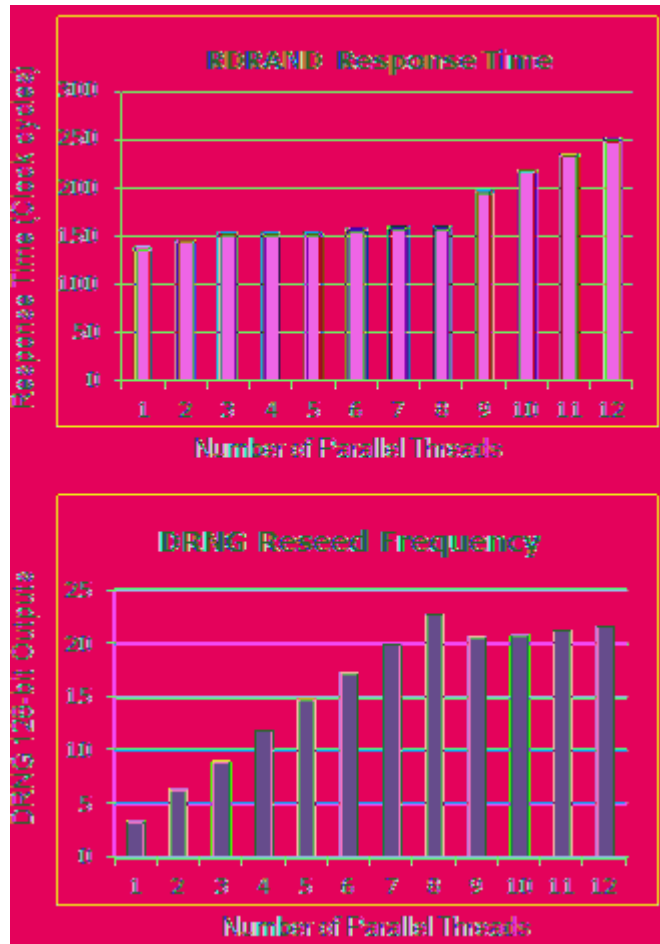
Notice that steady state is maintained at peak performance



RDRAND Response Time and Reseeding Frequency

- ~150 clocks per invocation
Note: Varies with CPU clock frequency since constraint is shared data path from DRNG to cores.
- Little contention until 8 threads
 - or 4 threads on 2 core chip
- Simple linear increase as additional threads are added
 - DRNG Reseed Frequency
- Single thread worst case: Reseeds every 4 RDRAND invocations
- Multiple thread worst case: Reseeds every 23 RDRAND invocations
- At slower invocation rate, can expect reseed before every 2 RDRAND calls

- NIST SP 800-90 recommends $\leq 2^{48}$



3.6 Power Requirements

The DRNG hardware resides on the processor chip. As such, it needs no dedicated power supply to run. Instead it simply uses the processor's local power supply. As described in section 3.2.1, it furthermore is designed to function across a range of process voltage and temperature (PVT) levels, exceeding the normal operating range of the processor.

The DRNG 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 RDRAND Instruction Usage

In this section, we provide RDRAND instruction reference information 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].

Determining Processor Support for RDRAND

Before using the RDRAND instruction, 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 [8] and [9].

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

Bit #	Mnemonic	Description
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction

Table 1: Feature information returned in the ECX register.

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.

Various examples in this chapter will illustrate both techniques. 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. This can quickly compromise readability. For this reason, we often favor defining an assembly routine in an independent file and then invoking it by its declared name. Now the original source code (the caller of the routine) can remain unchanged, while the build system can handle choosing among code versions of the same routine for different platform targets.

This Implementation Guide describes Linux implementation. Please see the DRNG downloads for Windows* and Mac* OS examples.

- Note: Nothing goes to memory. These samples were written a certain way for a reason.

Example 1 shows the definition of the function `get_cpuid_info_v1` for gcc compilation on 64-bit Linux.

```
.intel_syntax noprefix
    .text
    .global    get_cpuid_info_v1
get_cpuid_info_v1:
    mov r8, rdi    # array addr
    mov r9, rsi    # leaf
    mov r10, rdx   # subleaf
    push      rax
    push      rbx
    push      rcx
    push      rdx
    mov      eax, r9d
    mov      ecx, r10d
    cpuid
    mov      DWORD PTR [r8], eax
    mov      DWORD PTR [r8+4], ebx
    mov      DWORD PTR [r8+8], ecx
    mov      DWORD PTR [r8+12], edx
    pop      rdx
    pop      rcx
    pop      rbx
    pop      rax
    ret      0
#get_cpuid_info_v1 ENDP
#_TEXT      ENDS
```

Code Example 1: Using CPUID to detect support for RDRAND on 64-bit Linux.

The routine, defined in the file `get_cpuid_v1_lix64.s`, can be compiled into object code with gcc as follows:

```
gcc -g -c get_cpuid_v1_lix64.s -o get_cpuid_v1_lix64.o
```

To use the IA-64 assembly routine, first define the data structure to be passed to the routine in a header file like `get_cpuid_v1_lix64.h`:

```
typedef struct {
    unsigned int EAX;
    unsigned int EBX;
    unsigned int ECX;
```

```

        unsigned int EDX;
    } CPUIDinfo;

extern void get_cpuid_info_v1(CPUIDinfo *info, const unsigned int func,
const unsigned int subfunc);

```

Code Example 2: Header file to be used by assembly routine caller.

This header file also declares the function and uses `extern` to indicate that it is externally defined.

To invoke this assembly routine from a C/C++ program, include the above header file, create a `CPUIDinfo` object to hold the results, and invoke the externally defined function. One possible implementation of these steps is as follows:

```

#include "get_cpuid_v1_lix64.h"

void _CPUID(CPUIDinfo *info, const unsigned int func, const unsigned int
subfunc)
{
    get_cpuid_info_v1(info, func, subfunc);
}

typedef unsigned int DWORD;

int _rdrand_check_support()
{
    CPUIDinfo info;
    int got_intel_cpu=0;

    _CPUID(&info,0,0);
    if(memcmp((char *)(&info.EBX), "Genu", 4) == 0 &&
        memcmp((char *)(&info.EDX), "ineI", 4) == 0 &&
        memcmp((char *)(&info.ECX), "ntel", 4) == 0) {
        got_intel_cpu = 1;
    }

    if (got_intel_cpu) {
        _CPUID(&info,1,0);
        if ((info.ECX & 0x40000000)==0x40000000) return 1;
    }
    return 0;
}

```

Code Example 3: Invoking `get_cpuid_info_v1` to determine RDRAND support.

After the first `_CPUID` call in this example, the code checks whether the current processor is an Intel product. After the second `_CPUID` call, the code checks the RDRAND bit. Checking the manufacturer becomes important if another manufacturer uses bit 30 for a different purpose.

4.1 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.

The *Intel® 64 and IA-32 Architectures Software Developer's Manual* [7] provides a table describing RDRAND instruction usage as follows:

Opcode/ Instruction	Op Encoding	64/32bit Mode Support	CPUID Feature Flag	Description
0F C7 /6 RDRAND r16	ModRM:r /m(w)	V/V	RDRAND	Read a 16-bit random number and store in the destination register.
0F C7 /6 RDRAND r32	ModRM:r /m(w)	V/V	RDRAND	Read a 32-bit random number and store in the destination register.
REX.W + 0F C7 /6 RDRAND r64	ModRM:r /m(w)	V/I	RDRAND	Read a 64-bit random number and store in the destination register.

Table 2: RDRAND instruction reference

Essentially, the user invokes 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. 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 this case, the destination register will also be zeroed.

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 random value availability.

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 zeros. Random value not available at time of execution. May be retried.

Table 3: Carry Flag (CF) outcome semantics.

4.1.1 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)
```

In this approach, the return value of the function acts as a flag indicating to the caller the outcome of the RDRAND instruction invocation. If 1, the variable passed by reference will be populated with a usable random value. If 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 examples 4, 5, and 6 show how this approach can be implemented for 16-, 32-, and 64-bit invocations of RDRAND using the inline assembly approach.

```
int _rdrand16_step(unsigned short *therand) {
    unsigned char err;

    asm volatile("rdrand %0 ; setc %1"
        : "=r" (*therand), "=qm" (err));
}
```



```

return (int) err;
}

```

Code Example 4: Simple RDRAND invocation for a 16-bit random value

```

int _rdrand32_step(unsigned int *therand) {
    unsigned char err;

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

    return (int) err;
}

```

Code Example 5: Simple RDRAND invocation for a 32-bit random value

```

int _rdrand64_step(unsigned long long int *therand) {
    unsigned char err;

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

    return (int) err;
}

```

Code Example 6: Simple RDRAND invocation for a 64-bit random value

4.1.2 RDRAND Retry Loop

An alternate approach for handling random value unavailability 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.

```

int rdrand(unsigned int retry_limit, unsigned int *therand)

```

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 7 shows an implementation for 32-bit random values using the function from the previous section as an underlying primitive.

```

int _rdrand_get_uint_retry(unsigned int retry_limit, unsigned int *dest)
{
    int success;

```

```

int count;
unsigned int therand;

count = 0;

do
{
    success=_rdrand32_step(&therand);
} while((success == 0) || (count++ < retry_limit));

if (success == 1)
{
    *dest = therand;
    return 1;
}
else
{
    return 0;
}
}

```

Code Example 7: 32-bit RDRAND invocation with a retry loop

4.2 Initializing Data Objects of Arbitrary Size

A common random function within RNG libraries is seen below:

```
int rdrand_get_n_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 `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()`.

```

int rdrand_get_n_uints(int n, unsigned int *dest)
{
    int dwords;
    int i;
    unsigned int drand;
    int success;
    int total_uints;

    total_uints = 0;

```

```

for (i=0; i<dwords; i++)
{
    if (rdrand32_step(&drand))
    {
        *dest = drand;
        dest++;
        total_uints++;
    }
    else
    {
        i = dwords;
    }
}
return total_uints;
}

```

Code Example 8: Initializing an object of arbitrary size using RDRAND

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()`.

```

int rdrand_get_n_uints(int n, unsigned int *dest)
{
    int qwords;
    int i;
    unsigned long int qrand;
    int success;
    int total_uints;
    unsigned long int *qptr;

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

    for (i=0; i<qwords; i++)
    {
        if (_rdrand64_step(&qrand))
        {
            *qptr = qrand;
            qptr++;
            total_uints+=2;
        }
        else

```

```

        {
            i = qwords;
        }
    }
    if ((qwords > 0) && (success == 0))
        return total_uints;
}

```

Code Example 9: 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.

```

int _rdrand_get_bytes_step(unsigned int n, unsigned char *dest)
{
    unsigned char *start;
    unsigned char *residualstart;
    unsigned long long int *blockstart;
    unsigned int count;
    unsigned int residual;
    unsigned int startlen;
    unsigned long long int i;
    unsigned long long int temprand;
    unsigned int length;

    /* Compute the address of the first 64 bit aligned block in the destination
    buffer */
    start = dest;
    if (((unsigned long int)start % (unsigned long int)8) == 0)
    {
        blockstart = (unsigned long long int *)start;
        count = n;
        startlen = 0;
    }
    else
    {
        blockstart = (unsigned long long int *)(((unsigned long long int)start &
        ~(unsigned long long int)7)+(unsigned long long int)8);
        count = n - (8 - (unsigned int)((unsigned long long int)start % 8));
        startlen = (unsigned int)((unsigned long long int)blockstart - (unsigned
        long long int)start);
    }

    /* Compute the number of 64 bit blocks and the remaining number of bytes */
    residual = count % 8;
    length = count >> 3;
    if (residual != 0)

```

```

    {
        residualstart = (unsigned char *) (blockstart + length);
    }
    /* Get a temporary random number for use in the residuals. Failout if retry
    fails */
    if (startlen > 0)
    {
        if (_rdrand_get_n_qints_retry(1, 10, (void *)&temprand) == 0) return 0;
    }

    /* populate the starting misaligned block */
    for (i = 0; i<startlen; i++)
    {
        start[i] = (unsigned char)(temprand & 0xff);
        temprand = temprand >> 8;
    }

    /* populate the central aligned block. Fail out if retry fails */
    if (_rdrand_get_n_qints_retry(length, 10, (void *) (blockstart)) == 0) return
0;

    /* populate the final misaligned block */
    if (residual > 0)
    {
        if (_rdrand_get_n_qints_retry(1, 10, (void *)&temprand) == 0) return
0;

        for (i = 0; i<residual; i++)
        {
            residualstart[i] = (unsigned char)(temprand & 0xff);
            temprand = temprand >> 8;
        }
    }

    return 1;
}

```

Code Example 10: Initializing an object of arbitrary size using RDRAND

4.3 Library API Recommendations

Library APIs making RDRAND available to other applications may do so at two levels. First, a library may offer a low-level wrapper for invoking RDRAND to obtain 16-, 32-, or 64-bit random values. Results of the CF flag value indicating availability of a random value at the time of execution are returned directly to the caller.

```
int _rdrand_u16_step(unsigned short int *);  
int _rdrand_u32_step(unsigned int *);  
int _rdrand_u64_step(unsigned __int64 *);
```

Second, a library may offer high-level functions that manage various aspects of RDRAND invocation. Functions may, for example, handle the possibility of random value unavailability with retry loops.

```
int _rdrand_get_uint_retry(unsigned int retry_limit, unsigned int *dest);
```

Other functions may handle multiple invocations of RDRAND to fill arrays or other objects of arbitrary size and type.

```
int _rdrand_get_rand_step(unsigned int n, void *dest);  
int _rdrand_get_rand_step_retry(unsigned int n, unsigned int retry_limit,  
void *dest);
```

4.4 Guaranteeing DRBG 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 vendors will want to use the DRNG to seed and reseed in an ongoing manner their current software PRNG. Some may furthermore 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 makes use of 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 reseeded occurs long before the maximum number of samples can be requested by RDRAND.

There are two approaches to structuring RDRAND invocations such that DRBG reseeded 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.

The code example below exercises the second approach to guarantee that the random value returned is based on an entropy sample independent from the prior function invocation and independent from the subsequent function invocation.

```

/* CBC-MAC together 32 128 bit values, gathered with delays between, to guarantee
some intervening reseeds */
/* Creates a random value that is fully forward and backward prediction
resistant, suitable for seeding a NIST SP800-90 Compliant, FIPS 1402-2
certifiable SW DRBG */

int _rdrand_get_seed128_retry(unsigned int retry_limit, void *buffer)
{
    unsigned char m[16];
    unsigned char key[16];
    unsigned char ffv[16]; /* feed forward value */
    unsigned char xored[16];
    unsigned int i;

    for (i=0;i<16;i++)
    {
        key[i]=(unsigned char)i;
        ffv[i]=0;
    }

    for (i=0; i<32; i++)
    {
        usleep(10);
        if (_rdrand_get_n_uints_retry(2, retry_limit, (unsigned long long
int*)m) == 0) return 0;
        xor_128(m, ffv, xored);
        aes128k128d(key, xored, ffv);
    }

    for (i=0;i<16;i++) ((unsigned char *)buffer)[i] = ffv[i];
    return 1;
}

```

Note the use of `xor_128()` and `aes128k128d()`, two functions found in most AES library implementations. The random data gathered from the multiple RDRAND invocations should be combined using a suitable cryptographic function to yield a single 128-bit value that is suitable for use as a seed by a secure software PRNG. Here, `xor_128()` and `aes128k128d()` together implement the AES-CBC-MAC mode of operation with AES.

5 References

- [1] Makoto Matsumoto and Takuji Nishimura, (1998). Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1, January 1998.
- [2] Z. Gutterman, B. Pinkas, and T. Reinman. (March, 2006). Analysis of the Linux Random Number Generator. <http://www.pinkas.net/PAPERS/gpr06.pdf>
- [3] CVE-2008-0166 (<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166>)
- [4] Information Processing Standard Publication 197, Nov 26th 2001, Specification for the Advanced Encryption Standard (AES)
<http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.
- [5] SP800-90 reference is 'NIST Special Publication 800-90, Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised), March 2007' http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf
- [6] NIST Special Publication 800-38A, Recommendation for Block Cipher Modes of Operation, http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist_sp800-38A.pdf
- [7] [*Intel® 64 and IA-32 Architectures Software Developer's Manual*](#), Section 7.3.18 RDRAND.
- [8] [*Intel® Processor Identification and the CPUID Instruction*](#). April 2012.
- [9] [*Intel® 64 and IA-32 Architectures Software Developer's Manual*](#). Volume 2A and 2B: Instruction Set Reference.

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>

Intel, the Intel logo, VTune, Cilk and Xeon are trademarks of Intel Corporation in the U.S. and other countries.

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

Copyright© 2012 Intel Corporation. All rights reserved.

Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804