Project - Analysis, state and seed recovery of RNGs

Abstract

Study of (novel) methods for seed and state recovery using reduced number of outputs for general purpose random number generators like MT19937, MT19937-64, LCGs Truncated LCGs, LSFRs, using SMT/SAT solvers.

Introduction

Given a PRNG algorithm A, which is initialised using an initial value aka `seed` and $x_1, x_2, ..., x_k$ be the generated outputs from the random number generator, we wish to determine the starting `seed` or the state S required to predict the future outputs of the generator.

We were able to recover `seed` of standard mersenne twister (MT19937), which is the most used PRNG across all software systems, using only **3** outputs using SMT solvers in under 5 minutes, whereas all previous work is on state recovery using *624 consecutive outputs*.

We also employed SMT solvers to recover the state of other well known PRNGs like LCG, LSFRs and combiner generators using a set of LSFRs.

We aim to understand the predictability of PRNGs and further analysed the design of some cryptographically secure PRNGs and case study of the notorious DUAL_EC_DRBG CSPRNG for presence of a kleptographic backdoor to give NSA ability to predict all outputs.

Table of Contents

Mersenne Twister

Mersenne Twister (19937)

Mersenne Twister is by far the most widely used general-purpose PRNG, which derives its name from the fact that its period is the mersenne prime $2^{19937} - 1$

It is the default PRNG in Dyalog APL, Microsoft Excel, GAUSS, GLib, GNU Multiple Precision Arithmetic Library, GNU Octave, GNU Scientific Library, gretl, IDL, Julia,CMU Common Lisp, Embeddable Common Lisp, Steel Bank Common Lisp, Maple,MATLAB, Free Pascal, PHP, Python,R,Ruby,SageMath, Scilab, Stata, SPSS, SAS, Apache Commons, standard C++ (since C++11), Mathematica. Add-on implementations are provided in many program libraries, including the Boost C++ Libraries, the CUDA Library, and the NAG Numerical Library.

Algorithmic Details

The Mersenne Twister algorithm is based on a matrix linear recurrence over a finite binary field F_2. The algorithm is a twisted generalised feedback shift register (twisted GFSR, or TGFSR) of rational normal form, with state bit reflection and tempering.

The internal state is defined by a sequence of $n = 624$, 32-bit registers (w)

$$x_{k+n} \rightarrow x_{k+m} \oplus ((x_k^u \| x_{k+1}^l)A)$$

To compensate for reduced dimensionality of equidistribution, the state is cascaded with tampering transform (to improve the equidistribution) to produce the output

$$y \rightarrow x \oplus ((x >> u)\&d)$$
$$y \rightarrow y \oplus ((y << s)\&b)$$
$$y \rightarrow y \oplus ((y << t)\&c)$$
$$z \rightarrow y \oplus (y >> l)$$

The computed z is returned by the algorithm where the choice of constants is as follows

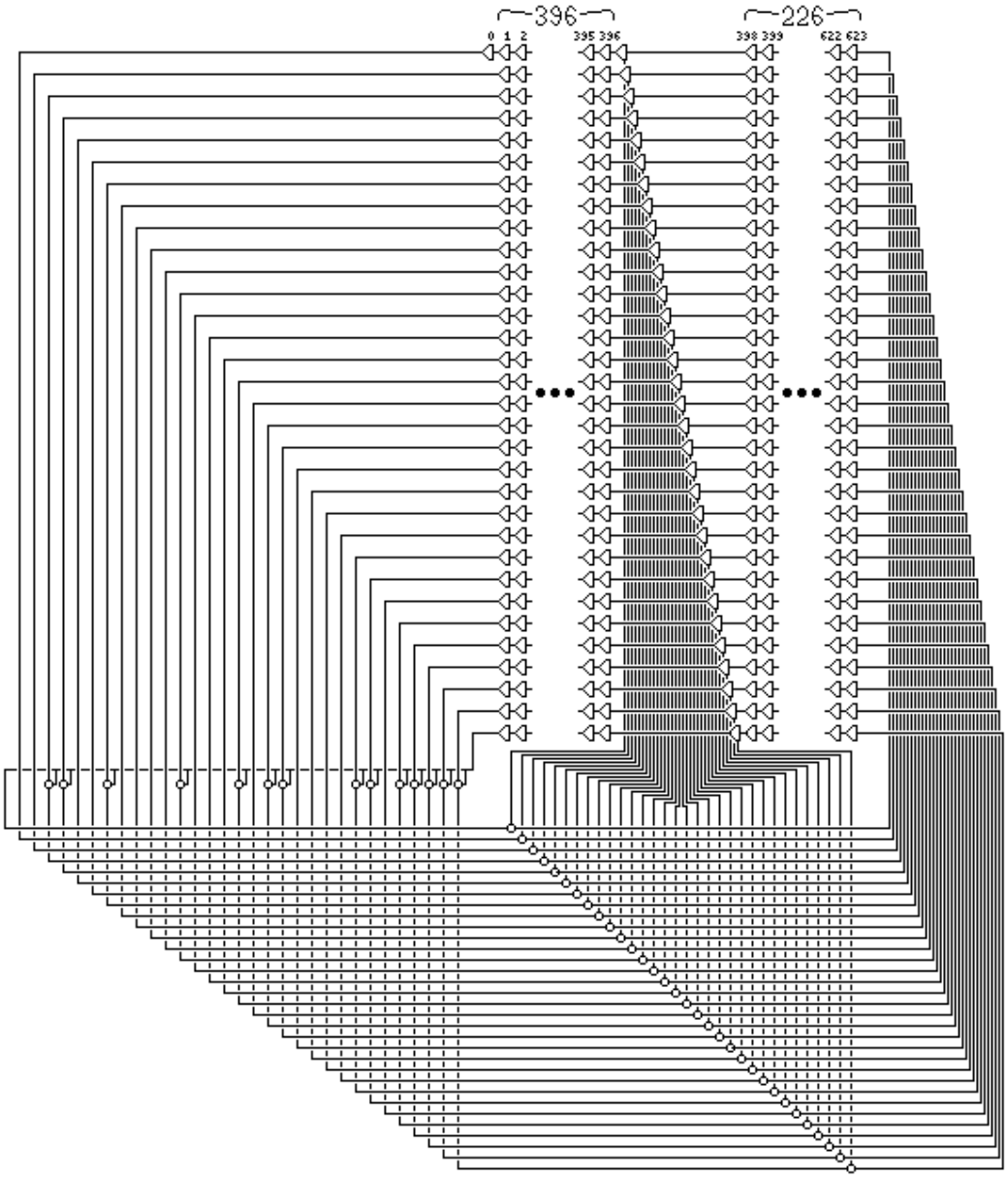
```
(w, n, m, r) = (32, 624, 397, 31)
a = 0x9908B0DF
(u, d) = (11, 0xFFFFFFFF)
(s, b) = (7, 0x9D2C5680)
(t, c) = (15, 0xEFC60000)
l = 18
f = 1812433253
```

Initialization

The state needed for a Mersenne Twister implementation is an array of n values of w bits each. To initialize the array, a w-bit seed value is used to supply x_0 through x_{n-1} by setting x_0 to the seed value and thereafter setting

$$x_i = f \times (x_{i-1} \oplus (x_{i-1} >> (w-2))) + i$$

for i from 1 to n-1. The first value the algorithm then generates is based on x_n



While implementing, we need to consider only three things 1. State initialization i.e. seeding 2. The `twist` operation to produce next 624 state registers by "twisting" the current state of 624 registers 3. The `tamper` operation to tamper a state register to the produced 32-bit output

Background

There exist various conference talks for mersenne twister seed and state recovery for the aid of pentesters at various security conferences e.g - untwister presented at B-Sides Las Vegas 2014, which recovers upto 32 bit seeds by a parallalized bruteforce using a pool of workers or

state recovery using 624 consecutive outputs (will be discussed soon).
- PRNG Cracker which in addition to parallalized seed bruteforcing, creates a rainbow table of outputs for lookup in seed database. - PHP mt_rand predictor achieves seed recover using two outputs which are 227 apart of each other exploiting the improper implementation of mersenne twister in PHP in particular. This works only for PHP as it doesnt use the standard MT algorithm.

State recovery from 624 consecutive outputs

The mersenne twister keeps a state of 624 registers `MT` and an index `i` to track the position in the state. Once `i` reaches the end of state array, the `twist` operation is called to twist the state to next 624 numbers in the sequence and `i` is set to 0. The output y_i is generated using the `tamper` function on the state `MT[i]`. This tamper function is completely reversible, hence given y_i we can recover `MT[i]`. Once we recover any 624 state registers, we can set $i = 0$ from there and predict any future outputs.

Untamper

Each of the step of instructions in `tamper` is reversible since it is simple xor of a register and right or left shifted select bits of it. Merely tracking which bits were xored with which bits of the input register to get the next value, we can undo the operation. Since in xoring with right shifting, the MSB of y would be MSB of x, and in xoring with left shifting, the LSB of y will be LSB of x.

```
y = undo_right_shift_xor_and(z, l)
y = undo_left_shift_xor_and(y, t, c)
y = undo_left_shift_xor_and(y, s, b)
x = undo_right_shift_xor_and(y, u, d)
```

Our work

We began with the implementation of standard MT19937 from algorithm described on Wikipedia. This involved a lot of debugging and testing against various random number library implementations, reading the source code of the MT implementations in Python, Ruby, PHP. And figuring out how each of these vary from the standard implementation on wiki.

Modelling

We modelled the seed recovery algorithm as a SMT problem using the SMT solver Z3Prover, as a sequential program written in theory of BitVectors(32) (since the algorithm is designed to work on 32bit architectures) and theory of BitVectors(64) for MT19937-64 . After (painfully) modelling the program, we begin a SAT solver search (all-SAT to give all satisfying models for possible seed values) which leads us to a given sequence of outputs (the generated random numbers).

Results

We were able to recover the seed of the mersenne twister for both MT19937 and MT19937-64 using any **3** consecutive outputs, in about ~200 seconds.

The modelling of `untwist` can reverse the `twist` operation to go back 624 outputs, which cannot be done easily by any of usual methods thus enabling us to predict unseen outputs which were produced before the observed set of outputs.

Our method is extremely memory and space efficient since SAT solvers work with negligible memory (<500 MB). And way faster and efficient considering the space time tradeoff involved.

The same methodology is applicable and extendible to various other cases where it might not be possible at all to come up with an angle of attack. For example. - Outputs of non-32 bit sizes, say `random.gerandbits(31)` is called - One of the most used methods from random library is usually `rand` which generates a random float in 0 to 1 (which internally makes two MT calls and throws away 11 bits to generate the random number) - `random.randrange(1,n)` is called which may internally make use of multiple MT calls.

All of the various methods from random libraries can be used to recover the state/seed whereas all other approaches merely work if only we have 624 consecutive `random.getrandbits(32)` calls which is quite rare to observe in a real life application.

Challenges

While the wikipedia implementation is treated as the standard mersenne twister, we found that our implementation was producing different outputs from the implementations in programming languages even after initializing by the same seed. After dissecting a lot of source code, we figured out that the wiki implementation serves as the base of merssene twister implementation everywhere with a difference in the way it is seeded. All modern mersenne twister are seeded with a function `init_by_array` which takes an array of 32-bit seeds (or 32 bit hashes of seed objects). Later we found that this was proposed as an enhancement to equidistribution property to the base mersenne twister MT2002.
This `init_by_array` is much more non-linear that the standard `seed_mt` operation and makes it much more difficult to recover the seed values from a few outputs. We tried following the same approach, and it turns out it was unable to deduce the seed even in a couple of hours.

Yet another major challenge was to understand the exact API and studying what exactly to use.
e.g `BitVec` class of z3 assumes the bitvector to be signed. Consequently, we will need to define the `>>` the shift right operator as either logical shift right (`z3.LShR`) or arithmetic shift right (which is `>>` the `__rshift__` magic method in `BitVec` class). Mistakably using `>>` to shift right logically, costed us a lot of debugging time and a lot of `unsat` examples, which were quite hard to figure out due to the unsuspecting sign of things.

Limitations

The most basic limitation of seed/state recovery using SMT/SAT solvers is figuring out the time complexity and approximate time required to find a satisfying assignment of our choice. While researching, it is almost like running through a jungle as we never know how much time it would take us to find a way.
The time required to find a satisfying assignment is also very dependent on the way the problem was encoded. Seemingly similar modelling/encoding can end up in drastically different run times as the solver may end up using completely different heuristics and paths

to find a goal. So the major drawback is finding out the best way or a good enough way to recover the seeds of a given PRNG algorithm.

Other drawback of our approach is that SMT solvers operate in the realm of first order logic. If a given algorithm can not be encoded in FOL, SMT solver wont be able to find a satisfying assignment (though all the work we did can be translated easily under FOL).

Another drawback can be when there are more than one possible seed/state to produce a given set of outputs, SAT solvers are designed and optimized to find a single satisfying assignment, finding successive assignments, may or may not translate equivalently.

Yet another drawback is lack of parallelism. The current design of SAT/SMT solvers is massively single threaded and may not use the full capabilities and cores of the machine to find a satisfying assignment.

References

- The Mersenne Twister
- Wikipedia

LFSR

Background

Previous work done in this topic

Our Work

Exactly what we did (setup up something, installed something, ran code from github etc. whatever small things we did for the project)

Challenges / Difficulties

Limitations / Assumptions

Compatibility issues

Could not run a particular version of software or a software didn't worked in a particular os

Understanding existing papers

sir ka example -> this particular paper was very cryptic and there were a lot of notions and after enormous discussion it took us 1 week to understand

Reading manual

Critique / Comparison

Critique an idea or paper or an attack senario Comparison with others work or idea

Extensions

LCG

Background

Previous work done in this topic

Our Work

Exactly what we did (setup up something, installed something, ran code from github etc. whatever small things we did for the project)

Challenges / Difficulties

Limitations / Assumptions

Compatibility issues

Could not run a particular version of software or a software didn't worked in a particular os

Understanding existing papers

sir ka example -> this particular paper was very cryptic and there were a lot of notions and after enormous discussion it took us 1 week to understand

Reading manual

Critique / Comparison

Critique an idea or paper or an attack senario Comparison with others work or idea

Extensions

Dual_EC_DRBG

Background

Previous work done in this topic

Our Work

Exactly what we did (setup up something, installed something, ran code from github etc. whatever small things we did for the project)

Challenges / Difficulties

Limitations / Assumptions

Compatibility issues

Could not run a particular version of software or a software didn't worked in a particular os

Understanding existing papers

sir ka example -> this particular paper was very cryptic and there were a lot of notions and after enormous discussion it took us 1 week to understand

Reading manual

Critique / Comparison

Critique an idea or paper or an attack senario Comparison with others work or idea

Extensions

Team members

1. Himanshu Sheoran 170050105
2. Lakshya Kumar 170050033
3. Sahil Jain 180050089
4. Yash Ajitbhai Parmar 170050004