Quadrivium: A Trivium-Inspired Pseudorandom Number Generator

Latoya Jackson[1] and Yeşem Kurt Peker[2]
(Corresponding author: Yeşem Peker)

TSYS School of Computer Science, Columbus State University[1]
Columbus, GA, USA
TSYS School of Computer Science, Columbus State University[2]
Columbus, GA, USA
(Email: peker_yesem@columbusstate.edu)

(Received July 4, 2018; Revised and Accepted Oct. 28, 2018; First Online Feb. 2, 2019)

Abstract

A pseudorandom number generator (PRNG) is an algorithm that produces seemingly random number sequences. They are employed in applications requiring randomness such as arbitrary sample selection in statistical sampling and secret key generation for ciphers. Where unpredictability is a concern, a cryptographically secure PRNG (CSPRNG) is the only type of PRNGs suitable for such applications. CSPRNGs are specially designed to withstand security attacks. In this paper, after describing a well-known lightweight stream cipher Trivium, we present Quadrivium, a PRNG inspired by the design of Trivium. We compare the statistical properties of Quadrivium by that of Trivium using NIST Statistical Test Suite and Dieharder: A Random Number Test Suite. The analyses show that Quadrivium performs as well as Trivium and has the advantage of producing longer sequences of random bits.

Keywords: Crytographically Secure Pseudorandom Number Generator; CSPRNG; PRNG; Pseudorandom Number Generator

1 Introduction

A random number generator is an object that produces number sequences emulating characteristics of truly random sequences. They are relevant in statistical sampling, Monte Carlo simulation, gaming, internet gambling, cryptography as well as other areas in need of random values. In statistical sampling, generators are used to select arbitrary samples for analysis. Monte Carlo simulation methods employ RNGs to solve optimization, numerical integration and probability distribution problems. Computer-controlled characters and procedural generation in electronic gaming use generators as a source of randomness. Internet gambling as well requires this same type of source to ensure game integrity and combat cheating. Randomness is also implemented to generate secret keys for well-known ciphers such as AES, RSA and Blowfish; It is used to encrypt messages for One Time Pads or to conceal information in protocols by converting the data to seemingly random sequences.

There are two approaches used to generate random sequences. One is a truly random number generator (TRNG), which outputs strings of random quantities using an unpredictable physical source. The other approach is a pseudorandom number generator (PRNG) which uses deterministic methods to generate ”random” sequences. Various methods for generating pseudorandom numbers are being proposed and studied such as [3, 8, 9].

PRNGs are considered more suitable for computing devices in comparison to its genuine counterpart. They are portable; Do not consume a lot of resources (in terms of memory); And operate on a wide range of devices. However, the deterministic nature of the generation process is a concern. PRNGs should be carefully tested to verify that their output approximates a sequence of true random numbers. There is no single test available that can determine if a PRNG generates numbers that have the characteristics of randomness. The best that can be done is to assess a PRNG via a series of tests. A PRNG must perform well on multiple tests to be considered random.

The basic construct of a PRNG is a seed, a generating algorithm and an output. The seed is from a finite set of seeds and is typically a truly random number. It is used to initialize the generator. The generating algorithm has an internal state comprised of all stored values, parameters and variables the generator relies on to function. Additionally, it possesses an update function to refresh the internal state as well as an output function which yields a pseudorandom output. Like the seed, the output is an element of a finite set of possible outputs and an elongated transformation of the seed.

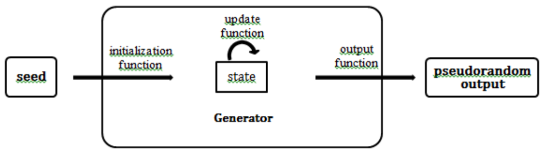

Figure 1: Diagram of a pesudorandom number generator

2 CSPRNGs

2.1 Definition and Properties

In addition to random-like statistical properties, a cryptographically secure PRNG (CSPRNG)—unlike a non-cryptographic PRNG—must possess the property of unpredictability. Unpredictability guarantees pseudorandom values produced by a generator lacks structure, cannot be controlled nor conform to some pattern. Unpredictability does not equate to true randomness. It is another form of randomness that requires high entropy. It is considered more practical than perfect randomness—which is not accessible for all systems. The degree of unpredictability in the "random" generation process directly affects the strength of the cryptographic algorithm. In cases of insufficient unpredictability, generators are susceptible to attacks. By definition, a CSPRNG is unpredictable if the next output value in a sequence is computationally infeasible even if a sequence of previous output values is known [10]. This is formally termed as the next-bit test.

2.2 PRNG Failures

Some PRNG failures are attributed to the lack of entropy or acquisition of entropy within a generating environment. Entropy is a collection of sources employed to seed, and for some PRNGs, update its internal state. Examples of entropy sources include mouse movement, keystroke timing and noise from a computing system's soundcard. Other failures may be based on short periodicity or linearity of the generating function. Low entropy, short periodicity, and linearity in a PRNG facilitate the prediction of the generated numbers in a feasible amount of time. As a consequence, the generator becomes vulnerable to attacks.

Debian experienced a security breach with its OpenSSL distribution. The pseudorandom generator included in the implementation was incapable of acquiring high levels of entropy. This caused the PRNG to produce 32,767 private keys. The small key space ensue highly predictable keys. Other Debian-based products, like Ubuntu, were affected by this PRNG failure.

Another case involves the internationally used MIFARE Classic chip. It has applications in contactless smart cards and proximity cards. In a 2008 paper by de Koning Gans, Hoepman and Garcia, the researchers were able to recover keystreams, read memory blocks and modify memory blocks from the chip. This was all due to the low entropy collecting PRNG implemented in MIFARE [5].

Security Socket Layer (SSL) uses a PRNG to generate a random key. The key is used in a cryptographic algorithm to encrypt information flowing between client and server. Netscape utilized its own implementation of SSL to protect transmission of sensitive data over its browser. However, two computer science students were able to decipher encrypted messages sent over Netscape Web by exploiting the flaws in the PRNG used in the Netscape SSL implementation. The flaw was due to poor seeding of the generator. Even though unique, the seed values taken from the running system (process ID, parent process ID and time of day) were predictable. Hence, the key was retrievable as well as the messages [6].

Shortly following a publication which analyzed the security of popular SecureRandom constructs, a Bitcoin incident occurred leaving its Android users vulnerable to theft [7]. The two events are related in that SecureRandom is a special PRNG for cryptographic applications and Android uses it for cryptographic Bitcoin procedures. However, the Android SecureRandom implementation had a bug that caused the generator to yield predictable sequences. The paper revealed how the generator produced colliding values—making the private key recoverable. The paper also discussed the PRNG's defects in entropy collection and the capability to overwrite the seed value.

3 Lightweight Cryptography and Trivium

Lightweight cryptography is a cryptographic protocol or algorithm intended for usage within constrained device networks. Constrained devices are objects that have limited processing power, memory storage capabilities, and power resources. Typically, they do not possess the proper resources needed to employ traditional cryptographic algorithms. There are some cases where traditional algorithms can be implemented but it is accompanied with significant performance reduction. (Performance encompasses power and energy consumption as well as latency and throughput.) Lightweight cryptography provides a solution for the performance-security tradeoff problem that exists for compact devices.

Trivium, designed by Christophe DeCanniere and Bart Preneel, is a stream cipher intended to operate in constrained spaces. It was selected for the eSTREAM portfolio of lightweight stream ciphers for hardware application. Trivium is also efficient in software-based environments. Additionally, it has been designated by International Organization for Standardization (ISO) as a keystream generator for lightweight stream ciphers [1]. Its keystream may be used as a source for pseudorandom bits.

3.1 Structure of Trivium

Trivium can be described as a bit-oriented stream cipher conducting operations at the bit level. The internal state of the cipher consists of three registers totaling to 288 bits. The first register holds 93 state bits, the second holds 84 state bits and the last register holds 111 state bits. The algorithmic component is broken down into two phases, the setup phase and the generation phase (which is also responsible for updating the internal state of the cipher). Trivium takes in a key and IV of 80 bits each and guarantees to generate up to 2^{64} keystream bits [4].

When creating Trivium, the authors had two mandatory specifications the construction must contain. First, the structure must generate seemingly uncorrelated keystreams. Second, the construction must also be efficient such that there is a high throughput of generated keystream bits per cycle per logic gate. The authors referenced the operations of block ciphers as a solution to their specifications. In comparison to stream ciphers, block ciphers are more developed. Many techniques have been uncovered to bolster the efficiency of block ciphers to operate speedily and with low space consumption. Additionally, the security of a block cipher is well researched and understood.

3.2 Trivium's Algorithm

Trivium requires an 80-bit key and 80-bit initialization vector for set up. Initialization begins with the key being copied to the first register. After copying the key to the first 80 slots, the remaining state bits (denoted as s) are set to zero. The initialization vector is then written to the second shift register. The rightmost four bits in this register are set to zero. The last register has all its bits set to zero except for the last three bits; They are set to one. The internal state is refreshed 1152 times to ensure that all bits are influenced by the key and the IV. The pseudocode is given below in (Algorithm 1).

Algorithm 1 Initialization of Trivium

```
 1: Begin
 2: Initialize *registers*.
 3: for i = 1 to 1152 do
 4:     $t_1 \leftarrow s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171}$
 5:     $t_2 \leftarrow s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264}$
 6:     $t_3 \leftarrow s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69}$
 7:     $[s_1, s_2, \cdots, s_{93}] \leftarrow [t_3, s_1, \cdots, s_{92}]$
 8:     $[s_{94}, s_{95}, \cdots, s_{177}] \leftarrow [t_1, s_{94}, \cdots, s_{176}]$
 9:     $[s_{178}, s_{279}, \cdots, s_{288}] \leftarrow [t_2, s_{178}, \cdots, s_{287}]$
10: end for
```

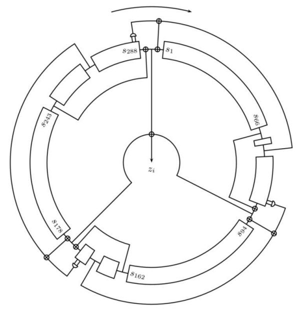

Figure 2: Structure of Trivium

The Trivium generation process actually begins by performing an exclusive or operation on two specific bits from each register. The resulting three bits collectively undergo another exclusive or operation. The result from this last step is a single bit that is added to the keystream. The pseudocode is given below in (Algorithm 2).

Algorithm 2 Generation, Update & Output Algorithm for Trivium

```
m = requested bits
 1: for i = 1 to m do
 2:     $t_1 \leftarrow s_{66} + s_{92}$
 3:     $t_2 \leftarrow s_{162} + s_{177}$
 4:     $t_3 \leftarrow s_{243} + s_{288}$
 5:     $z_i \leftarrow t_1 + t_2 + t_3$
        {Trivium Updates by doing the following:}
 6:     $t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171}$
 7:     $t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264}$
 8:     $t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69}$
 9:     $[s_1, s_2, \cdots, s_{93}] \leftarrow [t_3, s_1, \cdots, s_{92}]$
10:     $[s_{94}, s_{95}, \cdots, s_{177}] \leftarrow [t_1, s_{94}, \cdots, s_{176}]$
11:     $[s_{178}, s_{279}, \cdots, s_{288}] \leftarrow [t_2, s_{178}, \cdots, s_{287}]$
12: end for
```

Trivium produces only a single bit at a time. This entire process is reiterated until the desired length is reached. The pseudorandom output can be simplified to:

$$z_i \leftarrow s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}.$$

The unpredictability of the z_i is dependent on the constant rotation of the state bits and transformation of some bits with each state update. An attacker must be aware of the internal state to accurately predict the next bit. This is quite difficult given that each Trivium state is constructed to be linearly independent.

4 Quadrivium

Quadrivium is a pseudorandom number generator designed with a software implementation in mind. The structure is primarily modeled after Trivium but coalesce findings in [11]; The definition of primitive polynomials; And feedback functions found in linear feedback shift registers (LFSRs).

4.1 Design

Quadrivium is a 384 bit state PRNG. The generator is partitioned into four registers of 98-bit, 97-bit, 95-bit and

94-bit length. It requires a total of 160 random bits for initialization. To understand the design principles of Quadrivium, first, we will review some standard definitions relating to polynomials.

A polynomial p(x) is a mathematical expression consisting of a sum of terms where each term includes x raised to a non-negative integer power and multiplied by a coefficient. It can be written as:

$$p(x) := \sum_{i=0}^{n} a_i x^i. \qquad (1)$$

The variable a_i denotes the coefficient of x_i and an is different from zero. The value of n is the degree of $p(x)$. For this discussion, we restrict all $p(x)$ to polynomials in GF(2). Therefore, all coefficients are either zero or one. A polynomial $p(x)$ is said to be trivial if the degree of $p(x)$ is $-\infty$, indicating a zero polynomial, or 0, a constant polynomial; Otherwise, it is nontrivial. A polynomial $p(x)$ is an irreducible polynomial if it cannot be factored into two or more non-trivial polynomials.

A primitive polynomial is an irreducible polynomial with degree $n > 0$ and a polynomial order (or period) of $2^n - 1$. A linear feedback shift register (LFSR) is an object that employs the function $f : 0, 1^n \to 0, 1$ such that the output bit x_0 is:

$$x_0 = \sum_{i=0}^{n} a_i x^i. \qquad (2)$$

The variable a_i is a coefficient $\epsilon 0, 1$. The feedback function has a period of $2^n - 1$. It is a common practice to employ primitive polynomials as a feedback function.
In [11], the authors noted the following as the active bits in Trivium:

$$\{s_{66}, s_{69}, s_{93}\}\ \{s_{162}, s_{171}, s_{177}\}\ \{s_{243}, s_{264}, s_{288}\}.$$

Recognizing that each index is a multiple of 3, these bits can be generalized as

$$\{s_{a_{m_1}}, s_{a_{m_2}}, s_{a_{n_1}}\}\ \{s_{a_{m_3}}, s_{a_{m_4}}, s_{a_{n_2}}\}\ \{s_{a_{m_5}}, s_{a_{m_6}}, s_{a_{n_3}}\}.$$

For the next part of the discussion, we are only concerned with the family of variables $\{m_1, m_2, n_1\}$, $\{m_3, m_4, n_2\}$ and $\{m_5, m_6, n_3\}$. If we consider these variables as powers of x for non-zero terms in a polynomial, we get the following:

$$x^{m_1} + x^{m_2} + x^{n_1} + x^{m_3} + x^{m_4} + x^{n_2} + x^{m_5} + x^{m_6} + x^{n_3}.$$

Let $k \epsilon \mathbb{N}$ and $q(x)$ is a primitive polynomial. A polynomial $p(x)$ is a k-order primitive polynomial if

$$p(x) = (x + 1)^k q(x). \qquad (3)$$

According to [11], Trivium is a 3-order primitive polynomial with

$$q(x) = x^{22} + x^{23} + x^{31} + x^{54} + x^{57} + x^{59} + x^{81} + x^{88} + x^{96} \qquad (4)$$

We were motivated by this definition to extend Trivium to a k^{th} round and use primitive polynomials to select the active state bits in Quadrivium. Our construction differentiates from the one proposed in [11] in two major respects. First, Quadrivium is driven by the principles of PRNGs. This results in a pseudorandom sequence of lesser correlated bits. Second, the active state bits were redefined to be in agreement with the concept of PRNGs. Since Quadrivium takes a PRNG approach, our concern lies in the linearity of the pseudorandom output. We imposed several criteria to be in accordance with this approach. Recall in Trivium that the pseudorandom bit z_i is the sum of state bits 66, 92, 162, 177, 242 and 288.

In our construction, we redefine the active state bits to those responsible for pseudorandom bit z_i. Second, the active state bits must be derived from a primitive polynomial of degree 384. In Trivium, two state bits from each register is used to generate a single bit output. This should be the criterion to restrict the number of active state bits. As a result, the active state bits are s_{49}, s_{98}, s_{147}, s_{195}, s_{243}, s_{290}, s_{337} and s_{384}.

4.2 Algorithm

The initialization procedure, like Trivium, uses an 80-bit key and 80-bit IV. For the first and second registers, the key and IV is copied to the registers, respectively. The third register is filled with zeroes excluding the last three state bits. Those are set to one. The last register is initialized with one-bit bit values except for the last four bits. They are zeroes. Once the registers are loaded, the rotate procedure is executed. The rotate procedure is reiterated for four full cycles (Algorithm 3).

Algorithm 3 Initialization of Quadrivium

```
1:  Begin
2:  Initialize *registers*.
3:  **for** i = 1 to 1536 **do**
4:      $t_1 \leftarrow s_{49} + s_{96} \cdot s_{97} + s_{98} + s_{171}$
5:      $t_2 \leftarrow s_{147} + s_{193} \cdot s_{194} + s_{195} + s_{358}$
6:      $t_3 \leftarrow s_{243} + s_{288} \cdot s_{289} + s_{290} + s_{69}$
7:      $t_4 \leftarrow s_{337} + s_{382} \cdot s_{383} + s_{384} + s_{264}$
8:      $[s_1, s_2, \cdots, s_{98}] \leftarrow [t_2, s_1, \cdots, s_{97}]$
9:      $[s_{99}, s_{100}, \cdots, s_{195}] \leftarrow [t_4, s_{99}, \cdots, s_{194}]$
10:     $[s_{196}, s_{197}, \cdots, s_{290}] \leftarrow [t_1, s_{196}, \cdots, s_{289}]$
11:     $[s_{291}, s_{292}, \cdots, s_{384}] \leftarrow [t_3, s_{292}, \cdots, s_{383}]$
12: **end for**
```

The main procedure for Quadrivium is quite similar to Trivium (Algorithm 4). Unlike Trivium, we did not include the previous values of t_i in the update function to produce the current values of t_i. The inclusion of the values does not necessarily have a negative impact on the generator. The decision to exclude these values was to restrict their influence to only the output bit versus both the output and the new state bits in Trivium. Both Quadrivium and Trivium has a nonlinear internal state

Algorithm 4 Generation, Update & Output Algorithm for Trivium

```
m = requested bits
 1: for i = 1 to m do
 2:     $t_1 \leftarrow s_{49} + s_{98}$
 3:     $t_2 \leftarrow s_{147} + s_{195}$
 4:     $t_3 \leftarrow s_{243} + s_{290}$
 5:     $t_4 \leftarrow s_{337} + s_{384}$
 6:     $z_i \leftarrow t_1 + t_2 + t_3 + t_4$
 7:     $t_1 \leftarrow s_{96} \cdot s_{97} + s_{171}$
 8:     $t_2 \leftarrow s_{193} \cdot s_{194} + s_{358}$
 9:     $t_3 \leftarrow s_{288} \cdot s_{289} + s_{69}$
10:     $t_4 \leftarrow s_{382} \cdot s_{383} + s_{264}$
11:     $[s_1, s_2, \cdots, s_{98}] \leftarrow [t_2, s_1, \cdots, s_{97}]$
12:     $[s_{99}, s_{100}, \cdots, s_{195}] \leftarrow [t_4, s_{99}, \cdots, s_{194}]$
13:     $[s_{196}, s_{197}, \cdots, s_{290}] \leftarrow [t_1, s_{196}, \cdots, s_{289}]$
14:     $[s_{291}, s_{292}, \cdots, s_{384}] \leftarrow [t_3, s_{292}, \cdots, s_{383}]$
15: end for
```

so it is difficult to determine their periodicity. In [4], the authors noted that the period of Trivium is at least $2^{96-3} - 1$. This is under the assumption that the state evolves linearly. For Quadrivium, the period is at least $2^{384} - 1$, given the same assumption. This is based on the fact that the output function is derived from a primitive polynomial.

5 Statistical Testing

Statistical testing is one of the most common methods used to determine the output quality of PRNGs. In this section we provide brief descriptions of the three well-known statistical testing suits; Namely NIST Statistical Test Suite, Diehard Battery of Tests, and the Dieharder Random Number Test Suite.

5.1 NIST Statistical Testing Suite

NIST Statistical Test Suite (STS) for cryptographically secure RNGs and PRNGs is a standard for statistical testing. The suite contains fifteen tests which analyze the quality of a PRNG's output; And determine whether the outputs mimic the behaviors of truly random sequences. Each test uses a test statistic to determine whether to reject the null hypothesis or not. The null hypothesis is the tested sequence is random; It lacks a pattern and portrays irregularity. The alternative hypothesis is the sequence is not random, a pattern was detected therefore it is predictable [2].

The assessments focus on different behaviors which indicate predictability in a sequence; They can be classified into four main types. The first type is frequency tests. They are the Frequency test (Freq), Frequency Test within a Block (Block), Runs Test (Runs), Test for the Longest Run of Ones in a Block (Long). The following two tests, Binary Matrix Rank Test (Rank) and Discrete Fourier Transform Test (FFT), fall under the repetitive patterns type. Non-overlapping Template Matching Test (NOTemp), Overlapping Template Matching Test (OTemp), Maurer's "Universal Statistical" Test (Univ), Linear Complexity Test (LinCom), Serial Test (Serial) and Approximate Entropy Test (AppEnt) are pattern matching types. The fourth type is random walks and consists of Cumulative Sums Test (CuSum), Random Excursions Test (RanEx) and Random Excursions Variant Test (RanExV) [12].

Even though all tests focus on different aspects of an ensemble, there are three assumptions that they all hold about random outputs. These assumptions are taken in consideration when determining the quality of a PRNG's outputs and if they are comparable to a set of truly random sequences. The assumptions are uniformity, scalability and consistency. Looking at a random sequence of length n, uniformity means the occurrence of zeroes should be one-half of the sequence, likewise the occurrence of ones. Scalability determines to what degree is a sequence random. This property also expects that all subsets of a random sequence must also be random. Consistency expresses the behavior of a PRNG. According to the literature, a consistent PRNG will always produce the random sequences of equal quality. It is not necessary to conduct all tests in the suite when analyzing a PRNG. The analyst is responsible for selecting the appropriate combination of assessments used to study a generator [2].

5.2 Diehard Battery of Test

Diehard is a statistical testing suite created by George Marsaglia, who is also the creator of pseudorandom number generator Xorshift. Diehard includes sixteen tests—fifteen personally authored by Marsaglia—that gauge the randomness quality of a generator. The tests require a binary file of at least 80 million random bits as input. The number of bits needed for to execute each test varies.

A majority of the assessments in the suite uses a p-value to determine if a sequence is random. This is similar to the NIST STS which also has a number of tests that rely on p-values to draw a conclusion. In statistics, p-values represent the probabilities that some arbitrary event will occur; Their purpose is to accept or reject the null hypothesis, which is the tested claim. In Diehard, the null hypothesis is the analyzed sequence is random. Tested sequences are acknowledged as random if p-values are not close to zero or one. Contrarily, in NIST STS, the further the p-value is to one, the further a sequence is to being truly random.

Another difference between the two testing suites is the analysis of the results. NIST STS specifies the p-value needed to reject the null hypothesis. Diehard battery of tests is ambiguous and only states that the p-values should be uniform on the set $[0, 1)$.

The names of the exams included in the Diehard battery of tests are Birthday Spacings Test; Overlapping 5-

Permutation Test; Separate Binary Rank Tests for 31x31, 32x32 and 6x8 matrices; Bitstream Test; OPSO, OQSO and DNA (Overlapping Pairs Sparse Occupancy, Overlapping Quadruples Sparse Occupancy and DNA Test, respectively); Separate Count the 1s Test for byte-streams and specific bytes; This is a Parking Lot Test; Minimum Distance Test; 3DSpheres Test; Squeeze Test; Overlapping Sums Test; Runs test—which is a standard test; And Craps Test.

5.3 Dieharder: A Random Number Test Suite

Dieharder is a test pack for random number generators. The suite includes modified tests from Diehard battery of tests, NIST Statistical Test Suite as well as some assessments created by Robert G. Brown, the chief developer of Dieharder. The test suite is an open source project whose purpose is to become a one-stop source for quantifying randomness. The project encourages inclusion of other new testing schemes from other developers. Dieharder is primarily concerned with analyzing the randomness quality and speed of truly random and pseudorandom number generators.

In comparison to STS and Diehard, the suite prefers to examine the actual generator, not a random output file produced by the generator. The reasoning behind this is "perfect randomness is the production of 'unlikely' sequences of random number at an average rate." Looking at the output alone is not sufficient to declare randomness; The likelihood of the sequence as a whole cannot be determined. Even though Dieharder prefers the aforementioned method of testing, it can still accommodate file-based inputs.

In Dieharder, parameters from STS and Diehard are altered so failures are concluded without ambiguity. Moreover, the Diehard tests are improved in three ways. One, assessments that uses KSTEST, Kolmogorov-Smirnov test, imposes a higher default quantity of one hundred p-values. This coincides with Dieharder's aim to determine unambiguous failure. Two, analysts have more control over tests that use samples. Sample sizes are treated as a variable rather than a fixed constant. Three, assessments that employs overlapping techniques on sequences were adjusted to use non-overlapping techniques. Please note that these improvements were made only if it was possible.

There are ten additional tests in Dieharder .They were created by Robert G. Brown and are called RGB. They are the following: Bit Distribution Test, Generalized Minimum Distance Test, Permutations Test, Lagged Sums Test, KSTest (Kolmogorov-Smirnov Test) Test, DAB Byte Distribution Test, DCT (Frequency Analysis), DAB Fill Tree Test, DAB Fill Tree 2 Test and DAB Monobit Test.

6 Results

We employed NIST STS and Diehārder: A Random Number Test Suite to assess the performance of Trivium and Quadrivium. We used Trivium as a benchmark for the performance of the Quadrivium. For all analyses, three different pseudorandom data files from each generator were tested. Each file consisted of 122.88 million bytes. This was determined by the Dieharder test suite which requires about 31 million integers for proper analysis.

6.1 STS Results

All tests in the suite were conducted on each file. For testing purposes the data file was segmented into 700 subsequences, each one million bits in length. The significance level, $\alpha = 0.01$, determined the number of subsequences used. For this level, at least one hundred sequences must be available for testing. The subsequence length was chosen based on the Maurer's "Universal Statistical" Test. This assessment requires approximately 1.4 million bit-long sequences to evaluate a generator correctly. This is the largest quantity amongst all the tests in the suite.

For each STS run, the suite returns twelve values for each test. One value is the proportion of subsequences passing the respective test. Another value is a single p-value of all the p-values determined. The remaining values are the distribution of p-values over ten subintervals on $(0, 1]$. The p-value is used to determine the degree of uniformity amongst sequences. The closer a p-value is to one; The closer it is to perfect uniformity. A p-value ≥ 0.0001 and a proportions value of 0.978 are required to pass a test.

In Tables 1 and 2, 'P-val' denotes the p-value and 'Prop' stands for proportion. The Cumulative Sums test and the Serial Test assess in two directions, forward and backward. In the aforementioned tables, 'F' and 'B' signifies the results for the forward and backward direction, respectively. The Non-Overlapping Template, Random Excursion and Random Excursion Variant Test provide multiple sets of test results. The proportion values shown reflect the average of these tests' results.

Quadrivium outperformed Trivium on the Frequency within a Block, Tests for the Longest Runs, Overlapping Template Matching, Maurer's "Universal Statistical" Test and Linear Complexity Tests. Quadrivium had the highest average proportions for the Frequency within a Block, Tests for the Longest Runs, Overlapping Template Matching, Maurer's "Universal Statistical" Test and Linear Complexity Tests.

6.2 Dieharder Results

Diehard Battery of Tests and RGB tests were conducted under the Dieharder test suite. Each dataset was parsed into unsigned 32 bit integers totaling 30.72 million integers. The suite returned two results: a p-value and an assessment of passed, weak or failed. A weak assessment signifies the p-value ≤ 0.005. A failed assessment signifies the p-value ≤ 0.000001.

Tables 3 and 4 show the assessment counts for all collected data. Even though Diehard and RGB are sets of fifteen and ten tests, respectively, the total assessment counts are greater. This is attributed to the fact that some of the tests are conducted with multiple parameters. One of the RGB tests, Lagged Sums Test, for example, has 33 different variants.

Both generators had one data set that was considered weak for the OPSO test, Trivium dataset 1 and Quadrivium dataset 2. The Binary Matrix Rank Test 32x32 was also a common problem for the generators. Trivium failed this test with dataset 1 and was considered weak for dataset 3 while Quadrivium received a weak assessment for data sets 1 and 2. The other weak assessments are as follows: Trivium dataset 1, Craps 2 test; Trivium dataset 3, Count the ones test for bytes; Quadrivium dataset 2, Runs test; Quadrivium dataset 3, OQSO.

7 Conclusion

In this paper, we presented a revised model of Trivium that focused on improving the selection of state bits. We considered the entire state of Trivium to make improvements versus its individual registers in previous works. We were aware that the unpredictability of pseudorandom sequences is directly correlated to the entire set of state bits selected to yield stream bits and proposed a solution in our model.

The analyses we presented indicates that Quadrivium exhibits more characteristics of uniformity than Trivium. From Tables 1 and 2, we see that Quadrivium has more p-values closer to one than Trivium. Tables 3 and 4 shows us that Quadrivium has a higher passing rate on the Diehard and RGB tests. Given the data from all the tables, we can conclude that Quadrivium consistently performs well on tests that checks for linear complexity, pattern matching and pseudorandomness on a sequence-level.
Future work can be to improve Quadrivium such that it is seemingly random on a bit level. Potential research could also be to determine the period and security of Quadrivium.

Table 1: STS results of Quadrivium datasets

Tests	**Dataset 1**		**Dataset 2**		**Dataset 3**	
	P-val	*Prop*	*P-val*	*Prop*	*P-val*	*Prop*
Freq	0.074	0.986	0.577	0.988	0.009	0.991
Block	0.937	0.989	0.640	0.987	0.702	0.988
CuSum F	0.243	0.988	0.971	0.985	0.486	0.991
CuSum B	0.435	0.983	0.613	0.987	0.458	0.991
Runs	0.011	0.986	0.219	0.985	0.138	0.994
Long	0.211	0.992	0.341	0.994	0.218	0.990
Rank	0.911	0.990	0.426	0.978	0.634	0.995
FFT	0.174	0.981	0.955	0.988	0.013	0.982
NOTemp	—	—	—	0.989	—	0.990
OTemp	0.460	0.986	0.762	0.988	0.534	0.990
Univ	0.511	0.981	0.795	0.985	0.944	0.981
AppEnt	0.893	0.992	0.713	0.990	0.672	0.991
RanEx	—	—	—	0.990	—	0.991
RanExV	—	—	—	0.992	—	0.990
Serial F	0.559	0.987	0.957	0.992	0.622	0.991
Serial B	0.984	0.980	0.308	0.992	0.768	0.990
LinCom	0.229	0.991	0.719	0.992	0.592	0.988

Table 2: STS results of Trivium datasets

Tests	**Dataset 1**		**Dataset 2**		**Dataset 3**	
	P-val	*Prop*	*P-val*	*Prop*	*P-val*	*Prop*
Freq	0.277	0.998	0.291	0.992	0.374	0.994
Block	0.341	0.985	0.722	0.988	0.034	0.991
CuSum F	0.138	0.995	0.634	0.991	0.756	0.995
CuSum B	0.332	0.994	0.899	0.992	0.352	0.991
Runs	0.944	0.988	0.155	0.990	0.450	0.997
Long	0.876	0.994	0.534	0.987	0.348	0.984
Rank	0.210	0.992	0.264	0.991	0.223	0.984
FFT	0.592	0.991	0.158	0.990	0.474	0.980
NOTemp	—	0.989	—	0.989	—	0.983
OTemp	0.323	0.985	0.366	0.990	0.187	0.987
Univ	0.098	0.995	0.795	0.987	0.208	0.992
AppEnt	0.146	0.988	0.705	0.990	0.563	0.990
RanEx	—	0.991	—	0.988	—	0.990
RanExV	—	0.992	—	0.989	—	0.987
Serial F	0.023	0.987	0.352	0.987	0.932	0.985
Serial B	0.453	0.991	0.273	0.984	0.376	0.984
LinCom	0.657	0.994	0.260	0.991	0.000	0.978

Table 3: Diehard tests assessment counts

Assessment	**Quadrivium Datasets**			**Trivium Datasets**		
	1	2	3	1	2	3
Passed	18	16	18	16	19	17
Weak	1	3	1	2	0	2
Failed	0	0	0	1	0	0

References

[1] "Information technology — security techniques — lightweight cryptography — part 3: Stream ciphers,". Tech. Rep. ISO/IEC 29192-3, October 2012.

[2] et al A. Rukhin. "A statistical test suite for random and pseudorandom number generators for cryptographic applications,". Tech. Rep. NIST Special Publication 800-22 Revision 1a, April 2010.

[3] K. Szczypiorski B. Stoyanov and K. Kordov, "Yet another pseudorandom number generator," *Interna-*

Table 4: RGB tests assessment counts

Assessment	**Quadrivium Datasets**			**Trivium Datasets**		
	1	2	3	1	2	3
Passed	46	42	42	36	43	30
Weak	6	9	4	16	6	18
Failed	9	10	15	9	12	13

tional Journal of Electronics and Telecommunications, vol. 63, no. 2, pp. 195–199, 2017.

[4] C.De Canniere and B.Preneel, "Trivium," *Lecture Notes in Computer Science, New Stream Cipher Designs, The eSTREAM Finalists*, vol. 4986, pp. 244–266, 2008.

[5] J. Hoepman G. de Koning Gans and F. Garcia, "A practical attack on the mifare classic," in *Lecture Notes in Computer Science, vol 5189, Smart Card Research and Advanced Applications,(CARDIS 2008)*, pp. 267–288, Egham, United Kingdom, September 2008.

[6] I. Goldberg and D. Wagner, "Randomness and the netscape browser," *Dr. Dobb's Journal—Software Tools for the Professional Programmer*, vol. 21, no. 1, pp. 66–71, 1996.

[7] C. Meyer K. Michaelis and J. Schwenk, "Randomly failed! the state of randomness in current java implementations," in *Lecture Notes in Computer Science, vol 7779, Topics in Cryptology,(CT-RSA 2013)*, pp. 129–144, San Frnacisco, California, USA, February 2013.

[8] Z. El Abidine Guennoun K.Charif, A. Drissi, "A pseudo random number generator based on chaotic billiards," *International Journal of Network Security*, vol. 19, no. 3, pp. 479–486, 2017.

[9] H. Mahmood N. A. Saqib, M. Zia and M. A. Khan, "On generating high-quality random numbers," *Journal of Circuits, Systems and Computers*, vol. 26, no. 2, 2017.

[10] C. Paar and J. Pelzl, *Understanding Cryptography (1ed)*. London: Springer-Verlag Berlin Heidelberg, 2010.

[11] G. Chen Y. Tian and J. Li, "On the design of trivium," *IACR Cryptology ePrint Archive*, p. 431, 2009.

[12] J. Zaman and R. Ghosh, "A review study of nist statistical test suite: Development of an indigenous computer package," *arXiv preprint*, vol. 1208, no. 5740, 2012.

Biography

Latoya Jackson has a B.S. in Mathematics from Morgan State University, Baltimore, Maryland, USA and an M.S. in Applied Computer Science from Columbus State University in Columbus, Georgia, USA. She has special interest in cryptography, information security and algorithm analysis.

Yesem Kurt Peker is an Associate Professor of Computer Science at the TSYS School of Computer Science at Columbus State University in Columbus, Georgia. She holds B.Sc. degrees in Computer Engineering and Mathematics and Master's Degree in Mathematics from Middle East Technical University in Ankara, Turkey, and PhD in Mathematics from Indiana University Bloomington in the United States. Her research focuses on computer and network security, in particular cryptography. She also is interested in computer science and cybersecurity education for college students as well as younger generation.