DOKUZ EYLÜL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

DESIGNING AN RNG FOR SECURE DATA COMMUNICATION WITH WISP

by

Cem KÖSEMEN

July, 2019

İZMİR

DESIGNING AN RNG FOR SECURE DATA COMMUNICATION WITH WISP

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Master of
Science in Computer Engineering

by
Cem KÖSEMEN

July, 2019
İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **"DESIGNING AN RNG FOR SECURE DATA COMMUNICATION WITH WISP"** completed by **CEM KÖSEMEN** under supervision of **ASSOC. PROF. DR. GÖKHAN DALKILIÇ** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Gökhan DALKILIÇ

Supervisor

Asst. Prof. Dr. Enis KARAARSLAN

(Jury Member)

PROF. DR. YALÇIN ÇEBİ

(Jury Member)

Prof. Dr. Kadriye ERTEKİN
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I would like to thank to my supervisor Asst. Prof. Dr. Gökhan Dalkılıç for his guidance and help throughout this work. I also thank PhD student Ömer Aydın for his help on development and testing.

Cem KÖSEMEN

DESIGNING AN RNG FOR SECURE DATA COMMUNICATION WITH WISP

ABSTRACT

Requirement for security in lightweight devices such as radio frequency identification tags (RFID) is increasing. Pseudorandom number generator (PRNG) is an essential part of the authentication protocols that provides security. Aim of this thesis is producing a lightweight PRNG for cryptographic applications in wireless identification and sensing platform (WISP) family devices and other lightweight devices. The proposed PRNGs in this work is produced with a genetic programming (GP) methods using entropy calculation as fitness function. Proposed PRNG is tested with NIST statistical test suite (NIST STS) and passes it successfully. It also satisfies the requirements of EPCGen2 standards.

Also, a hardware random number generator (HRNG) that runs on WISP is designed, which uses the accelerometer sensor of WISP as an entropy source. This entropy source is post-processed with de-biasing and extraction methods to provide more uniformly distributed results that can be used in the authentication protocols between a RFID tag and an RFID reader. The obtained random number outputs of HRNG are also tested using the NIST STS and they pass all of the tests.

Keywords: PRNG, genetic programming, RFID, WISP, security, NIST STS, HRNG

WISP İLE GÜVENLİ VERİ TRANSFERİ İÇİN RANDOM SAYI ÜRETECİ TASARLANMASI

ÖZ

Düşük güç tüketimli radyo frekansı ile tanımlama (RFID) cihazlarındaki güvenlik ihtiyacı günümüzde artmaktadır. Sözde rastgele sayı üreteçleri (SRSÜ) güvenliği sağlayan tanımlama protokollerinin temel parçalarından biridir. Tezin amacı, Kablosuz Kimlik Tespiti ve Algılama Platformu (WISP) gibi düşük güç tüketimli cihazlardaki kriptografik uygulamalarda çalışabilecek bir SRSÜ oluşturmaktır. Bu SRSÜ, uygunluk fonksiyonu olarak bit entropi hesaplama yöntemi kullanan bir genetik programlama (GP) yöntemiyle oluşturulmuştur. Oluşturulan SRSÜ, NIST istatistiksel test takımı (NIST STS) ile test edilmiş olup, bu testleri başarıyla geçmiştir. Üretilen SRSÜ, EPCGen2 şartlarını da sağlamaktadır.

Ayrıca, WISP üzerinde çalışan ve yine WISP üzerindeki ivmeölçeri entropi kaynağı olarak kullan bir donanımsal rastgele sayı üreteci (DRSÜ) de geliştirilmiştir. Bu entropi kaynağından toplanan veriler, tarafsızlaştırma ve damıtım yöntemleriyle işlendikten sonra istatiksel olarak daha iyi bir dağılıma sahip olup, RFID cihazlarındaki tanımlama protokollerinde kullanılabilecek hale getirilmektedir. DRSÜ çıktıları da NIST STS ile test edildiklerinde tüm testleri geçen sonuçlar elde edilmiştir.

Anahtar kelimeler: SRSÜ, genetik programlama, RFID, WISP, güvenlik, NIST STS, DRSÜ

CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

CHAPTER ONE
INTRODUCTION

Pseudorandom number generators (PRNGs) are required in many cryptographic applications. Random numbers generated from PRNG are used as secret keys and nonce values. Secure communication protocols use random nonce values for authentication purposes, which are very critical parts of these protocols. In this thesis, a lightweight PRNG is produced to provide a secure end-to-end communication between wireless identification and sensing platform (WISP) family of devices and their readers.

Security, one of the main concerns of users of interconnected computers, is nowadays a bigger issue. Because the Internet is now evolved into the Internet of things (IoT), where billions of electronic devices are connected, in addition to the computers. These devices are usually small and constrained. Means, they have limited power sources, memory and processing capabilities. However, they sometimes handle and process very sensitive personal or commercial data. Yet, many modern security countermeasures are too heavy to be implemented on such devices. Even random number generation (for authentication and encryption purposes) may be too heavy for them. Nevertheless, random numbers are one of the primary elements for many cryptographic applications. PRNG seeds, encryption keys and authentication tokens need to be fed by random numbers. In some applications, the communication between the radio frequency identification (RFID) tag and the reader must be secure. This two-way transmission with the WISP tag and the reader should not be tracked nor modified by unauthenticated individuals. Also, in some situations, copying/tracking tags or authentication of the reader are critical security problems and must be solved.

Our motivation is providing a lightweight PRNG for devices with limited computing power. This produced PRNG must be usable as a part of security protocols.

In this thesis, a PRNG is proposed for WISP to use it in its security protocols. WISP is a lightweight device so, only the mathematical addition operator and the bitwise

logical operators are used in produced PRNGs. Produced PRNG can be used on any other lightweight device besides WISP, like various IoT devices, which have the task of transferring sensitive information that needs to be protected.

Our main contribution is showing that a statistically good PRNG can be generated with genetic programming (GP) methods using the entropy calculation as its fitness function. This produced PRNG satisfied NIST STS and EPCGen2 constraints which are important standards for most security protocols.

1.1 WISP

WISP is an RFID device with a microcontroller. It has a general-purpose, fully programmable 16-bit microcontroller, and external physical sensors like accelerometer, light sensor and temperature sensor (Chae, Salajegheh, Yeager, Smith, & Fu, 2013; Jiang et al., 2005; Sample, Yeager, Powledge, & Smith, 2007; Smith, Sample, Powledge, Roy, & Mamishev, 2006). WISP uses the received power of the reader's emitted radio signals to work, which allows it to run without a power source. WISP has a processor in its microcontroller and it can execute programs pre-loaded on it. This feature is not included in other passive RFID tags so, this makes WISP suitable for implementing security protocols and allows running these protocols easier on it. The PRNG produced in this work is also programmed and loaded onto WISP, and runs on its processor flawlessly.

WISP's development started at Intel Labs Seattle in 2006. WISP 5 is the latest version of WISP device family, which is developed at the University of Washington laboratories. A WISP 5 device obtained from University of Washington is used for testing purposes in this work. WISP 5 that is presented on the web site wisp5.wikispaces.com, conforms to the EPC Class 1 Generation 2 (EPC C1G2) standard for the ultra-high frequency (UHF) RFID tags. WISP 5 has an MSP430 microcontroller with FRAM non-volatile memory for ultra-low energy for data transmission. It also has ADXL362 accelerometer with very low power consumption.

WISP 5 communicates with an RFID reader that acts as a base station like previous versions (Yıldırım, Aantjes, Majid, & Pawełczak, 2016).

1.2 Random Number Generators

There are several approaches for generating random numbers. Hardware random number generators (HRNG) use non-deterministic sources, for example physical sources like accelerometer sensors, temperature sensors, magnetic fields sensors, as their entropy sources. They obtain data from these sensors and apply some whitening methods to generated number, true random numbers (TRN) are generated (Stipčević & Koç, 2014). HRNGs generally have slow rates of number production and generating TRNs are costly.

HRNGs collect the raw data from physical sources via sensors or other devices. After the digitized data is collected, they harvest the raw data with a method which collects as much entropy as possible (Sunar, 2009). For instance, taking the least significant bit (LSB) of each part of raw data is an acceptable method because LSBs have higher chance to differ.

The final step in HRNG design is extracting the randomness from that harvested data. Numbers collected from physical sources are usually biased and not uniform which means they do not have the same number of zeroes and ones. So, these harvested data should be post-processed with whitening and randomness extraction methods. Post-processing methods eliminate biases of the data. There are many kinds of post-processing methods like ad hoc simple correctors, cryptographic hash functions, extractor functions and resilient functions (Stipčević & Koç, 2014).

PRNGs generate random numbers with deterministic functions that are basically mathematical expressions with operators (Park & Miller, 1988). They include operators like addition, multiplication, division, modulus; and other logical operators like bitwise and, or etc. The same seed given to these functions always output the same sequence as the result. PRNGs have higher rates of number production, because it does

not wait for any data from outside. However, it is required that PRNGs to be seeded with true random numbers which is generated by an HRNG. This working logic can be seen in Figure 1.1.

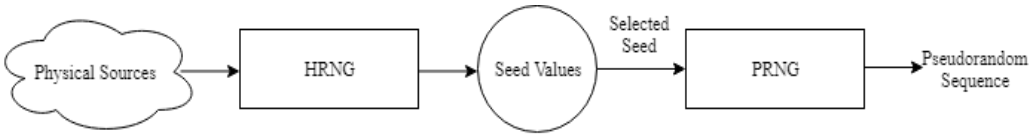

Figure 1.1 PRNG & HRNG Workflow

PRNGs should be tested with appropriate test tools to determine their statistical quality (randomness). Currently, there are several test softwares that use different methods. In this work, NIST statistical test suite (STS) is selected for validating the quality of the generated random numbers, because NIST STS is one of the well-known statistical test suites, and includes 15 different statistical tests (Barker & Kelsey, 2014, 2016b; Bassham et al., 2010; Turan et al., 2016). A selected PRNG, which will be tested, generates a binary output sequence and NIST STS applies a set of statistical tests to this binary sequence. These 15 tests of NIST STS evaluate the statistical properties of this binary sequence. Some of these tests are; uniformity of zeroes and ones, repeating patterns, bit entropy calculation, etc.

1.3 Genetic Programming

Genetic programming (GP) is a problem-solving method which provides a full or an approximate solution, among a population of initial computer programs. This initial set (called the population) of programs, or expressions, may change depending on the problem domain. For example, if the solution is a good PRNG, the possible population consists of PRNG functions, which consist of numbers, inputs and mathematical or logical operations. Each individual of the population has a fitness value based on how good it solves the problem. The fitness of an individual indicates that how well it solves the given problem (Koza, 1992).

In the starting generation, random population of individuals is generated. Initial population's individuals have low fitness scores because they are generated randomly

and so, they are far from being a viable solution. After generations pass by applying selection, crossover and mutation operations to these individuals, better generations with higher fitness values are obtained. The population is improved and individuals have higher chance to become a solution. If any of the individuals of the population reaches (or becomes near by a some tolerance value) the perfect fitness score in any specific generation, then this individual is considered as the solution of the problem (Koza, 1990, 1994).

1.4 Testing PRNGs & NIST STS

There are several considerations for PRNGs and the random numbers generated from them. First of them is uniformity. Uniformity means that, the numbers of a zeros and ones in sequence are equally likely. For example, in a binary sequence, ½ of the all sequence should be 0.

Second one is scalability. If any test is applied to the sequence and passed, it should also pass on the subsequences of that sequence. The last one is consistency. A PRNG should produce high quality results for all seed values. If a PRNG works good for few seed values, it is not adequate (Rukhin et al., 2010).

PRNGs can be tested with appropriate tools to determine their statistical quality. Currently, there are several testing softwares using different methods. NIST STS is designed for testing randomness of PRNGs which are used in cryptographic applications. NIST STS applies a set of statistical tests to binary sequences which are generated by the given PRNG. These tests evaluate statistical properties like; uniformity of zeroes and ones, uninterrupted series of same bits, repeating patterns, entropy etc. Because of there are infinite number of possible statistical test methods, no specific statistical test suit can be considered as complete.

First aim of a statistical test is testing a given null hypothesis (H_0). This null hypothesis indicates that the data which is going to be tested is random. Other than

null hypothesis, there is an alternative hypothesis (H_A) which indicates that the data which is going be tested is not random. Statistical tests decide that whether it accepts the null hypothesis or alternative hypothesis. For example, if the PRNG generates a sequence, and this sequence fails to confirm null hypothesis, this PRNG considered non-random.

In sum, a statistical test only has two results. It either accepts a sequence as random by accepting H_0 or rejects by accepting H_A which means data is not random.

There can also be some faulty results in these statistical tests. If the statistical test rejects the null hypothesis, even though the data actually is random, it is called Type I error. If the statistical test accepts the null hypothesis, even though the data actually is not random, it is called Type II error. Probability of the occurrence of these errors is indicated with α variable. In most cases, including cryptographic applications, α is acceptable below the probability of 0.01.

Another variable used in statistical tests is the P-value. It shows the strength of the clues that are opposing null hypothesis. If the P-value is acquired as 1 as a result of testing, the data are considered perfectly random. If the P-value is acquired as 0 as a result of testing, the data are considered completely not random.

With chosen significance level, a result can be obtained according the P-value. If the P-value is greater or equal than significance level, it means that the null hypothesis is accepted. If the P-value is lesser than significance level the null hypothesis is rejected as a result, and the data are not random. The significance level (α) indicates the chance of the Type I error occurring. Generally, α is chosen between 0.001 and 0.01 in NIST STS (Rukhin et al., 2010).

For example, if α is 0.001, it shows that a sequence can be rejected by a chance of 1/1000, even though the sequence is actually random. So, if the obtained P-value from the sequence is greater than α value, the sequence is random with a certainty of 99.9%.

Likewise, if α is 0.01 and the obtained P-value from the sequence is greater than that α value, it shows that a sequence can be rejected by a chance of 1/100, even though the sequence is actually random. So, the sequence is random with a certainty of 99% (Rukhin et al., 2010).

1.4.1 NIST STS Details

NIST STS includes 15 different tests to qualify the randomness of the given binary data. In these part, all of the NIST STS tests are examined in detail (Rukhin et al., 2010; Zaman & Ghosh, 2012). These tests help to decide whether the PRNG has a good quality or not. Also, they confirm that the implementation of good PRNGs which produce statistically good binary sequences. It should also be noted that, no statistical test suite can exactly say that, a PRNG is appropriate for using in a specific area like cryptographic applications. Every application has its own aim and requirements (Rukhin et al., 2010).

1. **Frequency (Monobit) Test:** This test analyzes all of the binary sequence, and counts the number of zeroes (0) and ones (1). It is expected that the sequence has the same number of zeroes and ones, which is accepted statistically good.

2. **Frequency Test in a Block:** This test analyzes the M-bit subsequences of the sequence and counts the number of zeroes and ones. It is expected that the subsequence has the same number of zeroes and ones, which is statistically good. For example, if M is defined as 10 and the sequence has 50 bits, there are 5 subsequences. Each subsequence's frequency is calculated separately. Basically, it checks whether or not frequency is evenly distributed among the subsequences.

3. **Runs Test:** This test analyzes the changing rate between zeroes (0) and ones (1). If there are too long or too short sequences of ones and zeroes, the sequence is expected to be not random. For instance, the sequence 1001101011 has 6 runs. Run count is calculated by how many times the repeating bit is changed.

4. **Test for the Longest Run of Ones in a Block:** Longest run test is similar to runs test but it works on M-bit subsequences. It looks for uninterrupted series of ones in subsequences.

5. **Binary Matrix Rank Test:** The aim of this is to calculate the rank of disjoint sub-matrices that are acquired from the sequence and checking for linear dependence on these fixed length substrings (Marsaglia & Tsay, 1985). First, matrices' sizes are determined so that the sequence will consecutively read and imported to the matrices. After forming the matrices (NIST STS uses 32x32 matrices to calculate the ranks), ranks of these matrices are calculated. By definition, the rank of the matrix is an integer greater than zero and it cannot be greater than the size of the either row or column of matrix. Also, only the zero matrix has rank of zero. Finally, NIST STS calculates the chi square of obtained values.

6. **Discrete Fourier Transform (Spectral) Test:** This test finds the peak points in a binary sequence with Discrete Fourier Transform (DFT) of this given binary sequence. It detects the periodic properties of the sequence which are not seemed statistically good.

7. **Non-overlapping Template Matching Test:** This test tries to find PRNGs that produce the same non-periodic pattern much. A specific m-bit pattern is selected for searching for other same m-bit patterns. If this specific m-bit pattern is not found in the tested sequence, this pattern is slide for one bit. If the pattern is found, it is reset to the bit after the found pattern, and searches again.

8. **Overlapping Template Matching Test:** This test tries to find PRNGs that produce the same non-periodic pattern much. Both this test and the non-overlapping template matching test work similarly. If a specific m-bit pattern is not found in the tested sequence, this pattern is slide for one-bit. If this pattern

is found, it slides only one bit before continuing to search. This sliding operation is the difference between this test and the non-overlapping template matching test.

9. **Maurer's Universal Statistical Test:** This test finds the bit counts between same patterns. This property can also be obtained from the length of the compressed form of this data. It finds out if the data are highly compressible without information loss. If data are significantly compressible, it is considered non-random (Coron & Naccache, 1999; Maurer, 1992).

10. **Linear Complexity Test:** This test basically determines if the given sequence has enough complexity as in a random sequence. Sequences generated by longer linear-feedback shift registers (LFSR) are considered more random. A short LFSR shows that sequence is not random. First, sequence is divided into parts. After that, linear complexity of each subsequence is calculated with Berlekamp-Massey algorithm (Menezes, Oorschot, & Vanstone, 1997). This algorithm finds the linear complexity of a binary sequence, so it will find the shortest LFSR for a given binary output sequence.

11. **Serial Test:** This test tries to prove if the times the 2^m-bit overlapping patterns occurred in given sequence is distributed statistically good.

12. **Approximate Entropy Test:** Approximate entropy test calculates the entropy value of the given sequence to decide its randomness. The overlapping patterns' frequency in all possible m-bit patterns with (m+1)-bit patterns in a sequence are compared to calculate the entropy of this sequence (Rukhin, 1997).

13. **Cumulative Sums Test:** This test first calculates the cumulative sum of the subsequences contained in the given sequence. If obtained cumulative sum is too far from the expected values of the cumulative sum of statistically good

sequences, the sequence is considered non-random. If a sequence is random, there should be nearly zero excursions of the random walk.

14. **Random Excursions Test:** This test looks for the number of visits to a specific cumulative sums state in a cycle. After that, it determines whether the sequence is random or non-random according to this number of visits.

15. **Random Excursions Variant Test:** This test finds the number of visits to a specific state in cumulative sums of random walk across the sequence. After that, it finds differences from expected number of visits in a random sequence. It determines whether the sequence is random or non-random according to this number of visits (Rukhin et al., 2010).

1.5 Outline of the Thesis

Second chapter contains a literature review of the past related works which are inspired and helped this thesis. Related works includes both works for PRNG and HRNG methods. In the third chapter, methods for generating a lightweight PRNG with a genetic algorithm (GA) is explained in more technical detail.

NIST STS and EPCGen2 results of the proposed PRNG is examined in the fourth chapter. Also for statistical analysis, a set of PRNGs are generated with this GA method and they are also tested and their statistical properties are shown.

In fifth chapter, a HRNG method is proposed which is using accelerometer of the WISP. Harvesting and post-processing methods of the acquired raw data are also explained. Chapter six includes a conclusion of the thesis and refers to possible future improvements.

CHAPTER TWO
RELATED WORKS

In the past, there are lots of different approach for generating a PRNG function (or just pseudorandom numbers) with GP methods.

2.1 Literature Review

One of the first approaches for producing a PRNG with GP methods is Koza's work that produces a randomizer (Koza, 1991). That randomizer converts a sequence of consecutive numbers, which starts from a specific starting point, to random numbers. Unlike PRNGs, it does not depend on a recursive seed. This work uses a bit entropy calculation method as its fitness function, where binary sequence outputs with higher entropy are considered better. Bit entropy calculation as the fitness function is also used in our work.

Koza's work includes some simple statistical test results, like frequency and gap tests, but not a complete test suite as NIST STS. It also uses expensive arithmetic operations like modulus and division, which can cause problems in lightweight IoT devices like WISP.

Tomassini et al.'s work generates pseudorandom sequences directly by using a cellular automaton (CA) method, instead of generating a deterministic PRNG (Tomassini, Sipper, Zolla, & Perrenoud, 1999). GA in this work is used for producing the suitable CA. This GA evolves the cells of a single, non-uniform CA, which has cell rules initialized randomly. This CA generator achieved good results within the Diehard test suite.

In Chlumecky et al.'s study, a methodology and a software using a GA and a generated hydro random number generator for the optimization of the rainfall-runoff model are introduced (Chlumecký, Buchtele, & Richta, 2017). Their new hydro random number generator generates random numbers using hydrological data and

gives better results than the PRNGs, as they claimed. Nevertheless, this paper does not include structural and statistical analysis of the HRNG's security.

Çabuk et al. in their work (Çabuk, Aydın, & Dalkılıç, 2017), proposed a new xorshift-based lightweight PRNG, called xorshiftR+, which is specially designed for the constrained devices of the IoT and tested on WISP. The authors have also obtained very satisfying results (in terms of speed and randomness) from the NIST STS and a more comprehensive suite, called TestU01. Nevertheless, they have not strictly followed the regulations of the EPC standards, like EPCGen2, which are binding for RFID-equipped devices.

Ibrahim and Dalkılıç focused on the security of healthcare systems, especially authentication protocols in their article (Ibrahim & Dalkiliç, 2017). In that work, a new mutual authentication protocol is developed using elliptic curve cryptography and advanced encryption standard (AES) for RFID tags. This protocol was tested and coded on WISP passive RFID tags. Although they claimed that it overcomes the security problem for the healthcare systems using RFID tags, they used WISP5 built-in random number generator for their authentication protocol. RNGs should be examined structurally and statistically to say that they are secure, and they did not give any information about the security status of that PRNG.

Knežević, in his study (Knežević, 2017), stated that the evolutionary computation can be used to create pseudorandom number generators, which has an important role against side channel attacks. Knežević concluded that the genetic programming method can speed up exponential computation or can be used to break the Merkle-Hellman cryptosystem in the asymmetric key cryptography and also can create boolean functions or good quality S-boxes for the symmetric key cryptography. In this article, the claims were not directly supported by experimental results, and only the articles that support the assertion were reviewed.

Challa et al. investigated the latest authentication protocols proposed for the implantable medical devices (IMD) and WISP in their study (Challa, Wazid, Das, &

Khan, 2018). In this paper, they gave comparative details considering IMDs' computation and communication costs and functional properties across existing authentication schemes. They called attention to some difficulties that should be lessened for authentication schemes, which are designed for IMDs. Although they mentioned the importance of random number generators in their paper, there is no evaluation about the resource usage and security analysis of those RNGs that have one of the key roles for authentication protocols.

Hernandez-Castro et al.'s work uses the avalanche effect and the strict avalanche criterion to measure the fitness of the PRNGs (Hernández-Castro, Isasi, & Seznec, 2004). It generates an initial set of random numbers with Mersenne Twister generator, and randomly changes a single bit. After that, it calculates the Hamming distance between these values and the output of their PRNG.

Hernandez-Castro and Tomassini's studies produce good results with Diehard, but they do not contain the NIST STS results, which is a more comprehensive package than the Diehard test package (Sýs & Ríha, 2014). Moreover, it is mentioned that only successful results are obtained with statistical tests and the results are not given for the measurements and tests performed on any equipment.

LAMED is another work that also uses strict avalanche criterion as its fitness function (Peris-Lopez, Hernandez-Castro, Estevez-Tapiador, & Ribagorda, 2009). As in our project, a PRNG for low-cost low-power RFID tags is produced considering the EPC standard; and a hardware implementation has been designed to test that PRNG, too. However, instead of running the algorithm on a real hardware, the authors of LAMED preferred to create only a conceptual design.

Khan and Bhatia's work applies GA on a binary sequence that is already generated and selects a key from that sequence (Khan & Bhatia, 2012). Coefficient of correlation calculation is used in the fitness function. Even though it has encouraging results, no known statistical tests are provided along with those results.

Leonard and Jackson's high entropy RNG production with single node genetic programming (SNGP) work is also based on Koza's bit-entropy calculation method (Koza, 1991; Leonard & Jackson, 2015). It also uses the same operator set that consists of arithmetical operators as in Koza's study. SNGP is a form of GP, which uses graphs and dynamic programming methods that improves efficiency. Although the resulting RNG has some good results, it could not pass all of the NIST STS tests.

Picek et al. generated a deterministic RNG function for lightweight devices with the Cartesian genetic programming method (Picek et al., 2016). In this method, expressions represented as an indexed graph. The avalanche criterion and approximate entropy tests are used in its fitness function. This work uses a different GA approach for generating the PRNG, but does not provide any testing results.

2.1.1 HRNG Method

Accelerometers are small and sensitive devices (or parts) those measure the physical vibrations, thus estimate the acceleration (or speed and position in some cases) of themselves and the objects those host them. Using accelerometers as the entropy sources for generating random numbers was used in different studies before.

The research of Voris also uses WISP and its accelerometer to design an HRNG (Voris, Saxena & Halevi, 2011). This research proposes a different approach to this problem while using similar process steps of a typical HRNG.s

Another accelerometer based HRNG system is proposed by (Hong & Liu, 2015). This work uses three steps to post-process the raw accelerometer data. First step cleans out the predictable patterns. Second step transforms the bare spots where pulses do not exist. Finally, in the third step, a PRNG is used to generate longer sequences of random numbers. This method generates good results that pass the NIST randomness test suite, but uses Fourier transforms which is not appropriate for lightweight devices used in the IoT, such as the WISP devices.

CHAPTER THREE
PRODUCING THE PRNG WITH GENETIC PROGRAMMING

Producing a PRNG with GP is tricky, because there is no certain testing method that ensures that a PRNG certainly has high quality. There are a vast number of statistical tests those determine different means of quality of the generated sequences from a particular PRNG. Because of that, different fitness calculation methods according to these tests are tested in previous works.

Shannon entropy calculation used in this work is one of these suitable statistical testing methods (Shannon, 1948). Binary sequences with higher entropy are statistically well distributed and tend to appear more random.

GA is a suitable method for generating a PRNG; since bunches of mathematical/logical expressions that can be used as PRNG functions, are essentially elements of a set with infinite members. Selecting a suitable PRNG function from that set can be done using a heuristic method such as GA.

3.1 Starting Population

GA starts working with an initial population consisting of randomly generated individuals (chromosomes). In this work, each individual is a mathematical prefix expression and each token of these prefix expressions are the genes of GA. The prefix form is used, because it is easier to store and to make operations on.

The optimal value for the population size is determined as 50; because different population sizes, which are bigger than 50, are experimented in the GA, but no positive or negative effect is observed on the results. Using a bigger population size also increases the running time of the GA.

For generating the initial population, an operator set must be determined. The operator set used is given in Eq. (3.1);

$$O = \{+, \&, |, \wedge, >, !\} \quad (3.1)$$

The operators are as follows; the addition, bitwise and, bitwise or, bitwise exclusive or, left rotation (left circular shift) and unary bitwise not operators. Other mathematical operators like; multiplication, division and modulus have higher costs and even are not implemented for lightweight devices like WISP, so these operators are not appropriate to be used.

There are also integer numbers and input parameters that are used in the expressions. Input is indicated with letter J. Integer numbers are selected between 1 and 16 to add variety to the generated PRNGs. We selected these numbers after testing a bigger range of numbers. Increasing the range of the numbers does not affect the results significantly.

3.2 Genetic Algorithm Steps

The first step in the genetic algorithm is the tournament selection (Miller & Goldberg, 1995). The tournament selection picks specific amount of individuals from the population randomly and selects the fittest of the picked individuals. In this work, 5 individuals are picked for the tournament selection. Since the population size of the GA is 50, 5 is an appropriate number for selecting one individual from that relatively small population. After two tournament selection operations are applied to the population consecutively, two individuals are gathered as the result.

These two individuals picked by the tournament selection are the inputs of the single-point crossover operation. For the single-point crossover, the tree representations of two expressions from Figure 3.1, exchange parts from randomly selected operator nodes, one from each of them. Basically, they exchange a subtree. The fitter of the new two individuals, which has the higher entropy, is selected as the result. Only the selected operator nodes are swapped between expressions, so the total token count does not change. On this basis, two expressions, Eq. (3.2) and Eq. (3.3),

are given to the crossover operation as inputs, the tree representations of these expressions given in Figure 3.1 are formed.

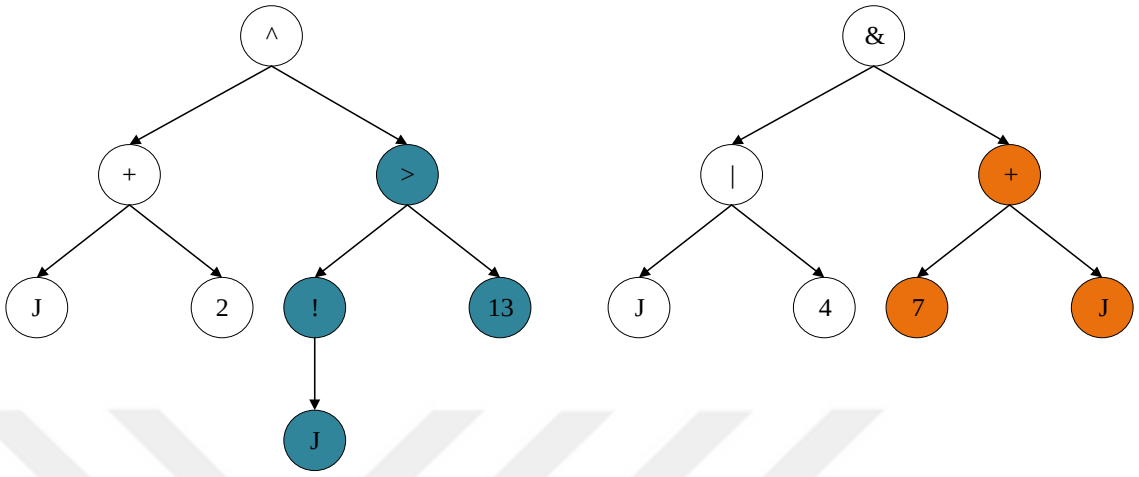

Figure 3.1 Tree representations of the expressions before the crossover operation

```
^ + J 2 > ! J 13
```
(3.2)

```
& | J 4 + 7 J
```
(3.3)

If “>” node of the first tree and “+” node of the second tree are selected as the crossover points, after the crossover operation, these two expressions become like in Eq. (3.4) and Eq. (3.5). The new trees are given in Figure 3.2.

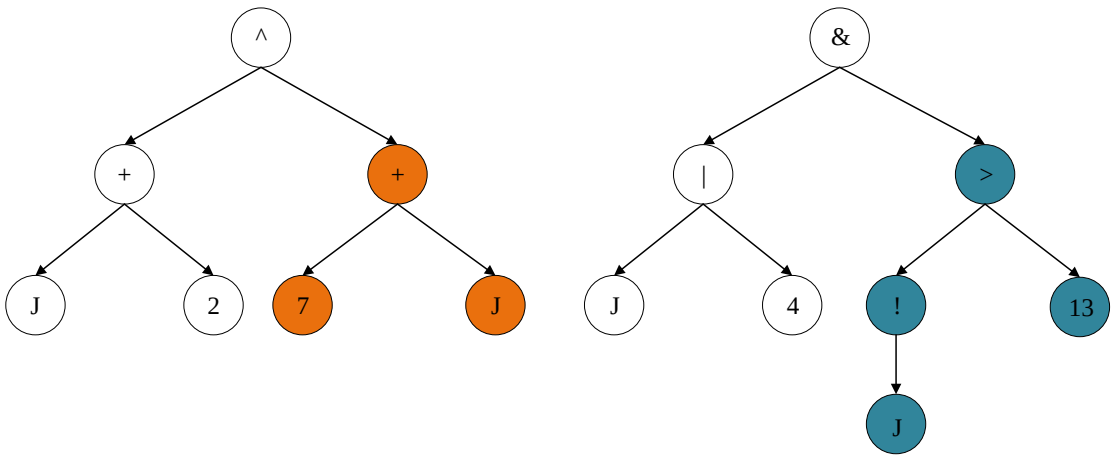

Figure 3.2 Tree representations of the expressions after the crossover operation

```
^ + J 2 + 7 J
```
(3.4)

```
& | J 4 > ! J 13
```
(3.5)

Maximum token count of the expression produced by the crossover operation is limited to 100. That means, there cannot be an expression that has more than 100 tokens as a result of a crossover operation.

After the crossover operation returns a new individual, the mutation operation is applied. The mutation operation only changes one operator of the expression with a probability of 5%. The rate 5% is selected experimentally, but in GAs, the mutation probability is generally low, and is set just to add some variety to the populations. For example, if a binary operator is selected as a target of the mutation operation, it is replaced with a new randomly selected binary operator. Finally, this individual becomes a new member for the next population.

Also, the fittest individual is preserved and remains unchanged in the next generation. This method is known as elitism in GP. The pseudo code of the genetic algorithm steps is indicated below;

```
do
   initialize newPopulation
   Elitism(population, newPopulation)
   for i <- 0 to populationSize do
     firstSelection <- TournamentSelection(population)
     secondSelection <- TournamentSelection(population)
     newIndividual <- Crossover(firstSelection, secondSelection)
     Mutate(newIndividual)
     newPopulation.Add(newIndividual)
   end for
   fittest <- newPopulation.fittest
 while fittest.fitness <= 18
```

3.3 Fitness Method

The fitness function determines/decides how good the solution is. In our case, the PRNG (individual) with the best entropy value considered as the fittest. The Shannon entropy method is used for calculating the entropy of the generated binary sequences (Shannon, 1948). It basically calculates the minimum number of symbols to encode the data as in data compression. The formula of Shannon entropy is shown in Eq. (3.6);

$$H = -\sum_{i=0}^{N} p_i \log_2 p_i \tag{3.6}$$

H value is separately calculated for every n-bit entropy value. The logarithmic base is 2, because the data is encoded into binary sequences. pi is the frequency of the specific character in the binary sequence. Assuming that we want to calculate the 2-bit entropy, first, all possible 2-bit characters' (00, 01, 10, 11) frequencies in the binary sequence are to be calculated. Likewise, 1-bit entropy calculates the frequencies of 0s and 1s. After these p values (frequencies) are obtained, H is calculated per to the formula. Since the frequencies vary between 0 and 1, we add a minus sign to the formula to negate the negative result.

In our GA's fitness function, 1-bit to 8-bit and 16-bit entropies are calculated, divided by its n-bit entropy value and then summed. For instance, 8-bit entropy value is divided by 8, which results 1 at maximum. The purpose of this division is that every n-bit entropy value should weight the same in the final fitness value. When they are between 0 and 1, each n-bit entropy value has the same effect on the final fitness value. So, the maximum entropy value that an individual can get is 9, because every n-bit entropy test returns 1 at maximum. The fitness function calculates the score over two sequences from two different seeds, so the maximum fitness value of an individual becomes 18.

For calculating the fitness of an individual, a pre-defined seed value is to be given and it generates 2^{18} values. The input and output values of the expressions are 64-bit

signed integers. Each output is used as the next step's input, which is similar in other PRNGs. The 2^{18} integers' entropy is calculated for two different seed values and the fitness of the individual is finally determined.

CHAPTER FOUR
DISCUSSION FOR PRODUCED PRNG AND GP METHOD

The fittest expression produced with this experiment is shown in Eq. (4.1) in the prefix form;

```
 + J + + J + > + J + ^ ! J 16 ! + J > J 6 11 ! + + + + J
+ J + J + + J + J + > J 6 ! ! + J > J 6 ! ^ ! J 16 ! > J 4
+ J + + J + J + > + ^ > J 6 14 ^ ! J 16 11 ! + J > J 6 ^ !
J 16 ! + J > J 6 ! ^ > J 6 16
```

(4.1)

C implementation of that PRNG algorithm is given below;

```
int64_t PRNG(int64_t J)
{
  int64_t a1 = (J >> 6) | (J << 58);
  int64_t a2 = (J >> 4) | (J << 60);
  int64_t a3 = J + J;
  int64_t a4 = ~J ^ 16;
  int64_t a5 = J + a1;
  int64_t a6 = J + a4 + ~a5;
  int64_t a7 = (a6 >> 11) | (a6 << 53);
  int64_t a8 = (a1 ^ 14) + a4;
  int64_t a9 = (a8 >> 11) | (a8 << 53);
  return ~(a1 ^ 16) + a3 + ~(~a2 + a3 + ~a5 + a3 + a3 + ~a4 + a5
 + a5 + a6 + a9) + a7;
}
```

This expression has 98 tokens, where 55 of them are operators, and it has a fitness value of 17.99429. It takes 11 generations in the GA to obtain this solution. However, it is experimented that the expressions with much higher entropy do not necessarily have good NIST STS results, when tested. Because of that, even if GA produces expressions with the entropy values higher than 17.999, they are not used.

4.1 NIST STS Results

NIST STS results for a sequence of 16 million bits, which are generated by the newly obtained PRNG are given in Table 4.1. As shown in the table, the produced PRNG successfully passes all of the NIST STS tests. This indicates that the resulting sequence is statistically well distributed, so that it provides the feeling and functionality of a random sequence at some extent (Rukhin et al., 2010). The bit-stream count property of NIST STS is given as 32 in this application. The proportion values mean the ratio of the bits in the 32 bit-streams that passed the test. NIST STS evaluates 29 bits and above as successful. That means proportions which are bigger than 29/32 (0.90625) are considered as random by NIST STS.

Table 4.1 NIST STS results for the proposed PRNG

Test	p-value	Proportion
Frequency	0.804337	1.00000
BlockFrequency	0.299251	0.96875
CumulativeSums	0.671779	0.96875
CumulativeSums	0.949602	1.00000
Runs	0.253551	1.00000
Longest Run	0.804337	1.00000
Rank	0.534146	1.00000
FFT	0.739918	1.00000
NonOverlappingTemplate (Avg.)	0.417842	0.98649
OverlappingTemplate	0.054199	1.00000
Universal	0.949602	1.00000
ApproximateEntropy	0.602458	1.00000
RandomExcursions (Avg.)	0.017591	0.97917
RandomExcursionsVariant (Avg.)	0.032163	0.97839
Serial	0.739918	1.00000
Serial	0.671779	1.00000
LinearComplexity	0.468595	1.00000

4.2 ENT Results

Another pseudo-randomness testing software is ENT Test Suite that is defined and detailed on www.fourmilab.ch/random. ENT applies a set of statistical tests to an output file generated by the subject PRNG. In Table 4.2, the three ENT test results for 128 MB output files that are generated from randomly selected seed values are given.

Table 4.2 ENT test results for the proposed PRNG

Test	**2,121,363,896 (1st seed)**	**897,893,119 (2nd seed)**	**1,732,455,681 (3rd seed)**
Entropy	7.999999 bits/byte	7.999999 bits/byte	7.999999 bits/byte
Compression Rate	0%	0%	0%
X^2 Statistic	276.17 (17.31%)	255.47 (48%)	255.1035 (48.64%)
Arithmetic Mean	127.4947	127.5085	127.5012
Monte Carlo π	3.141592609012 (0.00% error)	3.141444551072 (0.00% error)	3.141423987469 (0.01% error)
Serial Correlation Coefficient	-0.000100633592	-0.000123993062	-0.00000661128

The entropy result can be 8 at maximum, because each character is encoded in 1 byte that stores 8 bits. Since the calculated entropy values are high, as seen on Table 4.2, the compression rate is 0%, which is a good result. ENT Test Suite gives the entropy results with a maximum precision of 6. Since all of the results are very close to 8, they are represented as 7.999999 in Table 4.2. ENT Test Suite calculates the chi-square distribution of the binary sequence, too. The chi-square rate indicates how frequently a random binary sequence would exceed the calculated value. The calculated rate is expected to be in the range of 10% to 90%, which is considered as successful.

The arithmetic mean is calculated by dividing the sum of all the bytes generated by file length. The perfect score is 127.5 so; the generated PRNG has successful results for that value. Monte Carlo Value for Pi test returns a result that converges to a Pi value. The resulting values are close to Pi with a very small error rate between 0% and 0.01%, which is a good result, too.

The serial correlation coefficient shows how much a byte in the sequence depends on the previous bytes. It is expected that this value should be close to zero, while can be either negative or positive. As in Table 4.2, the generated PRNG has serial correlation coefficient values that are close to zero.

4.3 EPCGen2 Randomness Requirements

While EPCGen2 does not specifically instruct how a PRNG should be implemented, it expects that any suitable PRNG to pass the three statistical tests (EPCGlobal GS1, 2015);

Test 1. The probability of 16-bit random number selected from the generated sequence of the PRNG must be bounded by 0.8/216 and 1.25/216 as shown in Eq. (4.2) and Eq. (4.3).

$$0.8/2^{16} < P < 1.25/2^{16} \tag{4.2}$$

$$0.000012 < P < 0.000019 \tag{4.3}$$

This requirement is basically a 16-bit entropy calculation. Since the generated PRNG outputs are 64-bit numbers, they are divided into four parts to calculate the frequency.

After generating a 512 MB output file, the probability (P) for each number is found between 0.000013 and 0.000017. Since these numbers are within the boundaries, this requirement is satisfied.

Test 2. For 10,000 tags, probability that more than one tag produce the same sequence must be below 0.1%, whenever the tags are energized.

For this requirement, we have to calculate how many different seed values the PRNG can accept. It is calculated as in Eq. (4.4);

$$\left[10000 \times \left(\frac{1}{2^{64}}\right)\right] \times 100 \cong 5x10^{-14} < 0.1\% \tag{4.4}$$

Considering that the same sequence cannot be generated with different seeds, this requirement is satisfied by our PRNG as shown in Eq. (4.4).

Test 3. A 16-bit number selected from the PRNG should not be predicted with a probability higher than 0.025%.

This requirement can be checked with serial correlation calculation of the generated sequence by using ENT test suite. It is tested with 128 MB files generated with the PRNG and all results passed the test with success. The obtained results from different seeds are listed in Table 4.3. The requirement is satisfied since all the sample values are below 0.00025 (0.025%).

Table 4.3 Serial correlation test results of the proposed PRNG

Seed value	Serial correlation coefficient
1,824,581,738	0.000023774318
845,522,866	0.000042487019
290,056,763	0.000071459114
685,445,128	0.000041220041
2,115,072,129	0.000010820175

4.4 Time Constraint of EPCGen2

EPCGen2 standard also has time constraint for generating random numbers. Each tag has 18 milliseconds (ms) for encryption (Feldhofer, Dominikus, & Wolkerstorfer, 2004; Özcanhan & Dalkılıç, 2015). In our benchmark, for the test with the WISP 5 device, 10,000 64-bit number is generated in between 7200 and 7300 ms, which means PRNG is executed 10,000 times. So, a 64-bit number is generated approximately in 0.72 to 0.73 ms, which is clearly acceptable considering the total time slot of 18 ms.

4.5 Statistical Analysis of GP Method

For statistical analysis, 30 PRNGs are produced with genetic programming methods using bit-entropy calculation as the fitness function. The results are examined statistically.

All 30 PRNGs generated by the GA can be seen at Table 4.4 in prefix form. The fastest working expression (21st PRNG function) generated by GA has a total of 13 tokens, numbers and inputs. This result was achieved in the 3rd generation of the GA. The crossover used in the GA is precise at every step. Since the number of individuals in the population is 50, except for the individual protected by elitism, a total of 49 crossover operations are applied for each generation.

Table 4.4 PRNGs generated by GA in a prefix form

PRNG	#
+ J + + J ! > J 4 > + J 4 15	1
+ J + > + > ! & J 12 5 ! & > ^ > ^ J 7 14 & J 3 14 ! & J 12 5 ! > J 13	2
+ J + > ^ | > J 6 4 > ! + J 12 1 12 ! ! J	3
+ J + + J ^ > + > ^ J 8 14 ^ ! J 7 11 ! > ^ ! ! & J 16 | ! J 11 11 | > & J 4 4 & + J ^ > + > ^ J 8 14 ^ ! J 7 11 ! | J 4 8	4
+ J + ^ J 15 + J + + > ! J 16 6 ! > J 15	5
+ J + + J ^ ^ ^ > + ! J 6 9 ^ ! J 8 13 ! ^ J 15 > + J + + J 10 ! > J 10 9	6
+ J + J + ^ + + | + ^ + + J | > J 11 3 > ! J 9 14 9 ! > J 14 4 > ! J 9 14 9	7
^ J ^ > ^ > + + + | & ^ ^ > ^ > ! J 5 ! > J 11 5 ! > J 11 13 & ! J 6 10 10 ^ ! J 14 + J | ! J 12 5 ! > J 11 5 ! > J 11	8
+ J + | + J | & ! J 14 > ^ ^ ! ! J 3 & ! J 3 10 10 + + J 2 ! + ^ + J 2 13 > ! J 13	9
+ J + > + ! J + J + > + & | J 9 10 > ! J 1 9 | J 10 9 | J 10	10
^ J ^ | + ^ | > > + J 1 3 + ! J 4 3 9 > ! J 10 3 ! > ! J 12	11
+ J ^ ^ ^ | + | J 2 8 > ! J 14 12 ! > J 2 ! + J 16	12
+ J ^ | | > + + > > J 9 9 | ! J 13 9 7 + & | | J 2 7 ^ ! J 4 8 3 ! > J 5	13
+ J + > & > + J 16 13 ^ ! J 12 10 + J 1	14
^ J ^ ^ + + + J 9 3 + J > ^ > ^ J 6 & ! J 7 6 ! ^ J 12 12 ! ^ J 6	15
^ J | > & > > & > | J 7 10 ^ ! J 15 2 10 ^ ! J 15 2 > | J 16 9	16
^ J + + + & & J 6 9 + + J ^ J + + + > + J 13 4 + ! J 8 2 ! ^ J 13 8 2 ! ^ J 13	17
^ J ^ + + ^ | ^ > & J 16 2 > ! J 14 7 9 + ^ J > J 11 13 10 ! | J 2	18
^ J + | ^ ^ > J 1 16 > ! J 15 13 ! ^ > ^ | & + J 8 5 + | > + J 2 3 13 ! > ! J 15 3 ! + J 3 > ! J 15	19
^ J > ^ + ^ > J 7 2 ^ ! J 12 16 ! ^ ! J 16	20
+ J + > + ! J 8 16 ! > J 13	21
^ J + > + | & + > J 15 9 > ! J 3 6 ! J 10 ! & J 9	22
+ J ^ | + | > J 6 10 | ! J 14 13 ! + J + > + ! J 10 15 ! > J 13	23
^ J ^ + + ! J ^ J ^ > J 3 ! > J 13 3 ! > J 13	24
^ J > ^ | & + J ^ > + | + | & J 11 13 | ! J 4 14 & ! J 6 3 ! | J 4 9 + ! J 13 5 ! & J 2	25
+ J + + J > | ^ + | J 14 10 ! + J 15 8 ! > J 10 ! > J 10	26
+ J ^ | | + & & + J 11 9 | ! J 8 14 > ! J 12 16 ! + J ^ | ^ > J 13 4 16 ! | ! J 11	27
^ J + J ^ | + & + > J 8 12 9 > ! J 8 7 ! J	28
+ J + > ! J 14 ! ! & | + > + J 5 2 > ! J 3 2 ! & J 15	29
+ J + > & > ^ + | > J 11 15 ^ ! J 3 8 15 ^ ! J 10 10 + J + > J 16 8	30

Table 4.5 NIST STS results of PRNGs generated with GA

		Produced PRNGs with GA														
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
NIST STS TESTS	Frequency	√	√	√	√	√	√	√	√	√	√	X	X	√	X	√
	Block Frequency	X	X	X	X	√	X	√	X	√	X	√	X	√	X	X
	Cumulative Sums (Avg)	√	√	√	√	√	√	√	√	√	√	X	X	√	X	√
	Runs	√	√	√	√	√	√	√	√	√	√	√	X	√	X	X
	Longest Run	√	√	X	X	√	√	√	√	√	√	√	X	√	X	√
	Rank	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	FFT	√	√	X	X	√	√	√	√	√	√	√	√	√	X	X
	Non-Overlapping Template (Avg)	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Overlapping Template	√	√	√	√	√	√	√	√	√	√	√	X	√	X	X
	Universal	√	√	X	X	√	√	√	√	√	X	√	X	√	X	X
	Approximate Entropy	√	√	√	√	√	√	√	√	√	√	√	X	√	X	X
	Random Excursions (Avg)	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Random Excursions Variant (Avg)	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Serial (Average)	√	√	√	√	√	√	√	√	√	√	√	√	√	√	X
	Linear Complexity	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
X - It implies that the corresponding PRNG fails the test on the relevant line of the NIST STS.																
√ - It implies that the corresponding PRNG successes the test on the relevant line of the NIST STS.																

Table 4.5 continues

		Produced PRNGs with GA														
		16	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
NIST STS TESTS	Frequency	√	√	√	√	X	√	√	√	√	X	√	√	X	X	√
	Block Frequency	X	X	√	√	√	√	X	√	√	√	X	√	√	√	X
	Cumulative Sums (Avg)	√	√	√	√	X	√	√	√	√	X	√	√	X	X	√
	Runs	√	√	√	X	X	√	√	√	√	X	√	√	X	√	√
	Longest Run	√	X	√	√	X	√	√	√	√	X	√	√	X	√	√
	Rank	√	√	√	√	X	√	√	√	√	X	X	√	√	√	√
	FFT	X	√	√	√	X	√	X	√	√	X	√	√	X	√	√
	Non-Overlapping Template (Avg)	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Overlapping Template	√	√	√	√	X	√	√	√	√	X	√	√	X	√	√
	Universal	√	√	√	√	√	√	√	√	√	√	X	√	√	√	√
	Approximate Entropy	√	√	√	√	X	√	√	√	√	X	√	√	X	√	√
	Random Excursions (Avg)	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Random Excursions Variant (Avg)	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Serial (Average)	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
	Linear Complexity	√	√	√	√	√	√	√	√	√	√	√	√	√	√	√
X - It implies that the corresponding PRNG fails the test on the relevant line of the NIST STS.																
√ - It implies that the corresponding PRNG successes the test on the relevant line of the NIST STS.																

Furthermore, considering that different results are obtained in each run of the GA that is a stochastic algorithm, the algorithm was run 30 times under the same conditions. NIST test results of the 30 PRNGs produced using the GA can be seen in Table 4.5. 5^{th}, 7^{th}, 9^{th}, 13^{th}, 18^{th}, 21^{st}, 23^{rd}, 24^{th} and 27^{th} PRNGs have passed all NIST

STS tests successfully. 1st, 2nd, 6th, 8th, 19th and 30th PRNGs failed only in one test. 10th, 11th, 16th, 17th, 22nd and 29th PRNGs failed in two tests. The 26th PRNG failed in three tests, the 3rd and 4th generators failed in four tests, the 15th and 28th PRNGs failed seven tests, the 12th, 20th, and 25th generators failed in eight tests and the 14th PRNG failed in nine tests. All of the NIST STS results of the PRNGs generated by the GA can be seen at Table 4.5.

Table 4.6 GP information about 30 PRNGs generated by the GA

		Generation Count	Run Time (ms)	Average Token Count of Last Generation	Total Crossover Count
PRNGs Generated with GA	**1**	5	964722	28	5*49 = 245
	2	6	7021743	42	6*49 = 294
	3	6	7313761	40	6*49 = 294
	4	5	7038213	41	5*49 = 245
	5	5	10057198	22	5*49 = 245
	6	7	13859593	27	7*49 = 343
	7	5	5391627	26	5*49 = 245
	8	9	20987158	60	9*49 = 441
	9	11	25437796	54	11*49 = 539
	10	4	4374916	31	4*49 = 196
	11	3	5005431	29	3*49 = 147
	12	5	6895103	25	5*49 = 245
	13	3	5199664	25	3*49 = 147
	14	4	6988562	25	4*49 = 196
	15	3	5138736	27	3*49 = 147
	16	4	7171769	34	4*49 = 196
	17	6	7293231	31	6*49 = 294
	18	4	7023424	37	4*49 = 196
	19	4	6634589	32	4*49 = 196
	20	7	12257815	38	7*49 = 343
	21	3	4732673	23	3*49 = 147
	22	3	4823904	28	3*49 = 147
	23	3	4923051	25	3*49 = 147
	24	4	6876360	28	4*49 = 196
	25	3	5157252	28	3*49 = 147
	26	4	7086106	28	4*49 = 196
	27	4	7243786	36	4*49 = 196
	28	6	10844778	42	6*49 = 294
	29	3	5098134	28	3*49 = 147
	30	4	5774462	24	4*49 = 196

Table 4.6 shows the number of iterations, the elapsed time for production, the average number of tokens of the last generation, and the total number of crossovers realized for each of the 30 PRNGs produced by the GA. Initial population is considered as the first generation.

The nine PRNGs, passed all NIST tests, and was run on WISP separately. Using each PRNG, we generated 10,000 random numbers and then calculated the time for generation. We repeated this operation 3 times for each PRNG and calculated the average time for 10,000 numbers. At the end of it, we calculated the time for one number generation with dividing the average by 10,000. The estimated time for generating one random number on WISP is shown in Table 4.7 for each PRNG.

Table 4.7 Time for generating one random number on WISP for successful PRNGs

PRNG No	**PRNG**	**T (ms)**
5	J = (J + ((J ^ 15) + (J + ((((~J >> 16) | (~J << 48)) + 6) + (~((J >> 15) | (J << 49))))));	0.25473
7	J = (J + (J + ((((((((((J + (((J >> 11) | (J << 53)) | 3)) + ((~J >> 9) | (~J << 55))) ^ 14) + 9) | (~((J >> 14) | (J << 50)))) + 4) + ((~J >> 9) | (~J << 55))) ^ 14) + 9)))	0.46800
9	int64_t a1 = ((J ^ 3) ^ (~J & 3)); J = (J + (((J + ((~J & 14) | ((a1 >> 10) | (a1 << 54)))) | 10) + ((J + 2) + (~(((J + 2) ^ 13) + ((~J >> 13) | (~J << 51))))));	0.40557

13	int64_t a1 = ((J >> 9) | (J << 55));
	int64_t a2 = ((((a1 >> 9) | (a1 << 55)) + (~J | 13)) + 9);
	J = (J + (((((a2 >> 7) | (a2 << 57)) | ((((J | 2) | 7) & (~J ^ 4)) + 8)) | 3) ^ (~((J >> 5) | (J << 59)))));
		0.44923

18	int64_t a1 = (J & 16); J = (J ^ (((((((a1 >> 2) | (a1 << 62)) ^ ((~J >> 14) | (~J << 50))) | 7) ^ 9) + ((J ^ ((J >> 11) | (J << 53))) + 13)) + 10) ^ (~(J | 2))));	0.42273
21	int64_t a1 = (~J + 8); J = (J + (((a1 >> 16) | (a1 << 48)) + (~((J >> 13) | (J << 51)))));	0.24960
23	int64_t a1 = (~J + 10); J = (J + ((((((J >> 6) | (J << 58)) | 10) + (~J | 14)) | 13) ^ (~(J + (((a1 >> 15) | (a1 << 49)) + (~((J >> 13) | (J << 51))))))));	0.39313
24	int64_t a1 = ~((J >> 13) | (J << 51)); J = (J ^ (((~J + (J ^ (((J >> 3) | (J << 61)) ^ a1))) + 3) ^ a1));	0.31303
27	J = (J + (((((((J + 11) & 9) & (~J | 8)) + 14) | ((~J >> 12) | (~J << 52))) | 16) ^ (~(J + (((((J >> 13) | (J << 51)) ^ 4) | 16) ^ (~(~J | 11)))))));	0.40613
T : Elapsed time for one random number generation on WISP		

Mean, standard deviation and variance results of produced 30 PRNGs are obtained and are given in Table 4.8. As we inspect these results, if the standard deviation value of a test is greater than the mean value of the test, it is observed that the PRNGs generally fail this test. Also, if the means value of a test is greater than standard deviation of the test, that PRNG passes the test.

The fastest of 9 successful PRNGs is 21st that is selected as a result and can be seen at Eq. (4.5) in prefix form.

```
+ J + > + ! J 8 16 ! > J 13
```
(4.5)

This PRNG was run on the computer and WISP. On computer, we generated 250,000 64-bit numbers (approximately 15 megabytes) and tested this output using NIST STS. The results can be seen at Table 4.9.

Table 4.8 Mean, standard deviation and variance of NIST STS results of 30 PRNGs produced by GA

Test	Mean	Standard Deviation	Variance
Frequency	0.27491	0.30514	0.09311
Block Frequency	0.08295	0.18835	0.03548
Cumulative Sums (Average)	0.24625	0.24786	0.06144
Runs	0.23830	0.26629	0.07091
Longest Run	0.29308	0.33755	0.11394
Rank	0.41392	0.28928	0.08368
FFT	0.23435	0.30891	0.09543
Non-Overlapping Template (Average)	0.40638	0.08356	0.00698
Overlapping Template	0.28411	0.27601	0.07618
Universal	0.35656	0.32320	0.10446
Approximate Entropy	0.23997	0.29221	0.08539
Random Excursions (Average)	0.17080	0.16480	0.02716
Random Excursions Variant (Average)	0.17833	0.16325	0.02665
Serial (Average)	0.34666	0.25581	0.06544
Linear Complexity	0.42962	0.24951	0.06226

Table 4.9 Proposed PRNG NIST STS result

Test	p-value	Proportion
Frequency	0.122325	0.90625
Block Frequency	0.534146	0.96875
Cumulative Sums (Average)	0.139393	0.96875
Runs	0.407091	1.00000
Longest Run	0.739918	0.96875
Rank	0.911413	1.00000
FFT	0.299251	1.00000
Non-Overlapping Template (Average)	0.420549	0.98944
Overlapping Template	0.468595	1.00000
Universal	0.178278	0.93750
Approximate Entropy	0.671779	0.96875
Random Excursions (Average)	0.191373	0.97321
Random Excursions Variant (Average)	0.195760	0.97222
Serial (Average)	0.703397	0.98437
Linear Complexity	0.468595	0.93750

CHAPTER FIVE
PRODUCING TRUE RANDOM SEEDS WITH WISP

In this chapter, a hardware random number generator (HRNG) running on WISP is proposed. The accelerometer sensor of WISP is used as the physical entropy source. First, this physical entropy source is harvested for gathering raw data. After that, it is post-processed with de-biasing and extraction methods to provide more uniformly distributed results that can be used in the authentication protocols between an RFID tag and an RFID reader.

5.1 Harvesting Raw Data from Accelerometer of WISP

The accelerometer in WISP provides data as vectors with X, Y and Z components which are floating point numbers. Each X, Y and Z values are converted to their 32-bit representations and after that, LSB of each 32-bit value is taken. These LSBs form the final shape of the harvested data that will go through post-processing stage. Below is the explanation of the processing steps that are used to form the harvested data. Example raw data (X, Y and Z values) collected from accelerometer;

```
X: 76.804; Y: 78.721; Z: 6.77
```

Obtaining each value's binary representation;

```
X: 01000010 10011001 10011011 10100110
Y: 01000010 10011101 01110001 00100111
Z: 01000000 11011000 10100011 11010111
```

Final form of the harvested data is obtained by selecting LSBs from each component's binary representations;

```
011...
```

5.2 Post-processing Methods

Generally, results collected from hardware resources are processed with extractor functions for extracting more entropy from the raw data. These methods shorten the original sequence but provide more uniform sequences and extracts more entropy. Extraction methods mostly applied after harvesting (bit extraction) the raw data (Stipčević & Koç, 2014).

Ad hoc simple correctors like; feeding with linear-feedback shift register (LFSR) or Von Neumann de-biasing can be used for de-biasing the input (Tkacik, 2002; Von Neumann, 1963). Von Neumann de-biasing method processes the input and discards the pairs of “00” and “11”, replaces “10” with “1” and “01” with "0”. While it is very easy to implement Von Neumann de-biasing, a disadvantage of this method is that ¼ of the input bits are reduced in output, on average.

Cryptographic hash functions like message digest 5 (MD5), secure hash algorithm 1 (SHA-1), SHA-2, SHA-256 and SHA-512 can also be used in whitening the harvested data (Stipčević & Koç, 2014). However, WISP does not have any built-in hashing hardware chip. Because of that, executing these hash functions programmatically needs more computing power than other methods so they are inappropriate for WISP. These hash functions can also be programmed as software but with a low power device like WISP, processing those functions takes too much time.

Using extractor functions is another approach to post processing methods. Extractor functions converts long and biased sequences to more uniform but shorter sequences. One example one of them is M. Dichtl’s method of randomness extraction (Dichtl, 2007). This extractor function is given in Eq. (5.1).

$$H(X,Y) = X \oplus (X \ll 1) \oplus (X \ll 2) \oplus (X \ll 4) \oplus Y \tag{5.1}$$

16-bit input is separated into two 8-bit parts named as X and Y. The output of the H function is also 8-bit. So, H function reduces the bit count of the input by 50% in output. This function is used because it is a simple and lightweight extractor function that can work effectively on WISP. Harvested data is given to H function as 16-bit parts. For instance, one can collect long sequences of harvested data. So, every 16-bit subsequence of this data is processed with H function. Consider first 16-bit part of the harvested data is;

```
0010110100001000
```

Then its first 8-bit is X, and second 8-bit is Y. The output of H function, H(X,Y), with this input is;

```
00011001
```

5.3 Methods & Results of HRNG

16,000,000 bits are harvested from collected raw accelerometer data for testing. NIST statistical test suite applies 15 different tests to the given binary sequences. The sequence count used in the tests is 16. That means, for 16,000,000 data, there are 16 sequences each having 1,000,000 bits.

5.3.1 Harvested Raw Data

First of all, NIST STS test is applied to the harvested raw data from the accelerometer. As seen from Table 5.1, raw data results pass only 3 of 15 NIST STS tests. The failed tests are indicated with * symbol near their values.

Table 5.1 Test result of the data without post-processing

Test	p-value	Proportion
Frequency	0.000000 *	0.6250 *
Block Frequency	0.000000 *	0.0000 *
Cumulative Sums (Average)	0.000000 *	0.0000 *
Runs	0.000000 *	0.0000 *
Longest Run	0.000000 *	0.6250 *
Rank	0.534146	1.0000
FFT	0.000000 *	0.4375 *
Non-Overlapping Template (Average)	0.002621 *	0.5287 *
Overlapping Template	0.000000 *	0.0000 *
Universal	0.000000 *	0.0000 *
Approximate Entropy	0.000000 *	0.0000 *
Random Excursions (Average)	---	---
Random Excursions Variant (Average)	---	---
Serial (Average)	0.017587	0.6562 *
Linear Complexity	0.035174	0.9375

5.3.2 Extracting Methods

Harvested data is post-processed with H function that reduces the length of the data by 50%. So, 8,000,000 bits are tested with again 16 sequences and the result is shown in Table 5.2. As can be seen from Table 5.2, the test result is improved significantly with post-processing with H function.

Extracting randomness with hashing functions is also applied to harvested data. SHA-1 is used as the hashing function. As SHA-1 outputs 160-bit results, 320-bit sequences of harvested data are given as input, so the input size is reduced by 50% in output (Barker & Kelsey, 2016a). While the test results also pass all NIST tests, SHA-1 cannot be used in WISP devices because of performance issues.

Another well-known method for de-biasing data is feeding it with the LFSR output. In this method, every 32-bit subsequence of the harvested input data is XORed with

each consecutive output of the LFSR. This method is not considered as an extractor function while it is an acceptable post-processing method. The advantage of this method is that the input data is not reduced in output, so the number generation speed is higher. De-biasing with LFSR method improves the statistical properties of the harvested data significantly. However, it fails block frequency test of NIST STS.

Table 5.2 Test result of the data post-processed with H function

Test	**p-value**	**Proportion**
Frequency	0.066882	1.0000
Block Frequency	0.008879	1.0000
Cumulative Sums (Average)	0.108805	1.0000
Runs	0.066882	1.0000
Longest Run	0.122325	1.0000
Rank	0.002043	0.9375
FFT	0.004301	1.0000
Non-Overlapping Template (Average)	0.116383	0.9886
Overlapping Template	0.004301	1.0000
Universal	0.002043	1.0000
Approximate Entropy	0.213309	1.0000
Random Excursions (Average)	---	1.0000
Random Excursions Variant (Average)	---	1.0000
Serial (Average)	0.281897	1.0000
Linear Complexity	0.035174	1.0000

CHAPTER SIX
CONCLUSION AND FUTURE WORK

There are previous works those generate PRNGs with different GP and fitness methods, but very few of them can practically be used in lightweight devices, like WISP passive RFID tags. In this work, a lightweight PRNG that successfully passes all the tests of the NIST STS is produced with the GP methods. This brand-new PRNG consists of performance-efficient operators, like logical bitwise operators and the arithmetical add.

This study shows that a statistically good PRNG can be generated with GP methods using the entropy calculation as its fitness function. With this entropy calculation fitness method, a good PRNG can be produced in a small number of generations.

Unlike most of the other PRNGs generated in the past, the PRNG generated in this work tested on the real hardware and analyzed in terms of resource and time consumptions. It is also tested with EPCGen2 randomness conditions.

Using this method, 30 PRNGs were generated. These PRNGs use mathematical and logical operators that are appropriate for lightweight devices like WISP. Some of these generated PRNGs have good NIST STS results and pass all of the tests. Some of them failed one or more than one tests. Considering all this information, additional statistical analysis like average, standard deviation and variance of p-values of each NIST test results were calculated and evaluated to understand the produced PRNGs results. The result obtained shows that PRNGs that pass NIST STS can be produced with a GA method that uses bit-entropy calculation as fitness function.

For future works, other statistical properties (like in NIST STS) can be combined with the bit entropy calculation to improve the fitness function and generate better PRNGs.

It is also shown that, the accelerometer data, with some post-processing functions, can reliably be used as an entropy source for generating random numbers via hardware. Post-processing with H function gives the most satisfying NIST statistical test suite results. Unlike many others, it also complies with the tight resource limitations of the IoT devices. Thus, it is found to be a good complement for the accelerometer-based HRNG solutions.

To improve this HRNG; combining the output of other physical sensors, like temperature sensor of WISP, with accelerometer outputs can be tested for improving the speed of the entropy collection. Also, other de-biasing and randomness extraction methods can be applied to the data to observe their effects.

REFERENCES

Barker, E., & Kelsey, J. (2014). Recommendation for random number generation using deterministic random bit generators. *NIST special publication 800-90A.*

Barker, E., & Kelsey, J. (2016a). Recommendation for random bit generator (RBG) constructions. *NIST special publication 800-90C second draft*.

Barker, E., & Kelsey, J. (2016b). Recommendation for the entropy sources used for random bit generation. *NIST Special Publication (NIST SP) - 800-90B*

Bassham, L. E., Rukhin, A. L., Soto, J., Nechvatal, J. R., Smid, M. E. (2010). A statistical test suite for random and pseudorandom number generators for cryptographic applications. *Special Publication (NIST SP) - 800-22 Rev 1a*

Çabuk, U. C., Aydın, Ö., & Dalkılıç, G. (2017). A random number generator for lightweight authentication protocols : xorshiftR +. *Turkish Journal of Electrical Engineering & Computer Sciences*, 1–11.

Chae, H.-J., Salajegheh, M., Yeager, D. J., Smith, J. R., & Fu, K. (2013). Maximalist cryptography and computation on the WISP UHF RFID tag. *Conference on RFID Security*.

Challa, S., Wazid, M., Das, A. K., & Khan, M. K. (2018). Authentication protocols for implantable medical devices: taxonomy, analysis and future directions. *IEEE Consumer Electronics Magazine, 7*(1), 57–65.

Chlumecký, M., Buchtele, J., & Richta, K. (2017). Application of random number generators in genetic algorithms to improve rainfall-runoff modelling. *Journal of Hydrology, 553,* 350–355.

Coron, J.-S., & Naccache, D. (1999). An accurate evaluation of maurer's universal test. *Lecture Notes in Computer Science*, *1556*, 57–71.

Dichtl, M. (2007). Bad and good ways of post-processing biased physical random numbers. *Fast Software Encryption*, 1–13.

EPCGlobal GS1. (2015). EPC^{TM} radio-frequency identity protocols generation-2 UHF RFID specification for RFID air interface protocol for communications at 860 MHz – 960 MHz version 2.0.1 ratified, 1–152.

Feldhofer, M., Dominikus, S., & Wolkerstorfer, J. (2004). Strong authentication for RFID systems using the AES algorithm. *CHES: International Workshop on Cryptographic Hardware and Embedded Systems*, 357–370.

Hernández-Castro, J. C., Isasi, P., & Seznec, A. (2004). On the design of state-of-the-art pseudorandom number generators by means of genetic programming. *Proc. of the IEEE CEC'04*, 1510–1516.

Hong, S. L., & Liu, C. (2015). Sensor-based random number generator seeding. *IEEE Access*, *3*, 562–568.

Ibrahim, A., & Dalkiliç, G. (2017). An advanced encryption standard powered mutual authentication protocol based on elliptic curve dryptography for RFID, proven on WISP. *Journal of Sensors*, *2017*.

Jiang, B., Smith, J. R., Philipose, M., Roy, S., Sundara-Rajan, K., & Mamishev, a V. (2005). Energy scavenging for inductively coupled passive RFID systems. *Instrumentation and Measurement Technology Conference, 2005. IMTC 2005. Proceedings of the IEEE*, *2*(May), 984–989.

Khan, F. U., & Bhatia, S. (2012). A novel approach to genetic algorithm based cryptography. *International Journal of Research in Computer Science*, *2*(3), 2249–8265.

Knežević, K. (2017). Combinatorial optimization in cryptography. *40th International Convention on Information and Communication Technology*, 1324–1330.

Koza, J. R. (1990). Genetically breeding populations of computer programs to solve problems in artificial intelligence. *Proceedings of the Second International Conference on Tools for AI*, (1), 819–827.

Koza, J. R. (1991). Evolving a computer program to generate random numbers using

the genetic programming paradigm. *Proceedings of the Fourth International Conference on Genetic Algorithms*, 37–44.

Koza, J. R. (1992). The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems. *Dynamic, Genetic, and Chaotic Programming*, 203–321.

Koza, J. R. (1994). Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, *4*(2), 87–112.

Leonard, P., & Jackson, D. (2015). Efficient evolution of high entropy RNGs using single node genetic programming. *GECCO '15 Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 1071–1078.

Marsaglia, G., & Tsay, L. H. (1985). Matrices and the structure of random number sequences. *Linear Algebra and Its Applications*, *67*(C), 147–156.

Maurer, U. M. (1992). A universal statistical test. *Crypto*, *5*(2), 89–105.

Menezes, A. J., Oorschot, P. C. Van, & Vanstone, S. a. (1997). Handbook of applied cryptography. *Electrical Engineering*, *106*, 780.

Miller, B. L., & Goldberg, D. E. (1995). Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, *9*(3), 193–212.

Özcanhan, M. H., & Dalkılıç, G. (2015). Mersenne twister-based RFID authentication protocol. *Turkish Journal of Electrical Engineering and Computer Sciences*, *23*(1), 231–254.

Park, S. K., & Miller, K. W. (1988). Random number generators: good ones are hard to find. *Communications of the ACM*, *31*(10), 1192–1201.

Peris-Lopez, P., Hernandez-Castro, J. C., Estevez-Tapiador, J. M., & Ribagorda, A. (2009). LAMED - A PRNG for EPC Class-1 Generation-2 RFID specification. *Computer Standards and Interfaces*, *31*(1), 88–97.

Picek, S., Sisejkovic, D., Rozic, V., Yang, B., Jakobovic, D., & Mentens, N. (2016).

Evolving cryptographic pseudorandom number generators. *Parallel Problem Solving from Nature 2016*, 613–622.

Rukhin, A. L. (1997). Approximate entropy for testing randomness. *Journal of Applied Probability*, 37(1), 88–100.

Rukhin, A., Soto, J., Nechvatal, J., Miles, S., Barker, E., Leigh, S. (2010). A statistical test suite for random and pseudorandom number generators for cryptographic applications. *National Institute of Standards and Technology*, 131.

Sample, A., Yeager, D., Powledge, P., & Smith, J. (2007). Design of a passively-powered, programmable sensing platform for UHF RFID systems. *2007 IEEE International Conference on RFID, IEEE RFID 2007*, 149–156.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, *27*(3), 396–399.

Smith, J. R., Sample, A. P., Powledge, P. S., Roy, S., & Mamishev, A. (2006). A wirelessly-powered platform for sensing and computation. *UbiComp 2006: Ubiquitous Computing*, 495–506.

Stipčević, M., & Koç, Ç. K. (2014). True random number generators. *Open Problems in Mathematics and Computational Science*, 275–315.

Sunar, B. (2009). True random number generators for cryptography. *Cryptographic Engineering*, 381–406.

Sýs, M., & Říha, Z. (2014). Faster randomness testing with the NIST statistical test suite. *SPACE 2014: Security, Privacy, and Applied Cryptography Engineering*, *8804*, 272–284.

Tkacik, T. E. (2002). A hardware random number generator. *Cryptographic Hardware and Embedded Systems*, 450–453.

Tomassini, M., Sipper, M., Zolla, M., & Perrenoud, M. (1999). Generating high-quality random numbers in parallel by cellular automata. *Future Generation*

Computer Systems, *16*(2), 291–305.

Turan, M. S., Barker, E., Kelsey, J., McKay, K. A., Baish, M. L., & Boyle, M. (2016). Recommendation for the entropy sources used for random bit generation. *NIST special publication 800-4 90B second draft*.

Von Neumann, J. (1963). Various techniques used in connection with random digits. Içinde *John von Neumann Collected Works*, *5*, 768–770.

Voris, J., Saxena, N., & Halevi, T. (2011). Accelerometers and randomness: perfect together. *Proceedings of the fourth ACM conference on Wireless network security*, 115–126.

Yıldırım, K. S., Aantjes, H., Majid, A. Y., & Pawełczak, P. (2016). On the synchronization of intermittently powered wireless embedded systems. *Hillariously Low-Power Computing - Pushing the Boundaries of Intermittently Powered Devices*, 1–6.

Zaman, J., & Ghosh, R. (2012). *A review study of NIST statistical test suite: development of an indigenous computer package*. Retrieved November 1, 2018, from https://arxiv.org/abs/1208.5740