SFMT Pseudo Random Number Generator for Erlang

Kenji Rikitake

Institute of Information Management and Communication (IIMC), Kyoto University

kenji.rikitake@acm.org

Abstract

The stock implementation of Erlang/OTP pseudo random number generator (PRNG), *random* module, is based on an algorithm developed in 1980s called AS183, and has known statistic deficiencies for large-scale applications. Using modern PRNG algorithms with longer generation periods reduces the deficiencies.

This paper is a case study of *sfmt-erlang* module, an implementation of SIMD-oriented Fast Mersenne Twister (SFMT) PRNG with the native interface functions (NIFs) of Erlang. The test results show the execution speed of the implementation is approximately three times faster than the *random* module on the x86 and x86_64 architecture computers, and the execution own time for generating single random number sequences is proportional to the internal state table length.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.3.2 [*Programming Languages*]: Language Classifications—Erlang; G.3 [*Probability and Statistics*]: Random Number Generation

General Terms Algorithms, Performance

Keywords Erlang, NIF, Pseudo Random Number Generator, SFMT, Mersenne Twister

1. Introduction

Random number generators (RNGs) is an essential component of modern programming language library. Modern operating systems provide the following two types of RNGs.

"True" RNGs True RNGs, also known as hardware random number generators, obtain randomness from physical sources of unexpectable behaviors, such as avalanche diode noise. The number generation cost of True RNGs are expensive and the practical usage is limited to giving the seeds (initial vectors) of PRNGs for cryptography.

Pseudo RNGs Pseudo RNGs (PRNGs) are deterministic computed number sequences. PRNGs generate the same results from the same seed. The possible result values are within a finite range of numbers such as 32-bit integers, and the sequence is periodic. The pseudo-randomness is evaluated by how long the generation period is. Examples of practical usages of the PRNGs are hash table indices, parameters for testing, and other parameters for simulation and modeling.

Erlang/OTP [5] has a built-in PRNG library called *random* module. The module includes a multiplicative congruential algorithm called AS183 [1], published in 1982. The AS183 has the period length of $7 \times 10^{12} (\sim 2^{43})$ [2], which only sustains $\sim$ 81 days when 10^6 generations per second occur[1]. The internal state of AS813 is made of three 15-bit integers, which is susceptible to the brute-force attacks. A PRNG with much longer period length is needed to assure randomness of the generated number sequences for a long period of time.

Open source language systems such as Python [11] and R [12] have changed the default choice of built-in PRNG to the one with a longer period one called Mersenne Twister (MT) [9]. MT has a long period length of $2^{19937} - 1$, which is sufficiently larger for real-world simulation applications. MT passes a set of statistical tests called Diehard [7]. MT has the internal state table of 624 32-bit integers, which is significantly larger than AS183.

As an improved version of MT, SIMD-oriented Fast Mersenne Twister (SFMT) [15] is published in 2008 as an open source implementation [16]. SFMT generates number sequences with a better dimension of equidistribution and has a faster generation speed than MT. SFMT has a set of combination of parameters which allows the generation period to change from $2^{607} - 1$ to $2^{216091} - 1$, primarily depending on the internal state table size.

This paper is a case study of an implementation of SFMT for Erlang with the native interface functions (NIFs) called *sfmt-erlang*. We first implemented SFMT in pure Erlang and verified the correctness by comparing the generated sequences. The pure Erlang code was $\sim$ 300 times slower than the C code, so we decided to reimplement it by NIFs. The NIF code is $\sim$ 40 times faster than the pure Erlang code, and even faster than the *random* module. The experience presented here can help others in the Erlang/OTP development community to design NIF implementations in an efficient way, and to implement their own PRNGs in Erlang.

An outline of this paper's topics is as follows:

- summaries of the SFMT algorithm (Section 2);
- practical issues of modifying the reference code of SFMT in C to pure-Erlang code, and as Erlang NIFs (Section 3);
- performance test results of *sfmt-erlang* and comparison with the *random* module;
- related work (Section 5); and
- concluding remarks (Section 6).

2. The SFMT Algorithm

The SFMT algorithm of Saito and Matsumoto [15] is based on Linear Feedbacked Shift Register (LFSR) based on a recursion of

[1] $(7 \times 10^{12})/(24 \times 60 \times 60 \times 10^6) \simeq 81.0$.

Function name	Description
`gen_rand_all/1`	(NIF) Fills the internal state table with generated random numbers
`gen_rand_float/2`	Generates a float random number of [0,1] range
`gen_rand_list32/2`	(NIF) Generates a list of given size of 32-bit integer random numbers
`gen_rand32_max/2`	Generates an integer random number of [0,Max] range
`uniform_s/{1,2}`	Generate a float/integer random number, compatible with the *random* module; internally `uniform_s/1` calls `gen_rand_float/2`, and `uniform_s/2` calls `gen_rand32_max/2`

Table 1. List of *sfmt-erlang* exported functions referred in this paper.

128-bit shift registers. The internal state is represented as a table of 128-bit integers, and the size of the table N is 156 in the case where the generation period is $2^{19937} - 1$ (SFMT19937)[2].

Table 1 shows a description of *sfmt-erlang* exported functions referred in this paper. Figure 1 shows a brief description of the SFMT algorithm. In this algorithm, a block of N or more 128-bit integers are generated at once. The last N 128-bit integers of the generated block are also used as the content of the new internal state table. The generated block can be used as a pseudo random number pool of N 128-bit integers. See Figure 2 for the explanation of `do_recursion/4`, an example implementation of the LFSR function for SFMT19937.

The major differences between SFMT and AS183 are as follows:

- The internal state size of SFMT is much larger. The internal state of SFMT19937 has 156 128-bit integers (2496 bytes), while the AS183 has only three 15-bit integers (45 bits or less than six bytes).
- SFMT generates a block of the pseudo-random number sequences, while AS183 incrementally generates the sequence for each number.
- SFMT generates 128-bit integers. On the other hand, AS183 generates (0,1) real numbers which fails some tests at a probability level of $< 10^{-15}$ [20]; this indicates AS183 has no higher resolution than 50 bits.

3. Implementation Issues of SFMT in Pure Erlang and in C NIFs

Our implementation of *sfmt-erlang* proceeded in the following sequence:

- Revision of the SFMT reference C code for NIF use;
- Development of a pure Erlang version; and
- Modification of the C code into a NIF shared library.

3.1 Revision of the SFMT Reference Code

The first step of the code development was to revise the reference code called *sfmt-1.3.3* [16]. While this code was written mostly straight-forward, we had to solve the following problems:

Use of static arrays The internal state table was defined as `static` arrays. While this is effective to gain the execution speed, this renders the code totally unusable for a reentrant execution environment such as in the Erlang NIFs. We revised the code to remove all the `static` variable definitions.

Non-reentrant programming The functions manipulating the internal state table and performing the recursion were written to implicitly work on the single table. While this is effective for speeding up the running code, this also renders the code useless

[2] For a period of $2^n - 1$, $N = \lfloor n/128 \rfloor + 1$.

Recursion algorithm for `gen_rand_all/1` and `gen_rand_list32/2`:

- N: size of the internal table (in 128-bit integers, 156 for SFMT19937)
- S: size of output array (in 128-bit integers)
  - (For `gen_rand_all/1`, $S = N$)
  - (For `gen_rand_list32/2`, $S \geq N$)
- `POS1`: pickup offset position of the internal table (122 for SFMT19937) for `do_recursion/4`
- `w128()`: 128-bit integer (represented by a list of four 32-bit integers)
- `a[]`: output array (of S `w128()` elements)
- `i[]`: internal state (of N `w128()` elements)
- `r(a, b, c, d)`: `do_recursion/4` function (of 4 `w128()` arguments)

The following is the process of recursion (The array indices are in C notation, i.e, [0 ... J-1] for J elements)

```
a[0] = r(i[0], i[POS1],   i[N-2], i[N-1]);
a[1] = r(i[1], i[POS1+1], i[N-1], a[0]);
a[2] = r(i[2], i[POS1+2], a[0],   a[1]);
...
a[(N-POS1)-1] =
  r(i[(N-POS1)-1], i[N-1], a[(N-POS1)-3], a[(N-POS1)-2]);
a[(N-POS1)]   =
  r(i[(N-POS1)],   a[0],   a[(N-POS1)-2], a[(N-POS1)-1]);
a[(N-POS1)+1] =
  r(i[(N-POS1)+1], a[1],   a[(N-POS1)-1], a[(N-POS1)]);
...
a[N-1] = r(i[N-1], a[POS1-1], a[N-3], a[N-2]);
```

The assignments below are only applicable to `gen_rand_list32/2` when $S > N$:

```
a[N]   = r(a[0],   a[POS1],   a[N-2], a[N-1]);
a[N+1] = r(a[1],   a[POS1+1], a[N-1], a[N]);
...
a[X]   = r(a[X-N], a[X-(N-POS1)], a[X-1], a[X-2]);
...
a[S-1] = r(a[(S-N)-1], a[S-(N-POS1)-1], a[S-2], a[S-3]);
```

The last N `w128()` elements of `a[]` represents the new internal state `ni[]`, by the following copy operation:

```
ni[0] = a[S-N], ni[1] = a[S-N+1], ... ni[N-1] = a[S-1].
```

Figure 1. A brief description of SFMT algorithm.

```
%% in pseudo-Erlang code
%% A, B, C, D all represent 128-bit unsigned integers
do_recursion(A, B, C, D) ->
   A2 = (128-bit-left-shift A by 8 bits) bxor A,
   B2 = (quadruple 32-bit-right-shift B by 11 bits) band
        (16#BFFFFFF6BFFAFFFFDDFE7BCFDFFFFFEF),
   C2 = (128-bit-right-shift C by 8 bits),
   D2 = (quadruple 32-bit-left-shift D by 18 bits),
   A2 bxor B2 bxor C2 bxor D2.
```

Figure 2. What `do_recursion/4` does for SFMT19937 [15, Section 2.3].

```
%% To avoid appending two lists,
%% a and b of r(a, b, c, d) form ring buffers
%% (e.g., Int and AccInt, IntP and AccIntP,
%%  of gen_rand_recursion/8)
%% This makes the algorithm simpler and faster

gen_rand_recursion(0, Acc, _, _, _, _, _, _) ->
    lists:reverse(Acc);
gen_rand_recursion(K, Acc, Int, AccInt,
                   [], AccIntP, R, Q) ->
    gen_rand_recursion(K, Acc, Int, AccInt,
                       lists:reverse(AccIntP),
                       [],
                       R, Q);
gen_rand_recursion(K, Acc, [], AccInt,
                   IntP, AccIntP, R, Q) ->
    gen_rand_recursion(K, Acc,
                       lists:reverse(AccInt),
                       [],
                       IntP, AccIntP, R, Q);
gen_rand_recursion(K, Acc, Int,
                   AccInt, IntP, AccIntP,
                   [R0, R1, R2, R3],
                   [Q0, Q1, Q2, Q3]) ->
    [A0, A1, A2, A3 | IntN ] = Int,
    [B0, B1, B2, B3 | IntPN ] = IntP,
    [X0, X1, X2, X3] = do_recursion([A0, A1, A2, A3],
                                    [B0, B1, B2, B3],
                                    [R0, R1, R2, R3],
                                    [Q0, Q1, Q2, Q3]),
    gen_rand_recursion(K - 4,
                       [X3 | [X2 | [X1 | [X0 | Acc]]]],
                       IntN,
                       [X3 | [X2 | [X1 | [X0 | AccInt]]]],
                       IntPN,
                       [X3 | [X2 | [X1 | [X0 | AccIntP]]]],
                       [Q0, Q1, Q2, Q3],
                       [X0, X1, X2, X3]).
```

Figure 3. Recursion function of *sfmt-erlang*.

for reentrant programming. We revised the code to explicitly specify the internal state table by the pointers.

Redundant features The code contained optimized code for the *altivec* feature of the PowerPC architecture and the 64-bit C integer RNG operations. We removed the PowerPC code due to lack of the testbeds. We also decided to remove the 64-bit C integer RNG code since mixing it up with the 32-bit RNG can cause a fatal bug of the whole generation process.

We have released the code as *sfmt-extstate* [14], as a general purpose building block for other programs.

3.2 Development of a Pure Erlang Version of SFMT

The second step of the code development was to write the module of SFMT PRNG in Erlang with 32-bit integer operations, so that we could reuse it as a skeleton for the module with the NIF C shared library.

We started the work by translation of *sfmt-extstate* literally into Erlang, focusing on the following points:

- We decided to represent the internal table as an Erlang list so that the head-and-tail operation to a copy of the list could be used for faster generation of the random number sequence.
- We added a set of interface functions such as `uniform_s/1` so that the SFMT code could be a drop-in replacement of *random* module functions.
- The SFMT reference C code provided the example pairs of the seeds and the generated sequences for regression tests. Those data were useful for debugging.

We refactored the code following the rules below:

Extensive use of head-and-tail lists Erlang head-and-tail operations are generally much faster than using the tail-append (++) operator. Figure 3 shows a part of pure Erlang version of SFMT, which head-tail structures for two ring buffers to simplify the recursion operation. Our preliminary measurement showed this code was 50% faster than a former code using the tail-append operator.

Fixed-length integers Erlang integers are *not* fixed-length, so we had to explicitly limit the result bits in the fixed length by the `band` operator, for each time after performing the binary left shift by the `bsl` operator, and for any other operation which may exceed the assumed C integer length.

Binary right shift operator Erlang `bsr` operator always assumes *signed integers* and the *arithmetic* shift right operation, so the left-hand value must be positive or zero to use it for the *logical* shift right operation.

Array objects Objects handled by Erlang *array* modules are immutable. Setting the element value by `array:set/3` actually makes a modified copy, so excessive use of this function makes the code very slow. We avoided the use of arrays in the recursion function of RNG, but we had to use arrays during the internal state table verification to recover the state from 0-excess states [15, Section 6].

Despite the refactoring process, our preliminary measurement showed the pure Erlang version of SFMT code was ~ 300 times slower than the original C code. So we decided to further refactor the code by using NIFs for increasing the running speed.

3.3 Modification of the C code into a NIF shared library

We proceeded with the following conversion process of the pure Erlang version of SFMT code further into pieces of the NIF-based code:

Following the manual The *erl_nif* [4] C library provides the interface C functions for writing NIFs. Most of the necessary interface operations were included in the library during the coding work. Reading the library manual is surely the first thing to do.

Studying the sample code Erlang *crypto* module has been fully converted into NIFs from the R14B release. Reading the C source code[3] is a good starting point to write a NIF shared library.

Starting from the smaller functions When a NIF fails, it crashes the whole Erlang virtual machine (BEAM) at once. Incremental conversion from a small piece of code with a full regression test suite such as *eunit* is essential.

The code conversion process was proceeded by merging the pure Erlang SFMT code and the C functions of *sfmt-extstate*. We refactored the C code following the rules below:

NIF module assumes static C scope In a NIF C shared library, all the C functions must be declared in the same `.c` file with `static` scope to prevent namespace clash.

Large binaries must not be modified in a NIF For large Erlang binaries, BEAM assumes the actual objects are treated as immutable and the objects must not be modified within the NIF

[3] At `lib/crypto/c_src/crypto.c` for R14B03.

C function. Always make a copy of the passed binary object[4] *before* modifying it.

NIF blocks the BEAM scheduler Staying in a NIF function for a longer period may cause delay in execution of other Erlang processes and functions. While the delay will not cause serious problems with a dual- or multi-core execution environments, the blocking time should be minimized to retain responsiveness of BEAM in case of running the code in a single-core environment.

SSE2 code does not fit well The code contained the SSE2 feature code for the x86 and x86_64 architectures. While we first successfully ran the SSE2-optimized code, we decided to remove the feature due to the following reasons:

- the execution time of our testing code only reduced to ~ 80% to that of the unoptimized code, even when successfully executed;
- the SSE2 code occasionally started to crash, presumably due to address misalignment to 128-bit boundary of the memory allocated by `enif_alloc()`; and
- two versions of code, those with and without SSE2 features, will not coexist in the same code file without a compile-time macro definition.

During the code conversion to NIF, we compared the following two methods to handle the internal state table for generating the number sequence, represented in an Erlang binary:

List processing In this method, we first convert the Erlang binary representation of the internal table to an Erlang list of number sequences, then generate the sequence by the head-and-tail operation.

Random access through a NIF In this method, we generate the number sequence each time by calling a NIF which returns the number, with the position index and the Erlang binary internal table.

An early experiment showed that the speed of the list processing method was ~ 32% faster than the random access method through a NIF. So we decided to choose the list processing method. This result was opposite of our assumption before the measurement, and showed another example of a rule of thumb: *profile before optimize*.

4. Performance Test and Results

We conducted a performance comparison test between *random* and *sfmt-erlang* modules with the following three execution environments, all running Erlang R14B03:

leciel Intel Atom N270 dual-core CPU, 1.6GHz CPU clock, OS: FreeBSD/i386 8.2-RELEASE;

reseaux Intel Core2Duo E6550 dual-core CPU, 2.3GHz CPU clock, OS: FreeBSD/i386 8.2-RELEASE; and

thin AMD Opteron 8350 ×4 (16 CPU cores), 2.3GHz CPU clock, RedHat Enterprise Linux AS V4 of x86_64 architecture[5].

We chose the five SFMT generation periods of $2^n - 1$ where $n = 607, 4243, 19937, 86243, 216091$[6] for evaluation of the performance change possibly caused by the difference of the internal table length.

[4] For example, with a pair of `enif_make_new_binary()` and `memcpy()`.

[5] The test programs were executed in a batch queue to assign an individual run-time node for reduce the interference of other running programs on Kyoto University Supercomputer Thin Cluster.

[6] Saito and Matsumoto [16] have given precomputed values of SFMT parameters with the method proposed by Matsumoto and Nishimura [8].

n	607	4253	19937	86243	216091
N	5	34	156	674	1689
total wall clock execution time [ms]					
SFMT `gen_rand_list/2`	200	210	200	210	210
SFMT `uniform_s/1`	3070	2420	2430	2220	2570
`random:uniform_s/1`	6740	6810	7680	7410	7510
SFMT `gen_rand32_max/1`	2660	2230	2400	2050	2150
`random:uniform_s/2`	7830	7880	8310	7800	7920

$N = \lfloor n/128 \rfloor + 1$: size of internal tables (in 128-bit words)

Table 2. Total execution time of 100×10^5 random number generations on *thin* environment by wall clock time for each SFMT period of $2^n - 1$.

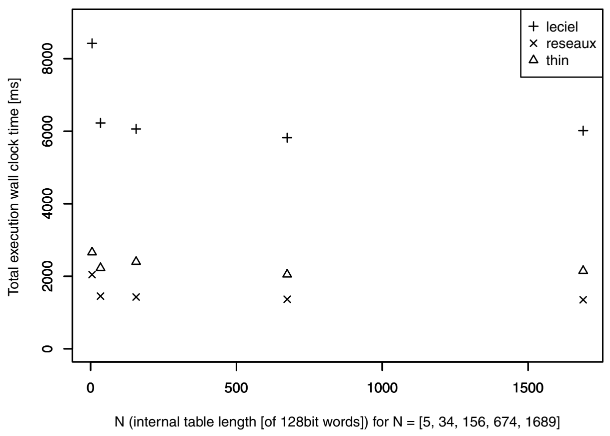

Figure 4. Total execution time of SFMT `gen_rand32_max/2` for 100×100000 calls.

What we wanted to measure on the behavior of *sfmt-erlang* is as follows:

- overall wall clock execution time difference between *random* and *sfmt-erlang* modules;
- execution overhead of the NIF calls;
- how much time exactly a NIF uses by profiling *own time*, i.e., how much time a function has used for its own execution, by *fprof* profiling tool provided in Erlang/OTP; and
- how execution environment differences affect the overall performance.

Table 2 shows the wall clock time for total execution of 100-time loop of 10^5 random number generations. The numbers show that the SFMT modules are roughly three times faster for floating point number generation[7] and three to four times faster for integer generation. Figure 4 shows the wall clock time for three different systems, specifically for SFMT `gen_rand32_max/2` integer generation function. The wall execution time for the period of $2^{607} - 1$ are much larger than those of other periods, while in the other periods the values of total wall clock time are almost the same with each other.

Figure 5 shows the total own time measured by Erlang *fprof* utility for total execution of 10 calls to generate a list of 10000

[7] In SFMT modules, each floating point number has 32-bit resolution between [0,1], which is smaller than ~ 50-bit resolution of *random* module.

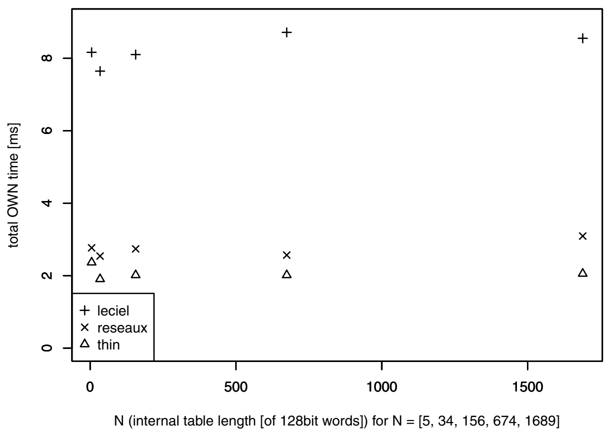

Figure 5. Total own time of SFMT `gen_rand_list32/2` for 10 calls of generating 10000 integer random number lists, measured by *fprof*.

n	607	4253	19937	86243	216091
N	5	34	156	674	1689
Number of calls	13173	1944	425	99	40
total sum of own execution time [ms]					
leciel	205.277	52.233	17.839	9.827	7.757
reseaux	94.610	14.763	4.234	2.149	2.051
thin	54.500	9.597	3.281	2.024	2.011
Own execution time for each call [μs]					
leciel	15.58	26.87	41.97	99.26	193.93
reseaux	7.18	7.59	9.96	21.71	51.28
thin	4.13	4.94	7.72	20.44	50.28

$N = \lfloor n/128 \rfloor + 1$: size of internal tables (in 128-bit words)

Table 3. Own time comparison for total and per-call execution times for generation of 10^5 integer and 10^5 float random numbers of SFMT modules with `gen_rand_all/1` for each SFMT period of $2^n - 1$, measured by *fprof*.

random integer numbers for each. The results show the sums of time `gen_rand_list32/2` used for its own execution to generate random integer number lists stay almost the same even the period of SFMT changes.

Table 3 shows the total sums and individual own time of SFMT `gen_rand_all/1`, which returns a 32-bit random number generated by SFMT, for the total execution of 10 loops of 10000 calls, measured by *fprof*. The number of calls is inversely proportional to the internal table length, and the total sums decrease as the period length become larger. This suggests the calling overhead time of NIFs takes a significant part of the total execution time for a NIF. Figure 6 shows the graph of own time for each tested system. The graph suggests that the own execution time for each call is proportional to the internal table length. During the own execution time the BEAM scheduler will be blocked, so the internal table length should not be set too large for a time-critical application.

The summary of our test results is as follows:

- *sfmt-erlang* functions are roughly three to four times faster than those of *random* module, while retaining the advantage of SFMT.
- Execution overhead of NIF calls is significant for shorter periods of SFMT especially with that of $2^{607} - 1$.

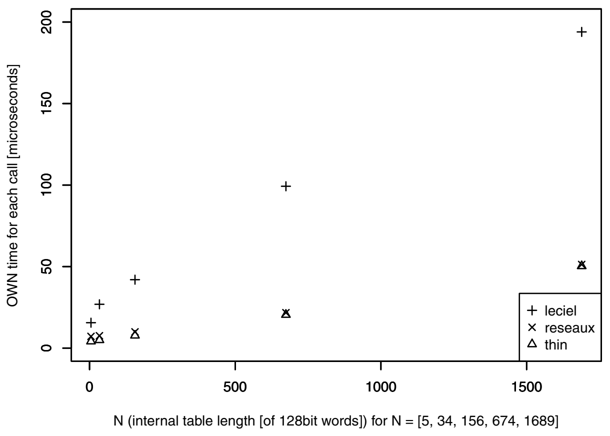

Figure 6. Own time for each call of SFMT `gen_rand_all/1`, measured by *fprof*.

- Execution own time of list generation of random integers for all tested SFMT periods are almost the same.
- Execution own time for each NIF call to update the internal state table is proportional to the internal table length. Generation of 10^5 32-bit integers takes $\leq 50\mu$s for the random number period of $2^{19937} - 1$.
- All above characteristics are similar among the three test execution environments.

5. Related Work

Tim Bates published an implementation of MT in pure Erlang on June 2007 [3]. His implementation was based on the code of David Wallin [19] in 2001, though Wallin's code is no longer publicly available as of May 2011.

A careless use of PRNGs may become a security vulnerability. One serious case has already been discovered on Erlang/OTP. Geoff Cant reports on April 2011 that an attacker can recover SSH session keys and DSA host keys through the weakness of RNG seeding with three guessable integers on the *ssh* module of Erlang/OTP R14B02 and older version releases [18][8]. Another cause of the vulnerability is seeding of the stock PRNG by three integers such as initialization of `random:seed/3` by the result of `erlang:now/3`. This initialization scheme, which we do not recommend, is unfortunately a common practice on Erlang/OTP. We recommend all OTP library and the other application software modules should be immediately and thoroughly reviewed for their PRNG usage.

A careless choice of initialization schemes and seeds may allow attackers unexpected intrusion vectors. Matsumoto et al. [10] report that many modern PRNGs have some manifest patterns when initial seeds are chosen via a linear recurrence. Amit Klein [6] reports on 2008 about the information leakage caused by the JavaScript `Math:random()` implementations in various Web browsers, which can be used for temporary user tracking.

We have tested Wichmann-Hill 2006 algorithm [20] as an replacement of the *random* module. The algorithm has the internal

[8] This vulnerability is fixed on the R14B03 release, by changing the key generation algorithm from that with the *random* module to the one that uses secure PRNG based on OpenSSL [17], which also results in adding new functions of `crypto:strong_rand_bytes/1` and `crypto:strong_rand_mpint/3`.

state of four 31-bit integers, and the generation period is $\sim 2^{120}$, much longer than that of AS183. The algorithm also has a seed generation method for generating non-overlapping random number sequences in parallel. We have implemented the algorithm in pure Erlang as the *random_wh06* module[9]. Our brief experiments [13] show that the execution own time of *random_wh06* functions increases from that of *random* in $< 10\%$ in modern x86 and x86_64 CPUs, although the own time may increase up to $\simeq 63\%$ on CPUs with lesser floating-point capability such as Intel Atom N270.

6. Conclusion

We have presented an implementation of SFMT PRNG for Erlang with NIFs, and have shown the implementation is faster than the stock *random* module, while retaining the PRNG characteristics which have been largely improved from the stock AS183 algorithm. We have also shown that the NIF calling overhead becomes significant when the generation period of SFMT is shorter, and that the execution time of the SFMT internal state table computation function is proportional to the internal state table size.

Our main contributions are to show that there are many rooms to improve Erlang/OTP stock PRNGs, and that computationally complex algorithms such as SFMT can be practically useful with Erlang NIFs. SFMT implementation in this paper, however, is iterative, and does not effectively utilize the concurrent and parallel nature of Erlang. Exploration of parallelism for PRNGs is still an open issue, though that will be possible through a dynamic creation method of PRNGs and a discovery of non-overlapping set of random number sequences.

Source Code Availability

The source code and documentation of *sfmt-erlang* is available at `https://github.com/jj1bdx/sfmt-erlang/` on GitHub.

Acknowledgments

We thank Tuncer Ayaz and Dan Gudmundsson for their contributions to the development of the *sfmt-erlang* software. We also thank Erlang Solutions for giving us the chance to make a work-in-progress presentation of this research at Erlang Factory SF Bay 2011 [13].

During the compatibility test of this software, we used the supercomputer service provided by Academic Center for Computing and Media Studies (ACCMS), Kyoto University.

References

[1] B. A. Wichmann and I. D. Hill. Algorithm AS 183: An Efficient and Portable Pseudo-Random Number Generator. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 31(2):188–190, 1982.

[2] B. A. Wichmann and I. D. Hill. Correction: Algorithm AS 183: An Efficient and Portable Pseudo-Random Number Generator. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 33(1):123, 1984.

[3] T. Bates. Mersenne Twister in Erlang. in mailing list erlang-questions@erlang.org, June 6, 2007. http://erlang.org/pipermail/erlang-questions/2007-June/027068.html

[4] Ericsson AB. erl_nif: API functions for an Erlang NIF library, from ERTS Reference Manual. http://www.erlang.org/doc/man/erl_nif.html

[5] Ericsson AB. Erlang Programming Language and The Open Telecom Platform (Erlang/OTP). http://www.erlang.org/

[6] A. Klein. Temporary user tracking in major browsers and Cross-domain information leakage and attacks, Nov. 2008. http://www.trusteer.com/list-context/publications/temporary-user-tracking-major-browsers-and-cross-domain-information-leakag

[7] G. Marsaglia. A Current View of Random Number Generators. In *Computer Science and Statistics: 16th Symposium on the Interface*, pages 3–10, 1985. http://stat.fsu.edu/pub/diehard/ http://www.evensen.org/marsaglia/keynote.ps}

[8] M. Matsumoto and T. Nishimura. Dynamic creation of pseudorandom number generators. In *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pages 56–69. Springer, 2000. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/articles.html

[9] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998. ISSN 1049-3301. http://doi.acm.org/10.1145/272991.272995

[10] M. Matsumoto, I. Wada, A. Kuramoto, and H. Ashihara. Common defects in initialization of pseudorandom number generators. *ACM Trans. Model. Comput. Simul.*, 17, September 2007. ISSN 1049-3301. http://doi.acm.org/10.1145/1276927.1276928

[11] Python Software Foundation. Python Programming Language. http://www.python.org/

[12] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. ISBN 3-900051-07-0. http://www.R-project.org/

[13] K. Rikitake. Erlang/OTP and how the PRNGs work. In *Erlang Factory SF Bay 2011, Burlingame, CA, USA*. Erlang Solutions, 2011. http://erlang-factory.com/conference/SFBay2011/speakers/kenjirikitake

[14] K. Rikitake, M. Saito, and M. Matsumoto. sfmt-extstate. https://github.com/jj1bdx/sfmt-extstate/

[15] M. Saito and M. Matsumoto. SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In A. Keller, S. Heinrich, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-74496-2.

[16] M. Saito and M. Matsumoto. SIMD-oriented Fast Mersenne Twister (SFMT). http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/

[17] The OpenSSL Project. OpenSSL. http://www.openssl.org/

[18] US-CERT. Erlang/OTP SSH library uses a weak random number generator. US-CERT Vulnerability Note VU#178990. http://www.kb.cert.org/vuls/id/178990

[19] D. Wallin. ANN: ERLMT - Mersenne Twister in Erlang. in mailing list erlang-questions@erlang.org, September 13, 2001. http://erlang.org/pipermail/erlang-questions/2001-September/003580.html

[20] B. A. Wichmann and I. D. Hill. Generating good pseudo-random numbers. *Comput. Stat. Data Anal.*, 51:1614–1622, December 2006. ISSN 0167-9473. DOI: 10.1016/j.csda.2006.05.019 http://portal.acm.org/citation.cfm?id=1219162.1219278

[9] The *random_wh06* module source code is available as a part of the *sfmt-erlang* software.