Erlang, random numbers, and the security

Kenji Rikitake
10-SEP-2013
for London Erlang User Group meetup
Executive summary

Random number generation is very difficult to do properly; so don’t try to invent yours!
Dilbert’s definition of randomness (NOT)

http://insanity-works.blogspot.jp/2008/05/debianubuntu-serious-opensslssh.html
What does randomness mean? (Wikipedia)

- Commonly, it means lack of pattern or predictability in events
- Non-order or non-coherence in a sequence, no intelligible pattern or combination
- Statistically, random processes still follow a probabilistic distribution
“Pseudo” randomness?

- The results of deterministic computations can be predicted; there is no real randomness from those algorithms.

- A **predictive** sequence generator with a very long period (e.g., $(2^{19937})-1$ times of computations) can be used as a list of pseudo random number generator (PRNG).

- Each generation needs a **seed** for to determine the starting point (aka **Initialization Vector**).
Then what is “real” randomness?

• Mostly depending on the physical events which is very unlikely to be predictable by the consumer of the randomness itself

• You can’t really avoid the interference of the exterior environment

• But at least statistically whether a sequence is random or not can be measured
Entropy pool and the level of randomness

• Entropy is the unit of randomness

• If 32-bit word of complete random number exists, it has 32 bits of entropy

• If the 32-bit word takes only four different values and each value has a 25% chance of occurring, the word has 2 bits of entropy

• The Dilbert’s case has ZERO bit of entropy

Sources of “real” high-entropy sources

- Thermal noise (of resistors)
- Avalanche noise of Zener diodes
- Optical noise (e.g., LavaCan)
- Radio noise (static) (random.org)
- Rolling the dice (Dice-O-Matic)
- Very expensive (= low entropy bit rate)
  - Practical for seeding purpose only
- Not necessarily statistically uniform or gaussian
Avalanche diode noise generator example

Arduino Duemilanove shield schematics for a hardware random number generator by Kenji Rikitake 6-APR-2009

Vin = +12V or +13.8V (+9V didn’t work)

100nF: 50V ceramic
470nF: film
2.2uF: 50V electrolytic
Arduino Duemilanove and the noise generator


Transistors as noise diodes

Photo by Kenji Rikitake 2009
Can OS gather entropy from various events in a computer?

- Possible events: hard disk access duration, mouse/keyboard timing, ethernet packet arrival timing, wall-clock values, etc.
- Note: those events can only happen after when the system is booted; at the boot time the entropy pool is virtually empty.
- Solutions: use an external entropy source to gather sufficient entropy, or wait for the enough entropy to be gathered and filled into the pool.

So what are good PRNGs?

- Output distribution
  - uniform distribution: equally covering all possible values in the same probability
  - Gaussian/normal distribution: representing the sum of many independent random values
- The generation period length should be very large (especially when sharing the same algorithm between multiple processes)
Gaussian/normal distribution

Uniformity is hard to achieve in the real world

An example: Cryptocat's buggy PRNG written in JavaScript

Another example: PHP5 `rand(0, 1)` on Windows: see the pattern?

http://boallen.com/random-numbers.html
Android Bitcoin Wallets had PRNG vulnerability to let 55 BTC stolen

- Java’s SecureRandom class (wasn’t secure enough on Android actually)
- “In order to remain secure the random numbers used to generate private keys must be nondeterministic, meaning that the output of the generator cannot be predicted”
- “Android phones/tablets are weak and some signatures have been observed to have colliding R values”

Random number generation is very difficult to do properly; so don’t try to invent yours!

```
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
    // guaranteed to be random.
}
```
So how Erlang/OTP does on PRNGs?

- crypto module: cryptographically secure PRNG, OpenSSL API in NIFs
- For generating passwords and keys, always use the **crypto** module
- There was a bug in SSH module using random function instead of crypto (CVE-2011-0766, discovered by Geoff Cant, fixed on R14B03)
Erlang’s AS183 512x512 bitmap: at least visually well-randomized
%% Code of AS183 PRNG
%% from Erlang/OTP R16B01 lib/stdlib/src/random.erl
%% This hasn’t been really changed at least since R14B02

(define(PRIME1, 30269).
(define(PRIME2, 30307).
(define(PRIME3, 30323).

uniform() ->
    {A1, A2, A3} = case get(random_seed) of
        undefined -> seed0();
        Tuple -> Tuple
    end,
    B1 = (A1*171) rem ?PRIME1,
    B2 = (A2*172) rem ?PRIME2,
    B3 = (A3*170) rem ?PRIME3,
    put(random_seed, {B1,B2,B3}),
    R = B1/?PRIME1 + B2/?PRIME2 + B3/?PRIME3,
    R - trunc(R).
Issues on random module, aka ASI83

• The period length is short (~ $2^{43}$)
• Very old design (in 1980s)
• Not designed for parallelism
• Only three 16-bit integers as the seed
• Not designed for speed (float division)
• Good thing: it’s pure Erlang code
Goals for solving issues on random module

• Larger period length PRNG needed
• New or modern design PRNGs preferred
• Addressing parallelism requirements
  • Non-overlapping = orthogonal sequences
• Larger state length for seeding
• Availability on both with and without NIFs
Required/suggested features on parallelism

- Each PRNG has to manage its own \textit{state}.
- \textbf{Independent} = \textit{orthogonal} sequences from the same algorithm.
- Splitting the internal state.
- Using orthogonal polynomials.
- Seed jumping: fast calculation for advancing the internal state.
Alternatives I implemented for the random module

- `sfmt-erlang`
- `tinymt-erlang`
- Both available at GitHub
- Details available as ACM Erlang Workshop papers, as well as from my own web site
- These are *non-secure PRNGs*
Mersenne Twister

- BSD licensed
- By Matsumoto and Nishimura, 1996
- Very long period \((2^{19937}) - 1\)
- Uniformly distributed in 623-dimension hypercube, mathematically proven
- Widely used on many open-source languages: R, Python, mruby, etc.
SIMD-oriented Fast Mersenne Twister (SFMT)

- Improved MT by Saito and Matsumoto (2006)
- SIMD-oriented, optimized for x86(_64)
- Various period length supported
  - $((2^{607})-1) \sim ((2^{216091})-1)$
- Faster recovery from 0-excess initial state
sfmt-erlang

- A straight-forward implementation of SFMT in both pure Erlang and with NIFs
- Suggested period length: \(2^{19937}-1\)
- with NIFS: >x3 faster than random module
- A choice on PropEr Erlang test tool
Porting tips of SFMT code from C to Erlang

- Make it reentrant: removed all static arrays for the internal state table (no mutable data structure)
- SFMT itself can be written as a recursion
  \[ a[X] = r(a[X-N], a[X-(N-POS1)], a[X-1], a[X-2]) \]
- An Erlang way: adding elements to the heads and do the lists:reverse/1 made the code 50% faster than using the ++ operator!
Speed difference between C and Erlang

- On sfmt-erlang
- Pure Erlang code: x300 \textit{slower} than C
- Erlang NIF code: still x10 slower than C
- Some dilemma: speed optimization is important, but Erlang VM can hardly beat the native C code
TinyMT

- MT for smaller memory footprint (and where shorter period length is OK)
- Saito and Matsumoto, 2011
- Period: \((2^{127})-1\), still much > \(2^{43}\)
- Seed: seven 32-bit unsigned integers
- Parallel execution oriented features: \(\sim 2^{56}\) independent generation polynomials
tinymt-erlang

- A straight-forward implementation of TinyMT in both pure Erlang and with NIFs
- Some of sfmt-erlang code are equally applicable (e.g., seed initialization)
- x86_64 HiPE optimization: x3 speed
- Wall-clock speed: roughly the same as random module
- In fprof: x2~x6 slower than random module
Observations on tinymt-erlang code

- 32-bit integer ops are on BIGNUMs
- Small integer on 32bit Erlang: max 28 bits
- Overhead of calling functions and BEAM memory allocation might have taken a significant portion of exec time
- sfmt-erlang batch exec is still x3~4 faster than tinymt-erlang batch exec
NIF scheduling issue

- **NIF blocks the Erlang scheduler**
- How long can a CPU spend time in a NIF?
  - Rule of thumb: <1msec per exec
  - Target for sfmt/tinymt-erlang: < 0.1msec
- On Riak this has been a serious issue
  - Scott Fritchie has an MD5 example code
- A preference for the pure Erlang code
TinyMT precomputed polynomial parameters

• On 2011, I decided to pre-compute the polynomial parameters at Kyoto University’s super-computer (x86_64) cluster (I could only use two 16-core nodes (32 cores))

• \(2^{28} \approx 256\text{M}\) parameter sets are available for both 32- and 64-bit implementations

• Took \(\sim 32\) days of continuous execution

• 18~19 sets/sec for each core
“Secure” PRNGs

- One-way function: guessing input or internal state from the output should be kept practically impossible.
- A PRNG is less secure when:
  - generation period is shorter
  - easy function exists to guess the internal state from the output
  - the output sequence is predictable from the past output sequence data (= a simple PRNG is unsecure)
For building more secure PRNGs

- Reducing predictability: always incorporate necessary external extropy

- Cryptographic strength: use well-proven algorithms, e.g., AES, to “encrypt” the PRNG output

- Rule of thumb: use crypto:strong_rand_bytes/1 to guarantee sufficient entropy is consumed
Tips for parallel execution of PRNGs

• Use completely orthogonal sequences
  • e.g., TinyMT independent polynomials

• If the seeding space is large, it is unlikely that PRNG sequences from the seeds generated from another PRNGs will collide with each other (but this is not mathematically proven)

• Beware of execution sequence change when predictability is important
Tips on secure PRNGs

• **Don’t invent yours** unless you hire a well-experienced cryptographers (i.e., stick to proven functions)

• Ensure the system has enough entropy to guarantee the unpredictability

• An external hardware RNG is suggested

• And don’t let NSA cripple the algorithms!
Random number generation is very difficult to do properly; so don’t try to invent yours!

---

<table>
<thead>
<tr>
<th>Affected System</th>
<th>Security Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fedora Core</td>
<td>Vulnerable to certain decoder rings</td>
</tr>
<tr>
<td>Xandros (Eee PC)</td>
<td>Gives root access if asked in stern voice</td>
</tr>
<tr>
<td>Gentoo</td>
<td>Vulnerable to flattery</td>
</tr>
<tr>
<td>OLPC OS</td>
<td>Vulnerable to Jeff Goldblum’s Powerbook</td>
</tr>
<tr>
<td>Slackware</td>
<td>Gives root access if user says elvish word for “friend”</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>Turns out distro is actually just Windows Vista with a few custom themes</td>
</tr>
</tbody>
</table>
Thanks

Questions?