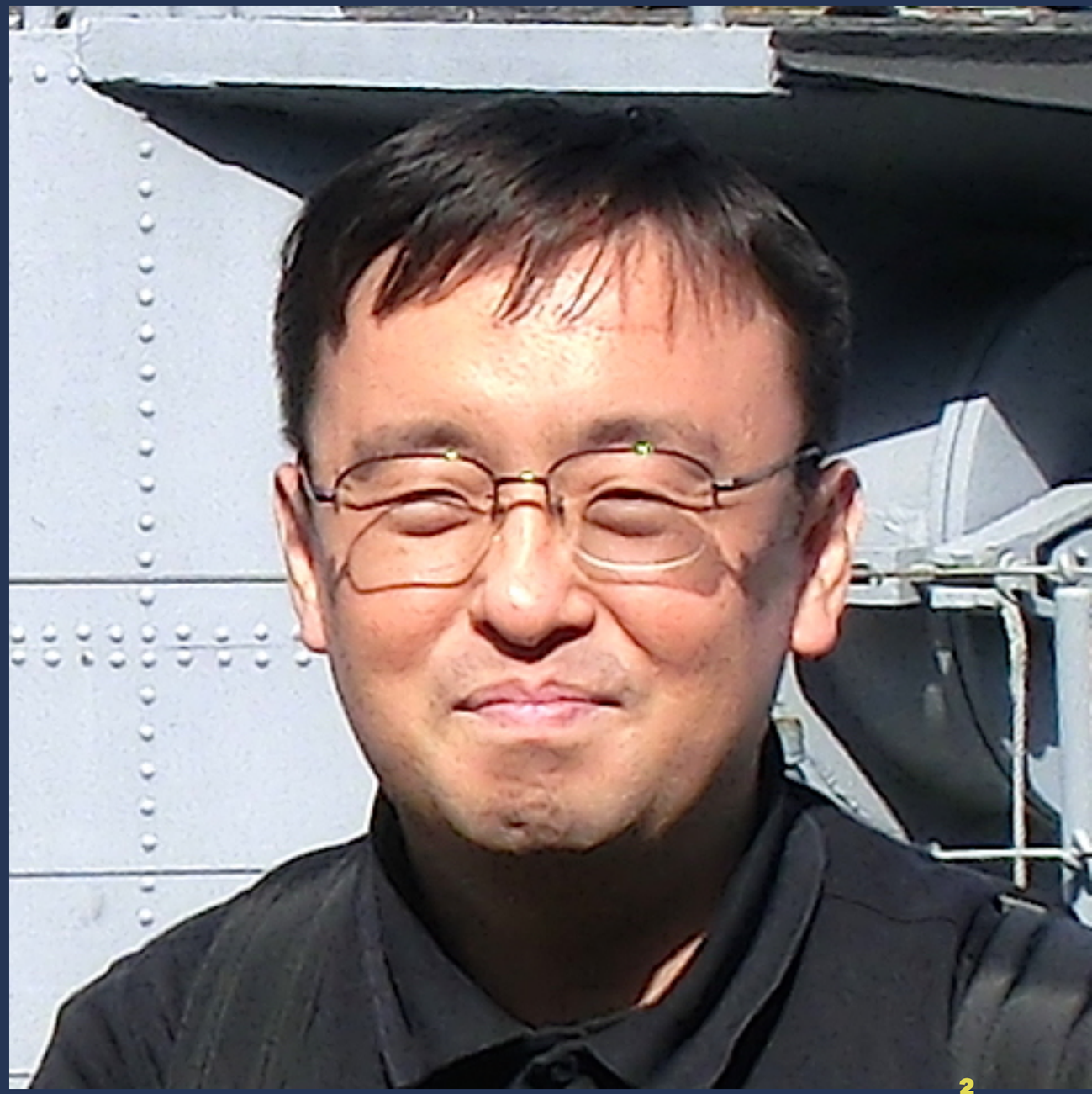
Fifteen Ways to Leave Your Random Module

Kenji Rikitake

9-SEP-2016
Erlang User Conference 2016
Stockholm, Sweden
@jj1bdx



Past random number talks sponsored by Erlang Solutions

- Erlang Factory SF Bay Area 2011: SFMT on Erlang
- London Erlang User Group September 2013: Erlang PRNG
- Erlang Factory SF Bay Area 2015: Xorshift*/+ on Erlang

... so *fourth* presentation this time!

So why I want you to leave the random module?

Random module is already deprecated in OTP 19.0 and will be removed in OTP 20!

AS183: the random module algorithm

- Originally written for 16-bit machines in 1982
- Relatively short period (6,953,607,871,644 = $2^{42.661}$)[1]
- Explorable in less than 9 hours with Intel Core i5 single core[2]

---

[1] B. A. Wichmann, I. D. Hill, "Algorithm AS 183: An Efficient and Portable Pseudo-Random Number Generator", Journal of the Royal Statistical Society. Series C (Applied Statistics), Vol. 31, No. 2 (1982), pp. 188-190, Stable URL: http://www.jstor.org/stable/2347988

[2] https://github.com/jj1bdx/as183-c

AS183 code on FORTRAN

Microsoft's implemantation on Excel 2003[3]:

```
C     IX, IY, IZ SHOULD BE SET TO INTEGER VALUES
C     BETWEEN 1 AND 30000 BEFORE FIRST ENTRY
      IX = MOD(171 * IX, 30269)
      IY = MOD(172 * IY, 30307)
      IZ = MOD(170 * IZ, 30323)
C23456 AMPERSAND SHOWS LINE CONTINUATION
      RANDOM = AMOD(FLOAT(IX) / 30269.0 +
     &         FLOAT(IY) / 30307.0 +
     &         FLOAT(IZ) / 30323.0, 1.0)
```

[3] Description of the RAND function in Excel, https://support.microsoft.com/en-us/kb/828795, modified by Kenji Rikitake for better readability (and FORTRAN 77 compatibility)

Issues of the random module

- AS183 is no longer safe in 2016; the period is too short
- *Without explicit seeding the result is always the same*
- Seeding with `erlang:now/0` can be easily exploited

```
%%% erlang:now/1 is also deprecated since 18.0!
_ = random:seed(erlang:now()). % DON'T DO THIS!
```

Think about the purpose of the randomness before using

- Security? Generating passwords or keys?
- Simulation? Needs a long period?
- Compatibility with older OTP 17.x or before?

Let's get down to the recipes

#1: Check the compile-time error message of deprecated functions

In OTP 19.0 or later, the compiler generates the warnings as in `otp_internal:obsolete/3`:

```
obsolete_1(random, _, _) ->
    {deprecated, "the 'random' module is deprecated; "
     "use the 'rand' module instead"};
```

#2: Use crypto module for secure random number generation

- `crypto:strong_rand_bytes/1`
- OpenSSL `RAND_bytes()` wrapper

```
1> crypto:strong_rand_bytes(10).
<<3,63,210,4,69,106,175,117,160,139>>
2> crypto:strong_rand_bytes(10).
<<69,169,134,65,238,118,51,203,47,125>>
```

#3: Use /dev/urandom for security

/dev/urandom is *not a regular file*[4]

```
1> Size = 10.
10
2> Cmd = lists:flatten(io_lib:format(
         "head -c ~p /dev/urandom~n", [Size])).
"head -c 10 /dev/urandom\n"
3> list_to_binary(os:cmd(Cmd)).
<<58,133,170,67,160,90,91,165,56,91>>
4> list_to_binary(os:cmd(Cmd)).
<<201,14,233,86,15,47,168,96,85,61>>
```

[4] See https://azunyanmoe.wordpress.com/2011/03/22/reading-device-files-in-erlang/ for the detailed explanation

#4: Use entropy-supplying system calls for security

- Linux (and Solaris) has `getrandom()` and `getentropy()`
- FreeBSD has `sysctl` MIB `KERN_ARND`/`kern.arandom` as:

```
%%% For FreeBSD only: Linux and Solaris need C code
9> list_to_binary(os:cmd("sysctl -X -b -B 10 kern.arandom\n")).
<<18,231,137,93,134,250,30,219,244,149>>
10> list_to_binary(os:cmd("sysctl -X -b -B 10 kern.arandom\n")).
<<188,136,104,118,223,21,21,142,121,225>>
```

#5: Use hardware random number generator for security

- Entropy generated in computers especially servers is low[5]
- Use external generator (with physical sources) such as: avrhwrng[6] / NeuG[7] / ChaosKey[8]

---

[5] Bruce Potter, Sasha Wood, Managing and Understanding Entropy Usage (pdf) (presented at BlackHat USA 2015 conference)

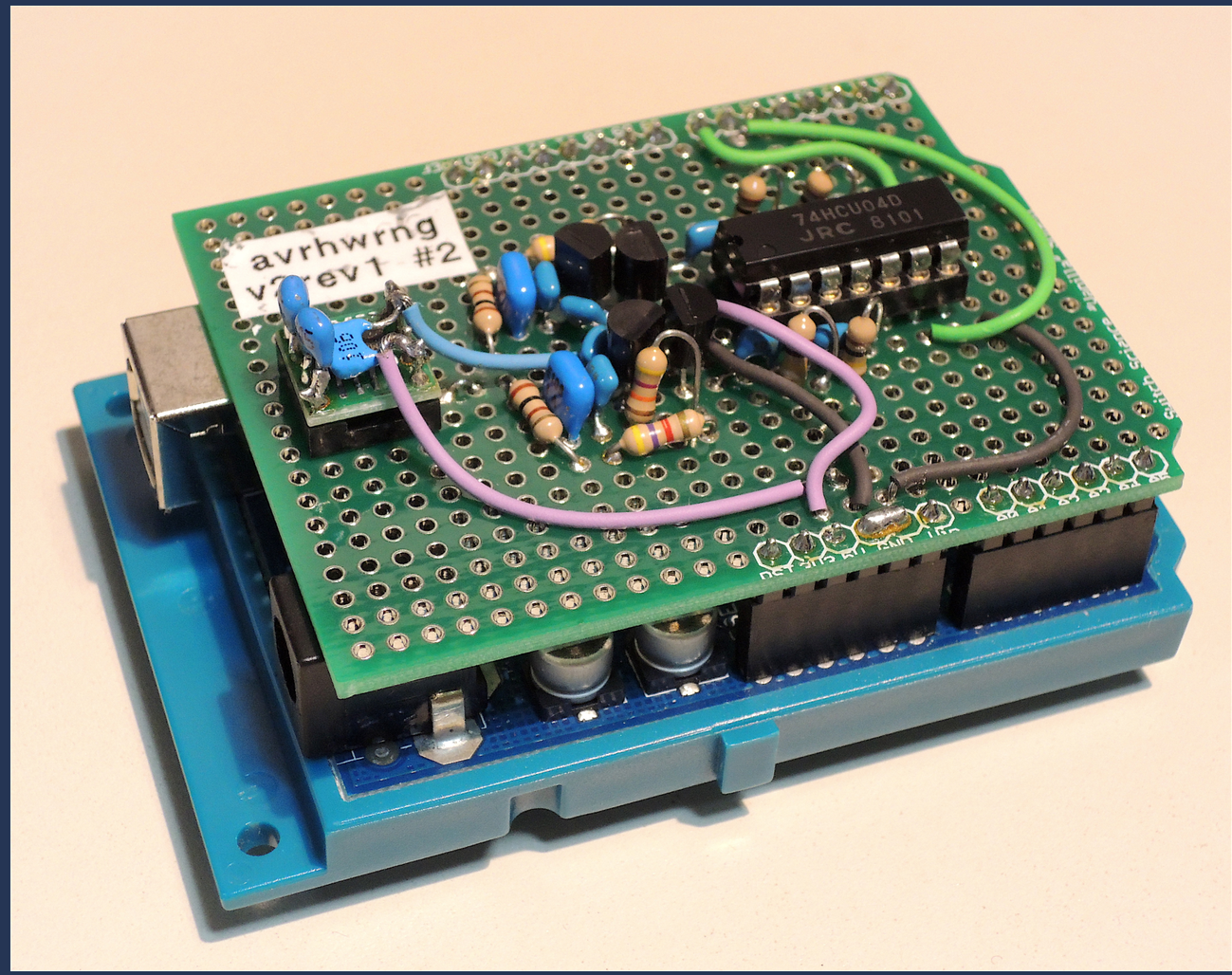
[6] Arduino UNO R3 + noise generator board: https://github.com/jj1bdx/avrhwrng/

[7] STM32F103 USB dongle: https://www.gniibe.org/memo/development/gnuk/rng/neug.html

[8] STM32F043 USB dongle: http://altusmetrum.org/ChaosKey/

avrhwrng v2rev1

- A shield for Arduino UNO R3 (and other compatible boards)
- Two digital random outputs from independent avalanche noise diodes and the amplifiers
- Generates ~80kbps with USB serial 115200bps port
- Design finalized on June 2016
- Source on GitHub



#6: Seeding rand module is different from seeding random module

#6.0: Seeding in per-process and functional APIs

- `rand:uniform/{0,1}` uses *per-process* seeding: the seed is in the *process dictionary*
- `rand:uniform_s/{1,2}` uses *functional* interface: the seed is given *in the function argument*
- These are the same in `random` module too

#6.1: random module needs *explicit and different* seeding for each process

- `random:seed/0` returns *a fixed value*: *explicit* seeding for each process as followis is *required*:

```
%%% Don't use erlang:now/0; use this for OTP 18.0 and later
random:seed({erlang:phash2([node()]),
             erlang:monotonic_time(),
             erlang:unique_integer()})
```

#6.1: Per-process API functions in rand module is *automatically seeded* on the first call

- You *do not need to call* `rand:seed/{1,2}` if you decide to use the process dictionary for storing the state
- For every process the seed is *different from each other* when it is automatically initialized in this way

#6.2: Seeding in `random:seed/3` no longer works in `rand:seed`

```
%%% Don't do this: this will fail
rand:seed(100, 200, 300) % no rand:seed/3 defined
%%% Do this
rand:seed(exsplus, {100, 200, 300}) % needs algorithm
%%% If you need the explicit state, use rand:seed_s/2
rand:seed_s(exsplus, {100, 200, 300}) % needs algorithm
```

#6.3: Do not assume the seed is stored as tuples on rand module!

- On rand module, seeds are *algorithm dependent*
- Seeds have *internal* and *external* format
- Internal format: algorithm handler and the seed
- External format: algorithm name (atom) and the seed

#6.3.1: Internal seed format

```
1> S = rand:seed_s(exsplus, {100, 200, 300}).
{#{max => 288230376151711743,
next => #Fun<rand.8.41921595>,
type => exsplus,
uniform => #Fun<rand.9.41921595>,
uniform_n => #Fun<rand.10.41921595>},
 [288090199732603799|1900797102015]}
```

#6.3.2: Use external format to transfer the state inside the process dictionary

```
2> ES = rand:export_seed_s(S).
{exsplus,[288090199732603799|1900797102015]}
3> S =:= rand:seed_s(ES).
true
4 > rand:seed(ES), rand:export_seed() =:= ES.
true
```

#7: Use default algorithm `exsplus` if you don't have other needs

- rand module have three Xorshift*/+ algorithms
- Default `exsplus` is fast, sufficient in most use cases
- `exsplus`: Xorshift116+, 58 bits, period: $(2^{116}-1)$
- `exs1024`: Xorshift1024*, 64 bits, period: $(2^{1024}-1)$
- `exs64`: Xorshift64*, 64 bits, period: $(2^{64}-1)$

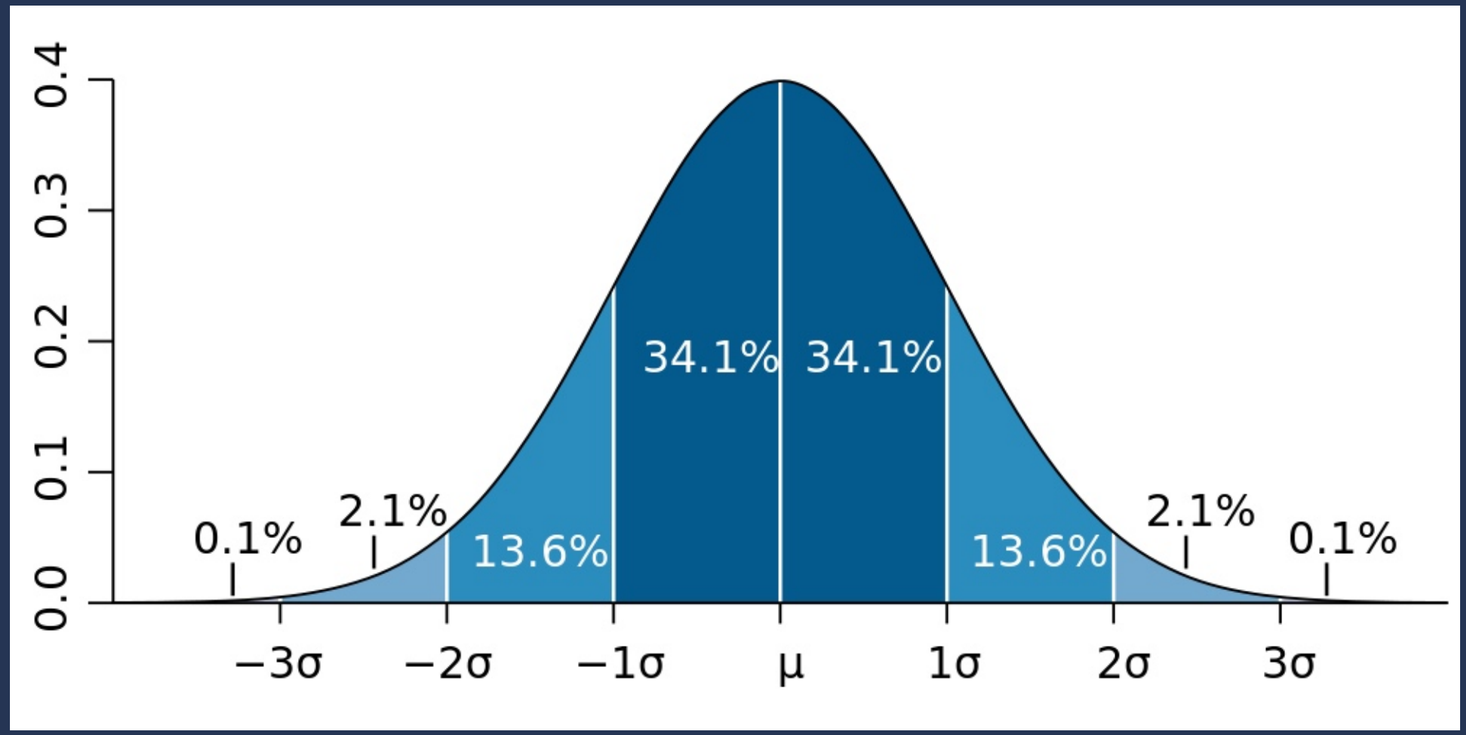
#8: Try exs1024 algorithm of rand module for simulation

- Longer periods are required for high-precision simulation
- `exs1024` has a sufficiently longer period than exsplus
- `exs1024` takes less than x2 execution time than `exsplus`

#9: Use rand:normal/0 for normal distribution

- `rand:normal/0` gives normal distribution output x of $\sigma = 1$ (standard deviation) and $\mu = \bar{x} = 0$ (mean value), based on fast ziggurat algorithm
- Normal distribution represents central limit theorem, where sums independent random variables follow

Normal distribution[9]



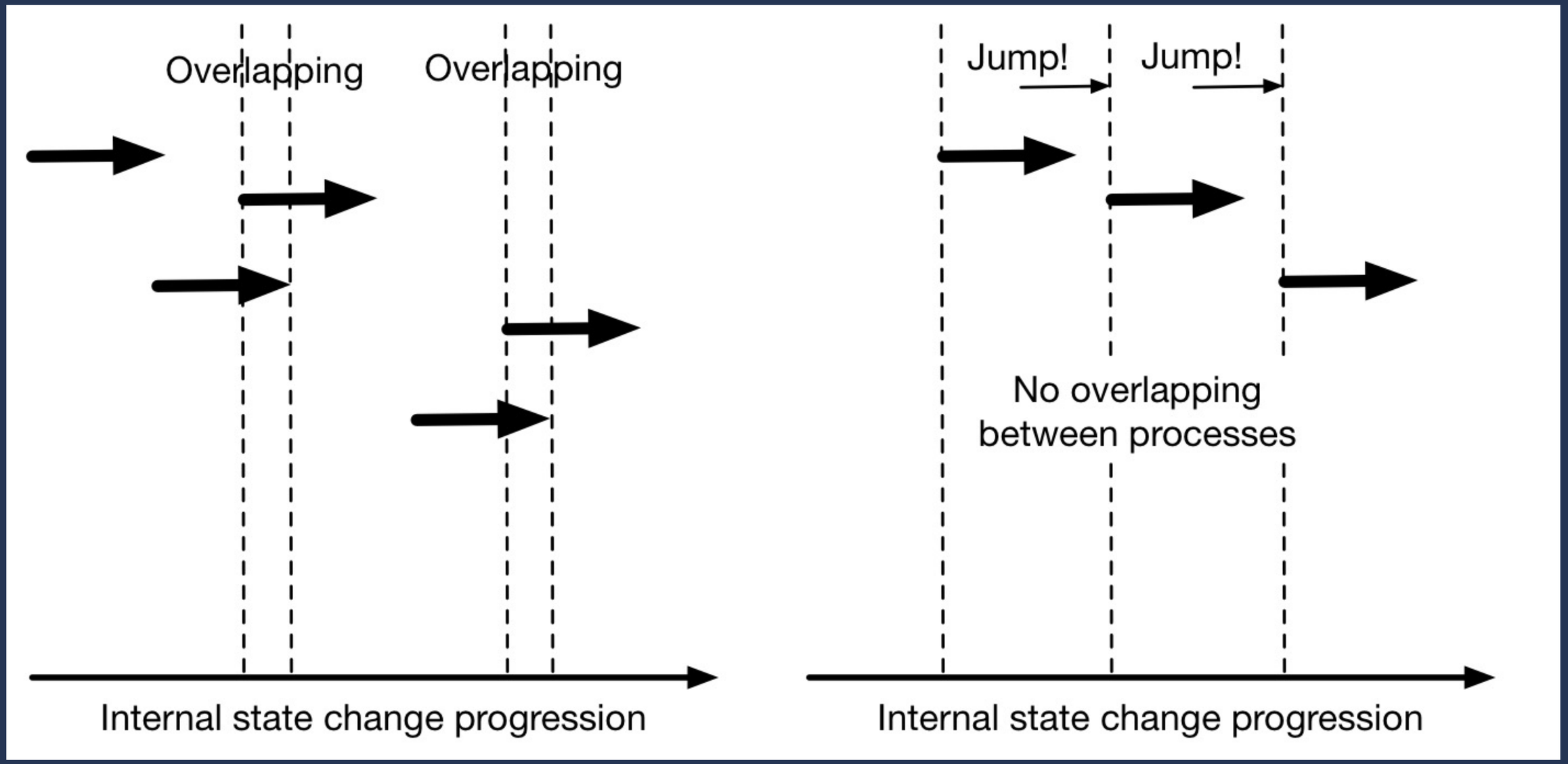
[9] By Mwtoews [CC BY 2.5] via Wikimedia Commons
https://commons.wikimedia.org/wiki/File%3AStandard_deviation_diagram.svg

#10: Use SFMT for a hard-core long-time simulation

- A typical SIMD-oriented Fast Mersenne Twister (SFMT) algorithm has the period of $(2^{19937} - 1)$
- The *extremely* long period may affect the results if the number of random samples is huge
- `sfmt-erlang` is a NIF-based implementation of 32-bit output streams and `rand`/`random` module compatible

#11: Check orthogonality of random generators for concurrent/parallel operations

- Each process must generate orthogonal sequences
- Use jump functions for ensuring orthogonality on Xorshift*/+ (exsplus116 and exs1024 are jump-function ready)
- `tinymt-erlang` can choose 2^{58} parameters (2^{28} subset available here) (period: $(2^{127}-1)$, 32-bit output)


Overlapping
Overlapping
Internal state change progression
Jump!
Jump!
No overlapping
between processes
Internal state change progression

#12: Use non-random external modules for OTP 17.x or before

- Use exsplus116, exs64, exs1024 (with HiPE for speed)
- sfmt-erlang and tinymt-erlang also work
- For proper seeding (from LYSE):

```
%% properly seeding the process
<<A:32, B:32, C:32>> = crypto:strong_rand_bytes(12)
random:seed({A,B,C}).
```

#13: Use wrappers for encapsulating the changes of random and rand modules

- With Tuncer Ayaz's `erlang-rand-compat` module, you can use `rand` if available, or fall back to `random` if not
- Examples: triq, rebar
- Rewriting code is still better, though (see a rebar3 commit)

#14: Implement your own modules for compatibility with old OTP versions (should be done *very* carefully)

- Jean-Sébastien Pédron did this on RabbitMQ
- Example: `src/rand_compat.erl` in rabbitmq-common
- Similar solution for time functions: erlang-time-compat

#15: If you do need to write your own code and algorithm, check at least stochastic and statistic consistency and quality

- Use checking tools: ent, Dieharder, TestU01
- Metrics: entropy, statistic estimators, pattern detection
- Measure at least for 1Gbytes, or even more

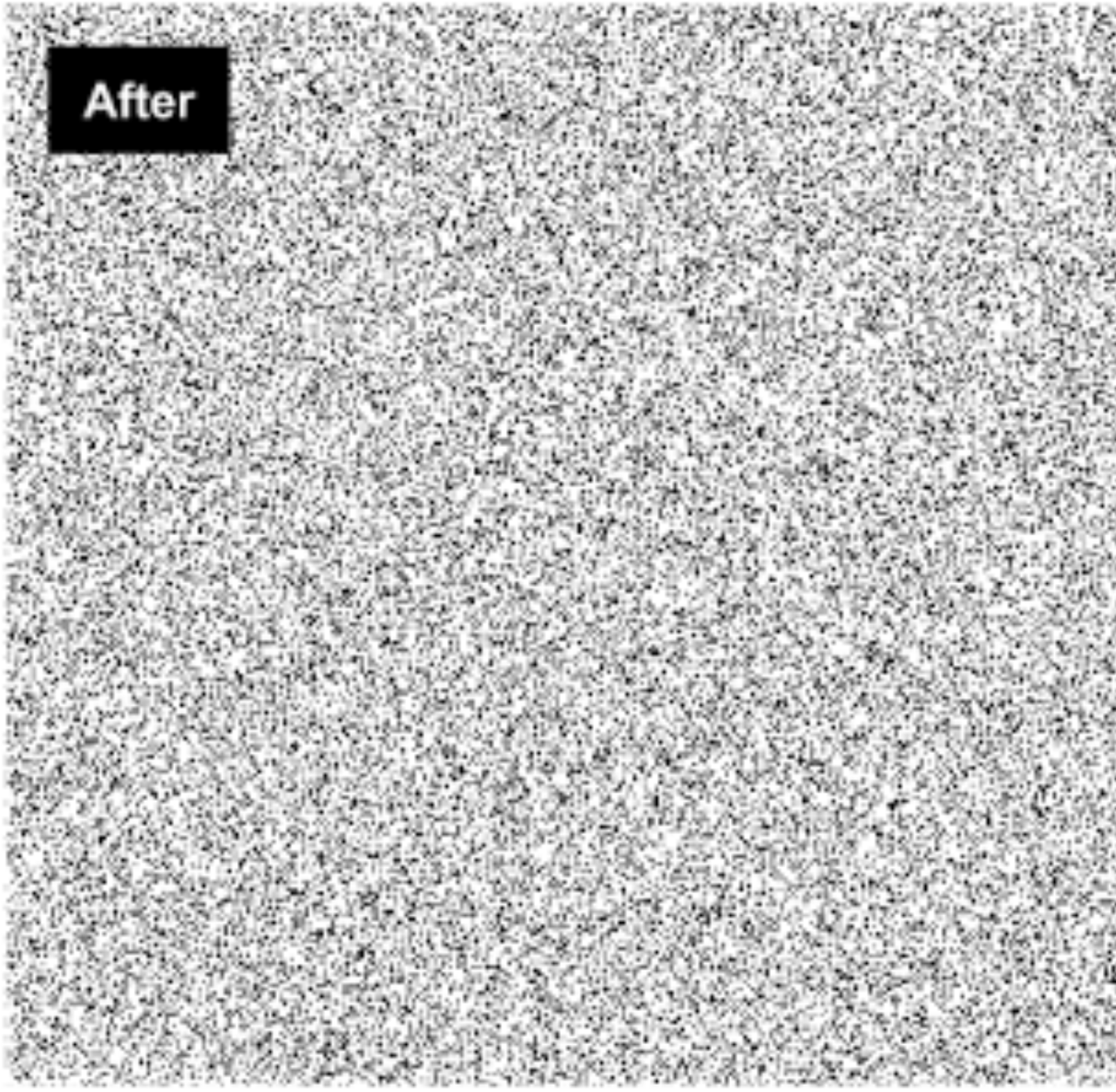
Before

After

Failure example: JavaScript V8 Engine[10]

[10] "There's Math.random(), and then there's Math.random()", V8 JavaScript Engine blog, 17-DEC-2015

Randomness visualization before PRNG update (V8 4.7) and after (V8 4.9) (visualization by Mike Malone, http://bl.ocks.org/mmalone/bf59aa2e44c44dde78ac, CC-BY-4.0)



Randomness visualization before PRNG update (V8 4.7) and after (V8 4.9) (visualization by Mike Malone, http://bl.ocks.org/mmalone/bf59aa2e44c44dde78ac, CC-BY-4.0)

Summary: Use rand module *now*

- There are already many ways and code samples to migrate to rand module from random module
- For security, use crypto module or `/dev/urandom`, preferably with hardware random number generators
- If you can't use 18.0 or later, **stop using random module** and use newer random number generator algorithms
- **Test your code before releasing it into production!**

Acknowledgment

- Dan Gudmundsson - rand module principal developer
- Sebastiano Vigna - Xorshift*/+ inventor
- Erlang Solutions
- ... and you all!
- Slides at https://github.com/jj1bdx/euc2016-erlang-prng/

Thank you
Questions?