CISC 601

Final project report

Zhentian Zhang

HUID 269972

I chose the paper A lightweight Pseudo random Number Generator for securing the Internet of Things.

# Introduction

I took a cryptography course during my undergraduate study and I was obsessed with one key sub-domain study – random number generator. For sure, RNG contributes more areas other than cryptography, but modern cryptography algorithm probably has the biggest needs for better RNG schema (in my view). There are generally two types of RNG: true random number generator (TRNG) and pseudo random number generator (PRNG). TRNG makes use of a random source that comes from the nature: lightening, hardware noises and etc., it usually just magnifies the source and expose the numbers as a service. Sometimes it’s not as good as PRNG! Because the results sometimes are not distributed as evenly as desired. I have to mention RNG tests. There are many types of tests to evaluate how good is an RNG algorithm. The most famous test or the most commonly known one is the threshold test: it basically checks if the sample results from one RNG algorithm can yield a given number of pass rate, so that it can be productized as a service. By the end of this paper research, the author claimed this lightweight PRNG algorithm can pass the LWC standard. Anyway, PRNG, on the other hand, does not make use the sources from the nature, but leverages the beauty of mathematics. More specifically, the algorithms find ways to permute / mash up numbers and usually, their results are more evenly distributed. The cost is occupying more hardware and software resources! They can be very complex and very costly as of time and space complexity. Thus, it’s hard to use PRNG on resource-contained devices. But with the rise of IoT and small smart devices and at the same time, in many situations, they are looking for high security standard. So, they have to pick PRNG by nature, and we see more researches on how to create better performance and lightweight PRNG algorithms.

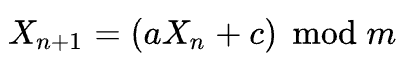
This paper proposed one, using two lagged Fibonacci generators (LFGs), permute the numbers, sum them up before mod m. It can be easy to reproduce, since the purpose of this PRNG is to be implemented on resource-constrained devices.

# Context

In order to understand what problem this paper is trying to solve; we need to understand what foundations this paper is based on.

## LCG

As we all know, LCG (linear congruential generator) is the most common pseudo-randomized number generator template. If I recall correctly, basically every famous language’s default random number generator function is based on this method. It’s very simple that it generates a series of numbers and the system will randomly pick one as the result to return. a, c and m are given parameters, and we need to seed x1 in order to start the calculation.

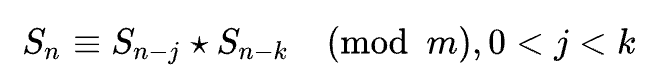


One thing to note that, we need to carefully choose a, c and m, so that we can maximize the security and performance of the random number.

## LFG

LCG is easy to identify and break, so it’s not used in cryptography. Instead, LFG (lagged Fibonacci generator) will be used.

The calculation template is as follow:



It also generates a series of numbers and the system will randomly pick one to return as the result. Following the nature of Fibonacci numbers, the later number is calculated by two previous numbers, star is a series of calculation instead of one step.

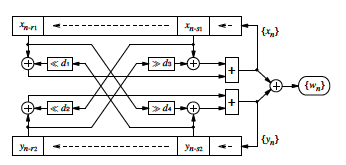
There are many optimizations to LFG, one is that, in order to save memory, we only need to remember k last numbers. But the drawback is that the theory to LFG is very complex, so that we cannot really be smart enough to come up with the best ways to pick the number j, k and m.

Although it’s better than LCG, but if the parameters are not carefully chosen, the generator can easily fail PRNG threshold testing. For example, the most failed one would be birthday spacing and generalized triple testing.

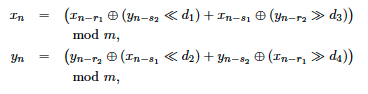
## Enhanced LFG: Arrow

This paper claimed, although LFG is performing good at resource-limited devices and is able to be used in cryptography context, it will easily fail to pass birthday spacing test of the Marsaglia’s diehard test suite. Furthermore, a good analysis of the passed numbers generated by LFG allows to predict the following numbers. Last, in order to ensure the security, more numbers are generated before the pick will be done, so the lag of the time will be the drawback of the method.

In order to avoid these problems, this paper proposed an enhanced version, which makes use of two simple lagged Fibonacci generators, and it’s called Arrow. It looks like this



The detailed calculation steps to generate xn and yn is as follow:



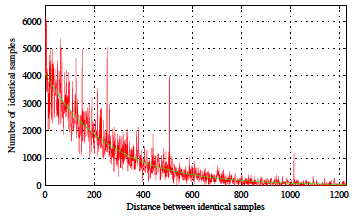
Where 4 ds are constants but will be smaller than the bits of the input, so that the output won’t be 0 after the calculation.

# Performance review

It’s still not clear, what’s the best combination of the constants will maximize the performance and security of Arrow, but this paper states some general rules:

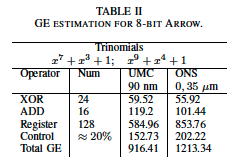
Moreover, the lags r1; r2; s1; s2 must be selected to satisfy that both trinomials xr1 + xs1 + 1 and yr2 + ys2 + 1 are irreducible and primitive over GF(2).

In order to examine the performance of Arrow, we need to understand period. Every Pseudo-random number generator will have a cycle that the same number series will present in all cycles, and thus we call the cycle as period of PRNG, and we usually want to maximize the period length, so that this generator is more secure. So when the period length is maximal, the linear complexity of any level of significance is equal to half of the period.

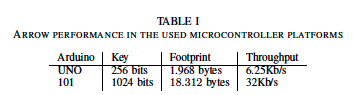


The paper tested Arrow on two IoT platform: Arduino UNO and Arduino 1010. They basically have around low-power 8-bit microcontroller, with 32kB memory, 2kB RAM and 16 MHz clock frequency.

The estimated hardware complexity of Arrow using a word size of 8 bits and 16bits, for these 2 microcontrollers are as follow, and it shows it’s very friendly to low-power computers (because it costs only around 1000 GE.



So, the paper claimed Arrow fulfils the general security requirements of the IoT and the test result is as follow:



# Conclusion

Arrow is fast-running, less resource-consuming, cryptographically secure PRNG, which is useful for resource-constrained devices, such as IoTs. It’s based on two mutually scrambled LFG and the paper proposed the technics to use the proper parameters so that the performance and security of the generator can be optimized. But works are needed to provide more theories on how to choose the best suite of parameters. The algorithm is simple, because only fast operations are used: addition, bitwise XOR, right and left shift. Thus, the estimated hardware complexity is low.

# Reproduction

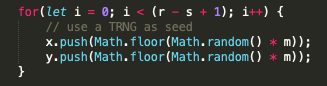
## Seed

Most PRNGs are built on algorithms involving some kind of recursive method starting from a base value that is determined by an input called the “seed”. So, given a set seed and an algorithm, the sequence of random number will be the same and it is because of the nature of pseudo-random number generating algorithms.

Random seeds can be deliberately chosen, but it will greatly affect the generator’s performance, especially in the field of computer security. When a secret encryption key is pseudorandomly generated, having the seed will allow one to obtain the key.

Random seeds can be given or be generated from the state of the computer system (such as the time).

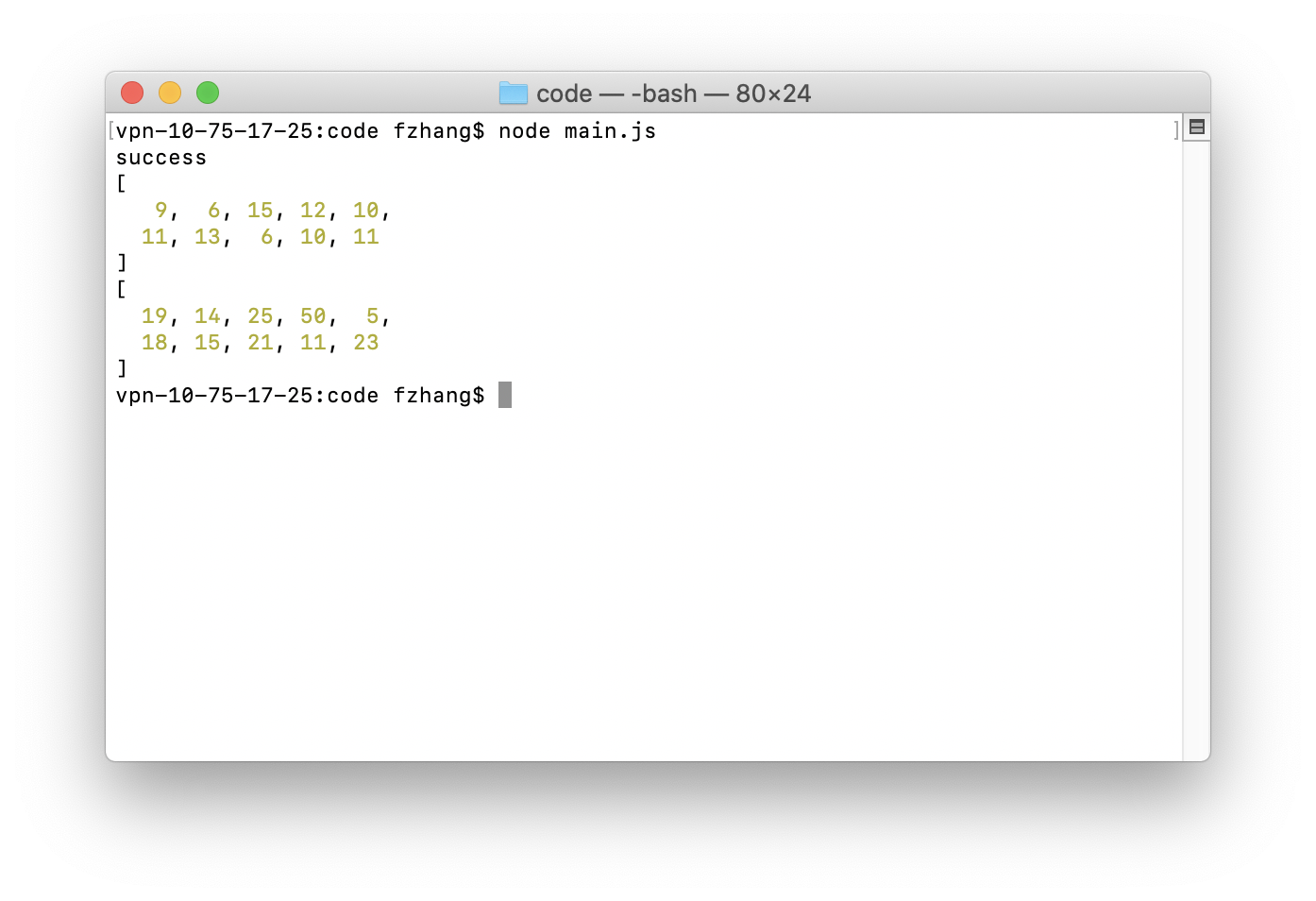
The paper did not mention what’s the best way to pick the seed, so I’m simplifying it by using the system provided random function.



## Result

In order to test Arrow class that I wrote, in the main.js file, I ran the code without providing a base, with a base and I wrote a threshold test.

The test result is as follow:



The success on the first line means the Arrow PRNG passed the test,

And the following 2 arrays are random numbers generated by Arrow,

The 1st array is when not providing the base, and the default base will be 4, that means, the generated number will be in the range of [0, 2 ^ 4), so [0, 16). We can tell from the result, the 10 numbers generated are random, in the range and evenly distributed.

The 2nd array is when I provide the base 6, so the numbers are the in the range of [0, 2 ^ 6), so [0, 64). We can easily tell the result set, all 10 numbers are also random, in the range and evenly distributed.

# Reference

* <https://stats.stackexchange.com/questions/354373/what-exactly-is-a-seed-in-a-random-number-generator>
* https://en.wikipedia.org/wiki/Random\_seed