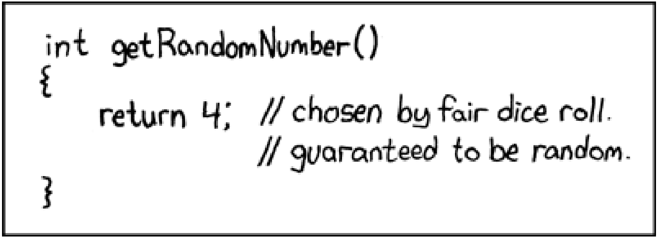
A study of entropy transfers
in the Linux Random Number Generator

Th. Vuillemin, F. Goichon, G. Salagnac, C. Lauradoux



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

The need for random numbers

Computers are built to be fully deterministic...

...but unpredictability is still required

- Cryptography
- Security
- Randomized algorithms
- Scheduling
- Networking

Random numbers a an OS resource

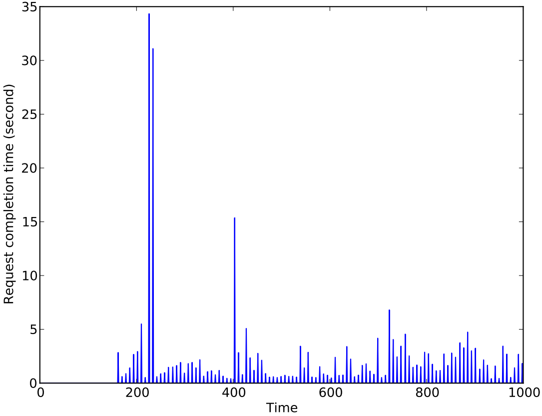
LRNG : Linux Random Number Generator

- Service provided by the OS kernel
- Shared among several (non-privileged) users
- `/dev/random` and `/dev/urandom`
- Essential for security-oriented software (SSH, SSL/TLS)

Depends on system *entropy*

- Prone to *entropy shortages* $\Rightarrow$ RNG stalls
- May have negative impact on application performance

Motivating example



Response time of `/dev/random` for 1000 one-byte requests.
Average 264 ms. Standard deviation 1.68 s.

Questions

- What is *entropy* anyway ?
- Why does the LRNG need it ?
- How to explain such variability in response time ?

Inria Research Report 8060 http://hal.inria.fr/hal-00738638

Agenda

1. Introduction
2. Random Number Generation
3. The Linux RNG
4. Experiments
5. Conclusion and perspectives

Desirable properties of "random" numbers

- X, Y random variables — e.g. the result of rolling a die
- Ω sample space — e.g. $\{1, 2, 3, 4, 5, 6\}$
- $\mathcal{X} = \mathcal{P}(\Omega)$ event space — e.g. $X \in \{2, 4, 6\}$
- $\{Pr(i)\}_{i \in \mathcal{X}}$ probability law

Uniform distribution

$$\forall x \in \Omega \qquad Pr(X = x) = \frac{1}{\text{card}(\Omega)}$$

Statistical independence

$$\forall x, y \in \Omega \qquad Pr(X = x | Y = y) = Pr(X = x)$$

Measuring randomness

Shannon Entropy

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

- expresses the "amount of uncertainty" contained in *X*
  - "how much information do I gain by looking at *X*"

Caveat Emptor

- Other entropy measures exist (e.g. Kolmogorov complexity)
- If we don't know *Pr*, we cannot directly apply the formula
- *Entropy estimation* is a very active research topic

Different types of generators

A Random Number Generator is a computer program imitating the behaviour of a random variable

PRNG : Pseudo Random Number Generator
CSPRNG : Cryptographically Secure Random Number Gen.
HRNG : Hardware Random Number Generator
TRNG : True Random Number Generator

Deterministic generators

PRNG : Pseudo-Random Number Generator

- finite-state machine
- transition function : updates internal state
- output function : produces actual numbers
- seed : initial internal state
- ▶ (hopefully) good statistical properties

CSPRNG : Cryptographically Secure PRNG

- ▶ A PRNG with stronger statistical properties (periodicity...)

Security issues

Threat model

What if an attacker *guesses* the internal state ?

► they can predict every future output of the RNG !

Solutions

- choose the output function such that it’s hard to reverse

- ... or just don’t be deterministic

Non-deterministic generators

HRNG : Hardware Random Number Generator

Based on some physical phenomenon

- really unpredictable, but often biased
- limited by the througput of the *entropy source*

TRNG : True Random Number Generator

- Pseudo-Random Number Generator
- internal state *reseeded* with entropy sources

Agenda

The Linux RNG

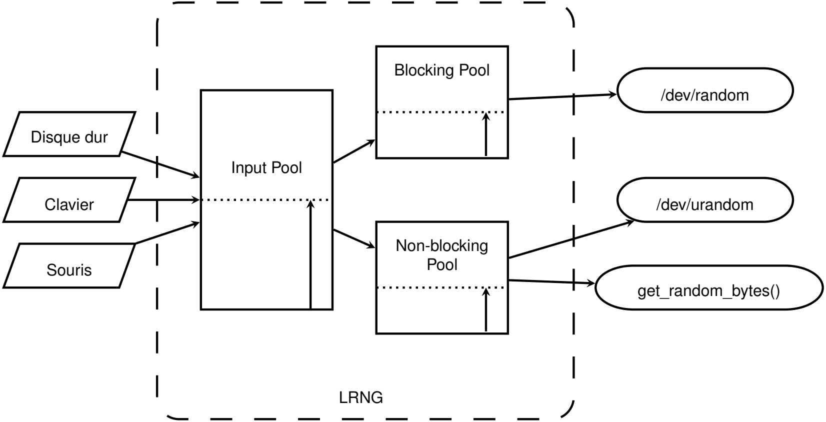
Authors

- Theodore Ts’o (1994–2005, 2012–now)
- Matt Mackall (2005–2012)

TRNG architecture

- uses a CSPRNG to produce numbers
  - internal state : 6Kb
  - output function : a variant of md5
- uses system events as entropy sources
  - opportunistic reseeding
  - hypothesis : inter-event timing is unpredictable
- tries to keep internal state *hard to guess* for an attacker
  - tracks the *entropy level* of state over time

Architecture



Output interfaces

`/dev/random`

- comsumes entropy
- in case of shortage → requests put on hold

`/dev/urandom`

- consumes entropy
- in case of shortage → PRNG

`get_random_bytes()`

- kernel function
- consumes entropy
- in case of shortage → PRNG

Entropy pools (internal state of the PRNGs)

Blocking pool

- 1Kb bitfield + entropy counter
- supplies data for `/dev/random`

Non-blocking pool

- 1Kb bitfield + entropy counter
- supplies data for `/dev/urandom` and `get_random_bytes()`

Input pool

- 4Kb bitfield + entropy counter
- supplies data for the two other pools
- refilled by opportunistically sampling *entropy sources*

Entropy sources

Callback functions exported by the LRNG to harvest entropy :

`add_disk_randomness()`

Hard drive events

`add_input_randomness()`

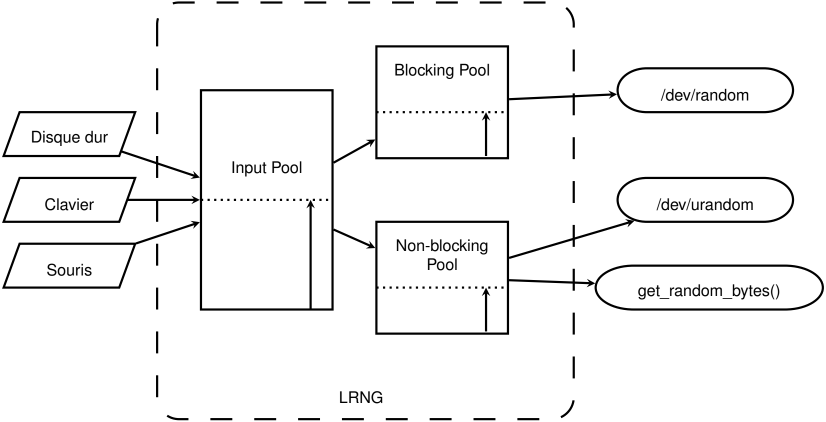
UI events : keyboard, mouse, trackpad

`add_interrupt_randomness()`

Other hardware events : USB, device drivers

`add_network_randomness()` removed, deemed too vulnerable

Architecture



Disque dur

Clavier

Souris

Input Pool

Blocking Pool

Non-blocking Pool

/dev/random

/dev/urandom

get_random_bytes()

LRNG

The need for entropy estimation

What if an attacker controls all the callbacks ?
What if hardware events happen to be predictable ?

Not all system events carry uncertainty

- Let’s try to assess randomness

► We need an *entropy estimator* !

The LRNG entropy estimator : detecting regularities

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

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

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

Example

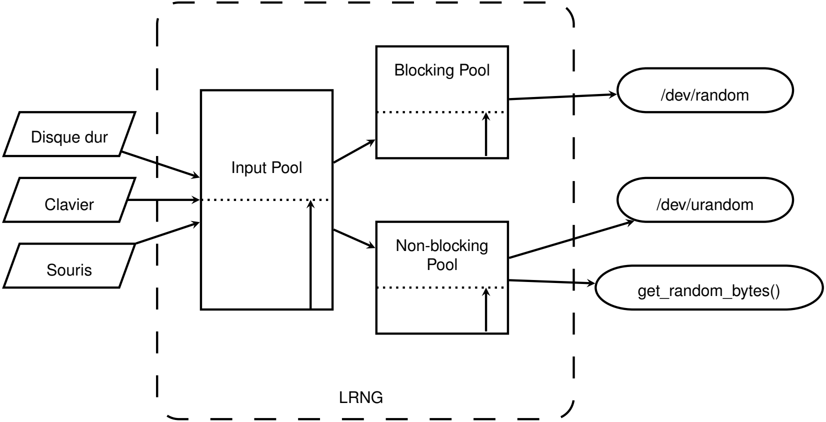
Time	1004		1012		1024		1025		1030		1041
1st diff		8		12		1		5		11	
2nd diff			4		11		4		6		
3rd diff				7		7		2			

$$H(1041) = 1,\ H(1030) = 2,\ H(1025) = 0$$

Agenda

1. Introduction
2. Random Number Generation
3. The Linux RNG
4. Experiments
5. Conclusion and perspectives

Architecture



Experimental setup

Prototype

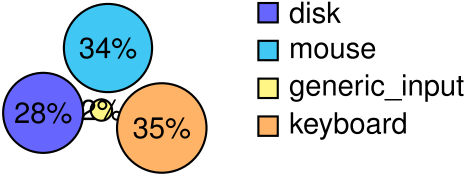
- ~~use a kernel debugger ?~~ → would kill timing
- ~~use `printk()` ?~~ → would generate disk events !
- ▶ instrument the LRNG itself (callbacks + output functions)
- use the *netpoll* API to send out UDP packets

Studied scenarios

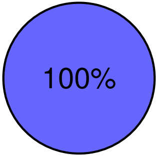
- Desktop workstation : web surfing, word processing
- File server : large file transfer
- Computation : CPU-intensive program only

each experiment : one hour long

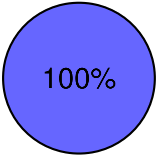
Entropy harvesting



(a) Workstation



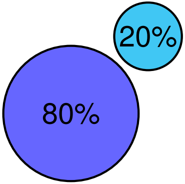
(b) File server



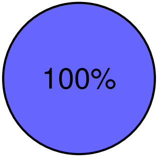
(c) Computation

Entropy extraction

(d) Workstation

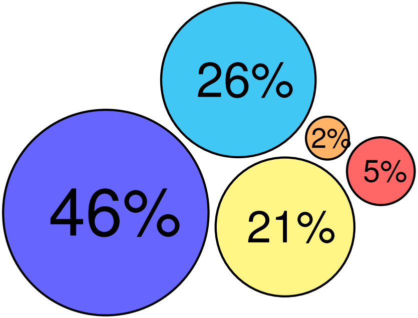


(e) File server



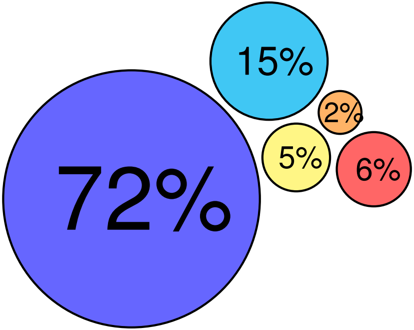
(f) Computation

Entropy consumers : Workstation



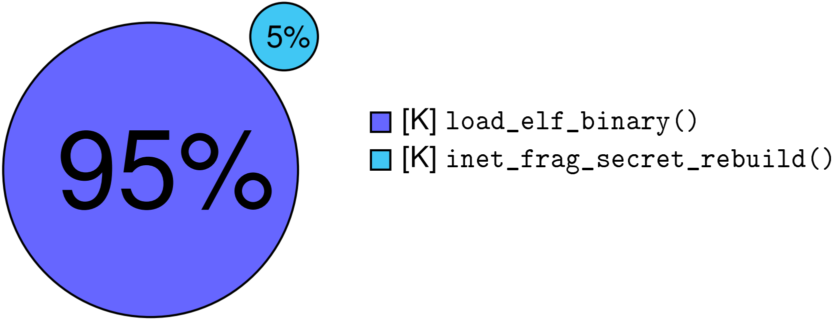
- 🟦 [K] `load_elf_binary()`
- 🟦 [U] svn
- 🟨 [U] chromium-browse
- 🟧 [U] php5
- 🟥 Others

Entropy consumers : File server



- 🟦 [K] `load_elf_binary()`
- 🟦 [U] php5
- 🟨 [K] `inet_frag_secret_reb...`
- 🟧 [U] apache2
- 🟥 Others

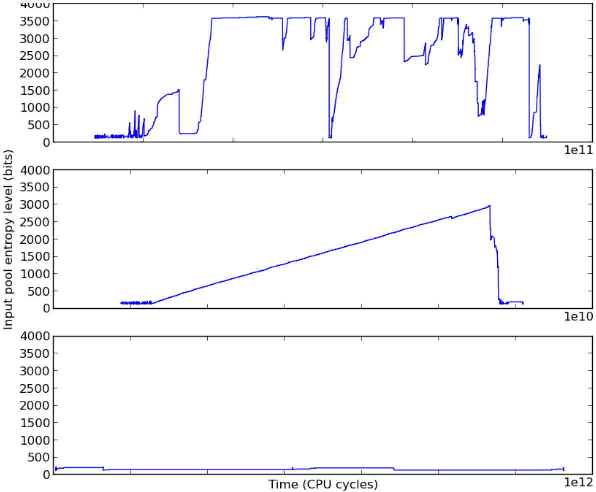
Entropy consumers : Computation



🟦 [K] `load_elf_binary()`

🟦 [K] `inet_frag_secret_rebuild()`

Entropy level in the input pool



Summary of experimental results

- only major entropy source : the hard drive
- `/dev/random` never used in practice
  - blocking `read()` considered too problematic by developers
  - doesn't even exist in other kernels (BSD)
  - security-oriented applications have their own CSPRNG
  - people believe that « there will soon be entropy » (true ?)
- major entropy consumer : the kernel itself
  - via `get_random_bytes()`
  - mostly for `load_elf_binary()` (i.e. ASLR)

Conclusions and perspectives

Summary

- Study of the architecture of the LRNG
- Measures of entropy transfers
- Study of entropy consumers
- see [Inria RR 8060] `http://hal.inria.fr/hal-00738638`

Perspectives

- Port experiments to diskless devices
  - Android phone, set-top box, SSD-based laptop
  - Entropy will be scarce
- Come up with new sources of entropy in the system
  - portability ?
  - availability ?