Entropy transfers in the Linux Random Number Generator

Thibaut Vuillemin, François Goichon, Cédric Lauradoux, Guillaume Salagnac

RESEARCH REPORT
N° 8060
September 2012
Project-Team PRIVATICS

ISSN 0249-6399 ISRN INRIA/RR--8060--FR+ENG

Entropy transfers in the Linux Random Number Generator

Thibaut Vuillemin, François Goichon, Cédric Lauradoux, Guillaume Salagnac

Project-Team PRIVATICS

Research Report n° 8060 — September 2012 — 23 pages

Abstract: One of the services provided by the operating system to the applications is random number generation. For security reasons, the Linux Random Number Generator is built upon the combination of a deterministic algorithm known as the cryptographic post-processing and an unpredictable physical phenomenon called an Entropy Source. While the various cryptographic post-processing algorithms and their properties are well described in the literature, the entropy collection process itself is little studied. This report first presents the different approaches to random number generation, and then details the architecture of the Linux Random Number Generator. Then, we present the experiments we performed to monitor entropy transfers. Our results show that the main source of randomness in the system is the behavior of the hard drive, and that most random numbers produced by the generator are actually consumed by the kernel itself.

Key-words: random number generation, operating systems, linux, entropy

Entropy transfers in the Linux Random Number Generator

Résumé : La génération de nombres aléatoires est l'un des services offerts par le système d'exploitation aux applications qu'il exécute. Pour des raisons de sécurité, le générateur de Linux est construit autour de la combinaison d'un traitement cryptographique déterministe et d'un mécanisme physique réellement non-déterministe appelé source d'entropie. Si les différents traitements cryptographiques et leurs propriétés sont abondamment décrits dans la littérature, le processus de collecte d'entropie lui-même est assez mal connu. Ce rapport, après une présentation des différentes approches de génération de nombres aléatoires, détaille l'architecture du générateur de Linux, puis les différentes expériences que nous avons menées pour observer les transferts d'entropie. Nos résultats montrent entre autre, que la plus grande source d'aléa est le comportement du disque dur, et que la majorité des nombres aléatoires produits dans le système sont consommés par le noyau lui-même.

Mots-clés : génération de nombres aléatoires, systèmes d'exploitation, entropie

1 Introduction

On a computer system, access to hardware devices like the processors, main memory, or network interfaces has to be properly shared between all the users. Not only has the operating system (OS) to provide software interfaces between these hardware resources and the user programs, but also the OS should enforce fairness and other desirable properties amongst users. Otherwise, a malicious user program may try and monopolize a particular resource like the disk drive or the CPU, which damages system performance and may lead to denial of service. Therefore, understanding and controlling the behavior of all shared resources is of critical importance to evaluate resiliency of the system.

One little-studied shared OS resource is the random number generator. Many applications need random numbers. For example, all security-oriented programs use random numbers to generate reliable cryptographic keys. Also, random numbers are key to implementing Monte-Carlo simulations. Because algorithmic generators are too weak to satisfy the randomness needs of all those applications, a majority of operating systems provide a system-level Random Number Generator (RNG). The Linux kernel is not an exception and contains such a generator: the Linux Random Number Generator (LRNG). This component is available both for use by the kernel itself and by user-space applications. A non-privileged user of a GNU/Linux OS obtains random numbers by reading the `/dev/random` or `/dev/urandom` devices.

The Linux Random Number Generator is not a mere deterministic algorithm producing numbers. In order to offer "good quality" randomness, the LRNG implements a so-called True Random Number Generator architecture, which collects so-called *entropy* from various parts of the system. These terms will be defined in Section 2. However, a major drawback of this architecture is that the generator sometimes has to wait for some entropy to arrive, thus yielding very long response times.

To illustrate this situation, we conducted a little experiment, with a program that repeatedly requests one byte from the `/dev/random` device. The results are presented on Fig. 1 below[1]. While the shortest request completion time is under 1μs, the longest response time is about 35 seconds. That is a seven order-of-magnitude difference.

The litterature about random number generation techniques, and about the different generator architectures, is numerous. However, to the best of our knowledge, little has been done on the entropy collection process itself. In this work, we investigate this subject by studying how Linux generates, collects, and handles entropy to be used by the LRNG. The report is organized as follows. Section 2 defines several terms related to random number generation (such as entropy) and presents the different RNG architectures. Section 3 gives an overview of the Linux Random Number Generator. Section 4 describes the monitoring system we implemented to understand to the LRNG and presents the results we obtained. Finally, the Section 6 summarizes our conclusions and perspectives.

2 Context: random numbers

2.1 Random numbers characterization

There are various ways to define or describe what is called randomness. Two important properties are often expected for a sequence of numbers to be called "random", that is *uniform distribution*, and *statistical independence*.

[1]This test is performed on a Ubuntu 12.04 machine, running the default desktop environment. The processors is an Intel Core i3 (2.27 GHz x 4), and the machine has 3.7 GB of RAM. The version of the Linux kernel is 3.2.0. We measure the completion time for each of the 1000 requests.

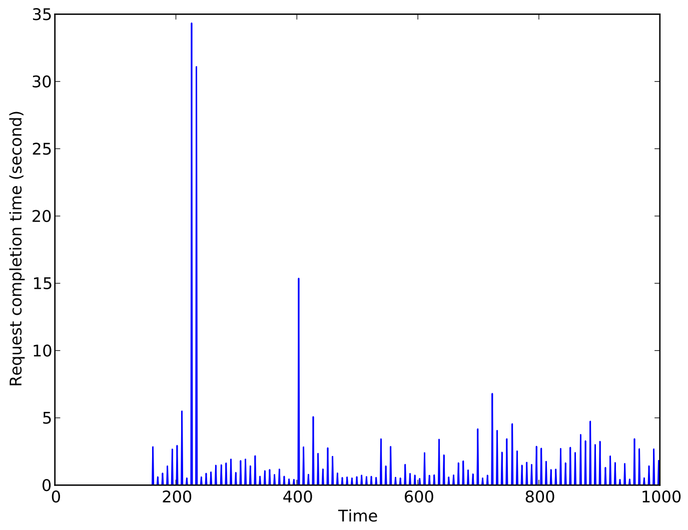


Figure 1: Response time of `/dev/random` for 1000 successive one-byte requests. The average response time is 264.518 ms with a standard deviation of 1675.535 ms.

Let $X_1, X_2, \cdots X_n$ be a sequence of random variables defined from the sample space Ω realizing their values in a measurable state space $\mathcal{X}$. Let $\omega \in \Omega$ be an event of the sample space. Let $x_i \in \mathcal{X}$ be the i-th realization of X_i such that $X_i(\omega) = x_i$ (i.e the definition of a random variable). Let $Pr(X_i = x_i)$ be the probability of the random variable X_i realizing the value x_i. Equation 1 defines that X_i follows a uniform distribution. Equation 2 defines statistical independence between two random variables X_i and X_{i-1}.

$$Pr(X_i = x_i) = \frac{1}{\mathrm{card}(\Omega)} \tag{1}$$

$$\forall i \in \{2, ..n\}, Pr(X_i = x_i | X_{i-1} = x_{i-1}) = Pr(X_i = x_i) \tag{2}$$

However, not all random variables satisfy these two properties. When it is not the case, then the numbers we get are not "purely random" but may still contain some amount of uncertainty. This amount is often named *entropy* and can be described by the following formula.

Let X be a random variable. Let $\mathcal{X}$ be the sample space of this random variable. Let x be a realization of this variable, with $x \in \mathcal{X}$. For any $i \in \mathcal{X}$, let $p(i) = Pr(x = i)$ be the probability of x being i. The *Shannon entropy* of variable X is defined as:

$$H(X) = -\sum_{\forall i \in \mathcal{X}} p(i) \log_2 p(i).$$

In order to use this formula, one has to know the probability distribution function p. If this function is not known, then the Shannon entropy can only be estimated from several realizations of the variable. In the literature, this problem of entropy estimation is an open topic [LPR11].

2.2 Random Number Generators

A *Random Number Generator* is a computer program intended to behave like a random variable. Random number generation always attracted attention from both computer science and mathematics communities. The work of Knuth [Knu97] is seminal in this field. In the literature, different classes of random number generators have been studied:

- pseudo-random number generators (PRNG);
- cryptographically secure pseudo-random number generators (CSPRNG);
- true random number generators (TRNG).

We choose to give a simple description for each of them. Our descriptions are close to the ones given by Viega in his book *Secure Programming* [VM03].

Pseudo-Random Number Generators

A *Pseudo-Random Number Generator* (PRNG) is a finite state machine. The initial value of the internal state is often called the *seed*. Upon each request, a *transition function* computes the next internal state, and an *output function* (sometimes the identity function) produces the actual number.

A PRNG will deterministically produce a sequence of values depending only on the initial seed. This sequence will always be periodic, because there are only so many distincts states the algorithm can visit before rolling over.

Remark If the internal state is stored as a n-bit vector, then the maximum period for the PRNG is 2^n. Indeed, a finite state machine has no other internal memory than its state, so after at most 2^n transitions it will necessarily end up in some already visited state, and from then on it will perform the same transitions again and again.

Example Linear congruential generators [PM88] are a common class of PRNGs. Let x_t be the internal state of the PRNG at time t. Let a, b and m be arbitrary integer constants. The PRNG transition function is:

$$x_{t+1} = a \cdot x_t + b \mod m.$$

Cryptographically Secure Pseudo-Random Number Generators

A *Cryptographically Secure Pseudo-Random Number Generator* (CSPRNG) is a PRNG that satisfies cryptographic conditions [Rö5] in addition to statistical conditions. Two classes of CSPRNGs can be distinguished depending on their security model: so-called *stream ciphers* [RB08] offer empirical security by resisting to the state of the art of cryptanalysis attacks, while *provably secure CSPRNGs* have more theoretical security properties derived by reduction to a difficult problem [BBS86].

As stated by Viega [VM03]:

> Cryptographic pseudo-random number generators are still predictable if you somehow know their internal state. The difference is that, assuming the generator was seeded with sufficient entropy and assuming the cryptographic algorithms have the security properties they are expected to have, cryptographic generators do not quickly reveal significant amounts of their internal state. Such generators are capable of producing a lot of output before you need to start worrying about attacks.

True Random Number Generators

Opposed to a PRNG, which is implemented entirely in software, a *hardware RNG* is based on some unpredictable physical phenomenon called an *entropy source* [Kel04]. Sampling this source is the electronic equivalent of rolling a die or flipping a coin. Examples of such generators include the Bull Mountain RNG implemented in recent Intel CPUs [Int11] which uses thermal noise within the silicon as an entropy source.

However, this architecture is limited by the speed of the physical phenomenon itself. Naive hardware RNGs thus typically offer insufficient throughput for most applications. This is why people generally build so-called *True Random Number Generators* [BH05] by cascading a hardware RNG with a PRNG: the slow-but-very-unpredictable hardware RNG is used to periodically refresh the internal state of the fast-but-deterministic PRNG, in a process called *reseeding*. In this context, the hardware RNG is often referred to as a *entropy harvester* [BIW04].

The resulting behavior is between that of a pure hardware RNG and a PRNG: when sufficient entropy is harvested from the source, then the PRNG is reseeded often and the output numbers are really non-deterministic. However, if the TRNG fails to harvest enough entropy from the source, then it does its best and acts as a pure PRNG.

3 Context: the Linux Random Number Generator

The Random Number Generator implemented by the Linux kernel [MT12, GPR06] is a True RNG: it contains both a PRNG and entropy harvesting mechanisms. In this section, we present the architecture of the LRNG as well as entropy transfer mechanisms.

3.1 Output interfaces

The Linux Random Number Generator provides three output interfaces:

- `/dev/random`
- `/dev/urandom`
- `get_random_bytes()`

The `get_random_bytes()` function is available only from within the kernel. It always return the requested amount of random data. On the other hand, the two `/dev/(u)random` devices are accessible from user-space, even to a non-privileged user. Whereas `/dev/random` can block the requesting process, `/dev/urandom` never blocks.

The Linux source code [MT12] reads that `/dev/urandom` will return as many bytes as are requested. If too many many bytes are required, this will result in “random numbers that are merely cryptographically strong, but they are still acceptable for many applications”. On the other hand, “`/dev/random` is suitable for use when very high quality randomness is desired (for example, for key generation or one-time pads), but it will only return a maximum of the number of bits of randomness contained in the entropy pool”.

3.2 Entropy pools

The internal state is separated into three arrays of unsigned integers called the *entropy pools*: the input pool, the blocking pool, and the non-blocking pool.

The blocking pool, which size is 1024 bits, is related to `/dev/random`. When bits are requested to `/dev/random`, the LRNG reads numbers is this pool. The non-blocking pool, which size is also 1024 bits, is related to `/dev/urandom`. When bits are requested to `/dev/urandom`, the LRNG reads numbers is this pool. It is also used by another function named `get_random_bytes()`. This function can be called from inside the kernel to request random numbers. The input pool, which size is 4096 bits, is connected to the blocking pool, the non-blocking pool and the entropy sources.

Every pool is associated with an integer called *entropy counter*. It is an estimation of the amount of entropy contained in the pool. The entropy counter of a pool varies over time, as explained below:

Random numbers request on the blocking pool When n random bits are requested to `/dev/random`, the blocking pool entropy counter is decreased by n. If it falls to zero, the read operation blocks until more entropy is acquired.

Random numbers request on the non-blocking pool When n random bits are requested to `/dev/urandom` or `get_random_bytes()`, the non-blocking pool entropy counter is decreased by n. However, if it falls to zero, it will not block the request and the device will act as a PRNG.

Entropy transfer When the blocking or the non-blocking pool has to provide n entropy bits but has an entropy counter lower than n, then entropy is transferred from the input pool to the demanding pool. The input pool counter is decreased by n while the counter of the demanding pool is increased by n. If the input pool entropy counter is lower than n, then the transfer is rejected.

Entropy harvesting As a TRNG, the LRNG uses entropy sources to harvest entropy (see Section 2). When n entropy bits are gathered from an entropy source, the input pool is refreshed using those bits and the input pool entropy counter is increased by n.

3.3 Entropy sources

The LRNG depends on the rest of the kernel to provide it with entropy. It exports three functions, which are intended to be called by various device drivers:

- `add_disk_randomness()`,
- `add_input_randomness()`, and
- `add_interrupt_randomness()`.

The `add_disk_randomness()` function is called by the hard disk driver. It uses the the seek time of block layer request events as an entropy source. The `add_input_randomness()` function is called by the drivers of input peripherals such as the mouse, the keyboard, but also a joystick or a tablet. For every event, it uses values representing the event (e.g. "KEY" event, "A" key, key released) as an entropy source. The `add_interrupt_randomness()` function can *a priori* be called in any part of the kernel that generates interrupt. For every interruption, it uses the inter-interrupt timing as an entropy source. Every interruption is not satisfying: the inter-interrupt timing of a timer is predictable. We provide a big picture of the LRNG architecture in figure 2, including the entropy sources, the entropy pools and the output interfaces.

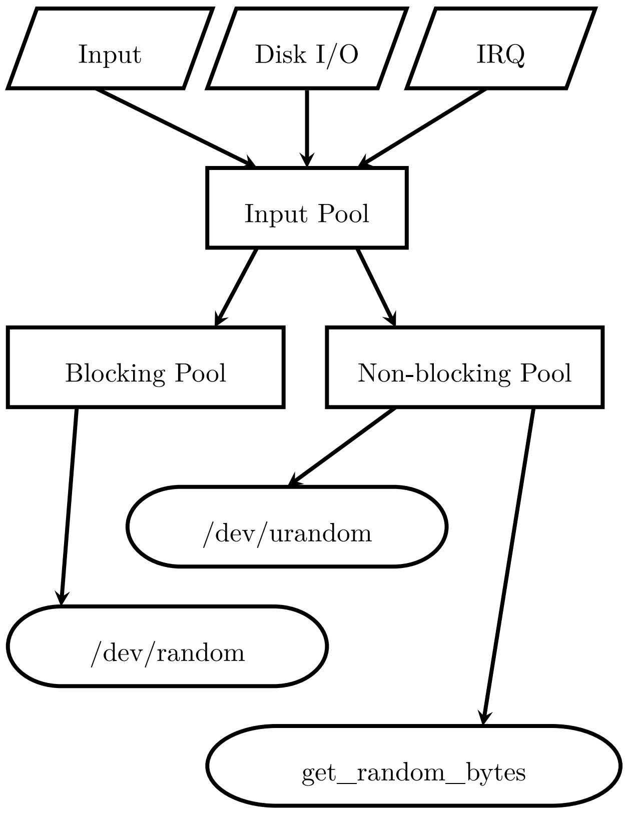


Figure 2: Global architecture of the the LRNG. Lozenge boxes are entropy sources ; rectangle boxes are the three entropy pools, together making up the internal state of the generator. Rounded boxes are the three output interfaces. An arrow between two boxes represents the possibility of an entropy transfer.

3.4 Entropy estimation

In Subsection 3.2, we stated that when n entropy bits are gathered from a source, the input pool entropy counter is increased by n. We now explain how the LRNG computes n, that is, how it estimates how much uncertainty was conveyed by the event. The LRNG has to do such an estimation in order to detect regularities and reject non-uncertain sequences.

Let us consider an entropy source that produces events for the LRNG. Let $t_0, t_1, t_2, ...$ be the jiffies associated with those events. A jiffy is the current value of the kernel timer when an event happens.

First, the estimator calculates the jiffies differences on three levels:

$$\begin{aligned} \delta_i &= t_i - t_{i-1}, \\ \delta_i^2 &= \delta_i - \delta_{i-1}, \\ \delta_i^3 &= \delta_i^2 - \delta_{i-1}^2. \end{aligned}$$

Then, the estimator takes the minimum of the differences absolute values:

$$\Delta_i = min(|\delta_i|, |\delta_i^2|, |\delta_i^3|).$$

Finally, it applies the following function [LRSV12]:

$$H_i = \begin{cases} 0 & \text{if } \Delta_i < 2 \\ 11 & \text{if } \Delta_i \geq 2^{12} \\ \lfloor log_2(\Delta_i) \rfloor & \text{otherwise.} \end{cases}$$

The choice of this particular estimator and of its parameters are not justified in the Linux source commentaries. It is a crude but cheap entropy estimator. It was chosen for its cost, not for its accuracy [LRSV12]. Still, it is pretty good at detecting regularities. A retro-engineering [Pou12] study proposes an interpretation based on Newton polynomial interpolation.

Let us give an example:

```
Jiffies             1004   1012    1024    1025    1030    1041

1st differences         8       12       1       5       11

2nd differences             4        11      4       6

3rd differences                 7        7       2
```

$$\begin{aligned} \delta_{1041} &= 1041 - 1030 = 11 \\ \delta_{1041}^2 &= 11 - 5 = 6 \\ \delta_{1041}^3 &= 6 - 4 = 2 \\ \Delta_i &= min(|11|, |6|, |2|) = 2 \\ H_{1041} &= \lfloor log_2(2) \rfloor = 1 \end{aligned}$$

According to the LRNG estimation, the event at time 1041 brings 1 bit of entropy.

3.5 Alternatives to the LRNG

The LRNG is not the only random number generator that can be used on a GNU/Linux operating system.

Many programming languages like C or Python include a random number generator in their standard library, but they are generally PRNG, not TRNG.

The Entropy Gathering Daemon [War02] is a user-space TRNG, that is intended to be used on a Linux system running applications that need random numbers but for some reason are denied access to `/dev/(u)random`.

HAVEGE [SS03] is a TRNG that uses the state of volatile hardware components (caches, branch predictors, long pipelines, instruction level parallelism) as entropy sources.

The kernel module `frandom` is a kind of add-on to the LRNG. It provides the system with two new user-space output interfaces: `/dev/frandom` is 10-50 times faster than `/dev/urandom` but may consume some of the LRNG entropy, `/dev/erandom` is slower but does not consume any LRNG entropy. Both of them are not meant to be used for cryptographic purposes.

4 Contribution

In the previous part, we studied the architecture of the LRNG, including its components and algorithms. We know the LRNG principles. Now we would like to investigate the behavior of the LRNG while in use. Consequently, we implemented a system that monitors the inputs, the outputs and the internal state of the LRNG, and ran it in several test scenarios.

4.1 Experimental setup

We used virtual machines in order to develop our system, because it is both faster and safer to test a kernel on a virtual machine rather than on a real computer. We use the QEmu emulator to host the virtual machines, because it allows us to change the kernel image very easily at each boot.

However, we used real machines for the actual experiments, in order to avoid any influence from the emulator on the results.

We chose to implement the monitoring system directly inside the kernel. We could have used a kernel debugger, but it would have been very difficult to know how it modifies the LRNG behavior. By using our system, we have both a better knowledge and a better control on what happens.

The easiest way to log something from within the kernel is to use the `printk` function. It works like `printf`, and writes the output in the operating system log files. We chose not to use this function, because it could have generated disk events that would have been used as an entropy source by the LRNG. We want our monitoring system to have as less influence as possible. Instead of `printk`, we chose to send packets over network.

To our knowledge, there are two different kernel APIs to send UDP packets, available via `include/linux/net.h` and `include/linux/netpoll.h`. The first one is pretty similar to the standard userspace socket API. The second one is more low-level and not commonly used. Still, we chose to use it, because it works even in IRQ contexts, which is required to instrument input events.

4.2 Tested scenarios

We ran three experiments, corresponding to different scenarios:

1. a desktop workstation,
2. a file server,
3. a computing server.

In the desktop scenario, we installed a graphical environment and a web browser on the virtual machine. Once ready, we rebooted it and started to monitor its LRNG activity. For one hour, we did some typical desktop tasks, mainly reading articles on the internet and installed a few programs. Finally, we shut it down properly, in order to include machine booting and halting activities in the results.

In the file server scenario, we did something similar (one hour, from boot to shutdown). There were no graphical packages installed on the system, and we ran `wget` to download a large data file from a Debian image mirror. Approximately 1.6 GB of data was transferred during the experiment.

In the computing server scenario, we launched a mathematical C program, which looks for integers $n \in \mathbb{N}$ such that $\sqrt[2]{n-1} \in \mathbb{N}$ and $\sqrt[3]{n+1} \in \mathbb{N}$. For instance, the number 26 is such an integer ($26 - 1 = 25 = 5^2$ and $26 + 1 = 27 = 3^3$).

We chose those scenarios because they were likely to stress different entropy sources of the LRNG. The desktop workstation is likely to generate mouse, keyboard and other input events; the file server mainly uses the disk and the network; and the computing server has a heavy CPU load.

4.3 Entropy inputs

In this experiment, we study the origin of the entropy harvested by the LRNG. For that we measure the proportions of entropy that comes from the three functions:

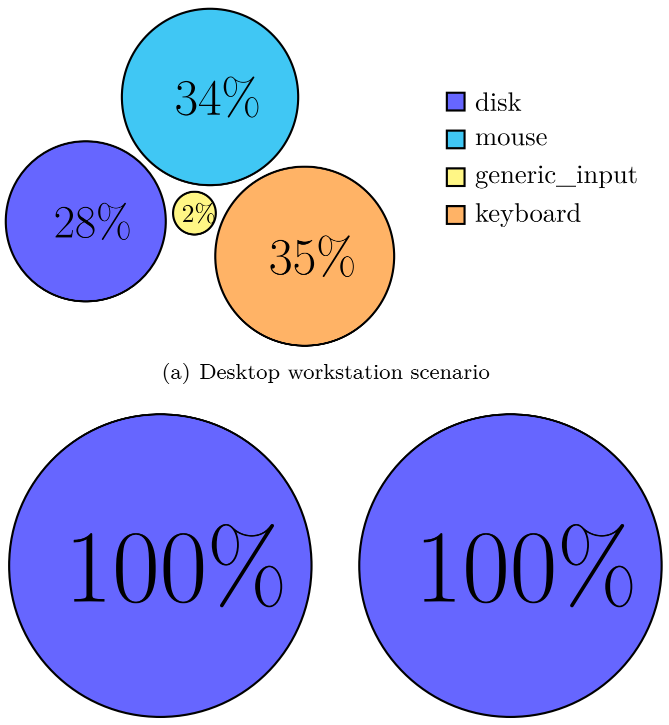
- `add_disk_randomness()`,
- `add_input_randomness()`, and
- `add_interrupt_randomness()`

We summarize the results in the table below, and then discuss the lessons learned from our observations. A graphical representation of those results can be found in Fig. 3. The term "generic input" represents an input event that cannot be associated with a particular device.

Desktop workstation scenario		
Input	*Entropy (bits)*	*Entropy (%)*
Total	26727	100
Disk	7520	28.14
Mouse	9121	34.13
Keyboard	9452	35.36
Generic input	634	2.37
IRQ	0	0.00

File server scenario		
Input	*Entropy (bits)*	*Entropy (%)*
Total	4773	100
Disk	4773	100.00
Mouse	0	0.00
Keyboard	0	0.00
Generic input	0	0.00
IRQ	0	0.00

Computing server scenario		
Input	*Entropy (bits)*	*Entropy (%)*
Total	1377	100
Disk	1377	100.00
Mouse	0	0.00
Keyboard	0	0.00
Generic input	0	0.00
IRQ	0	0.00



(a) Desktop workstation scenario

(b) File server scenario (c) Computing server scenario

Figure 3: Randomness inputs

Primary result: *The disk is an important entropy provider in every scenario, even in the desktop workstation scenario.*

Secondary result: In our three scenario, we did not observe a single "IRQ event". We assume than our experiments did not include any activity likely to generate interrupts that can be used by the `add_interrupt_randomness()` function, such as an interaction with a USB-key.

Secondary result: In the server scenarios, the only entropy provider is the disk. In those scenarios, all the interactions have been performed remotely, without using the machine mouse or keyboard, so there was no reason to observe any entropy generated but an input device.

Secondary result: The desktop workstation obviously produces far more entropy than the server scenarios. The amount of collected entropy in this scenario is almost 5 times higher compared the file server scenario, and more than 15 times higher compared the computing scenario. As expected, the more user activity there is, the more entropy the system collects.

4.4 Entropy outputs

In this experiment, we measure the quantity of entropy that flows through the three output interfaces:

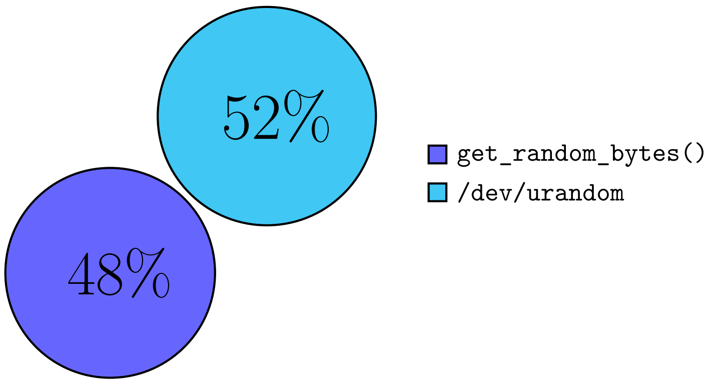
- `/dev/random`,
- `/dev/urandom`, and
- `get_random_bytes()`

We summarize the results in the table below, and a graphical representation of those results can be found in Fig. 4.

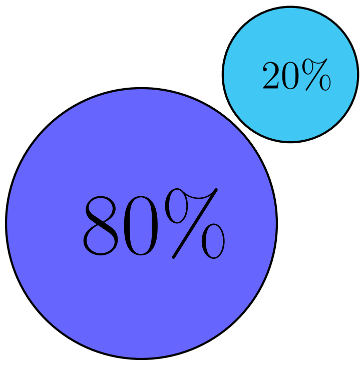
Desktop workstation scenario		
Input	*Entropy (bits)*	*Entropy (%)*
Total	-26096	100
`get_random_bytes()`	-12472	47.79
`/dev/random`	-0	0.00
`/dev/urandom`	-13624	52.21

File server scenario		
Input	*Entropy (bits)*	*Entropy (%)*
Total	-4685	100
`get_random_bytes()`	-3765	80.36
`/dev/random`	-0	0.00
`/dev/urandom`	-920	19.64

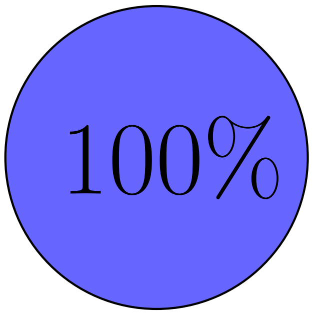
Computing server scenario		
Input	*Entropy (bits)*	*Entropy (%)*
Total	-1377	100
`get_random_bytes()`	-1377	100.00
`/dev/random`	-0	0.00
`/dev/urandom`	-0	0.00



(a) Desktop scenario



(b) File server scenario



(c) Computing server scenario

Figure 4: Randomness outputs

Primary result: *No request has ever been made on `/dev/random` during all the experiments.*

Primary result: *In our three scenarios, most entropy consumption is caused by requests on `get_random_bytes()`. It means the kernel is the first entropy consumer.*

Secondary result: The total amount of consumed entropy almost equals the total amount of produced entropy, in every scenario. This result was to be expected, because 1) obviously the system cannot consume more entropy than it has collected, and 2) the LRNG does not produce useless entropy : once the entropy counter for the input pool has reached its maximum value (4096 bits), the LRNG starts ignoring any incoming events in order to save CPU resources. Thus, it collects no more entropy.

4.5 Entropy consuming applications

In this experiment, we study the userspace clients requesting data on the `/dev/(u)random` devices, in order to find out which applications need random numbers and how much entropy they consume. Due to major differences between the results in each scenario, we choose to present them one after another. We summarize the results in the tables below, and graphical representations are available in figures 5, 6 and 7.

4.5.1 Desktop Workstation scenario

Desktop Workstation scenario		
Input	*Entropy (bits)*	*Entropy (%)*
[K] load_elf_binary	-11896	45.59
[U] svn	-6728	25.78
[U] chromium-browse	-5512	21.12
[U] php5	-640	2.45
[K] inet_frag_secret_rebuild	-320	1.23
[U] gnome-terminal	-256	0.98
[U] gconfd-2	-128	0.49
[U] gconftool-2	-128	0.49
[U] gdm3	-128	0.49
[K] nl_pid_hash_rehash	-128	0.49
[U] python	-104	0.40
[K] rt_cache_invalidate	-64	0.25
[K] rt_genid_init	-64	0.25

The main entropy consumer is the ELF binary loader, which is part of the kernel (implemented in `fs/binfmt_elf.c`). The ELF file format is the default format for executable binaries on x86 Linux systems. The loader needs random numbers to implement Address Space Layout Randomization [2]. Other important consumers are :

- SVN, because we were using an HTTPS repository ;
- the Chrome browser, mainly for use by various subsystems like TLS, SSL, (the network-related part of) ffmpeg, and python (for seeding its own "random" module)

We also note that a small but non-negligible part of the entropy is consumed by a process named "php5". This process is not part of a web server, but a mere php interpreter invoked during the start-up phase to run some system script.

The remaining 5% of entropy consumers include both userspace processes related to the desktop environment and kernel functions related to network.

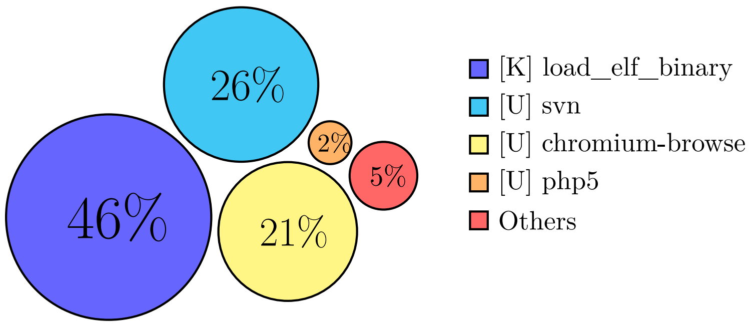


Figure 5: Randomness consumers - Desktop workstation

[2] Address Space Layout Randomization (ASLR) is a computer security method which involves randomly arranging the positions of key data areas, usually including the base of the executable and position of libraries, heap, and stack, in a process's address space. cf `http://en.wikipedia.org/wiki/Address_space_layout_randomization`

4.5.2 File server scenario

File server scenario		
Input	*Entropy (bits)*	*Entropy (%)*
[K] load_elf_binary	-3392	72.23
[U] php5	-704	14.99
[K] inet_frag_secret_rebuild	-256	5.45
[U] apache2	-80	1.70
[U] smbd	-72	1.53
[U] winbindd	-64	1.36
[K] reqsk_queue_alloc	-64	1.36
[K] br_fdb_init	-64	1.36

Once again, the main entropy consumer is the ELF binary loader. The php5 consumer is still there ; it is the second consumer, but the amount of consumed bytes is equivalent.

The third consumer is a kernel function called `inet_frag_secret_rebuild()` and located in the `net/ipv4/inet_fragment.c` file, which is part of the IPv4 stack. The implementation uses a hash table to store IP datagram fragments. If the hash function was the always the same, then it would be practical for an attacker to set up a denial of service attack just by sending well-known specific datagrams which caused hash collisions happen. That's why a random seed is used during the initialization in order to make the hash function unpredictable from the outside.

The fourth consumer is the Apache web server, even if there was no HTTPS request during the experiment. The server needs random numbers when it starts in order to initialize a seed used by the SSL protocol.

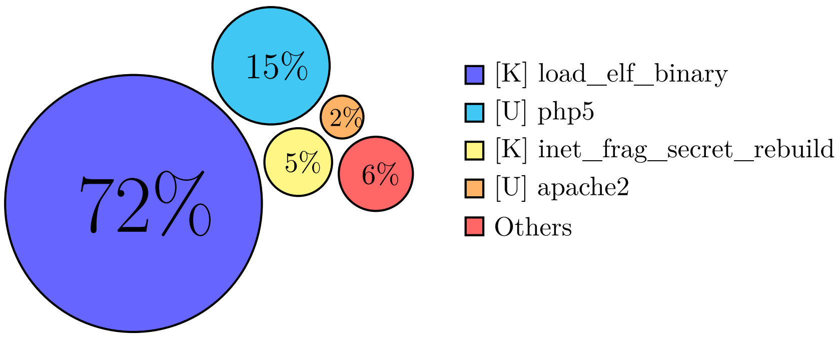


Figure 6: Randomness consumers - File server

4.5.3 Computing server scenario

Computing server scenario		
Input	*Entropy (bits)*	*Entropy (%)*
[K] load_elf_binary	-1320	95.38
[K] inet_frag_secret_rebuild	-64	4.62

In this scenario we find only two consumers, and both are kernel functions. The ELF binary loader is still the main consumer, eating an overwhelming majority of the entropy. Only 5% are required by the second consumer, a function of the IPv4 stack which has been presented before.

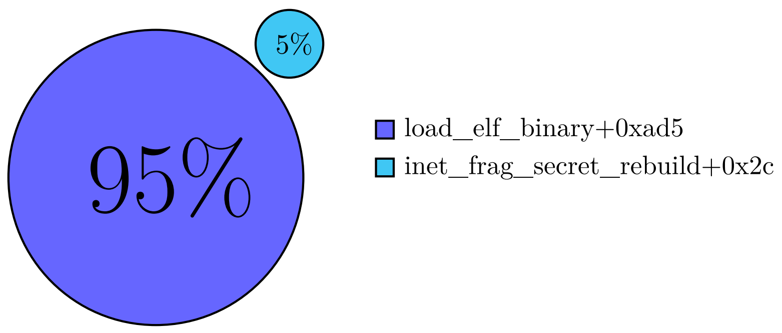


Figure 7: Randomness consumers - Computing server

4.5.4 Subsection conclusion

In this subsection, we studied how entropy is consumed. For each request on `/dev/(u)random`, we identified the corresponding process. For each call to `get_random_bytes()`, we analyzed the corresponding kernel function. In this conclusion, we remind the results that seem really significant.

Primary result: *In our three scenarios, the vast majority of entropy is consumed by one kernel function : the ELF binary loader.*

Secondary result: The other consumers vary greatly according to the scenario.

4.6 Entropy counter of the input pool

In this experiment, we measure the evolution over time of the input pool entropy counter. We present the results on Fig. 8.

The vertical axis represents the input pool level. It goes from 0 (empty pool) to around 4096, which is the size of the input pool bit field. The level cannot increase over this size : if all the bits of the bit field are unpredictable, no additional entropy can be stored.

Moreover, we denote that the entropy counter does not fall to zero exactly, and does not cap to 4096 exactly. The behavior is due to thresholds. When the entropy counter goes over 3584 bits, the LRNG starts dropping most samples to avoid wasting CPU time and reduce lock contention. On the other hand, if the entropy counter falls under 64 bits, entropy transfers are blocked until the input pool level reaches 64 bits, so that the transfer is significant enough.

The horizontal axis represents the time, expressed in CPU cycles. Even if the scenario duration is always the same (one hour), the number of CPU cycles is different according to the scenario. It is about 10^{10} cycles in the server scenario, ten times higher in the computing scenario and again 10 times higher in the desktop workstation scenario. These results are logical regarding the to the recent CPUs sleep mode. On the one hand, the CPU load is extremely heavy in the computing scenario. One the other hand, it is very light in the file server scenario, because the computer is mainly waiting for the network. The desktop workstation scenario stands for an average between the two precedent cases.

Primary result: *The entropy counter varies greatly in the desktop scenario. It can go from its maximum to its minimum very quickly. The unstable behavior is due the constant user interaction.*

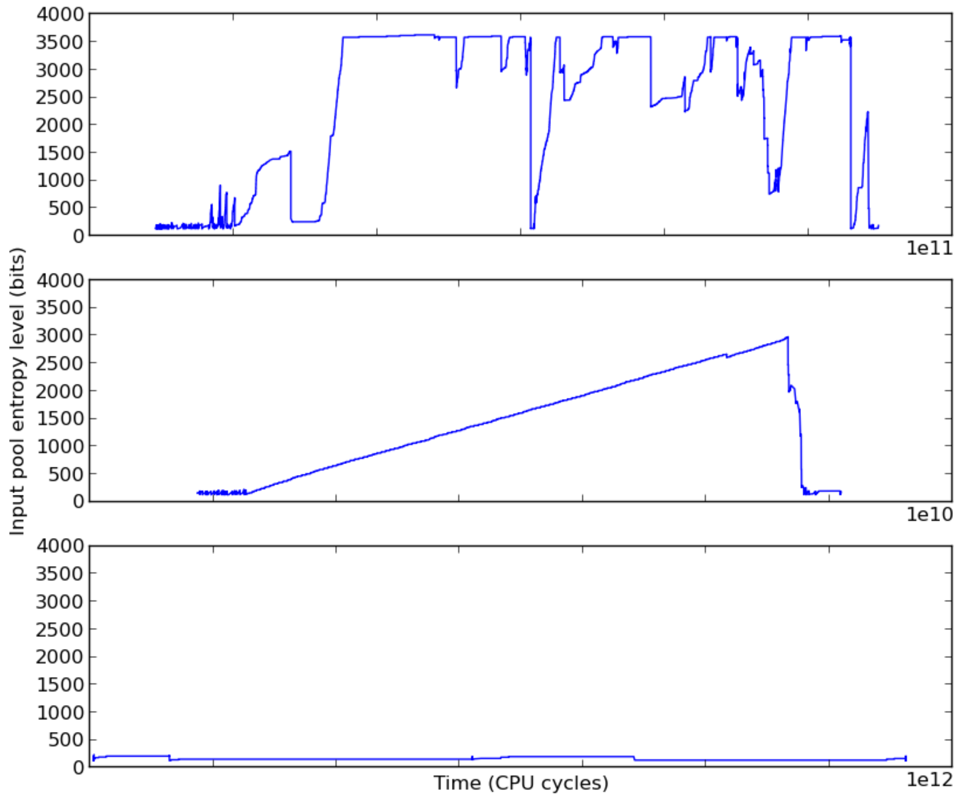


Figure 8: Evolution of the input pool entropy counters over time. The top graph corresponds to the desktop workstation scenario. The middle graph corresponds to the file server scenario. The bottom graph corresponds to the computing server scenario.

A long entropy shortage seems to be almost impossible in such a scenario.

Primary result: *In the file server scenario, the entropy counter increases continuously and slowly until the computer halts. We assume that the slow increase is due to the reduced number of CPU cycles, because the number of jiffies is also reduced and the LRNG estimator is based on the jiffies differences. If we would have represented the desktop workstation scenario and the file server scenario on the same X axis, the slopes would have been similar. We assume that a moderate entropy consumption would not cause any shortage.*

Primary result: *In the computing scenario, the entropy counter remains extremely low during all the experiment, because both entropy production and consumption are really basic. The situation is almost a constant entropy shortage.*

4.7 Proportion of events generating entropy

In this experiment, for each event type, we compare the number of events that actually generate entropy with the total number of event, in order to find the proportion of "useful" events. We summarize the results in the table below.

Desktop workstation scenario			
Input	*All events (bits)*	*Events generating entropy (bits)*	*(%)*
Disk	26732	3708	13.87
Mouse	13396	2279	17.01
Keyboard	56577	3940	6.96
Generic input	55440	216	0.39
IRQ	0	0	0.00

File server scenario			
Input	*All events (bits)*	*Events generating entropy (bits)*	*(%)*
Disk	9083	1686	18.56
Mouse	0	0	0.00
Keyboard	0	0	0.00
Generic input	0	0	0.00
IRQ	0	0	0.00

Computing scenario			
Input	*All events (bits)*	*Events generating entropy (bits)*	*(%)*
Disk	2649	715	26.99
Mouse	0	0	0.00
Keyboard	0	0	0.00
Generic input	0	0	0.00
IRQ	0	0	0.00

Secondary result: Only a tiny portion of the generic input events generates entropy, whereas the other entropy sources are more efficient.

Secondary result: The less disk events there are, the more entropy-efficient the events are.

4.8 Differences between VMs and real machines

Where as the final experiments have been run on physical computers, a huge part of the development has been led on virtual machines. Consequently, we have noticed the differences between the results on virtual and physical machines and we present them below.

Secondary result: In every one-hour scenario we played on virtual machines, the number of cycles elapsed was almost the same. Besides, the number of cycles seems to be a maximum, like there is no sleep mode on the CPU. This specificity has an unexpected consequence: the slopes on the input pool level graph tend to be higher, especially for the file server scenario. Surprisingly, it means that the virtualization would have a rather positive effect on the entropy harvesting process.

Secondary result: The repartition of entropy inputs and outputs are more or less the same. Only the disk looks a bit more efficient in the case of a virtual machine.

5 LRNG customers

Thanks to our experiments, we acquired information about the LRNG "customers", who use the LRNG random numbers and thus consume entropy. Mainly, we have got valuable information about the first of the customers: the kernel itself. However, even if we have some information about other customers, it is highly specific to our scenarios.

In this last part, we try to cross-check our results with the source code and the documentations of widely used applications:

- google_authenticator,
- openssl,
- openssh.

The Google authenticator is a enforced authenticator with a two-step verification, available for GMail or GApps users who feel concerned with security. It uses `/dev/urandom` to get random numbers. When someones suggests to switch from `/dev/urandom` to `/dev/random` be cause the latter is supposed to be more secure[3], the developers oppose a categorical refusal. One the one hand, the don't want to rely on a device that can block indefinitely. One the other hand, they argue that no practical attack has been shown on `/dev/urandom` despite its theoretical weaknesses.

The OpenSSH library only uses `/dev/urandom`, even when generating keys with `ssh-keygen.`[4]. Once again, the developers don't want to rely on a blocking device. According to them, it is so much easier to compromise a key by social engineering rather than by an attack on `/dev/urandom`, it would be a useless loss of time to use `/dev/random` instead.

The OpenSSL library is able to use both devices, but it will always try to use the non-blocking one. Although no explanation is given in the documentation, we assume that the reasons are similar to the previous ones.

Primary result: *Even if entropy shortage seems to be a common phenomenon according to our results, the end users never notice them because the LRNG "customers" only use the non-blocking interfaces: `/dev/urandom` and `get_random_bytes()`.*

6 Conclusions and perspectives

In this work, we studied the randomness resource in the GNU/Linux operating system with a system-oriented approach. Thanks to a built-in monitoring system, we did several measures on the random number generator of the Linux kernel in order to understand the underlying mechanism that transform unpredictable interactions like disk seeks of mouse events into reliable random numbers, meant to be used by security programs.

Even when entropy is rare, it does not prevent the kernel from running normally. This is due to the fact that the `/dev/random` device was never used in our experiments, whereas `/dev/urandom` and `get_random_bytes()` are commonly used and non-blocking. Between the two methods, i.e `/dev/urandom` and `get_random_bytes()`, the latter one is the most often called meaning that the kernel is consuming a lot of random numbers. Inside the kernel, the most entropy is consumed by the function used to load the binary files in memory. It needs random numbers in order to

[3] `http://code.google.com/p/google-authenticator/issues/detail?id=107`

[4] `https://groups.google.com/forum/?fromgroups#!topic/comp.security.ssh/UG8KDZBAM7g%5B1-25%5D`

randomize the address space of the binaries for security reasons. In all our experiments, the disk is a significant entropy provider even in the desktop scenario which includes interactions with a graphical environment.

If we compare our experiments, we observe that the total amount of entropy gathered is far higher in the desktop scenario than in the other two. From the perspective of the entropy gathered, a file server produces only a bit more entropy than a computing server. However, the input pool level for the file server is far higher than in the case of the computing server. Therefore, a quantitative comparison is not fair. The desktop workstation scenario and the computing scenario are both likely to have entropy shortage. They are very brief for the desktop workstation and should not cause any problem. However, the computing scenario is almost always in situation of entropy shortage.

To our knowledge, no attack on `/dev/urandom` has been shown in the state of the art. However, the systematic use of `/dev/urandom` instead of `/dev/random` relies on the postulate that "there will be enough entropy soon". The supposition is quite true in the desktop workstation scenario; but according to our experiments, it is absolutely false in the situation of a computing server scenario, and not certain in a file server scenario. Besides, the overwhelming majority of Linux systems are servers, not desktop workstations. And on servers, the only entropy provider is the disk, whereas this device plays a less and less important role in modern systems. In the future, it might be not acceptable anymore to consider that there will always be enough entropy soon.

Contents

1 Introduction **3**

2 Context: random numbers **3**
2.1 Random numbers characterization 3
2.2 Random Number Generators 5

3 Context: the Linux Random Number Generator **6**
3.1 Output interfaces 6
3.2 Entropy pools 6
3.3 Entropy sources 7
3.4 Entropy estimation 9
3.5 Alternatives to the LRNG 10

4 Contribution **10**
4.1 Experimental setup 10
4.2 Tested scenarios 11
4.3 Entropy inputs 11
4.4 Entropy outputs 13
4.5 Entropy consuming applications 14
4.5.1 Desktop Workstation scenario 15
4.5.2 File server scenario 16
4.5.3 Computing server scenario 16
4.5.4 Subsection conclusion 17
4.6 Entropy counter of the input pool 17
4.7 Proportion of events generating entropy 19
4.8 Differences between VMs and real machines 19

5 LRNG customers **20**

6 Conclusions and perspectives **20**

References

[BBS86] Lenore Blum, Manuel Blum, and Michael Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15:364–383, may 1986.

[BH05] Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In *ACM conference on Computer and communications security - CCS '05*, pages 203–212, Alexandria, VA, USA, 2005. ACM.

[BIW04] Boaz Barak, Russell Impagliazzo, and Avi Wigderson. Extracting randomness using few independent sources. In *Symposium on Foundations of Computer Science - FOCS 2004*, pages 384–393, Rome, Italy, 2004. IEEE Computer Society.

[GPR06] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the Linux random number generator. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006)*, pages 371–385, Oakland, California, USA, May 2006. IEEE Computer Society.

[Int11] Bull mountain software implementation guide. Technical report, Intel Corportation, 2011.

[Kel04] John Kelsey. Entropy sources. In *NIST RNG Workshop*, July 2004. `http://csrc.nist.gov/groups/ST/toolkit/random_number.html`.

[Knu97] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[LPR11] Cédric Lauradoux, Julien Ponge, and Andrea Röck. Online entropy estimation for non-binary sources and applications on iphone. Technical report, INRIA, 2011.

[LRSV12] Patrick Lacharme, Andrea Röck, Vincent Strubel, and Marion Videau. The Linux Pseudorandom Number Generator Revisited. Cryptology ePrint Archive, Report 2012/251, 2012.

[MT12] Matt Mackall and Theodore Ts'o. random.c – a strong random number generator, April 2012. in Linux Kernel 3.3.6.

[PM88] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[Pou12] Benjamin Pousse. An interpretation of the linux entropy estimator. Cryptology ePrint Archive Report available at `http://eprint.iacr.org/2012/487`, 2012.

[R05̈] Andrea Röck. Pseudorandom number generators for cryptographic applications. Master's thesis, Naturwissenschachtlichen Fakultät der Paris-Lodron-Universität Salzburg, 2005.

[RB08] Matthew J. B. Robshaw and Olivier Billet, editors. *New Stream Cipher Designs - The eSTREAM Finalists*. Lecture Notes in Computer Science 4986. Springer, 2008.

[SS03] André Seznec and Nicolas Sendrier. HAVEGE: A user-level software heuristic for generating empirically strong random numbers. *ACM Transactions on Modeling and Computer Simulation - TOMACS*, 13(4):334–346, 2003.

[VM03] John Viega and Matt Messier. *Secure Programming Cookbook for C and C++*. O'Reilly, 2003.

[War02] Brian Warner. EGD: The Entropy Gathering Daemon, 2002. `http://egd.sourceforge.net`.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399