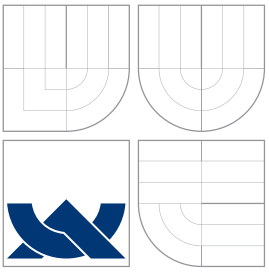
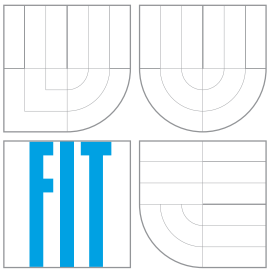


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

RDRAND: IA-64 A IA-32 INSTRUKCE PRO GENEROVÁNÍ NÁHODNÝCH ČÍSEL

RDRAND: IA-64 AND IA-32 INSTRUCTION FOR RANDOM NUMBER GENERATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN ŤULÁK
AUTHOR

VEDOUCÍ PRÁCE — Ing. TOMÁŠ KAŠPÁREK,
SUPERVISOR

BRNO 2014

Abstrakt

Tato práce popisuje implementaci a testování Linuxové knihovny, vytvořené pro poskytnutí abstrakce mezi uživatelem a instrukcí RdRand od Intelu (Ivy Bridge RNG). Samotná instrukce je rovněž zběžně popsána a jsou ověřeny některé její vlastnosti, jako kryptografická bezpečnost výstupních dat a reálná rychlost této instrukce.

Abstract

This thesis describes implementation and testing of a Linux library, created for providing an abstraction layer between a user and the Intel's RdRand instruction (Ivy Bridge RNG). The instruction itself is briefly described and some of its properties are tested here, like cryptographic strength of its output and the real speed of the instruction.

Klíčová slova

RdRand, Entropie, Ivy Bridge, IA-32, IA-64, náhodná čísla

Keywords

RdRand, Entropy, Ivy Bridge, IA-32, IA-64, random numbers

Citation

Jan Ťulák: RdRand: IA-64 and IA-32 Instruction for Random Number Generation, bakalářská práce, Brno, FIT VUT v Brně, 2014

Rozšířený abstrakt

Počínaje generací Ivy Bridge, Intel začal vybavovat své procesory hardwarovým generátorem náhodných čísel. Tento generátor, pojmenovaný Intel Secure Key, je možné využívat pomocí strojové instrukce RdRand pro rychlé získání náhodných čísel s vysokou entropií.

Intel Secure Key je založen na tepelném šumu ovlivňujícím chování dvou vzájemně propojených invertorů. Tyto invertory jsou oba nejprve nastaveny do stejného stavu 1 (tedy jsou nestabilní) a po následném odpojení řídící části se jeden z nich přepne do stavu 0. Pravděpodobnost překlopení konkrétního invertoru by měla být 1:1 a podle toho, který invertor se překlopil, je vygenerován jeden bit. Na tento jednoduchý generátor navazuje filtrovací logika, která se snaží odstranit korelace mezi vygenerovanými hodnotami. Protože tato logika výrazně zpomaluje generování, je její výstup použit jako neustále se měnící inicializační vektor pseudonáhodného generátoru založeného na AES algoritmech.

Rychlost RdRandu by podle Intelu měla dosahovat až 800 MiB/s, přičemž rychlost dostupná pro jednu výpočetní jednotku procesoru odpovídá 800 / počet jednotek. Reálné zkušenosti s RdRandem jsou však malé a jeho využití není zatím příliš rozšířeno. Bylo tedy nutné provést výkonnostní testy a ověřit jak tato tvrzení, tak statistické vlastnosti generátoru.

Statistické testy výstupu generátoru nenalezly žádné chyby ani v několika desítkách TB dat. Výkonnostní testy potvrdily, že rychlost RdRand instrukce na procesorech generace Ivy Bridge se blíží udávané rychlosti a pro její dosažení je třeba využít všech výpočetních jednotek. V následující generaci je však výkon pouze poloviční, zřejmě kvůli změnám v architektuře. Tato informace byla následně potvrzena Intelem.

S ohledem na odhalení aktivit NSA Edwardem Snowdenem ve věci úmyslného oslabování kryptografických standardů a spolupráce s výrobci HW a SW je vhodné věnovat zvýšenou pozornost i bezpečnosti tohoto generátoru náhodných čísel. V textu práce jsou zběžně popsány některé možnosti, jak by RdRand mohl negativně ovlivnit bezpečnost systému, který jej využívá. Z možných vektorů útoku lze jmenovat například změnu elektrických vlastností polovodiče vedoucí k odlišnému chování, která přitom není zjistitelná ani při optické inspekci čipu.

K RdRandu je možné přistupovat i skrz API poskytované knihovnami jako je OpenSSL. To je však komplikovanější, než v případě úzce zaměřené knihovny. Jednak to přináší řadu závislostí na dalších knihovnách a jejich verzích a jednak je takové rozhraní zbytečně komplikované.

Část software vytvořeného pro tuto práci byla tedy uvolněna jako knihovna pro operační systém Linux, poskytující abstrakci nad instrukcí RdRand a umožňující snadno generovat větší množství dat, než 16/32/64 bitů poskytovaných instrukční sadou procesoru. Rovněž testování, zda konkrétní procesor disponuje touto instrukcí bylo zjednodušeno, neboť AMD pracuje na vlastní variantě generátoru náhodných čísel na procesoru a tato knihovna může být snadno rozšířena i pro další implementaci. Vytvořená knihovna má jen minimální závislosti na dalším SW.

Citace

Jan Ťulák: RdRand: IA-64 and IA-32 Instruction for Random Number Generation, bakalářská práce, Brno, FIT VUT v Brně, 2014

RdRand: IA-64 and IA-32 Instruction for Random Number Generation

Declaration

Hereby I declare that I wrote this work on my own and all used sources are stated and correctly noted as citations.

.....
Jan Ťulák
May 19, 2014

Acknowledgement

I want to thank to Jiří Hladký, my supervisor in Red Hat for all his help and ideas.

Contents

1 Introduction

Generation of true random numbers is a stochastic process. In opposite, computers are deterministic machines and thus they are unable to generate true random numbers by using abilities of a Turing machine. But random numbers are crucial in cryptography and once computers began to be used in this domain, people tried combine these two conflicting requests – to allow a deterministic machine to act stochastically.

There are two solutions of this problem. We can stay with completely deterministic machines and through series of mathematical operations compute pseudo random numbers, that seems to be random, but when using the same initial state and algorithm, we get the same numbers again. Or we can add some source of entropy to the system, a device that is physically stochastic and measure the stochastic process (thermal noise, radioactive decay, etc.).

The second approach can provide real random numbers, but it requires online testing for case of hardware failure and also, finding of an entropy source with a good speed, an uniform distribution (without a bias) and with a reliable price, size, energy consumption and other parameters is difficult. Because of this, a lower quality HW generator is connected with a device that tests the bias and selects only some bits and then the generated random values are used as a seed for a cryptographically secure pseudo-random generator, which can lead to great speed enhancement without loosing much of the randomness.

The problematic of quality of the entropy for a specific purpose is wide and this work is not intended to cover it to great length, but still this area has to be shortly mentioned. Clearly, different request have a developer of a video game, a researcher performing a simulation and an army for encrypting their information. The researcher needs numbers that seem to be random but aren't – he or she needs to be able to repeat the simulation with the same initial state to get the same result[1].

The video game developer can also need the repeatability (e.g. for generating a terrain), but in another case, like decisions of an artificial intelligence, the repeatability may not be requested and in case of gambling highly unwanted. And the army needs the random numbers generator to be completely stochastic to prevent an enemy to decipher their messages. Another example, where random numbers are used, is the *Monte Carlo* method of solving definite integrals. Each of these cases has different requests for quality, speed and price.

Due to prices of *Hardware Random Number Generators (HW RNG)* and because the few cheap solutions never got widely used, they were for long time domain of governments and big corporations only, leaving the consumer electronic to rely on *Pseudo-Random Number Generators (PRNG)*. PRNG algorithms developed to great success over time, providing enough entropy for what most people usually do and also for most of cryptographic needs (*Cryptographically Secure Random Number Generators - CSPRNG*), but still it needs to be seeded by data with at least some entropy from the beginning. And as more and more of our daily life depend on computers, the importance of keeping our data secure have raised up.

In 2012 [13] Intel added a Digital Random Number Generator (DRNG) on their chips in Ivy

[1] In some cases of computing, researchers are even keeping the same machines, as a different machine would provide a different result due to inner number representation and architectural differences [32]

Bridge generation and allowed programmers to use it as part of instruction set of that CPUs. Intel named the instruction RdRand and its own implementation and the underlying DRNG hardware implementation is named *Intel Secure Key* (previously code-named Bull Mountain Technology) [25]. Intel used combination of HW RNG and CSPRNG, solution known as *Cascade Construction RNG*, where the relatively slow HW RNG[2] works as a seed generator for much faster CSPRNG. As is showed later in this thesis, in section 6.2 Testing, the difference in speed is about thousand times.

This step brought HW RNGs to general public, but the limitation on Intel CPUs only means that there is still big part of the information technology market without such solution - in x86 world there is another big player, AMD, who did not implemented this instruction yet and also many different architectures, like ARM, in mobile devices. So programmers cannot count on the presence of a HW RNG on casual computers yet.

Furthermore, the Intel's RdRand instruction is still mostly unknown and there are also questions about reliability of this generator, which is sealed into the chip and unable to be audited [33] if it is really manufactured according of published scheme [22] or if there is a backdoor.

It is important to notice that if there could be a backdoor in the RNG, there could be possibly backdoors also in any other parts of the CPU, opening possibilities for many others attacks which could achieve *the same* result. But hiding a backdoor to RdRand could be done more easily than to, for example, a HW acceleration for AES encrypting, so I agree with Linus Torvalds [5] that RdRand alone is great if we do not need cryptographically secure numbers, but for cryptographic usage, it is better to use it just as one of more sources of entropy and seed some CSPRNG by it.

As I'm interested in computer security (on some level), as well as I'm interested in Linux, when I got the possibility to work on implementation of a library using RdRand in production environment of Red Hat corporation, I was interested in it and choose it as my thesis. During the work, we have discovered unexpected issue with half performance on some CPU. This was handed to Intel, yet without a result.

In spring 2014, when this thesis was written, no other implementation of RdRand than Intel's one existed[3], so the term *RdRand* is used just as the instruction implemented by *Intel Secure Key* technology.

[2] The HW RNG itself has output about 3 gigabits per second [9], but these values are biased, so amount of the bites is reduced to concentrate the randomness.

[3] Although AMD is working on their own implementation for their future Excavator architecture [30], probably named RDRND.

2 Random Numbers and Deterministic Machines

The term *Entropy* is usually used as the *measure of disorder and uncertainty of a system* [6]. Claude E. Shannon defined it for information theory as the average of uncertainty (unpredictability) of an information source [29, p. 396]. The important properties of an information source with high entropy are [31, p. 150]:

Uniform distribution: Probability of each value should be the same; no values should be generated with higher frequency of occurrence than others.

Independence: No value can be inferred from the others.

In this report the term *entropy* can also refer to a random value itself from a information source.

2.1 Pseudorandom Numbers

When we pass any value to a *pseudorandom number generator* (PRNG), the PRNG will produce a long sequence of values, that seems to be random and statistical tests on these sequences (if the PRNG is strong enough) should not find any correlation between the produced values. However, the produced sequence is finite and thus after a time it become to repeat itself. This is because the PRNG is just an implementation of an algorithm, computing mathematical operations on the previously computed value (on the start of the PRNG, we need to fill it with a *seed* - a starting value) and after finite count of steps the algorithm gets to the point of its start.

A simple example of such PRNG, one of the most basic but the most widely used [31, p. 151], is the *linear congruential random number generator* (LCRNG) defined as

$$X_n = (aX_{n-1} + b) \mod m \tag{2.1}$$

where X_n is the nth number of the sequence (X_{n-1} respectively the previous). a, b and m are constants and the selected values have big impact on quality of the output. X_0 is a seed.

The LCRNGs are still useful because of their speed and easy implementation in various non-cryptographic situations, when their disadvantages (most importantly their predictability [31, p. 152, 153]) are not important [27, chapter 16.1].

Because with knowing the initial seed we can repeat all the pseudorandom sequence, it is important to have a secure, random seed if we want to use a PRNG in cryptography, except of having a *cryptographically secure pseudorandom number generator* (CSPRNG). In computers, the seed can be frequently extracted from hard-to-predict events like user interaction or network communication.

We can say a sequence generator is pseudorandom, if its output looks random, even under statistical tests.

2.1.1 Cryptographically Secure PRNG

Encryption algorithms can sometimes be used as the PRNG, for example DES or AES. These provide a very good result [31, p. 153-156]. Using of an encryption algorithm is planned as a feature for the future version of the library.

The important property of a PRNG, that can be considered cryptographically secure, is *unpredictability*. That means that it is impossible under our computational knowledge to predict (forward or backward) any number, even if we know the used algorithm[1] and/or the entropy source(s). The forward/backward security has to apply also for cases when the inner state of the RNG is disclosed, for example by reading it in memory.

The *forward security* grants us, that knowing of the current state of the generator, it is not possible to learn the previously generated values. The *backward security*, also known as *break-in recovery* means, that even if an attacker knows the state of the generator in a specific time, it is not possible recover the state and thus predict future values.

2.2 True Random Numbers

To produce random values, *true random number generators* (TRNGs), are using physical events that are hard to be predicted. The TRNGs can measure absolute values, timing or occurrence of such events. An examples of what can be used for generating true random numbers are radioactive decay, avalanche effects on reverse-biased electronic components and thermal, atmospheric and other sources of noise, and others [21, p. 6].

However, real using of such devices is problematic. The sources of entropy are frequently somehow biased and cyclic, so it is necessary to have an online testing and filtering, which is usually significantly slowing the output speed. Also, the price can be a problem too[2].

2.3 Random Numbers in Linux

The *Linux random number generator* (LRNG) is gathering its entropy from events that are very hard to predict: mouse movements, key-presses, interrupt sources of the system, jitter of access times to disks. These events are saved with a timestamp into an entropy pool [36].

LRNG is offering an API for its use. Except of C function `get_random_bytes`, it also provides two device drivers, `/dev/random` and `/dev/urandom`. The difference between these two devices is in way, how they handle user requests, providing two different levels of security. The `/dev/random` is producing more secure randomness and if there is not enough entropy in the entropy pool, then it can block the reader, until enough entropy is gathered. `/dev/urandom` then never blocks its output and produce less secure values, but is faster[3] [36, chapter 1].

[1] This is the reason, why LCRNGs are not considered as CSPRNGs - if an attacker have just 3 following numbers of the basic LCRNG, X_i, X_{i+1} and X_{i+2}, he can create an equation for each of them from the equation 2.1 $X_n = (aX_{n-1} + b) \bmod m$. The three equations creates a system of equations with 3 unknowns a, b and m, which is very easy to solve. After solving it, the attacker knows all the constants and then he can compute any preceding or following number in the sequence.

[2] They usually cost from hundreds to thousands Euros. For a brief overlook, see a comparison on Wikipedia [35].

[3] Just for illustration, the speed of `/dev/random` on my machine is about 300-400 KiB/s, while `/dev/urandom` is about 16 MiB/s

The LRNG is composed from three independent, asynchronous processes. Firstly, there is the collection of the entropy from all system. In the second process, the entropy is saved into the entropy pool and on last, when a random value is requested, the third process generates it using SHA-1 algorithm [4] [36, chapter 2].

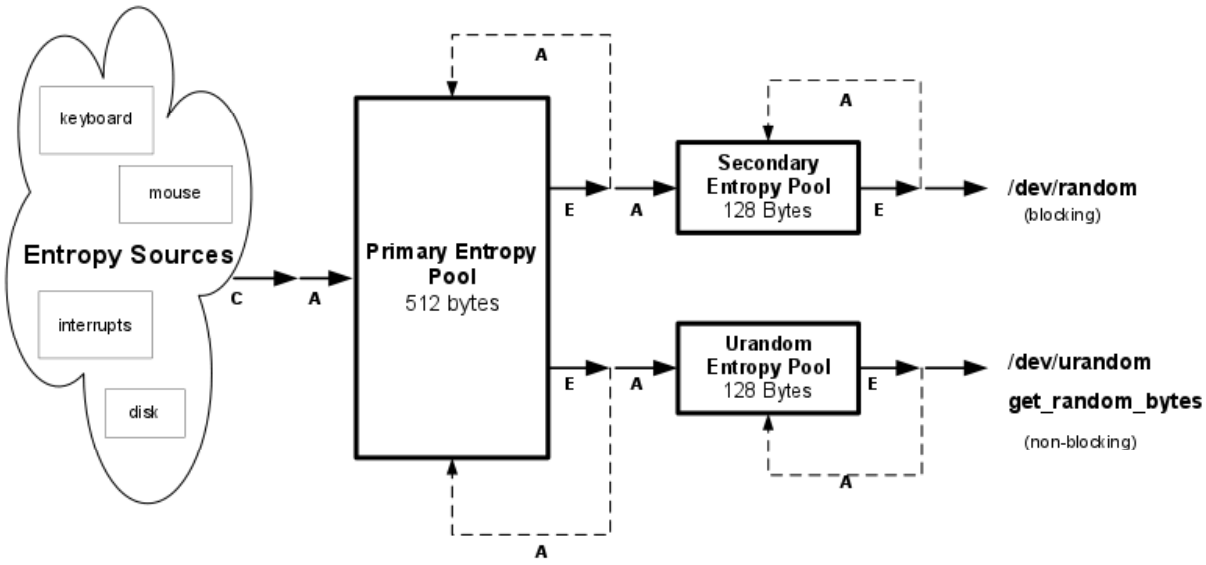


Figure 2.1: Scheme of the LRNG. The entropy is collected (C) and added (A) to the primary pool. Entropy is extracted (E) from the secondary or urandom pool. Whenever entropy is extracted from a pool, some of it is also fed back into this pool (broken line). The secondary pool and the urandom pool draw entropy from the primary pool [36].

When a random value is generated, the process uses the relevant secondary or urandom pool as the seed and the level of entropy in the pool is lowered. When the level of entropy is too low, the pool is refreshed from the primary pool. In case of low entropy in all pools, the blocking `/dev/random` waits until the system gathers more entropy, while `/dev/urandom` just generates randomness from any entropy it has.

2.4 Summary

As was noted, good true random number generators are hard to find, especially for an end-user usage. Although physical computers has some sources of entropy, with moving to virtualized systems the sources are disappearing [21] and furthermore, with raising requirements of security, the importance of good random numbers is more important than before.

Also, as is showed in *Analysis of the Linux Random Number Generator* [36], even kernel generators can has security flaws and allow some form of forward or backward attack.

For these reasons, it would be useful to have widely accessible and widespread fast and high quality TRNG or at least slower TRNG with CSPRNG. That device should be available to all users, even within virtualized systems. How RdRand can help with these points is shown in other chapters of this thesis.

[4] In all the process, the only one non-linear cryptographic function is SHA-1, that is used three times, along with some register shifting and pool mixing [36, section 2.6].

3 The RdRand instruction

First public information about RdRand came sometime during year 2011 [13], a year before the CPUs with it were released and Intel itself sends patches to add support into Linux in summer of the same year [2]. Later, RdRand was added between Linux entropy sources for `/dev/[u]random`. According to known information [33], Intel tried to have `/dev/[u]random` rely only on their instruction, but that was denied.

After disclosure of extends of NSA spying activities by Edward Snowden in summer of 2013 [8, 26], a petition for removing RdRand from Linux entropy sources was created [5]. Although supported by just 8 signatures, it got wide attention on information-technology aimed news pages and magazines, like Slashdot.org [14]. The petition was closed after Linus Torvalds responds with scorn:

> ...
>
> Short answer: we actually know what we are doing. You don't.
>
> Long answer: we use rdrand as _one_ of many inputs into the random pool, and we use it as a way to _improve_ that random pool. So even if rdrand were to be backdoored by the NSA, our use of rdrand actually improves the quality of the random numbers you get from `/dev/random`.
>
> ...

3.1 History

This is not for the first time Intel is producing a HW RNG. Around 1999, their chipsets of 8xx series had a TRNG [15, chapter 1.3.5][4] as part of the Intel FWH (82802AB or 82802AC) component. This RNG was analog – a thermal noise was affecting a resistor. The noise was amplified and forwarded to a voltage–controlled oscillator. Its output was combined with another oscillator with much higher frequency and the drift between these two frequencies provided the requested entropy [20].

Pairs of generated bits were digitally processed, using Von Neumann Corrector to enhance its statistical properties by removing some bias. Due to this, the RNG has a variable bitrate, averaging around 75 Kbit/sec.

Input Bits	Output
0, 0	None
0, 1	1
1, 0	0
1, 1	None

Table 3.1: A Von Neumann corrector

3.2 Intel Secure Key

The Intel Secure Key (ISK) uses cascade construction, combining a HW RNG with CSPRNG into one sealed block on CPU, which is compliant with many security standards, including NIST SP800-90, FIPS-140-2, and ANSI X9.82 [18]. Although it is impossible to audit it, there was found no evidence of low entropy or anything that would deny the security standards compliance - neither with tests in section 6.1 Testing, nor any other tests anyone else did[1].

3.3 Physical Implementation

One important thing about ISK with a big impact on the performance, but also price of that solution is that there is only one unit on a die and the unit should be the same on all CPUs[2] [18, Chapter. 3.1]. Because all processing units (PUs)[3] on one die share the RNG, one thread reading random numbers from it can never be faster than $Totalspeed\frac{1}{PUs}$. The effect of this is that the performance is scalable until amount of PUs and the maximum output speed are exceeded (see section 6.2 Testing), and price of the CPU is less affected.

Each ISK unit consists three basic parts: A hardware entropy source, a conditioner and a *deterministic random bit generator* (DRBG) [18]. The frequency of the RNG is independent on the rest of the CPU and is set to 800 MHz.

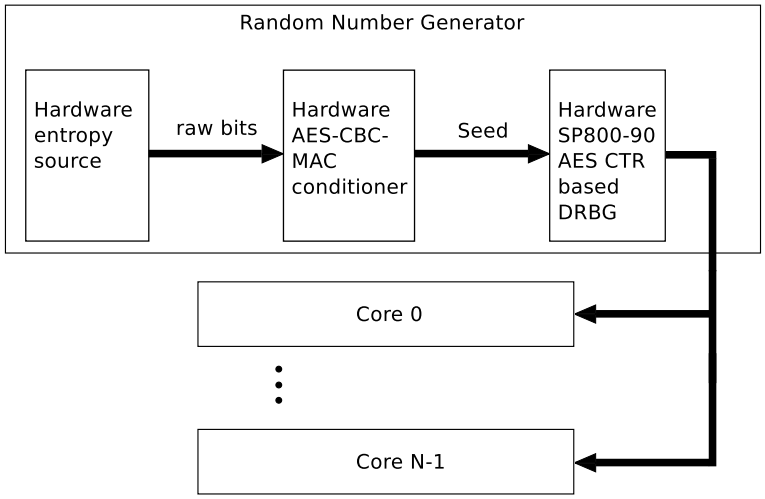


Figure 3.1: An Intel Secure Key unit

3.3.1 Entropy Source

The entropy source is a metastable circuit, with unpredictable behavior based on thermal noise [10]. In figure 3.2, the middle (red) part is the heart of the circuit, a RS-NOR latch. As its *reset* and *set* inputs are wired together, when an impulse is brought on these inputs,

[1] I assume that such revelation would become quickly known and broadly discussed, but all tests I have found have the same conclusion as mine.

[2] We didn't find the „be the same" officially confirmed and it can change in future CPUs. But empirical test results made on range of different CPU types provided the same characteristics, with exception of Intel Xeon CPUs as is described in the 6.

[3] Two physical cores with hyper-threading count as four PUs.

the latch sets to 1 or 0 based on thermal noise. To provide better distribution, there is the bottom (blue) negative feedback. Based on the output of the latch, charge on capacitors in the negative feedback is adjusted and then this negative feedback slightly shifts the chance for next bit to be opposite. The longer is a sequence of the same bits, the higher charge is on the capacitors and bigger effect it has.

Finally, when the latch is settled in one state, the top (green) part of the circuit detect it, saves the bit (and send it further), wait a little time and then it sends a pulse on the R/S inputs of the latch to produce a new bit. This entropy source has its own frequency, different from the rest of the RNG, about 3 GHz.

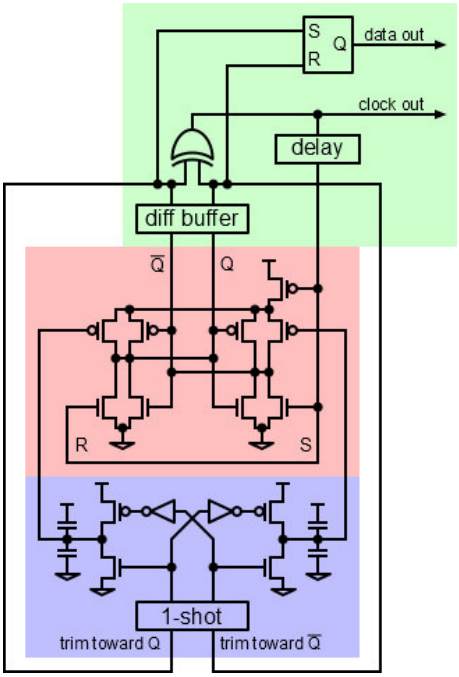


Figure 3.2: Circuit scheme of the entropy source [10].

3.3.2 Conditioner

The entropy source provides random bits with some entropy, but due to implementation and the feedback, it can be unbalanced and produce values in something close to oscillating patterns. For this reason, there is the conditioner that adds them to a 256 bit pool, make a set of XOR and AES operations over lower and upper half of the pool and test health of the pool[4] [22, 10]. If the pool is not healthy, the set of operations is repeated.

3.3.3 DRBG

Because the output speed of the conditioner is too slow (just 256 bits per few microseconds), a deterministic random bit generator is connected to the 256 bit pool and every few microseconds (if the pool is healthy) takes it as a new seed. This pseudo random generator then computes 65.536 bits values using a 128-bit AES and put them to an output buffer. From there, they

[4] The health check is done by counting six different bit patterns in the 256–bit pool and comparing the counts with empirically chosen values. The pool is healthy only if the numbers are in the specified ranges.

can be taken by the RdRand instruction after 64-bit blocks[5] [22, 10]. Reseeding of the DRBG is required after all 1024 blocks are used, but usually it will reseed more frequently, about each 10 microseconds [18, Chapter 4.4].

3.3.4 Built-in Self-Tests (BIST)

Intel Secure Key contains also built-in self-tests. After a reset, the BIST at first test health of the DRBG and conditioner, then it tests the entropy source. In the first part, ES is disconnected, a previously determined bit sequence is 'generated' and the BIST checks the output with a built-in value. In second phase, the entropy source is connected and few sequences are generated. If the entropy source would be bad, the previously checked health check in conditioner would detect it.

In case one of the BIST parts would not be finished correctly and the RdRand instruction is called, just zeros are returned and carry flag is cleared [22].

3.4 Existing Usages

RdRand is already used in Linux kernel for both blocking and non-blocking pools. For the non-blocking pool(`/dev/urandom`), it is possible to completely feed the pool with RdRand [2]. For blocking pool (`/dev/random`), the RdRand is used as one of more entropy sources in `drivers/char/random.c`.

Listing 3.1: Adding RdRand (and any other platform-depending HW RNG) to blocking pool with XOR, at line 1042 in `drivers/char/random.c` [1].

```
for (i = 0; i < LONGS(20); i++) {
        unsigned long v;
        if (!arch_get_random_long(&v))
                break;
        hash.l[i] ^= v;
}
```

Another well-known and widely spread system already using RdRand is OpenSSL. This library added the support in version 1.0.1 [24, Chapter. 3.2 Generation]. There the RdRand is provided as one engine of many and requires to be explicitly enabled.

Listing 3.2: Simplified example of usage of RdRand in OpenSSL [24].

```
OPENSSL_cpuid_setup();
ENGINE_load_rdrand();
ENGINE* eng = ENGINE_by_id("rdrand");
ENGINE_init(eng);
ENGINE_set_default(eng, ENGINE_METHOD_RAND);

// Here use the engine with some RAND_ method

ENGINE_finish(eng);
ENGINE_free(eng);
ENGINE_cleanup();
```

[5] 64 bits are always taken out, no matter what size is the target register of the instruction. Thus, it is not possible to achieve the full speed of the DRBG on 32-bit system, as there is no difference for the Intel Secure Key usage from pulling 64 bits. Just some bits are thrown away. See subsection 6.2.3 Testing for a performance test.

In Windows 8, RdRand is used for example during boot for `ASLR` *(Address Space Layout Randomization)* [28, 34].

3.5 Security

Security of the RNG is important, but it is hard to check. During the last few years, the possibility of hardware Trojans (malicious manipulation during the manufacturing process) gained attention, but none was yet found. This doesn't necessarily means that no such exists, because detection methods are much more difficult than with their software kindreds, and frequently destructive.

If such Trojan is not detected in software because of altering verifiable outputs (like AES HW Acceleration, which would not encrypt the data), there is not much other ways, than to slice the chip and optically inspect it. The verification against a „golden chip" is expensive, destructive and slow and requires to have the „golden", trusted chip for comparison. Yet it can find alterations in circuits that could open some backdoors. But recently, another possible attack vector was described by international group of scientists in their work *Stealthy Dopant-Level Hardware Trojans* [3], in which they directly presented such attack on the Intel's RdRand.

This new category of attack is not modifying the circuits' layout and is undetectable by the optical inspections, because it affects the functionality of transistors instead of changing the layout. By slight changes in doping[6], the modified gate can become static.

While with a correct RdRand an attacker has a chance of $1/2^{128}$ to correctly guess a random number, the described attack could increase the chance to $1/2^{n}$ [3, Chap. 3.2, page 9], but the output of such modified RdRand is still looking randomly. In the work $n = 32$ is achieved, while it still pass NIST random number test suite.

This kind of attack (predictability) is different from what the authors of the petition [5] for removing RdRand from `/dev/random` fears. Their point is that RdRand, as a part of the CPU, can actively try to reduce entropy of the pool. If the RdRand knows with which values its output is XORed (they can be in cache of the CPU), it can produce inverted value and effectively impact the entropy pool. But I don't have enough understanding of this Linux subsystem to be able to evaluate this claim.

[6] Doping means adding specific impurities to the clear silicon to change its electric properties.

4 The Library

According the needs of Red Hat I created a library providing basic interface over the RdRand instruction as well as a simple application using this library. The most important reason for Red Hat's own implementation was licensing; Intel has supplied its own library providing similar interface to this one, but under its own license, which could cause problems with modification, redistribution and in combination with GNU GPL. Thus, this work has been released under GNU LGPL 2.1[1]. Also, it was important to test the library and to create documentation for it.

When drafting the library interface, I have at first surveyed the library example from Intel [18], as well as OpenSSL API [23]. The first draft included functions described in API in 4.1.2.2 (simple wrappers) and 4.1.2.3 (single numbers) and one function for longer sequences: `rdrand_get_bytes_retry`.

Also, I planned to implement not only the usual, fast method, but also some more secure, that would avoid relying on the security of the CSPRNG (see section 3.3 The RdRand instruction for more details), yet I wasn't sure about an interface for this functionality.

This draft was discussed with Jiří Hladký. A need for more functions for longer sequences emerged from the discussion, so other functions from 4.1.2.4 (multiple numbers) were added and also the more secure generation was discussed. Initially, I wanted to implement the fast or more secure method switch as a state variable, set up in an initialization function or passed as an argument to the generating functions. After deeper analysis of such solution and its impact on performance and usability, I decided to implement the more secure methods as two independent functions, described in 4.1.2.5 (secure generating).

Also, the library could be used on machines with current hardware, but with legacy software with no support for RdRand in system libraries (for example RHEL 5). For this reason, dependency on system libraries of specific versions was declined. This means that although Linux has a `X86_FEATURE_RDRAND` flag for testing whether the RdRand is available and function similar to wrappers in 4.1.2.2, it wasn't used.

In March 2014, this library (and the generator) was pushed into Fedora package repository [37, 38] in version 1.0.5. In later versions, the functionality of this library is going to be extended above the requested and described range. Specifically, some sort of encryption of generated values is going to be added to counter the possibility of predicable output.

4.1 API

The library, if installed into the system, can be included by using `#include <librdrand.h>`. In the time of this work, the library is using the following API.

4.1.1 Constants

RDRAND_SUCCESS Returned by function if a random number(s) was generated correctly.

[1] For full text of the license, see [7].

RDRAND_FAILURE Returned by function if a random number(s) was NOT generated correctly.

RDRAND_SUPPORTED Returned by `rdrand_testSupport` function if the CPU supports RdRand.

RDRAND_UNSUPPORTED Returned by `rdrand_testSupport` function if the CPU doesn't know RdRand.

4.1.2 Functions

4.1.2.1 Non-Generating Functions

These functions are not generating any random numbers.

int `rdrand_testSupport` **(void)** – Detect if the CPU support RdRand instruction. Returns `RDRAND_SUPPORTED` or `RDRAND_UNSUPPORTED`.

4.1.2.2 Simple Wrappers

These methods are simply wrappers of an ASM code, which generates only one n-bits number. Although these functions are provided, I expect that they will be used only infrequently. Returns `RDRAND_SUCCESS` or `RDRAND_FAILURE`.

int `rdrand16_step` **(uint16_t *x)** – Generates 16 bits of entropy through RdRand.

int `rdrand32_step` **(uint32_t *x)** – Generates 32 bits of entropy through RdRand.

int `rdrand64_step` **(uint64_t *x)** – Generates 64 bits of entropy through RdRand.

4.1.2.3 Generating Single Number

More complex functions than the previous – in case of RdRand failure, these functions will try it again for the specified amount of times. Negative `retry_limit` implies default value with which the library is compiled. Returns `RDRAND_SUCCESS` or `RDRAND_FAILURE`.

int `rdrand_get_uint16_retry` **(uint16_t *x, int retry_limit)** – Generates 16 bits of entropy through RdRand.

int `rdrand_get_uint32_retry` **(uint32_t *x, int retry_limit)** – Generates 32 bits of entropy through RdRand.

int `rdrand_get_uint64_retry` **(uint64_t *x, int retry_limit)** – Generates 64 bits of entropy through RdRand.

4.1.2.4 Generating Multiple Numbers

As a single random value is usually not enough, the library provides also functions for generating multiple bytes of random values. For higher speed, all these functions are generating values in 64bit blocks when it is possible. These functions also accept `retry_limit` as the previous ones. Returns bytes of successfully generated values.

size_t `rdrand_get_bytes_retry` **(void *dest, const size_t size, int retry_limit)** – Generate `size` bytes of random data.

size_t `rdrand_get_uint64_array_retry` **(void *dest, const unsigned int count, int retry_limit)** – Generate `count` of 64bit blocks of random data.

size_t `rdrand_get_uint32_array_retry` **(void *dest, const unsigned int count, int retry_limit)** – Generate `count` of 32bit blocks of random data.

size_t `rdrand_get_uint16_array_retry` **(void *dest, const unsigned int count, int retry_limit)** – Generate `count` of 16bit blocks of random data.

size_t `rdrand_get_uint8_array_retry` **(void *dest, const unsigned int count, int retry_limit)** – Generate `count` of 8bit blocks of random data.

size_t `rdrand_fwrite` **(FILE *f, const size_t count, int retry_limit)** – Generate `count` bytes of random values and write it to the `f` stream

4.1.2.5 Secure Generating

As documented in the chapter 3 The RdRand instruction, the CPU is using an pseudorandom generator in connection with an entropy source. If the user want to avoid the risk of taking two numbers from the same pool, that were generated from the same seed by the PRNG for some reason, it is possible to use these functions, that guarantee[2] by reseeding the internal entropy pool, that each 64-bit generated value is independent on the previous or the next one.

These methods should be used only in a single thread. If more threads or processes try to generate random numbers with these two methods, the library has no possibility to enforce the reseeding of the PRNG and the numbers generated in two different threads can be from the same, non-regenerated pool. However, between numbers in one thread, the reseeding is always guaranteed[3] with all reliability we can have without knowing the implementation details.

size_t `rdrand_get_uint64_array_reseed_delay` **(uint64_t *dest, const size_t count, int retry_limit)** – Generate `count` of 64bit values. Forces reseed by waiting 20 microseconds before each generating. The delay duration was selected according a delay in Intel's reference

[2] Based on description of Intel Secure Key in subsection 3.3.3 The RdRand instruction. Verification of this claim is not possible due to sealed implementation in the CPU.

[3] According to known information [22, sec. 2.4.2], the DRBG requires reseeding after 512 128-bit outputs, that is 1024 of 64-bit. Thus if this amount of generated values is skipped, the pool has to regenerate.

implementation, but was doubled for sake of higher security. It can happen that the reseeding speed can be slower than this delay (if the speed with non-secure methods is markedly – more then half – slower than the ideal 800 MiB/s) and in such situation this delay may not be enough.

size_t `rdrand_get_uint64_array_reseed_skip` **(uint64_t *dest, const size_t count, int retry_limit)** – Generate `count` of 64bit values. Forces reseed by generating and throwing away 1024[4] 64-bit values per one saved.

4.2 RdRand Calling

The RdRand instruction is called in three functions: `rdrand16_step`, `rdrand32_step` and `rdrand64_step`. In case that the compiler compiling this library knows RdRand instruction and `x86intrin.h` header file is included, the three named functions are just a renaming of `_rdrandXX_step` functions. But if the compiler does not know the instruction (for example when the version is too old), or the header file is not installed on the system, then the functions directly call a byte code.

Listing 4.1: Byte code called in `rdrand64_step`.

```
asm volatile (".byte␣0x48;␣.byte␣0x0f;␣.byte␣0xc7;␣.byte␣0xf0;␣setc␣%1"
                    : "=a" (*x), "=qm" (err));
```

Byte code instead of instruction in the assembly language is used to support compilers that do not know RdRand instruction. A specific example of such compiler can be *Red Hat Enterprise Linux 5*, whose `gcc` compiler is from year 2008[5].

If the library is compiled for a 32-bit system, then the `rdrand64_step` function contain two calls of the RdRand instruction to fill lower and higher half of a 64-bit number, as it is not possible to use 64-bits registers on a 32-bit system.

4.3 Intelligent Generating

Most functions of the library that generates an array fill the array values one by one, as it was passed to the function, just using larger data type if possible, having as little operations as possible. The `rdrand_get_bytes_retry` is the only one that is applying a simple heuristic for achieving the best possible speed even when passed memory area is not aligned to 64-bit blocks.

The function computes offset of the first 64-bit aligned block within the given memory space and then, if the offset is different than 0, it fills the few unaligned bytes at the beginning individually. After that, the generating continues like in other methods by filling 64-bit blocks until the end, potentially ending again by less than 8 bytes that need to be filled individually (not in 64-bit chunk). Because of this approach, the function has a performance impact that can be notable on small memory areas[6], but on large aligned areas the performance difference is almost undetectable.

[4] 1023 should be enough, but 1024 is better to remember.
[5] According information provided by gcc itself on any machine with RHEL 5.
[6] See section 6.2 Testing for details.

4.4 Support Testing

Because the RdRand instruction is not accessible on all machines, it is necessary to provide an easy tool to check it. This is done by function `rdrand_testSupport`. This function gets information about a CPU through `cpuid` assembly language instruction and tests a vendor string of the CPU. If the string is "GenuineIntel", the CPU vendor is Intel[7] and it is possible to test feature bits of the CPU for RdRand flag. Without the vendor check, it would be possible that some other vendor has another feature flag on the same bit as Intel has RdRand.

4.5 Development and Testing Options

Testing whether some functions that should produce random numbers are correctly working (and for example not just reusing part of memory without overwriting it) is difficult. Thus, it is possible to compile the library with defined constant `STUB_RDRAND`[8]. This overwrites the RdRand instruction calling, resulting in all returned bits enabled (each byte is `0xFF`). This allows to easily see, whether the generated values are correctly used in full length.

[7] Currently the only vendor providing this instruction. See chapter 3 The RdRand instruction

[8] When using gcc, flag `-dSTUB_RDRAND` can be used.

5 Rdrand-Gen

Because the library is for C language only, using it for example with shell scripts would be difficult. For this reason, a simple application was also created, which is installed with the library, and which can be used for generating random values without the need of using C language by the user.

The generator has four optional command-line parameters to modify its behavior. Firstly, `--amount` can be used to generate specific amount of bytes of randomness. Suffixes K, M, G and T are accepted for easier use and when this option is not used, the application is generating indefinitely until it is stopped, for example by KILL signal.

The second parameter is `--method`, which allows user to change the default method `rdrand_get_bytes_retry` for the two reseeding functions. The names of the methods are made shorter for the interaction with the user. Third parameter `--output` is used for specifying the output file – without it, the random values are printed on `stdout`.

The `--threads` can specify, how many threads the generator will run in parallel for better performance (as measured in section 6.2 Testing, a single core can't utilise RdRand fully). By default it is set to 2, because according to Intel [17], in Ivy Bridge generation of CPUs (in which the instruction was added) there are still CPUs with just two processing units.

5.1 Underflow Recovery

Although is stated in Intel's Software Developer Manual [16, chapter 7.3.17] that exceeding the speed of the internal generator is unlikely, and although according to *unverified* information (for example on StackOverflow [19]) it should not be possible to achieve it in current generations (specifically on Ivy Bridge) of Intel's CPU, we decided that the application should be working with good performance even in case of slower internal generator. The importance of this decision become even more obvious after finding that on *dell-pr1700-02.lab.bos.redhat.com* the CPU wasn't able to handle more than four parallel threads reading from RdRand[1].

The principle is simple: By default, there is tolerance for few failures, implemented in the library itself. In such case, a new call of the RdRand instruction is made immediately after a failure. But if the RdRand is too slow, amount of the failures in a row exceeds a specific limit[2].

In case of exceeding of the HW RNG speed, the generator application tries to lower the speed of acquiring. This is at first done by decrementing threads count by one and new try. If this solution is not working or is not possible (that means, when the threads count was lowered to a single thread, or was such from the beginning), delays are being inserted between calls. The delays are then lengthened with each unsuccessful call. If even in this case the HW RNG is not able to provide enough random values, the application ends with an error message[3].

[1] Unfortunately, I cannot provide a statistic probability of such situation – only one machine from all I have tried had this problem and I wasn't able to find that anyone other would experienced this too, which is, with regards of the currently low usage of the instruction, not surprising.

[2] Currently 3, but can be changed in the source code.

[3] Such situation would be clearly a sign of a hardware error and thus it is questionable if the generated values would be still really random

6 Testing

During the development, we measured speed of the library and after getting the code on decent performance and finishing the main development work, we also used some statistical tests batteries. More information about these can be found in section 6.1 Testing.

For the tests, multiple operating systems were used.

OS	Arch.	Kernel version	GCC version
RHEL 5.10	i686	2.6.18-348.el5PAE	Red Hat 4.1.2-54
RHEL 5.10	x86_64	2.6.18-348.el5	Red Hat 4.1.2-54
RHEL 6.4	i686	2.6.32-358.el6.i686	Red Hat 4.4.7-3
RHEL 6.4	x86_64	2.6.32-358.el6	Red Hat 4.4.7-3
RHEL 7.0	x86_64	3.10.0-54.0.1.el7	Red Hat 4.8.2-3

Table 6.1: Used systems versions.

All tests were done on all listed systems[1]. Fresh instalation of the operating system was made on every machine for these tests. No additional than default services were running and no other work was done with the machine during the tests.

6.1 Statistical Testing

OS	Arch.	Machine
RHEL 7	x86_64	*hp-aladdin-01.lab.bos.redhat.com*
RHEL 5	x86_64	*intel-brickland-02.lab.eng.rdu.redhat.com*
RHEL 7	x86_64	*intel-canoepass-01.lab.eng.rdu.redhat.com*
RHEL 5	x86_64	*intel-canoepass-02.lab.eng.rdu.redhat.com*

Because it is important to be sure that generated values are truly random, two test suites were used: PractRand and TestU01. From PractRand, test battery `RNG_test` was used and from TestU01, `BigCrush` and `Alphabit`. These test suits found no deviation at all.

6.1.1 PractRand

From PractRand suite, just the shipped `RNG_test` battery was used in the manner of listing 6.1. About 16 TB of randomness was tested for a single run of PractRand battery.

Listing 6.1: PractRand test battery usage

```
(time ( stdbuf -eL -oL rdrand-gen -t$SIMPLE_THREADS -o \
>(stdbuf -oL RNG_test stdin32 -tlmax 16T  -tlfail) ) )
```

6.1.2 TestU01

Test01 is, in opposite of PractRand, a C library, without any shipped binaries usable for testing. Thus a testing program `TestU01_raw_stdin_input_with_log` by Jiří Hladký [12] was used.

[1] For raw data see Appendix A Attachments

This program can read values from standard input and pass it to the test batteries, which can be selected. The used commands are in listing 6.2 to listing 6.4. On every machine, threads count for the generator was set to number of PUs -1, leaving one PU for system and the test itself.

Listing 6.2: TestU01 BigCrush battery

```
(time ( stdbuf -eL -oL rdrand-gen  -t THREADS -o \
>(stdbuf -oL $TESTU01 -b) ) )
```

Listing 6.3: TestU01 Alphabit battery for all bits

```
(time ( stdbuf -eL -oL rdrand-gen  -t THREADS -o \
>(stdbuf -oL $TESTU01 --Alphabit=40:0:32) ) )
```

Listing 6.4: TestU01 Alphabit battery for one bit

```
(time ( stdbuf -eL -oL rdrand-gen  -t THREADS -o \
>(stdbuf -oL $TESTU01 --Alphabit=35:0:1) ) )
```

The TestU01 batteries tested on each run about 20 terabytes of randomness at sum.

6.1.3 Conclusion

The statistical tests passed successfully with tens of TB of generated data. This can't provide reliable information whether the RdRand has a backdoor or some side-channel security hole (see section 3.5 The RdRand instruction for more details), but it shows that the generated values have good statistical properties and cannot by distinguished from real random numbers.

6.2 Performance Testing

Because the performance of the RNG is important, we had to measure it. There are generally two options how the performance can be measured:

- Speed of the library itself, minimizing other influences
- Speed of the application using the library, including printing the generated values on stdout

Although during the development both options were measured, here the first option is tested primary, because it provides better information about the Intel Secure Key, not so biased by output routines or by speed of a memory medium.

A Bash script named `perftest.sh` in the `tests/` directory contains a battery of performance tests, which are described later in subsections. The battery runs the specified set of tests ten times to get a median values, in cyclic way; between two runs of each test, all other tests were started. Within each test description, the used command called in the test is included. All other options that are not specified within such command were left with default values of the `RdRand` executable[2]:

- `--numbers, -n`: *printing of generated values is not enabled*
- `--thread, -t`: *2 threads used for generation*

[2]Only relevant options that has an impact on the performance are described.

- `--duration, -d`: *3 seconds* – empirically measured that 3 seconds are the minimum duration when the result is not influenced by initialization overhead.
- `--repetition, -r`: *2 times repeat the test, print the average value* – important when testing all methods at once.
- `--chunk-size, -c`: *size of the memory space filled in one call: 2048 of 64-bit values* – empirically selected the minimum size which is not slowing the speed by overhead, that means bigger value has no performance effect.

But usually, the duration was set to 5 seconds to be sure that the measured speed is not influenced by startup overhead. Repetition was disabled (set to 1) as only one method is tested at a time and all the tests are started repeatedly. Amount of threads was depending on the specific test as well as the size of the chunk and generated values weren't printed at all.

The listing 6.5 shows how the speed was measured on the generator application, as it doesn't have any built-in speed test like the test in listing 6.6.

Listing 6.5: Measuring speed of the generator application.

```
./rdrand-gen -t THREADS -n AMOUNT | pv -c >/dev/null
```

Listing 6.6: Usage of the testing application

```
./RdRand -m METHOD -r1 -d5 [OTHER OPTIONS]
```

The test application creates a memory space of given size (`chunk-size`)and call the specified library function to fill it. The library function is then called repeatedly for the given time, so the measured speed is not biased by memory allocations or other overhead in the test application and is close to what is the ideal speed of the library, at least if the running time is long enough.

6.2.1 Speed Scattering

OS	Arch.	Machine
RHEL 7	x86_64	*hp-aladdin-01.lab.bos.redhat.com*
RHEL 6	x86_64	*hp-aladdin-01.lab.bos.redhat.com*
RHEL 5	x86_64	*hp-aladdin-01.lab.bos.redhat.com*

To be able to evaluate results of tests, it is necessary to know the spread of speeds in the same conditions during a time. The listing 6.7 shows command used in the test, which was started 1000 times overall (in ten of one-hundred blocks) and measured the speed of a single thread.

Listing 6.7: Test script for scattering testing.

```
./RdRand -m rdrand_get_bytes_retry -r1 -d5 -t1
```

Each version of RHEL has a different histogram and statistical properties, caused by differences in used scheduler settings, system libraries, and compiler and kernel version. The differences weren't deeply investigated, because important was the performance on an unmodified system, not on a system optimized for best speed of this library.

Results of RHEL 5 have the smallest standard deviation (about 1.7 %), but also the lowest median of speed. As the histogram shows, most of runs had speed about 213 MiB/s or little less. Because the speed differences are small, this operating system was selected as the main, from which the results are usually showed in other tests.

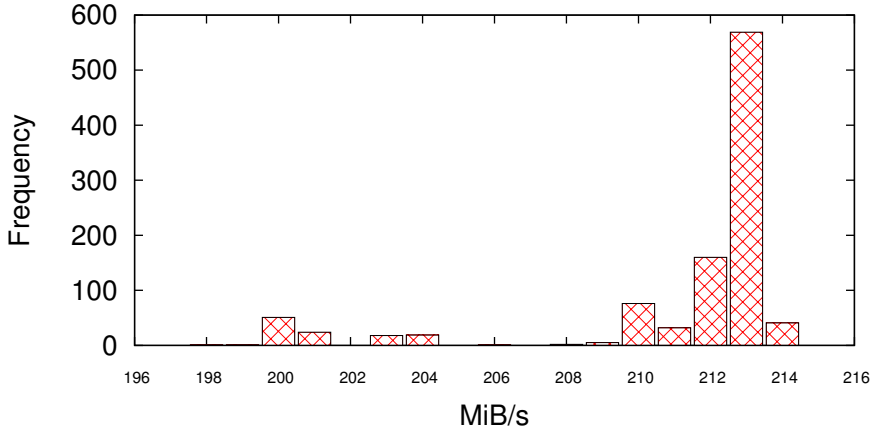


Figure 6.1: Histogram of measured speeds on RHEL 5.

Minimum value	198.194 MiB/s
Maximum value	214.821 MiB/s
Ar. mean	211.700 MiB/s
Median	213.166 MiB/s
Std. deviation	3.711 MiB/s

Table 6.2: Statistical properties on RHEL 5

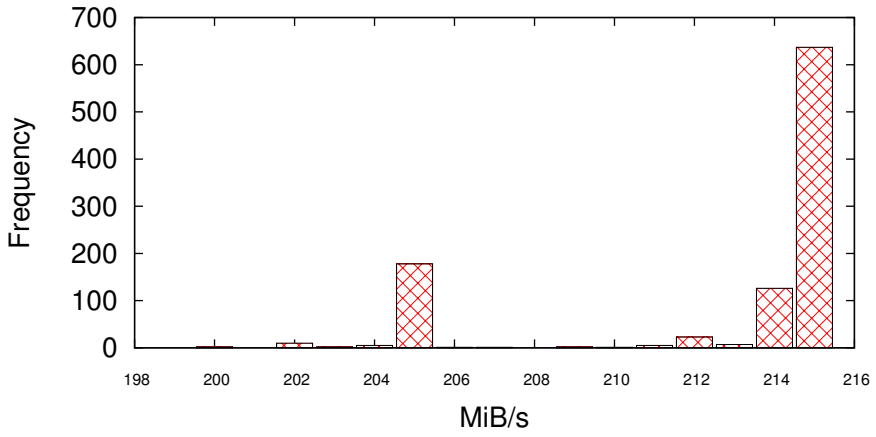


Figure 6.2: Histogram of measured speeds on RHEL 6.

Minimum value	200.201 MiB/s
Maximum value	215.662 MiB/s
A. mean	213.165 MiB/s
Median	215.207 MiB/s
Std. deviation	3.985 MiB/s

Table 6.3: Statistical properties on RHEL 6

The measured speeds are mostly about 215 MiB/s, but quite large amount is gathered on

205 MiB/s mark. RHEL 6 has the best performance results on a single thread, but as can be seen in subsection 6.2.3 Testing, the performance has absolutely different profile from other versions of RHEL once it runs in more threads than the machine has processing units.

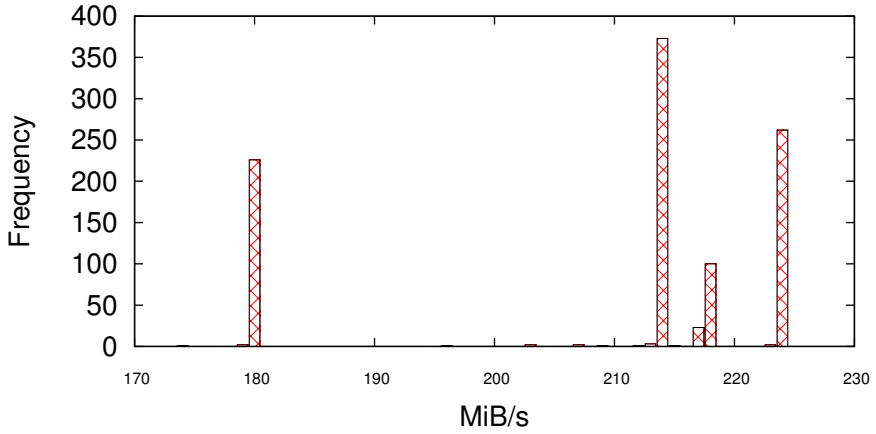


Figure 6.3: Histogram of measured speeds on RHEL 7.

Minimum value	174.253 MiB/s
Maximum value	224.970 MiB/s
A. mean	209.822 MiB/s
Median	214.774 MiB/s
Std. deviation	16.615 MiB/s

Table 6.4: Statistical properties on RHEL 7

RHEL 7 has far the worst standard deviation and while on RHEL 5 and 6 there are one or two dominating speeds, in case of RHEL 7 there are three values, pretty far away (about 180, 214 and 224 MiB/s). On this version of RHEL, the measured speed was the highest, but also the lowest.

6.2.2 Scaling

OS	Arch.	Machine
RHEL 5	x86_64	*hp-aladdin-01.lab.bos.redhat.com*

This test shows how the output speed depends on count of used threads for both the `rdrand-gen` application shipped with the library and `RdRand` performance testing application.

Listing 6.8: Test application command.

```
./RdRand -r1 -d5 -m rdrand_get_bytes_retry -t COUNT
```

Listing 6.9: Generator application command.

```
./rdrand-gen -t THREADS -n\$((THREADS*400))M
```

On the Figure 6.4 Amount of generated bytes in dependency of threads count can be seen that the average performance per one thread is about 170 MiB/s up to four threads. Then, on about 730 MiB/s is the performance peak, where it is not possible to get higher speed

anymore by adding more threads. Because the performance gets stable now (does not drop), this stabilization cannot be caused by scheduler (as the scheduler cause the drop down later in the graph). Thus this has to be the performance limit of the Intel Secure Key.

Because the machine has 8 processing units, but the speed of ISK is reached on 4 threads, the performance is constant from reaching the peak to 8 threads. After that, the operating system began to interrupt the threads to allow all threads to use the CPU and the performance drops to about 500 MiB/s. With more threads, the PUs are better utilized and the performance again rises, but will not achieve the previous value.

Also the difference between the test application and the shipped generator is visible on the peak. The difference is rising with the amount of generated data and on the peak it is about 14 %.

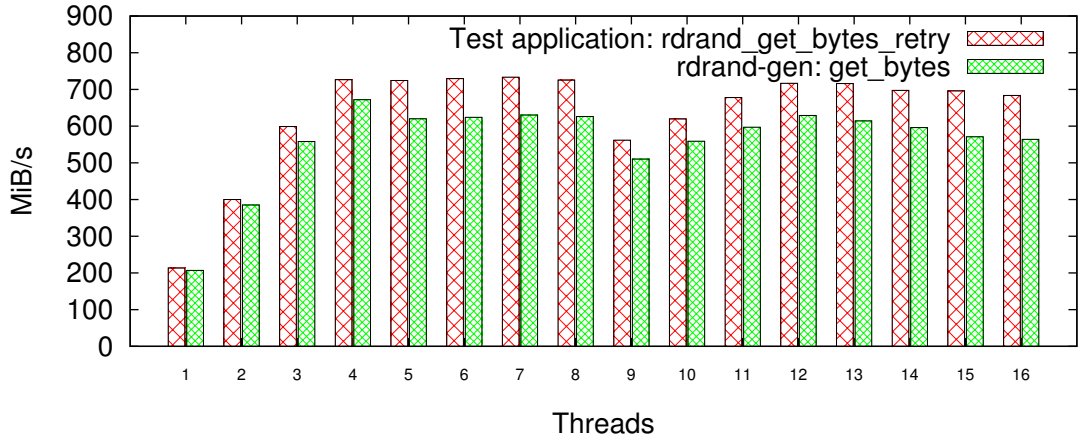


Figure 6.4: Amount of generated bytes in dependency of threads count.

For a single thread, the performance of the thread is above average and it seems to be because the system is giving more computing time to the thread in comparison when another thread is started.

The maximum speed of about 730 MiB/s is equal to $\frac{730 \times 2^{10} \times 2^{10}}{10^6} = 765$ Hz. This is similar to the ideal 800 Hz value given in the section 3.3 The RdRand instruction.

6.2.3 Differences Between OS Versions

OS	Arch.	Machine
RHEL 5	i686	*hp-aladdin-01.lab.bos.redhat.com*
RHEL 5	x86_64	*hp-aladdin-01.lab.bos.redhat.com*
RHEL 6	x86_64	*hp-aladdin-01.lab.bos.redhat.com*
RHEL 7	x86_64	*hp-aladdin-01.lab.bos.redhat.com*

Performance was measured on 32 and 64 bits version of RHEL 5 and 6, but not 7 as RHEL 7 has just 64 bits version [11]. To make the figure 6.5 more readable, just RHEL 5 is included to demonstrate the difference between 32 and 64 bits of the same system. The 32 bits version is according the expectation from the section 3.3 The RdRand instruction almost exactly half.

During the first stages of development we thought that the multiple variants of the RdRand instruction (16, 32 and 64-bits) were created primary because of performance, to avoid wasting of generated bits. But performance testing and later finding of some more documents about

Intel Secure Key showed that the these variants are there probably just for compatibility with non 64-bit operating systems and for programmer's comfort. As it is described in the subsection 3.3.3 The RdRand instruction, 64 bits are always pulled out internally.

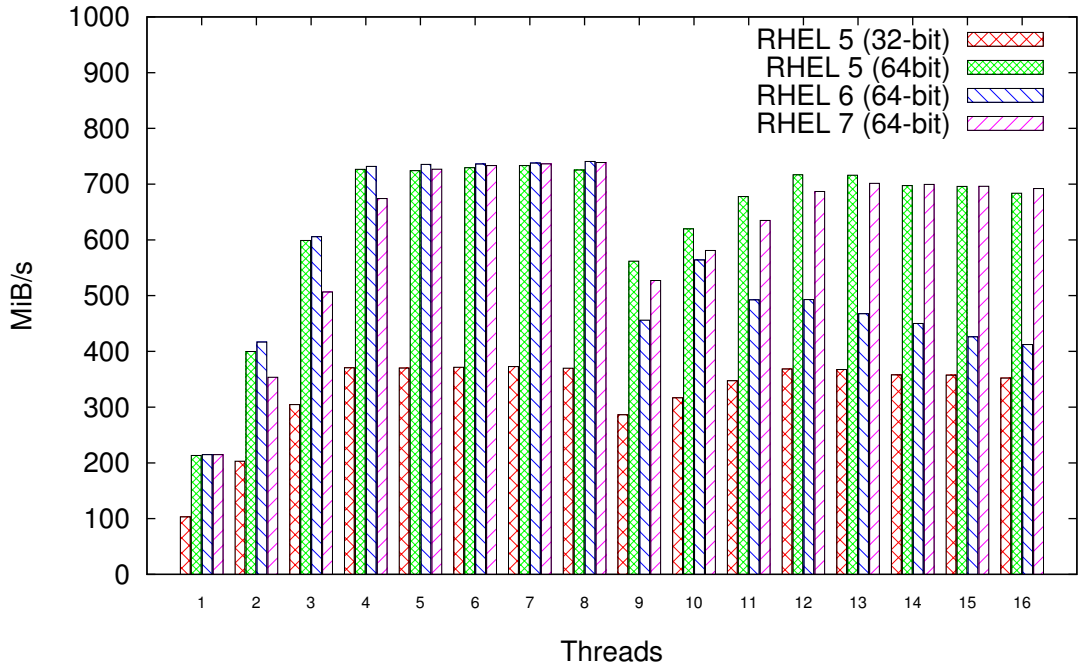


Figure 6.5: The differences between 32 and 64-bit of RHEL 5 and RHEL 7 installation.

The differences between 64-bit RHELs are much smaller, yet the RHEL 7 has worse performance when not on a peak. Furthermore, the peak speed is the same, but is reached with one thread more, so it seems that on RHEL 7, the system is interrupting the threads more frequently. It can be also by worse optimization, but this seems to be less probable, because for a single thread, the speed is the same for all three versions.

Another interesting point is what happens with RHEL 6 after amount of threads reaches the limit of PUs. At first is still close to other 64 bits systems, but after 10 threads, the performance is decreasing to almost half of RHEL 5 and 7.

In an effort to find reason of this difference on RHEL, these tests were done (each of them was independent and not affecting the others):

- Scheduler settings of RHEL6 were changed through `sysctl` to have the same configuration as on RHEL7.
- Current versions of GCC and OpenMP[3] libraries were manually installed.
- The tested application was run also as multiple parallel processes with a single thread, instead of one process with multiple threads using GNU `parallel` program.

Scheduler settings and compiler/library versions had just a small effect on the performance drop on RHEL6. There were just small differences. Running the application as multiple processes

[3]GCC 4.8.2 with other requested libraries was installed alongside of the distribution's 4.4.7 into a custom directory. `Makefile` was edited respectively.

with a single thread, on the other side, was completely different and proved even better, than the results from other systems.

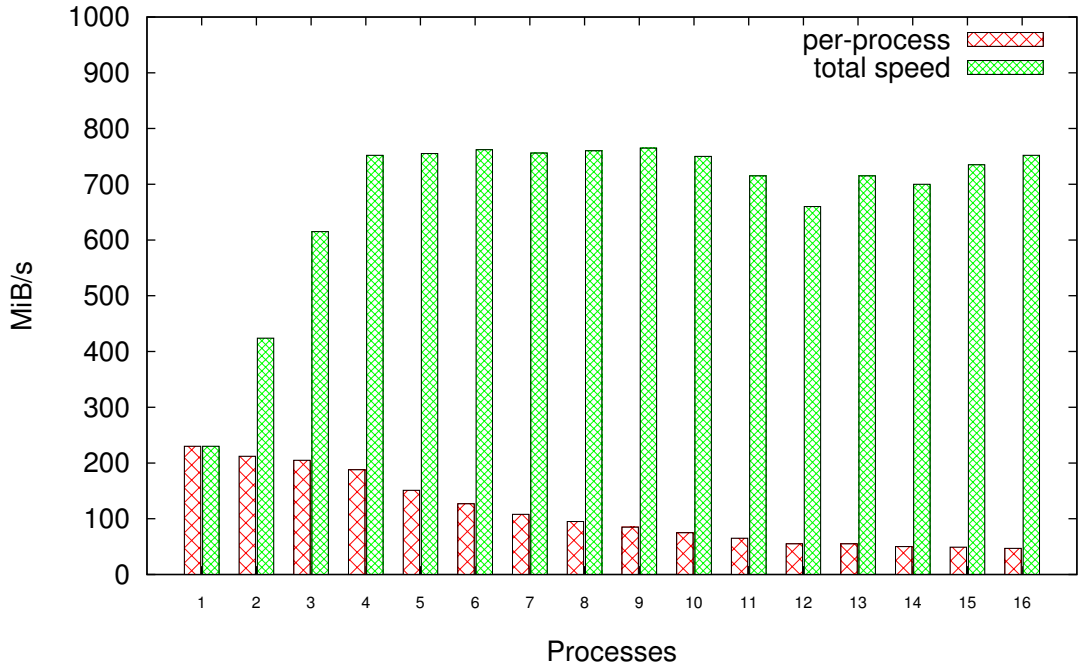


Figure 6.6: The speed of single thread runs of RdRand test in dependency of amount of parallel runs. Per-process and total speeds are showed.

Because changing of the GCC and OpenMP library version did not change the behavior, nor the scheduler options, it seems that the cause is rather in kernel code itself. According of Jiří Hladký, RHEL 6 has also confirmed similar behavior for other applications that uses more threads: it seems that the threads are not running concurrently, but sequentially.

This theory was verified by running time measurement, using `clock()` from `time.h`. Results for 1–16 threads are in section A.2 Attachments. This test measured time for generating in a single thread and also total time. This found that up to 8 threads (the breakpoint) is the total time the same as time of each thread, continually growing. With 9 and more thread, speed of a single thread fall down to values similar to 4-5 threads, but the total time goes up to multiply times of any of the thread. Besides, the times of the threads become more fluctuating. This supports the theory about sequential ordering of threads by kernel, when there are more threads than processing units.

A possible way how to remove this performance issue, is to use a producent–consument model, where the generating threads would run independently and fed a buffer as fast as they could and a single slower thread would not slowdown all application.

6.2.4 Size Dependency

OS	Arch.	Machine
RHEL 5	x86_64	*hp-aladdin-01.lab.bos.redhat.com*

In this test, functions `rdrand_get_bytes_retry` and `rdrand_get_uint64_array_retry` were compared in different sizes of the memory area that was filled with random numbers.

The goal of this test was to find if there is some difference between these two functions; if the additional logic in the first one has an measurable impact.

Listing 6.10: Test command for chunk size dependency.

```
./RdRand -r1 -d5 -m METHOD -c SIZE
```

For a better visibility, the figure with measured values is split to two parts. The figure 6.7 shows the difference from 1 to 32 of 64-bit numbers (quadwords) and figure 6.8 shows the rest, from 64 to 8192 generated numbers.

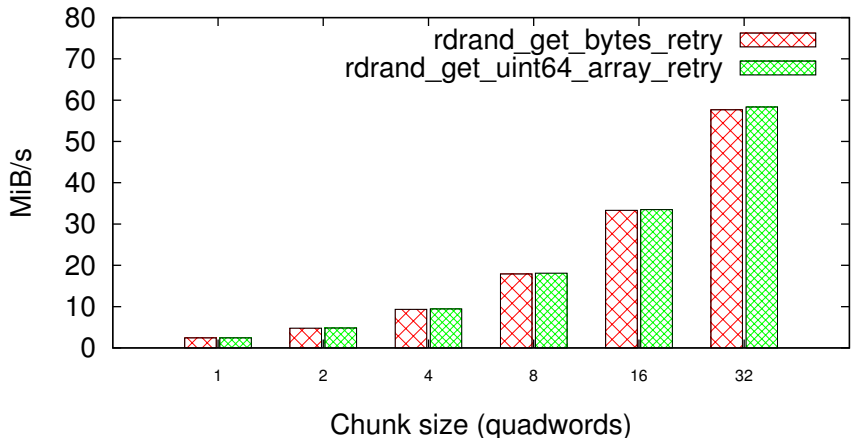


Figure 6.7: The difference of two functions on different sizes of filled memory area, from 1 to 32 quadwords.

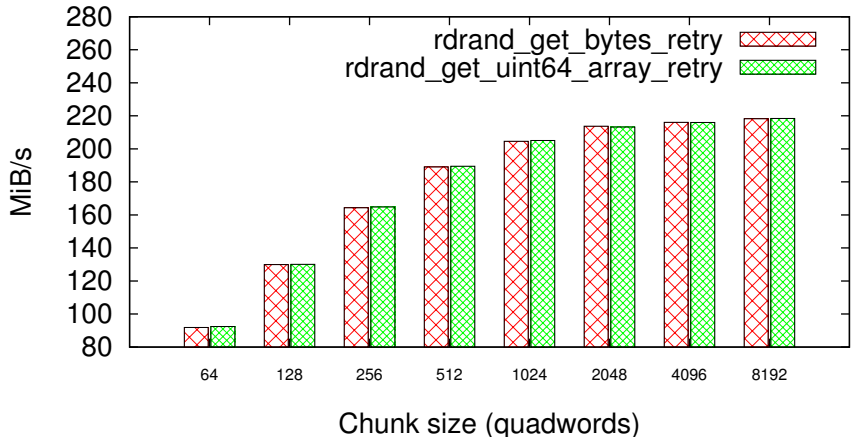


Figure 6.8: The difference of two functions on different sizes of filled memory area, from 64 to 8192 quadwords.

On these figures it is not much apparent, but in numeric data can be found that there is a very small difference and the `rdrand_get_bytes_retry` is about 1 % slower in the figure 6.7. Then the difference is getting smaller and at the size of 8192 quadwords it is just 0.06 %.

The difference is more apparent on small sizes, but as it is not bigger than the standard deviation, the `rdrand_get_bytes_retry` can be safely used in most times.

6.2.5 Fast and Secure Generating

OS	Arch.	Machine
RHEL 5	x86_64	*hp-aladdin-01.lab.bos.redhat.com*
RHEL 5	x86_64	*intel-brickland-01.lab.eng.rdu.redhat.com*
RHEL 6	x86_64	*hp-aladdin-01.lab.bos.redhat.com*
RHEL 7	x86_64	*hp-aladdin-01.lab.bos.redhat.com*

Because the secure methods, described in the section 4.1.2.5 The Library (both functions `rdrand_get_uint64_array_reseed_delay` and its `_skip` twin), should not be used in parallel threads[4], only a single thread comparison between them and the `rdrand_get_bytes_retry` as a fast method was made. The speed was measured on two different machines with different type of CPUs (Intel Core i7 and Intel Xeon).

In tables in this section is shown that the delay method is the slowest.The common, fast method is about one thousand times faster than the skipping method; this is according expectations, because just one per thousand values is used.

The reason, why on RHEL 5 is the delay method just 0.008 MiB/s, is unknown, but it seems probable, that it depends on how kernel sleeps and wakes sleeping processes.

	hp-aladdin-01	intel-brickland-01
Fast	214.530 MiB/s	73.846 MiB/s
Delay	0.008 MiB/s	0.008 MiB/s
Skip	0.218 MiB/s	0.074 MiB/s

Table 6.5: Comparison of speed of a fast method of generating (`rdrand_get_bytes_retry`) and two variants of secure generating on RHEL 5 on two machines.

	RHEL 6	RHEL 7
Fast	216.660 MiB/s	218.347 MiB/s
Delay	0.103 MiB/s	0.101 MiB/s
Skip	0.223 MiB/s	0.215 MiB/s

Table 6.6: Comparison of speed of a fast method of generating (`rdrand_get_bytes_retry`) and two variants of secure generating on hp-aladdin-01.

The speed of the given by `rdrand_get_uint64_array_reseed_skip` should be possible to enhance by using more threads for the skipping, but implementing of this functionality directly in the library would add unwanted dependency for OpenMP (it is used in the test and generator application for multithreading, but library itself can be compiled without it). Because of this, I decided to do not implement it and just note it here that if someone would need higher speed, he or she will probably need to implement its own variant of the reseeding function.

Also note, that while the skipping speed on the Brickland machine is about one third of the Aladdin (see subsection 6.2.6 Testing for more about this difference), the delay is roughly the same. This is because the skipping method speed depends on speed of RdRand per thread, while the delay method does not.

[4] See 4.1.2.5

6.2.6 Half performance on some machines

All *tested* machines with an Intel Xeon CPU (the Brickland, for example) had just half performance of expected values (and in comparison with desktop/laptop processors).

We passed this to Intel Premier program, yet the only information we received through private communication were:

> The expectation need to be set that RDRAND is slower on IVT-EX because IVT is missing some optimizations that IVB has and the measurements ran confirm that after 5 threads, the RdRand throughput maxes out at 375 MiB/s.

And:

> This is not a bug in gcc or any other software. It is a deliberate choice by the CPU designers. Both Ivytown and Haswell have lower RDRAND bandwidth than Ivybridge. Is there a customer case for having higher RDRAND bandwidth? If so, it would be good to get that information in front of CPU architects so they can make the right tradeoffs for future CPUs.

Unfortunately, this is not telling us anything useful and just it raises more questions about security of this solution, whether the optimizations can affect not just the speed. The only thing we know for sure is that on newer CPUs the speed is deliberately slower.

6.2.7 Underflow

On *dell-pr1700-02.lab.bos.redhat.com*, when acquiring values from RdRand in more than four threads (no matter whether this library was used, or some other program), the RdRand wasn't able to meets the requirements and calls of the instructions began to fail. No such another case was found, so it seems to be just a flaw on the specific silicon. With regards to the fact that the CPU in this machine is a prototype, this flaw is understandable.

6.3 Specifications of Referenced Machines

Number of cores is sum of all CPUs on each machine.

dell-pr1700-02.lab.bos.redhat.com
CPU: *Prototype Intel(R) Xeon(R) CPU E3-1285 v3 @ 3.60GHz*
RAM: *4 GB*
Notes: *Dell Precision T1700, 4 GB RAM. The internal RNG was not able to handle more than four parallel threads at November 2013.*

hp-aladdin-01.lab.bos.redhat.com
CPU: *Intel(R) Core (TM) i7-3920XM CPU @ 2.90GHz (1 NUMA node, 4 cores, HT[5])*
RAM: *4 GB*
Notes: *HP elitebook 8770w, used for performance and statistical tests.*

intel-brickland-01.lab.eng.rdu.redhat.com
CPU: *4x Intel(R) Xeon(R) CPU E7-4890 v2 @ 2.80GHz (4 NUMA nodes, 60 cores, HT)*

[5] Hyper-Threading

RAM: *128 GB*
Notes: *Used for performance tests.*

intel-brickland-02.lab.eng.rdu.redhat.com
CPU: *4x Intel(R) Xeon(R) CPU E7-4890 v2 @ 2.80GHz (4 NUMA nodes, 60 cores, HT)*
RAM: *128 GB*
Notes: *Used only for statistical tests.*

intel-canoepass-01.lab.eng.rdu.redhat.com
CPU: *2x Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz (2 NUMA nodes, 24 cores, HT)*
RAM: *32 GB*
Notes: *Used only for statistical tests.*

intel-canoepass-02.lab.eng.rdu.redhat.com
CPU: *2x Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz (2 NUMA nodes, 24 cores, HT)*
RAM: *32 GB*
Notes: *Used only for statistical tests.*

7 Conclusion

In this thesis I have described the Intel Secure Key instruction RdRand and the library I made for its easier usage. Performed tests presented that the real performance is not far from the expected one and that the produced values have a good statistical properties and can be considered at least as pseudorandom (section 6.1 Testing).

Performance testing confirmed that the maximum speed of RdRand is close to the presented value 800 MiB/s, but it also found, that a single thread can never achieve better speed than $800/PUs$ MiB/s, because the current HW implementation is splitting the *maximum* performance, not the current load. On newer CPUs (Ivy Town, Haswell) the production speed of RdRand is just 400 MiB/s. Also, the tests uncovered few performance issues on specific versions of RHEL, namely performance drop on RHEL 6 when there is more reading threads in one application than PUs (subsection 6.2.3 Testing), and a very slow secure–generating method on RHEL 5 (subsection 6.2.5 Testing).

The RdRand is a good and fast source of entropy. Unfortunately, revelations about NSA and other espionage agencies throw a shadow over this technology. One of possible ways to eliminate a possible compromising is encrypting the generated values by AES in counter mode. This option is going to be implemented in future versions of the library, which is already under development.

In current situation, there are some examples of usage, where a potential security risk is not harmful: The speed allows to use it for erasing hard drives with random values instead of zeroes, when the erasing is not slowed down by the RNG. Another example is ASLR or similar cases, when the operating system has just a little entropy collected from other sources soon after the boot, like Windows 8 currently do [28].

Bibliography

[1] Linux kernel source drivers/char/random.c. https://github.com/torvalds/linux/blob/0891ad829d2a0501053703df66029e843e3b8365/drivers/char/random.c, [quoted 2014].

[2] H. Peter Anvin. Direct support for the x86 rdrand instruction. http://lwn.net/Articles/453651/, 2011-6-29.

[3] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy dopant-level hardware trojans. http://people.umass.edu/gbecker/BeckerChes13.pdf, 2013.

[4] Kevin Cernekee. Rng tools. https://github.com/cernekee/rng-tools/, [quoted 2014].

[5] Kyle Condon. [petition] linus torvalds: Remove rdrand from /dev/random. http://www.change.org/en-GB/petitions/linus-torvalds-remove-rdrand-from-dev-random-4, 2013.

[6] Merriam-Webster dictionary. Entropy. http://www.merriam-webster.com/dictionary/entropy, [quoted 2014].

[7] GNU. Gnu lgpl 2.1. http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html, February 1999.

[8] Glenn Greenwald. Nsa collecting phone records of millions of verizon customers daily. http://www.theguardian.com/world/2013/jun/06/nsa-phone-records-verizon-court-order, 2013-6-6.

[9] George Cox Greg Taylor. Behind intel's new random-number generator. http://spectrum.ieee.org/computing/hardware/behind-intels-new-randomnumber-generator, [quoted 2014].

[10] Michael Hamburg. Understanding intel's ivy bridge random number generator. http://electronicdesign.com/learning-resources/understanding-intels-ivy-bridge-random-number-generator, 2012-12-11.

[11] Red Hat. Are 32-bit applications supported in rhel 7? https://access.redhat.com/site/solutions/509373, [quoted 2014].

[12] Jiří Hladký. Csprng. http://code.google.com/p/csrng/, [quoted 2013].

[13] Gael Hofemeier. Find out about intel's new rdrand instruction. http://software.intel.com/en-us/blogs/2011/06/22/find-out-about-intels-new-rdrand-instruction, 2011.

[14] hypnosec. Linus responds to rdrand petition with scorn. http://slashdot.org/story/13/09/10/1311247/linus-responds-to-rdrand-petition-with-scorn, 2013.

[15] Intel. Intel 810 chipset. http://download.intel.com/design/chipsets/designex/29065701.pdf, 1999.

[16] Intel. Intel 64 and ia-32 architectures software developer's manual, volume 1: Basic architecture. http://download.intel.com/products/processor/manual/253665.pdf, 2013. Order Number: 253665-047US.

[17] Intel. Ark. http://ark.intel.com, [quot. 2013-11-15].

[18] Intel. Intel digital random number generator (drng) software implementation guide. http://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide, [quot. 2013-11-15].

[19] David Johnston. What are the exhaustion characteristics of rdrand on ivy bridge? http://stackoverflow.com/a/14443763, [quot. 2013-11-16].

[20] Benjamin Jun and Paul Kocher. Analysis of the intel random number generator. http://www.cryptography.com/public/pdf/IntelRNG.pdf, April 22, 1999.

[21] Peter Krempa. *Analysis Of Entropy Levels In The Entropy Pool of Random Number Generator*. FIT VUT Brno, 2013. Diploma Thesis.

[22] Mark E. Marson Mike Hamburg, Paul Kocher. Analysis of intel's ivy bridge digital random number generator. http://www.cryptography.com/public/pdf/Intel_TRNG_Report_20120312.pdf, 2012-12-11.

[23] OpenSSL. Openssl - rand(3). http://www.openssl.org/docs/crypto/rand.html, [quoted 2013].

[24] OpenSSL. Openssl - random numbers. http://wiki.openssl.org/index.php/Random_Numbers, [quoted 2014].

[25] Eliana Penzner. Digital random number generator (drng) analysis project. http://software.intel.com/en-us/articles/digital-random-number-generator-drng-analysis-project, 2013.

[26] Reuters reporter. U.s. monitored german chancellor angela merkel's phone since 2002. http://www.dailymail.co.uk/news/article-2477593/U-S-monitored-German-Chancellor-Angela-Merkels-phone-2002.html, 2013-10-26.

[27] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, inc., 1996. ISBN 0-471-11709-9.

[28] PT Security. Windows 8 aslr internals. http://blog.ptsecurity.com/2012/12/windows-8-aslr-internals.html, December 4, 2012.

[29] C. E. Shannon. A mathematical theory of communication. http://www3.alcatel-lucent.com/bstj/vol27-1948/articles/bstj27-3-379.pdf, 1948.

[30] Anton Shilov. Amd excavator core may bring dramatic performance increases. http://www.xbitlabs.com/news/cpu/display/20131018224745_AMD_Excavator_Core_May_Dramatic_Performance_Increases.html, 2013-10-18.

[31] William Stallings. *Cryptography And Network Security*. Prentice Hall, 1998. ISBN 0-13-869017-0.

[32] Timothy. Ask slashdot: How reproducible is arithmetic in the cloud? http://slashdot.org/story/13/11/21/2255209/ask-slashdot-how-reproducible-is-arithmetic-in-the-cloud, [quot. dec 2013].

[33] Theodore Ts'o. Theodore ts'o's post on google plus after relevations of extend of nsa spying. https://plus.google.com/+TheodoreTso/posts/SDcoemc9V3J, 2013-9-5.

[34] Chris Valasek and Tarjei Mandt. Windows 8 heap internals. http://illmatics.com/Windows%208%20Heap%20Internals%20%28Slides%29.pdf, [quoted 2014].

[35] Wikipedia. Comparison of hardware random number generators. http://en.wikipedia.org/wiki/Comparison_of_hardware_random_number_generators, [quoted 2014].

[36] Tzachy Reinman Zvi Gutterman, Benny Pinkas. Analysis of the linux random number generator. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1624027&isnumber=34091, http://eprint.iacr.org/2006/086.pdf, 2006. vol., no., pp.15 pp.-385, 21-24.

[37] Jan Ťulák. Fedora package. https://admin.fedoraproject.org/pkgdb/acls/name/RdRand, [quoted 2014].

[38] Jan Ťulák. Fedora package review. https://bugzilla.redhat.com/show_bug.cgi?id=1048815, [quoted 2014].

A Attachments

A.1 Content of the CD

List of content on the attached CD.

bin - Compiled library and generator (for Fedora 20 x86_64).

data - Full results of tests used for this report.

docs - This report and its source files (LaTeX).

literature - Copy of used literature and webpages, if it is publicly available.

README.txt - Text file containing this same list of content and simple instructions how to compile the library.

source - Source files of the library, ready for `./configure && make && make install`.

A.2 Times of threads on RHEL 6

Result of running time measurement on RHEL 6 on *hp-aladdin-01.lab.bos.redhat.com*.

```
 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 1
Data will be printed to stdout if -p is specified.
total clocks (avg): 70
thread 0 clocks (avg): 70
rdrand_get_bytes_retry 215.772 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 2
Data will be printed to stdout if -p is specified.
total clocks (avg): 145
thread 0 clocks (avg): 143
thread 1 clocks (avg): 144
rdrand_get_bytes_retry 418.631 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 3
Data will be printed to stdout if -p is specified.
total clocks (avg): 226
thread 0 clocks (avg): 224
thread 1 clocks (avg): 223
thread 2 clocks (avg): 223
rdrand_get_bytes_retry 600.268 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 4
Data will be printed to stdout if -p is specified.
total clocks (avg): 334
```

```
thread 0 clocks (avg): 330
thread 1 clocks (avg): 330
thread 2 clocks (avg): 329
thread 3 clocks (avg): 329
rdrand_get_bytes_retry 726.714 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 5
Data will be printed to stdout if -p is specified.
total clocks (avg): 522
thread 0 clocks (avg): 515
thread 1 clocks (avg): 516
thread 2 clocks (avg): 515
thread 3 clocks (avg): 516
thread 4 clocks (avg): 516
rdrand_get_bytes_retry 731.115 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 6
Data will be printed to stdout if -p is specified.
total clocks (avg): 753
thread 0 clocks (avg): 741
thread 1 clocks (avg): 741
thread 2 clocks (avg): 738
thread 3 clocks (avg): 739
thread 4 clocks (avg): 741
thread 5 clocks (avg): 742
rdrand_get_bytes_retry 733.988 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 7
Data will be printed to stdout if -p is specified.
total clocks (avg): 1022
thread 0 clocks (avg): 1009
thread 1 clocks (avg): 1004
thread 2 clocks (avg): 1003
thread 3 clocks (avg): 1007
thread 4 clocks (avg): 1009
thread 5 clocks (avg): 1006
thread 6 clocks (avg): 1007
rdrand_get_bytes_retry 733.639 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 8
Data will be printed to stdout if -p is specified.
total clocks (avg): 1330
thread 0 clocks (avg): 1314
thread 1 clocks (avg): 1312
thread 2 clocks (avg): 1312
thread 3 clocks (avg): 1313
thread 4 clocks (avg): 1312
thread 5 clocks (avg): 1313
```

```
thread 6 clocks (avg): 1312
thread 7 clocks (avg): 1312
rdrand_get_bytes_retry 736.122 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 9
Data will be printed to stdout if -p is specified.
total clocks (avg): 1602
thread 0 clocks (avg): 529
thread 1 clocks (avg): 508
thread 2 clocks (avg): 547
thread 3 clocks (avg): 413
thread 4 clocks (avg): 501
thread 5 clocks (avg): 547
thread 6 clocks (avg): 520
thread 7 clocks (avg): 568
thread 8 clocks (avg): 471
rdrand_get_bytes_retry 421.251 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 10
Data will be printed to stdout if -p is specified.
total clocks (avg): 1488
thread 0 clocks (avg): 480
thread 1 clocks (avg): 463
thread 2 clocks (avg): 501
thread 3 clocks (avg): 499
thread 4 clocks (avg): 476
thread 5 clocks (avg): 458
thread 6 clocks (avg): 418
thread 7 clocks (avg): 431
thread 8 clocks (avg): 508
thread 9 clocks (avg): 588
rdrand_get_bytes_retry 603.341 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 11
Data will be printed to stdout if -p is specified.
total clocks (avg): 1611
thread 0 clocks (avg): 237
thread 1 clocks (avg): 376
thread 2 clocks (avg): 460
thread 3 clocks (avg): 493
thread 4 clocks (avg): 380
thread 5 clocks (avg): 390
thread 6 clocks (avg): 396
thread 7 clocks (avg): 365
thread 8 clocks (avg): 446
thread 9 clocks (avg): 390
thread 10 clocks (avg): 353
rdrand_get_bytes_retry 487.343 MiB/s 100.00 %
```

```
 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 12
Data will be printed to stdout if -p is specified.
total clocks (avg): 1660
thread 0 clocks (avg): 435
thread 1 clocks (avg): 296
thread 2 clocks (avg): 389
thread 3 clocks (avg): 474
thread 4 clocks (avg): 470
thread 5 clocks (avg): 419
thread 6 clocks (avg): 442
thread 7 clocks (avg): 284
thread 8 clocks (avg): 354
thread 9 clocks (avg): 440
thread 10 clocks (avg): 376
thread 11 clocks (avg): 443
rdrand_get_bytes_retry 516.715 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 13
Data will be printed to stdout if -p is specified.
total clocks (avg): 1760
thread 0 clocks (avg): 272
thread 1 clocks (avg): 379
thread 2 clocks (avg): 355
thread 3 clocks (avg): 447
thread 4 clocks (avg): 331
thread 5 clocks (avg): 398
thread 6 clocks (avg): 401
thread 7 clocks (avg): 412
thread 8 clocks (avg): 397
thread 9 clocks (avg): 331
thread 10 clocks (avg): 334
thread 11 clocks (avg): 363
thread 12 clocks (avg): 353
rdrand_get_bytes_retry 482.253 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 14
Data will be printed to stdout if -p is specified.
total clocks (avg): 1834
thread 0 clocks (avg): 152
thread 1 clocks (avg): 365
thread 2 clocks (avg): 339
thread 3 clocks (avg): 439
thread 4 clocks (avg): 370
thread 5 clocks (avg): 441
thread 6 clocks (avg): 279
thread 7 clocks (avg): 395
thread 8 clocks (avg): 308
```

```
thread 9 clocks (avg): 455
thread 10 clocks (avg): 354
thread 11 clocks (avg): 368
thread 12 clocks (avg): 276
thread 13 clocks (avg): 377
rdrand_get_bytes_retry 459.098 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 15
Data will be printed to stdout if -p is specified.
total clocks (avg): 1940
thread 0 clocks (avg): 385
thread 1 clocks (avg): 279
thread 2 clocks (avg): 407
thread 3 clocks (avg): 284
thread 4 clocks (avg): 290
thread 5 clocks (avg): 357
thread 6 clocks (avg): 253
thread 7 clocks (avg): 286
thread 8 clocks (avg): 248
thread 9 clocks (avg): 237
thread 10 clocks (avg): 436
thread 11 clocks (avg): 261
thread 12 clocks (avg): 292
thread 13 clocks (avg): 396
thread 14 clocks (avg): 286
rdrand_get_bytes_retry 427.849 MiB/s 100.00 %

 cmd: ./RdRand -m rdrand_get_bytes_retry -r1 -t 16
Data will be printed to stdout if -p is specified.
total clocks (avg): 2046
thread 0 clocks (avg): 188
thread 1 clocks (avg): 327
thread 2 clocks (avg): 315
thread 3 clocks (avg): 271
thread 4 clocks (avg): 382
thread 5 clocks (avg): 329
thread 6 clocks (avg): 329
thread 7 clocks (avg): 350
thread 8 clocks (avg): 364
thread 9 clocks (avg): 299
thread 10 clocks (avg): 323
thread 11 clocks (avg): 392
thread 12 clocks (avg): 362
thread 13 clocks (avg): 396
thread 14 clocks (avg): 295
thread 15 clocks (avg): 323
rdrand_get_bytes_retry 411.730 MiB/s 100.00 %
```

B Glossary

ASLR – Address Space Layout Randomization. Security mechanism involving randomisation of memory addresses of various system data structures to make it harder to change them in an attack.

RNG – Random number generator.

PRNG – Pseudorandom number generator.

CSPRNG – Cryptographicaly secure PRNG.

TRNG – True random number generator.

LCRNG – Linear congruential random number generator – a basic PRNG, probably most widely used [31, p. 151].

LRNG – Linux Random Number Generator – a RNG used in Linux Kernel.

Entropy – Measure of disorder and uncertainty of a system. In this report the term *entropy* can also refer to a random value itself from an information source. See chapter 2 Random Numbers and Deterministic Machines.

PU – Processing Unit. A CPU with 2 cores and Hyper–Threading has 4 PUs.