A Comparative Study on Pseudo Random Number Generators in IoT devices

Efe Alkan
Daily Supervisor: Miray Aysen
Responsible Professor: Dr. Zekeriya Erkin
Cyber Security Group, Department of Intelligent Systems
Delft University of Technology

Abstract

Pseudo-random number generators are the essential part of many security protocols such as signature schemes, key-exchange protocols and encryption algorithms. The security of these protocols is usually dependent on the quality of the generators they use. The generation of unpredictable random numbers supplies refreshment to the protocols, which makes them harder to break. In this paper, an extensive comparative study made on some of the well known and some interesting pseudo-random generators. These generators consist of xorshift, xorshiftStar, PCG, CMWC and Fortuna. They are tested according to some criteria, which consists of efficiency, security and statistical randomness quality. Possible IoT usages are suggested for each generator according to their results. In the end, for the xorshift and xorshiftStar generators, improvements are suggested to increase their statistical quality and security.

1 Introduction

Internet is one of the most fastest-growing technology with more than 3 billion increase in usage in the past twenty years [1]. With internet usage, many life-impacting technologies such as messaging systems, e-commerce, web technologies, cloud systems exist. However, these technologies cannot be used without a device. We can combine all the devices that can connect to the internet under a paradigm called the Internet of Things (IoT). This paradigm refers to the connection of physical devices with virtual networks. IoT devices connect to the internet using wireless networks that generate massive amounts of data using their sensors/actuators and shares data with other devices with or without human intervention [2].

In the last decade, the number of IoT devices has increased significantly. These devices are essential as they are used as an integral part of almost every field, ranging from healthcare to the manufacturing industry to smart homes to wearable technology [3]. Nowadays, even the most essential appliances such as coffee machines or microwaves can connect to the internet and controlled by smartphones.

Unfortunately, every IoT device is vulnerable to cyber-attacks. Data gathered by IoT devices might be considered sensitive information, including personal and private information such as the camera view of someone's home, personal messages, log consisting of date and time of someone being at home and many more. DoS (Denial of Service), MITM (Man-In-The-Middle), and modification-of-message [3] are some examples of the attacks which can thread someone's social/personal life or

security. Thus, security protocols such as authentication protocols, signature schemes and encryption algorithms are used in IoT devices to protect the devices against cyberattacks. These security protocols all have one common property, that is, they all use Random Number Generators (RNG).

RNG's are an integral part of the security protocols because they should provide random and unpredictable sequences of bits to be used as initialization vectors, keys and nonces for these protocols. RNG's refresh the algorithms by changing the inputs of the algorithm every time the algorithm is used. RNG's are essential for obtaining high levels of security because if an attacker breaks the RNG for some protocol, then the attacker can bypass the protocol by guessing the states and keys in most cases.

There are two types of Random Number Generators; True Random Number Generators (TRNG) and Pseudo-Random Number Generators (PRNG) [4]. TRNG's are defined to be non-deterministic generators that are dependent on the hardware. Properties like nuclear decay, photons, thermal noise and atmospheric noise influence the true randomness. However, in practice, generating true randomness with computers is not possible. Thus, in computer software, PRNG's are used to simulate TRNG's by generating pseudo-random numbers [4]. PRNG's are defined as deterministic, meaning that if a computer can generate a number, another computer can generate the same number with the same inputs. However, lack of true randomness does not influence security since pseudo-randomness with great unpredictability is usually enough.

In the IoT world, the biggest challenge is efficiency. Algorithms used in IoT have to work efficiently due to IoT device's low computational power and memory. Thus, all algorithms used in IoT devices, including PRNG's have to be efficient as possible to decrease the load on the device and the memory consumption.

In this paper, PRNG's are compared in a detailed way to suggest a suitable usage for these in IoT devices. Four PRNG's are chosen, consisting of Complementary-Multiply-With-Carry (CMWC), Permuted Congruential Generator (PCG), xorshift and xorshiftStar. Although there are many studies on the xorshift family, there are very few in-depth comparative studies that include most of these selected PRNG's.

Later in this paper, we do investigate the usage of lightweight block ciphers as PRNG's to possibly ensure more unpredictability compared to the traditional PRNG's studied in this paper. The lightweight block cipher used as a PRNG that is discussed in this paper is called Fortuna. There are some other examples of lightweight block cipher usage as a PRNG, such as the Yarrow algorithm, AES-CTR. However, Fortuna is chosen for this study because it is one of the most recent and known ones. This paper aims to be a comparative study of these PRNG. The comparison criteria consist of the following; security, empirical statistical randomness, time and space efficiency.

The structure of the paper is the following; section 2 gives brief background information about the studied PRNG's. The main properties and structures of the studied PRNG's are described to serve as preliminary knowledge. The section 3 is about the responsible research where the ethical implications of this research are discussed. The following section is about the methodology and the results of the experiment. This section describes the experiment and the comparison criteria. In this section, results and the observations are presented as well. Then in section 5, results presented in the previous chapters are discussed to derive conclusions for the studied PRNG's. Section 6 gives future recommendations about the possible improvements related to the studied PRNG's. Last but not least, in the section 7, a conclusion is given, which summarises the results obtained.

2 Background Information

In this section, brief information about the structure and the properties of the PRNG's, xorshift(star), CMWC, PCG and Fortuna, are given.

2.1 Xorshift

Xorshift is a pseudo-random number generator that George Marsaglia designed in 2003. It is similar to linear-shift-feedback registers (LSFR) [5] as it only uses xor (exclusive or) and shift operations without requiring complex mathematical operations. Since xor and shift operations are fast to perform in computers, xorshift can generate a 32-bit random number super fast and efficiently. As can be seen in Figure 5 in the appendix, Xorshift generates the initial number by first seeding a 32-bit arbitrary value to the state and then taking the xor of the state and the shifted-bit version of itself three times. (Note that, "$\wedge$" is the xor operation, "$\ll$" is the left shift and "$\gg$" is the right shift). The number of shifts (ex. 13, 17, 5) used in xorshift is suggested by the Marsaglia, but the implementer can change them. The result of xor and shift operations becomes the generated random number. Finally, xorshift overrides the value of the state with the generated number. The algorithm uses the updated version of the state to generate future random numbers.

2.2 XorshiftStar

There are multiple improved versions of xorshift where they all aim to add some non-linearity; xorshiftstar is one of them. XorshiftStar uses the xorshift with an additional non-linear multiplicative operation. As can be seen from Figure 6 in the appendix, in xorshiftStar, after the algorithm overrides the value of the state, it applies an arbitrary multiplication to the generated number. Only after this operation, the algorithm returns the generated number. This way, it adds confusion to the state. There are multiple versions of xorshiftStar, such as xorshiftStar64, xorshiftStar128, xorshiftStar1024, where the numbers represent the bit-length.

As a side note, improved versions of xorshift does not mean xorshift cannot be used anymore. The usage of these generators all depends on the application's purposes. If the security has no priority, then people can still use xorshift to generate a random number. More about this is discussed in section 5.

2.3 Permutated Congruential Generator

Linear-Congruential-Generators (LCG) [6] are not cryptographically secure and easy to predict due to their linear structure. Permutated Congruential Generator (PCG) is a pseudo-random number generator based on the classical LCG. It was developed in 2014 by M.E O'Neill with an aim to improve the statistical properties of the classical LCG.

Figure 7 in the appendix shows the structure of the PCG. First, the state is initialized in the Initialization step by summing the random seed with an arbitrary increment value, as can be seen from the inner structure of the Initialization step from Figure 8 in the appendix. PCG consists of two main parts; one called the state transition function, and the other called the output function [7]. PCG uses the classical LCG in its state transition function to generate a number. In the transition function, first, the state for the next round gets calculated, then the number is generated from XORing and shifting the old state (inner structure of transition function from Figure 8). Then the rotation value is generated, which is used in its output function (inner structure of output function from Figure 8). The non-linearity comes in with its output function where the generated number is

distorted by using the rotation value, bit-shift, AND and OR operations.

There are multiple types of PCG; XSH-RR, XSL-RR, RXS-M-XS [8]. The most significant difference between these types is the bit lengths of the seed value and the output value. XSH-RR starts with a 64-bit seed value and outputs a 32-bit number, XSL-RR starts with a 128-bit seed value and outputs a 64-bit number, and RXS-M-XS starts with a 32-bit (or 64-bit) seed value and outputs a 32-bit (or 64-bit) number. The type that is considered in this paper is XSH-RR.

2.4 (Complementary) Multiply-with-Carry

Multiply-with-Carry (MWC) is a pseudo-random number generator that George Marsaglia discovered in 1994. Marsaglia even mentioned MWC as the mother of all PRNG due to its simplicity and efficiency. Later in the 1997's, R. Couture and P. L'Ecuyer improved the MWC and came up with the Complementary Multiply-with-Carry generator.

It consists of two parts; initialization and the generator. In the initialization part, like all other PRNG's, arbitrary seed values are initialized. However, the number of the seeds is more than one and generally equals the cycle length, which is suggested as 4096 by the designer [9]. Also, the numbers "a" and "b" must be chosen before starting the generation process. All of the initially chosen values must not be consists of zero's.

The idea of the generator comes with the following equation;

$$x_n = (b-1) - (a \cdot x_{n-r} + c_{n-1}) \mod b,\ c_n = \lfloor \frac{a \cdot x_{n-r} + c_{n-1}}{b} \rfloor,$$

where "r" is the seed that is being used, "b" is the base, and "a" is the multiplication value [9]. "x_n" is the output of the generator where "n" represents the index of the generated number.

2.5 Fortuna

Fortuna is a pseudo-random number generator developed by Bruce Schneier and Niels Ferguson in 2003. It is an improved alternative to the Yarrow algorithm [10]. Figure 1 shows the structure of Fortuna, which consists of two main parts; generator and entropy accumulator [11].

As a generator, it uses a secure block cipher such as AES [12], but choosing the block cipher is open to users. The block cipher works in the counter mode, which uses block cipher as a stream cipher. In counter mode, the block cipher generates the keystream in a blockwise fashion. The counter is a function, and the algorithm uses it as an input taking a different value every time the block cipher generates a new keystream. The algorithm regenerates the key after every usage to prevent exposing the older outputs.

Fortuna uses an entropy accumulator to collect data from different sources such as computer-dependent events, time, mouse movement and more. It stores the data in its pools, waiting to (re)seed the generator. In Fortuna's original design, there are 32 pools, but the implementer can alter this number. The algorithm uses the pools with lower id's more frequently, while the pools with higher id collect random data. Once there are enough random data in the pools, it can seed the generator and generate a random number.

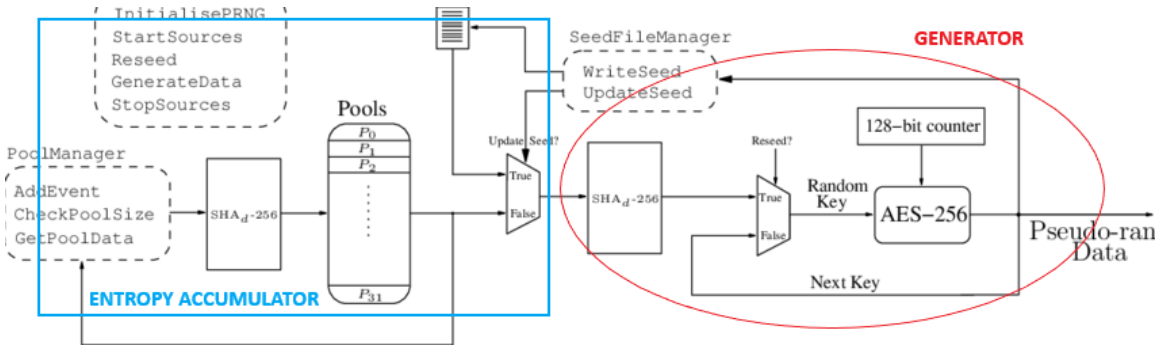

Figure 1: Structure of the Fortuna generator using AES [13].

3 Responsible Research

This research aims to compare certain PRNG comparatively. This is done by first literature survey, and then implementation and testing the PRNG according to some criteria mentioned before. The main possible ethical problems with this paper could be biased analysis due to mistakes in implementations or picking one PRNG and representing it in a better way than it is. To prevent these kinds of possible issues, all implementations are implemented using multiple references to reduce possible misunderstandings and mistakes. The tests have been performed using a well-respected test suite correctly by first understanding their introductive examples. The results of the CPU time measures are the mean of hundred trials to be more precise. None of the PRNG's except the improved xorshift family is done by us, so we tried to be concise and unbiased as much as we can. We analyzed the PRNG by using the results of the tests and the information found by literature surveys to minimize the bias towards a certain PRNG. All of the test results are presented throughout the paper.

4 Results and Methodology

In the results section, the results of the tests and the observations made about the studied PRNG's are presented to be later discussed in section 5.

In this experiment, the studied PRNG algorithms are implemented in the C programming language. An empirical randomness test suite called TestU01 [14] is used to test the implemented PRNG's. We chose to use TestU01's Big Crush test suite because it is one of the most well-respected test suites in the industry for testing the randomness of the generators. Big Crush test contains 106 different tests, and for some tests, it uses different parameters, making the total test count 160. For every test, it calculates a p-value. For the test to be a pass, its p-value needs to be in the range which is [0.001, 0.999] by default. One drawback of this test suite is that it only accepts 32-bit inputs meaning that generators producing numbers more than 32-bits cannot be used directly [18]. Thus, if the generated number has more than 32-bits, then the generated number needs to be split into pieces, and these pieces need to be tested individually. Out of all the studied PRNG's in this experiment, only the xorshiftStar generates a 64-bit number. Therefore, the 64-bit output is split into lower and upper bits, tested individually and then combined.

We measured the CPU time it took to generate a hundred million numbers in hundred trials and took the mean of these trials. We also included the file sizes of the implemented PRNG in the

results section. Even though the file sizes are implementation-dependent, they still give average comparative results.

All of the tests have been performed on Ubuntu virtual machine with a 4GB ram in windows based Hp-laptop with an i7 processor, 16GB ram and GTX 1050 graphics card. In order to run the Big Crush test suite, TestU01's source package needs to be downloaded from their website [15].

Table 1: Big crush test results, showing the number of test failures and the names of the systematic failures. Higher number means it's randomness statistics is worse.

Big Crush Test Results				
Name of the PRNG	Number of test fails for higher bits	Number of test fails for lower bits	Total number of test fails	Systematic Test Failures
Xorshift-32	(59/160)	-	(59/160)	MatrixRank, LinearComp, Permutation, ClosePairs, Fourier, CollisionOver, SerialOver, Gap, MaxOft
Xorshiftstar-64	(1/160)	(4/160)	(5/160)	MatrixRank, LinearComp, BirthdaySpacings
PCG	(1/160)	-	(1/160)	PeriodsInStrings
CMWC	(0/160)	-	(0/160)	-
Fortuna	-	-	-	-

Table 2: Shows the time the generators take to generate hundred million numbers and the total file sizes of the algorithms.

Comparison Table		
Name of the PRNG	CPU time to generate 10^8 numbers (in sec)	Total file size (in bytes)
Xorshift-32	1.93	563
Xorshiftstar-64	1.95	897
PCG	1.96	1200
CMWC	2.11	2300
Fortuna	-	-

5 Discussion

In this section the results that are presented in section 4 are discussed to find possible vulnerabilities and then suggest possible usages of the studied PRNG's in IoT device usage.

Secure generators and cryptographically secure generators (CSPRNG) do not have the same meaning. A secure generator means a non-cryptographically secure generator (NCSPRNG) that performs great on statistical randomness tests. The outputs of some NCSPRNG might represent true randomness with possibly having a somewhat high unpredictability with no or minimal patterns between the numbers. However, this does not mean they are not entirely unpredictable. For an average user who has very little or no knowledge of the security topics, the outputs of NCSPRNG can look genuinely random and unpredictable. However, an experienced attacker might break NCSPRNG and predict the bits to be generated next. Some NCSPRNG's are easier to break, while other NCSPRNG's can be more challenging to break. The difficulty of breaking NCSPRNG depends on how they are designed structurally (number of periods, non-linearity, state). All NCSPRNG are designed to be used for different purposes, so it is essential to consider their usage purposes.

In applications where security is essential, it is crucial to use CSPRNG instead of NCSPRNG [17]. Generally, CSPRNG's are tough to break unless the attacker learns the inner state of the generator. Most CSPRNG's use a secure encryption algorithm as their generators that follow block or stream cipher structure. These encryption algorithms are usually combined with different kind of entropy sources. Thus, when analyzing the studied PRNG's, it is vital to cover both their security levels and the efficiency in order to suggest proper usage in IoT devices.

5.1 Xorshift

The xorshift fails 59 test cases of Big Crush, as can be seen in Table 1. It is the worst one performing out of all the studied PRNG's. It fails all the cases (with different parameters) of MatrixRank, CollisionOver, SerialOver, ClosestPairs and LinearComp tests. We can call these failures systematic failures because their p-values are less than 10^{-10}, which are way out of the expected range; [0.001, 0.999]. These high number of systematic failures indicate a severe problem with the randomness of the xorshift generator. For example, the SerialOver test divides the interval [0, 1) into d pieces and uses the PRNG to generate n vectors with overlaps [14]. Then it maps these vectors to the divided pieces to compare the number of corresponding vectors. Failure of SerialOver indicates that some of the generated numbers by the xorshift generator are closer to each other than the rest. This means users may find patterns between the generated numbers which can lead to high predictability.

Xorshift is not cryptographically secure due to its design. The next and previous numbers can easily be predicted due to their linear structure. As described in section 2, it contains three consecutive xor and shift operations without any non-linear operation. This makes the xorshift generator easily guessable.

According to Table 2, the xorshift generator is the fastest in terms of CPU time to generate a number out of all the studied PRNG's. Its file size is 563 bytes which are the smallest compared to the other studied PRNG's. The reason for xorshift to be very fast and very small in code size is because of its design; it is designed to be super-fast [18]. Computers can perform the operations xorshift uses, meaning bit-wise operations, even faster than arithmetic operations.

Even though xorshift generators are very speed and time-efficient, they should not be used in any IoT application due to their security concerns. There are some improved versions of xorshift; one is

called xorshiftStar, which is discussed next.

5.2 XorshiftStar

XorshiftStar is the improved version of the xorshift with the additional multiplicative operation as described in section 2. Since the version of the XorshiftStar used in this experiment produces a 64-bit output, we had to divide the output as higher and lower bits. When we run the Big Crush tests on the higher bits, it fails only 1 test case called BirthdaySpacings as can be seen in Table 1. However, when we run the lower bits, 4 test cases of the Big Crush fail. These failures indicate, its lower bits are more vulnerable and predictable. These failed tests are called the MatrixRank and LinearComp tests. When we combine the results, it fails a total of 5 tests of the Big Crush. These failures are systematic failures since most of the p values of the failed tests are less than 10^{-300}, which are immensely out of the expected range. However, we can see that a simple arbitrary multiplicative operation adds a massive improvement to the xorshiftStar generator as it fails five tests, whereas the original xorshift generator fails 59 tests. With this operation, xorshiftStar can hide its state from the users.

According to the statistical tests, the xorshiftStar generator performs worse than PCG and CMWC. According to the Table 2, the xorshiftStar generator is the second-fastest in generating a number and the most size efficient after the xorshift. The slight difference between these two generators is that they are mainly similar except for a multiplicative operation and the generated output bit length. The reason for xorshiftStar to be faster than the rest of the studied PRNG's is that it has a simpler structure compared to the rest. The other studied PRNG's are designed to require more complex operations instead of having just three shit, xor and a multiplicative operation.

Although xorshiftStar performs not bad in statistical tests and contains a non-linear operation, it is still considered as an NCSPRNG because it is designed to provide high-quality and fast number generation [18] instead of providing cryptographic security. Thus, xorshiftStar should be used in IoT applications where security is not the main priority, but speed and memory efficiency is essential. Any IoT application that contains vulnerable or personal private data should not have the xorshiftStar generator as their PRNG. The applications for xorshiftStar could be smart lighting systems (LED animation shows), personal robotics projects, smart appliances such as smart coffee machines, smart fridges and such.

5.3 PCG

PCG is the most recent generator out of all the studied PRNG's. It has a more complex design than the xorshift family and CMWC, using two primary functions as described in section 2. The version we used in this study is XSH-RR, which is initialized with 64-bit numbers and produces 32-bit outputs. For this reason, we do not need to split the output into lower and higher bits.

PCG performs excellent on the Big Crush test, failing only 1 test called PeriodsInStrings as can be seen in Table 1. The p-value of the failing test is $3.1 * 10^{-4}$, which is close to the expected range [0.001, 0.999]. Since the p-value of the test is too close to the expected range, we run the Big Crush for PCG five times, and out of all the three rounds, PeriodsInStrings fail with a similar p-value while it does not fail in the other two runs. It is not possible to detect the reason why sometimes it fails and sometimes not, but we believe that this failure difference is happening because of either two reasons; one is due to the initial seed value, which is implemented as dependent to the time, and the other one is that the Big Crush shows some bias or false-positives towards some results or seeds for the PCG. Thus, it is not right to call this failure directly a systematic failure, but we still

keep this test as a fail since it fails half the time. PCG performs better than the xorshift family and slightly worse than CMWC in statistical quality.

According to the Table 2, the CPU time to generate a hundred million numbers takes 1.96 seconds, which is nearly the same as the xorshiftStar's. The file size of PCG is slightly more than the xorshiftStar, with a total size of 1200 bytes. However, if we compare the CPU time and file size of the PCG and the xorshiftStar, PCG is the winner. PCG takes 1.96 seconds with a code size of 1200 bytes, whereas xorshiftStar takes 1.95 seconds with a code size of 897. These numbers show that even though the structure of the PCG contains more operations than the xorshiftStar's, PCG is more speed efficient with larger code size. The reason for xorshiftStar being less efficient compared to the PCG could be that it becomes slowed down while running three parallel 64 bit XOR and shift operations consecutively using a single core (Although all of the implementations are tested in the same environment).

PCG generators are designed to be fast general-purpose generators that perform great on statistical tests [19]. The output function provides high non-linearity better than xorshiftStar and CMWC by revealing only some tiny part of the state. However, this does not mean that an attacker cannot crack them. According to the designer of PCG, it is very challenging to break the PCG [20]. There is not much study and information on breaking the PCG, but according to a study made in 2020, they predicted the secret information of the PCG using 2.3 CPU years of computation [20]. Thus, PCG is considered as an NCSPRNG with high toleration to attacks more than most of the other NCSPRNG. After considering the tests and the security of the PCG, we recommend that PCG can be used in all IoT applications where xorshiftStar can be used, plus in applications where the security and the efficiency needs a balance. These applications can include personal and private data that has not much importance in case of an attack. Examples of IoT applications for using the PCG could be smart home systems like home entertainment systems, interactive systems like smart public route guides, smart airport guides or machines like smart farm irrigation systems.

5.4 CMWC

CMWC is the oldest generator in this study and the best one performing in the Big Crush test with 0 failing tests, as can be seen in Figure 1. To ensure the correctness of the test result, we performed the test a couple of times, resulting in the same result. This result shows that the randomness of the generated numbers is perfect, with minimal patterns between the generated numbers.

Figure 2 shows that the CPU time it takes to generate a hundred million numbers is 2.11 seconds with a large file size of 2300 bytes. These numbers are the larges ones in Figure 2, although this generator is mentioned as a simple and efficient generator by its designer. However, if we compare the ratio of the code size and CPU time, just like we did in the PCG section, CMWC becomes the winner. PCG generates a hundred million numbers in 1.96 seconds with a code size of 1200 bytes ($1200 \div 1.96 = 612$), whereas CMWC takes 2.11 seconds with a 2300 bytes ($2300 \div 2.11 = 1090$).

One of the biggest advantages of the CMWC comes with its period. Its period length can reach up to 10^{453} efficiently, depending on the chosen "a", "b" and "r" values [21]. Every program must end up before starting to repeat itself, and the number of this is called the period. A higher period means larger amounts of non-repeating numbers, which is one of the aspects we want from a PRNG.

Although CMWC has many advantages mentioned previously, it is considered as an NCSPRNG, meaning that it is not secure for cryptographical applications. Even though it has extensive periods, an attacker can still predict the numbers by exposing its states. This is, however, not easy as

breaking the xorshift or xorshiftStar.

We can see that CMWC is similar to PCG in terms of results and security. We cannot decide which one is more secure as it requires more in-depth cryptoanalysis knowledge, but we believe they are similar to each other due to their states and inner structures. CMWC is actively being used in games. However, we can suggest that CMWC can be used in applications where PCG can be used. The choice of using either PCG or CMWC depends entirely on the user. If the size of the program is essential, PCG should be used, but if the overall efficiency is important, then we suggest CMWC.

5.5 Fortuna

Fortuna is the only generator that uses a block cipher in this study. It is considered one of the nicest CSPRNG's out there. It might seem like an unfair comparison with the other studied PRNG's. However, we believe that this comparison is valid as all of these PRNG's have different usage purposes due to their security and efficiency levels. The main focus is to see which one should be used for what purpose. One of the drawbacks of this comparison is that we do not have the Big Crush and efficiency results for Fortuna. One of the main reasons this is the case is that implementing the very complex structure of the Fortuna and using it with the Big Crush test is a very challenging and problematic process that we could not manage to finish in the given time. Thus, we primarily use references to other studies to discuss Fortuna.

One of the main advantages of Fortuna is the usage of AES as its generator. AES is proven to be one of the most secure generators of all time. It gives a massive advantage in terms of security as it is impossible to gain the state of the generator just by analyzing the output [17]. If an attacker cannot find out the state, then they have very, very little chance to guess the previous or future generated numbers. Another advantage of the Fortuna comes with its pools. Thirty-two pools seed the generator, and these pools can take up several years before being emptied [22]. This means that when an attacker attacks with an injection attack to the pools, it is very challenging for them to control all of the pools at once. Thus, if an attacker cannot control all of the pools, the generator continues to be secure as it can use its other untouched pools to seed the generator.

One of the main drawbacks of the Fortuna comes with its greatest advantage. AES has a deterministic nature [4], which means an attacker can predict all of the future generated numbers once they get to learn the inner state. To prevent the exploitation of the inner state values, the key used for the AES has to be randomly changed every time the generator is used.

According to a study, hardware implementation of the Fortuna instead of software implementation increases its security [13]. It is because when information about the state and the pools are stored inside of the RAM, it becomes more challenging for the attacker to access them [13].

Fortuna is designed to be used in cryptographical-secure applications, unlike the other studied PRNG's. Thus, Fortuna is the most secure PRNG studied in this paper. However, although we do not have concrete test measures, we believe that the efficiency (CPU time to generate numbers and file size) of the Fortuna is the worse out of all the studied PRNG's. It is because, as can be seen in section 2, its structure is the most complicated with 32 pools, pool manager, key generation and a block cipher, namely AES-256, while the other PRNG's contain basic mathematical operation such as AND, OR, XOR, bit-wise shift, multiplication and addition operations, which are fast to perform in computers. Thus, Fortuna should be used when security has very high importance, while efficiency has a lower priority. These IoT applications can include sensitive private data that others should not know. Examples for these applications can be home monitor systems (camera's),

smart-watches, home interaction systems (Alexa, Siri), smart electric cars, and public systems such as train, bus schedule monitoring.

6 Possible Improvements and Recommendations

In this section, we are discussing the improvements that are made for the xorshift and xorshiftStar generators and presenting the results of these improved versions.

Xorshift and xorshiftStar generators are high-speed and efficient generators but weak in terms of security. The main goal was to improve their non-linearity by making their structure more confusing. We came up with an idea and implemented it in C language to see whether it differs in statistical tests. We discuss only the statistical aspect and the possible drawbacks of these improvements. Since we are not experts in cryptanalysis, we cannot say that the improvements increase the security of the generator or not, but we can only give suggestions. Thus, all of the improvement ideas mentioned in this chapter should be further studied by a cryptanalysis expert.

The main idea is to combine these generators with secure S-boxes. S-boxes are usually used in encryption ciphers to distort the data [23]. For an S-box to provide confusion, it must be designed in a way that it should be resistant against cryptoanalysis attacks. The security of many block ciphers are dependent on many things but most importantly, their underlying S-boxes. We aimed to increase the non-linearity of these PRNG's, and we decided to achieve these by combining the generators with S-boxes. That is because security proven S-boxes can create high amounts of confusion by substituting the data. This is already done with many existing block ciphers but not with a single NCSPRNG's. Thus, we wanted to test the combination of S-Box's with xorshift family generators to see if they actually work. One reason why this approach has not been used could be that NCSPRNG's are usually designed for speed and efficiency instead of providing high security. Our initial thought was that by adding S-boxes, generators efficiency would decrease, making their design and usage's quite different from their original design, but we believe that it would make them more unpredictable. These implementations are tested and compared later in this chapter.

One other important point to consider is that designing secure S-boxes is challenging even for cryptography experts. These boxes are tested with many cryptoanalysis attacks before being used in applications. Thus, we chose to use an S-box of an existing secure cipher called PRESENT [24]. It uses a 4x4 S-box as can be seen in Figure 2.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Figure 2: 4x4 S-Box used in PRESENT block cipher [25].

We could have chosen a larger secure S-Box such as AES's 8x8 S-box, but the larger the S-box becomes, the heavier the process works. Although we aim to improve the unpredictability, we also try to keep a balance between security and efficiency.

As can be seen in Figure 3, our version of the xorshiftStar adds additional steps after the multiplication operation. After the multiplication, either the higher 32-bits or, the lower 32-bits (depending on which we want to test) is selected. Then selected bits are divided into 4 bytes in order to be used in a 4x4 S-Box. These 4 bytes are individually fed into the S-Box, and then the resulting

outputs of the S-Box are combined in a permutated order. The state has to be used for later number generations. Thus, protecting the state of the generator is trivial for the security of a PRNG. After saving the state for the subsequent usage, with the help of multiplicative operation plus the additional S-Box and permutation, we distort the state to become different from the original state. Thus, it becomes harder to trace back to the original state, increasing its unpredictability for future generated numbers.

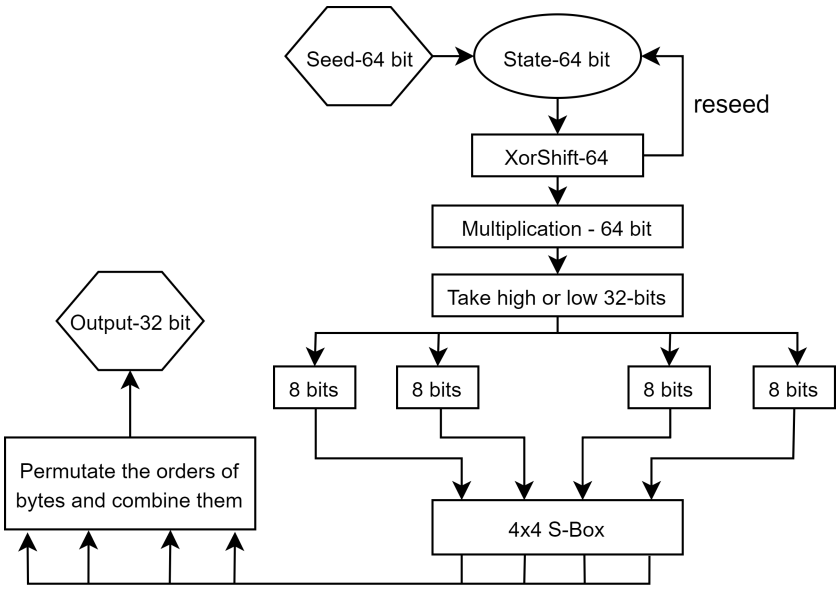

Figure 3: Structure of the improved xorshiftStar.

The idea for the xorshift is similar to the one we did on xorshiftStar but in a more complex and heavier way. The structure that we applied to the xorshiftStar did not increase the complexity of the xorshift generator the way we expected. In the statistical tests, there was an improvement of 16 tests (59 fails reduced to 43 fails), but this was not enough for our aims. We wanted to achieve a clear improvement in statistical tests, reducing most of the systematic failures. We did not want to add multiplicative or additive operation between the S-Box and xorshift steps because they are already done in xorshiftStar and xorshiftPlus generators. After tons of trial and errors, we decided to use two S-Boxes; one is the original S-Box of the PRESENT, and the other one is the inverse of it, which made the structure way complicated.

As can be seen in Figure 4, the structure after the xorshift part is the following; first of all 32-bit number is divided into 4 bytes and fed into the initial S-Box. Then two of the resulting bytes are fed into the same S-Box again. However, the other two bytes are fed into the inverse of the S-Box. Additional S-Box is used to separate two of the bytes selected randomly to create more confusion. After all of the bytes are fed into the S-Boxes, we take the xor of the outputs of the S-Boxes with some of the initial bytes that were calculated before the S-Box step. These xor steps are performed between different rounds to reduce the probability of re-tracing the S-Boxes. These steps are repeated 20 times before they are combined in permutated order, and returned to the user.

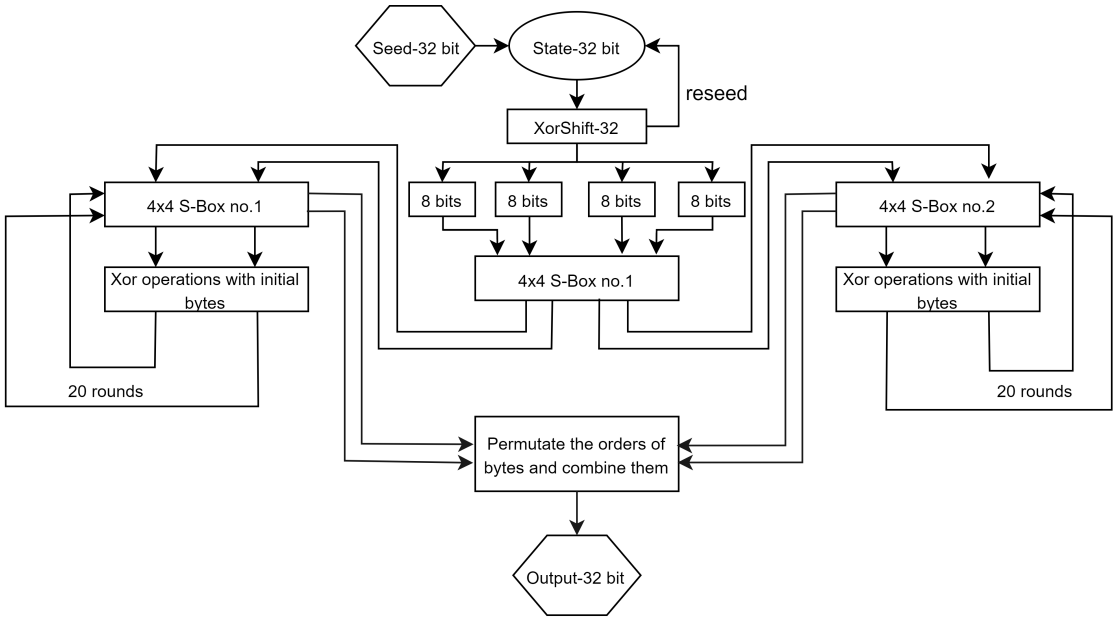

Figure 4: Structure of the improved xorshift.

As can be seen from the Table 3, improved versions of the generators increase the statistical quality of the original generators. Improved xorshiftStar fails zero tests of Big Crush while the original one fails five tests. This statistical improvement comes with a new and more non-linear structure, as described before. We believe that the improved version is more secure than their original version, and the other studied NCSPRNG's (This claim is just a suggestion that should be further studied!). The generated numbers are confused and distorted in a way that it becomes harder to find patterns. Re-tracing the state of the generator becomes more challenging due to a more compact structure. Thus, the non-predictability of the improved version can be higher than its original version. One problem could be that if the inner structure gets exposed, it can become weaker, just like some block ciphers or any other PRNG's because an attacker can simulate the generation process and possibly expose the state. Thus, it is essential to protect the inner structure (the seed, multiplicative value, and the permutation logic) to provide high security.

Table 3: Comparison of the Big Crush test results of the original Xorshift(Star) and our version.

Big Crush Test Results		
Name of the PRNG	Number of total test fails for the original version	Number of total test fails for our version
Xorshift-32	(59/160)	(8/160)
Xorshiftstar-64	(5/160)	(0/160)

The improved xorshift fails eight tests of Big Crush, while the original version fails 59 tests. These results show a substantial statistical improvement as the original xorshift is considered a weak generator in security and statistical quality. Even though the structure of the improved xorshift is

more complicated than the improved xorshiftStar, xorshiftStar still performs better than it. The main difference between the two is that one contains arbitrary multiplicative operation and the other does not. For testing purposes, we also tried improved xorshift (Figure 4) with an additional multiplicative operation in between the S-Box and xorshift steps, and the result of the Big Crush was higher (0 failing tests). As a side note, the original xorshiftStar performs even better than the improved xorshift. Thus, it can be concluded that a single arbitrary operation that the users do not know can create huge differences. We believe that it is the case because by multiplicating the number k with number b, we can end up with a number z that is way different and far away from the original number. Thus, predicting the final number becomes challenging when the users do not know the multiplication and the original state.

Table 4: Comparison of the GPU times and file sizes of the original Xorshift(Star) and our version.

Comparison Table				
Name of the PRNG	GPU time to generate 10^8 numbers for the original version (in sec)	GPU time to generate 10^8 numbers for our version (in sec)	Total file size for the original version (in bytes)	Total file size for our version (in bytes)
Xorshift-32	1.93	7.35	563	2800
Xorshiftstar-64	1.95	2.32	897	2100

Table 4 shows the time the generators take to generate a hundred million numbers and their total file sizes. Xorshift generator family is originally designed for speed and efficiency. However, with our improvement's, this is not further the case. We mainly focused on increasing their security, which resulted in less efficiency than the original versions and the other studied PRNG's. Improved xorshiftStar generates a hundred million numbers in 2.32 seconds while the original takes 1.95 seconds with an increase of 0.38 seconds. However, the improved xorshift is even performing worse in terms of efficiency as it takes 7.35 seconds while the original version takes 1.93 seconds, an increase of 5.42 seconds due to its heavier structure.

After testing the improved xorshift with 20 rounds, we played with the number of rounds and found that rounds larger than 10 give the same statistical results. Thus, to increase efficiency, we could use ten rounds instead of 20. When we use ten rounds, the time it takes to generate a hundred million numbers decreases to 4.5 seconds from 7.35 seconds. However, this number is still too large for a PRNG that was initially designed for speed.

If we look at both the statistical quality and the efficiency, improved xorshift does not add many benefits compared to the improved xorshiftStar. Thus, we do not recommend the usage of the improved xorshift until it is further studied to be more efficient. However, we think that the improved xorshiftStar becomes the perfect PRNG compared to the other studied PRNG's for IoT device usage. This is because it performs a perfect score on the statistics, and we believe that the combination of multiplicative operation and the additional S-Box structure makes the generator more secure than the PCG, CMWC and xorshift (note that an expert should further study this claim!). Even though it performs quite lower on efficiency scores, the difference between the other studied PRNG's is not that much. Thus, for IoT applications requiring a balance between security and efficiency, improved

xorshiftStar usage is recommended. Maybe further development of this improvement idea can turn the NCSPRNG xorshiftStar into a CSPRNG xorshiftStar.

A similar structure applied to xorshiftStar may be applied to PCG or CMWC as well. However, due to the time limit, we do not consider these cases in this study. Thus, it could be a beneficial future study to test and experiment with the PCG and CMWC generators using the S-Box structure.

7 Conclusion

This research aims to compare PRNG's straightforwardly by analyzing their statistical quality, security and efficiency. It has been done using literature studies and with actual testings. All of the studied PRNG's are designed for specific purposes where their security and efficiency levels differ. For a PRNG to be secure, its inner state must not be exploited, and the resulting outputs must be unpredictable with minimal or no visible patterns. As a result, we see that xorshift is a not-secure but most efficient generator, which should not be used in IoT applications where security is essential. XorshiftStar is the improved version of xorshift, which is more secure than the original one but still can be breakable. PCG is one of the most balanced NCSPRNG in terms of security and efficiency, which can be used in most IoT applications where security and efficiency needs a balance. The results of the CMWC is very similar to PCG, just with larger code size. Fortuna is the only CSPRNG studied in this paper, and because of its design (usage of pools and AES-256), it is the most secure PRNG out of all studied PRNG's. However, due to its complex and heavy structure, it is the least efficient one.

As a possible improvement, xorshift and xorshiftStar generators are combined with S-Box structure to increase their complexity and confusion. We see that these improvements increased their statistical quality but decreased their efficiency. We also believe that these improvements made them more secure than the original versions, but a cryptoanalysis expert should further study this claim.

A Figures

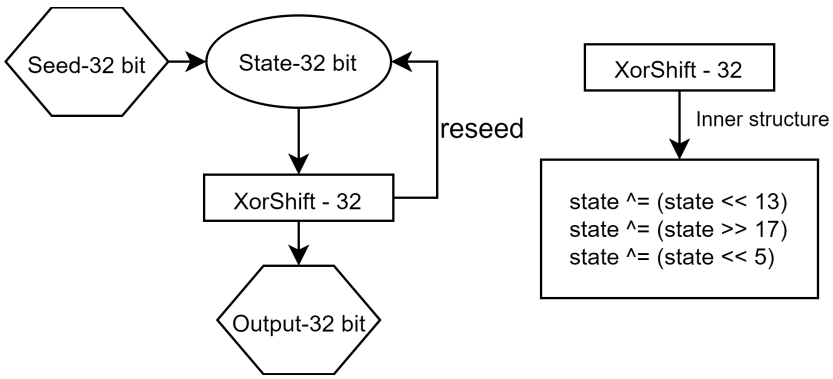

Figure 5: Structure of the Xorshift-32.

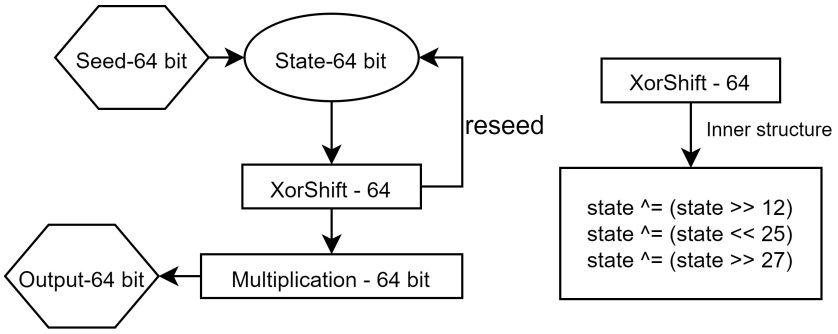

Figure 6: Structure of the XorshiftStar-64.

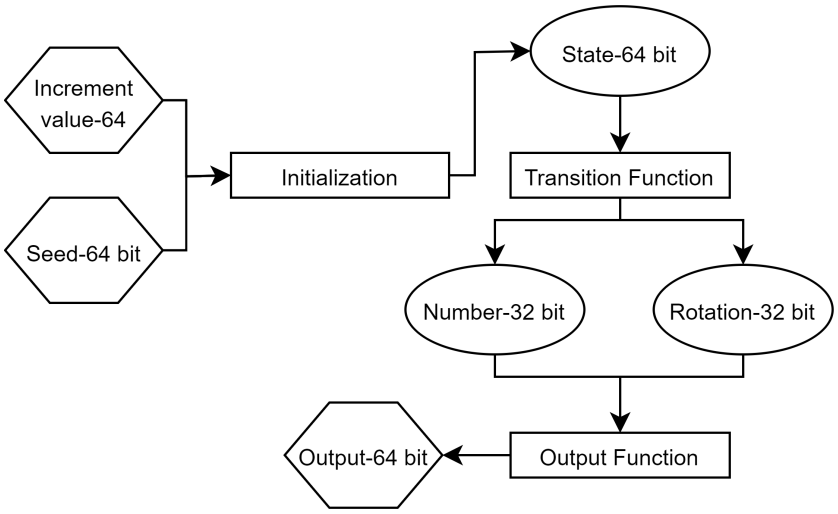

Figure 7: Structure of the PCG XSH-RR. (Note that output is 32-bit instead of 64-bit)

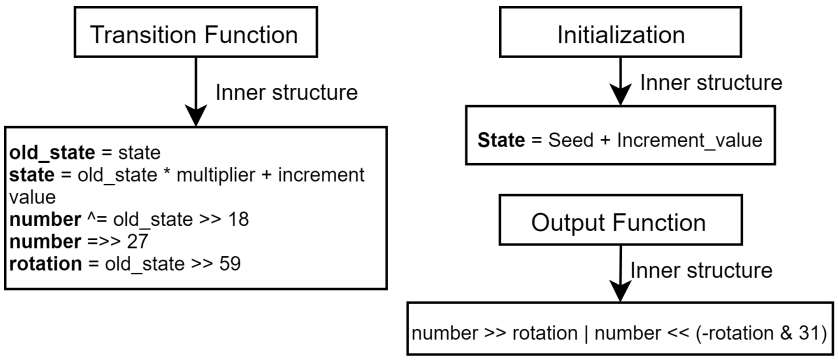

Figure 8: Inner structures of the PCG Functions.

References

[1] M.Roser, H.Ritchie, E. Ortiz-Ospina (2015). Internet [Online]. Available: https://ourworldindata.org/internet

[2] P. Sethi and S. R. Sarangi, âInternet of Things: Architectures, Protocols, and Applications,â Journal of Electrical and Computer Engineering, vol. 2017, pp. 1â25, 2017.

[3] V. Rao and K. V. Prema, âA review on lightweight cryptography for Internet-of-Things based applications,â Journal of Ambient Intelligence and Humanized Computing, 2020.

[4] T. Prescot, "Random Number Generation Using AES," 2011.

[5] Richard P. Brent, "Journal of Statistical Software, Oxford University," Oxford University, 2004.

[6] S. Tezuka, âLinear Congruential Generators,â Uniform Random Numbers, pp. 57â82, 1995.

[7] M. E. O'Neill, âPCG, A Family of Better Random Number Generators,â PCG, A Better Random Number Generator, 20-Aug-2014. [Online]. Available: https://www.pcg-random.org/.

[8] M.E. O'Neill, "PCG, A Family of Simple Fast Space-Effecient Statistacilly Good Algorithms for Random Number Generation," CA 91711, USA, 2014.

[9] âMultiply-with-carry pseudorandom number generator,â Wikipedia, 06-May-2021. [Online]. Available: https://en.wikipedia.org/wiki/Multiply-with-carry_pseudorandom_number_generatorComplementary-multiply-with-carry_generators.

[10] J. Kelsey, B. Schneier, and N. Ferguson, âYarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator,â Selected Areas in Cryptography, pp. 13â33, 2000.

[11] T. Kohno, N. Ferguson, and B. Schneier, Cryptography engineering: design principles and practical applications. Indianapolis, IN: Wiley Pub., Inc., 2010.

[12] A. Bogdanov, F. Mendel, F. Regazzoni, V. Rijmen, and E. Tischhauser, âALE: AES-Based Lightweight Authenticated Encryption,â Fast Software Encryption, pp. 447â466, 2014.

[13] R. McEvoy, J. Curran, P. Cotter, and C. Murphy, âFortuna: cryptographically secure pseudo-random number generation in software and hardware,â IET Irish Signals and Systems Conference (ISSC 2006), 2006.

[14] P. LâEcuyer and R. Simard, "A Software Library in ANSI C for Empirical Testing of Random Number Generators," Universite de Montreal, 2013.

[15] âTestU01,â Empirical Testing of Random Number Generators. [Online]. Available: http://simul.iro.umontreal.ca/testu01/tu01.html.

[16] TestU01 Installation Guide. [Online]. Available: http://simul.iro.umontreal.ca/testu01/install.html.

[17] A. Rock, "Pseudorandom Number Generators for Cryptographic Applications," Paris-Lodron-Universitat Salzburg, 2003.

[18] S. Vigna, âAn Experimental Exploration of Marsaglia's xorshift Generators, Scrambled,â ACM Transactions on Mathematical Software, vol. 42, no. 4, pp. 1â23, 2016.

[19] M. E. O'Neill, âPredictability,â PCG, A Better Random Number Generator, 17-Oct-2014. [Online]. Available: https://www.pcg-random.org/predictability.html.

[20] C. Bouillaguet, F. Martinez, and J. Sauvage, âPractical seed-recovery for the PCG Pseudo-Random Number Generator,â IACR Transactions on Symmetric Cryptology, pp. 175â196, 2020.

[21] M. Goresky and A. Klapper, âEfficient multiply-with-carry random number generators with maximal period,â ACM Transactions on Modeling and Computer Simulation, vol. 13, no. 4, pp. 310â321, 2003.

[22] âFortuna (PRNG),â Wikipedia, 13-Jan-2021. [Online]. Available: https://en.wikipedia.org/wiki/Fortuna_(PRNG).

[23] H. M. Heys, "A Tutorial on the Implementation of Block Ciphers: Software and Hardware Applications," Memorial University of Newfoundland, St. John's, Canada, 2020.

[24] D. Irwin, P. Liu, S. R. Chaudhry, M. Collier, and X. Wang, âA Performance Comparison of the PRESENT Lightweight Cryptography Algorithm on Different Hardware Platforms,â 2018 29th Irish Signals and Systems Conference (ISSC), 2018.

[25] A. Bogdanov, I.R Knudsen, G. Leander, C. Paar, A. Poschmann, M.J.B Robshaw, Y. Seurin and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher," Horst-Gortz-Institute for IT security, Germany and Technical University Denmark, 2007.