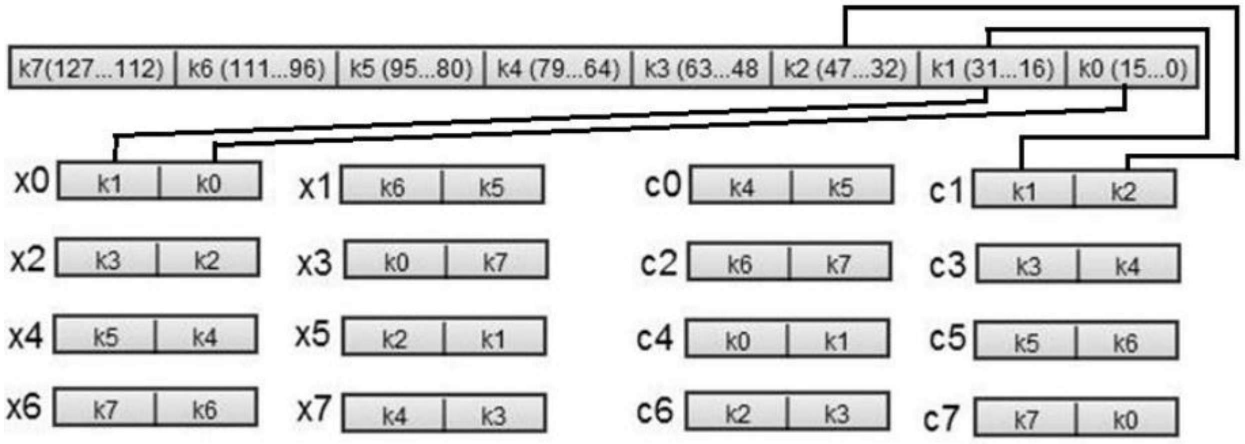
Michael Adams Dmitry Bespalov

Analysis of Rabbit Stream Cipher

I. Algorithm Description

Rabbit is a stream cipher developed by Martin Boesgaard, Mette Vesterager, Thomas Christensen, and Erik Zenner [1]. The cipher takes as input a 128 bit key and (optionally) a 64 bit nonce. The original design specifies 4 initiation rounds to shuffle the key and nonce amongst all of the internal state bits. Once initialized, the keystream is produced in chunks of 128 bits, and encryption/decryption is performed with a simple XOR of the data and the keystream. The internal state of the cipher is comprised of 513 bits. There are eight 32 bit state variables (making up 256 bits), eight 32 bit counters (making an additional 256 bits), and 1 counter carry bit.

The first step of the algorithm is to permute the key bits throughout the state variables and the counters. The key is split up into eight 16 bit chunks as follows: k_0 = bits 15…0 (lowest order bits), k_1 = bits 31…16, and so on, with k_7 = bits 127…112 (highest order bits). Once the key has been split up, the chunks are permuted through the state variables and counters by concatenating two of the chunks (thereby making up the 32 bits of the state variable or counter). The actual permutation is defined by the diagram and equation shown in figure 1.



$$x_{j,0} = \begin{cases} k_{(j+1 \bmod 8)} \diamond k_j & \text{for } j \text{ even} \\ k_{(j+5 \bmod 8)} \diamond k_{(j+4 \bmod 8)} & \text{for } j \text{ odd} \end{cases}$$

$$c_{j,0} = \begin{cases} k_{(j+4 \bmod 8)} \diamond k_{(j+5 \bmod 8)} & \text{for } j \text{ even} \\ k_j \diamond k_{(j+1 \bmod 8)} & \text{for } j \text{ odd.} \end{cases}$$

Figure 1 Permutation of key bits throughout the state variables and counters (x is the state variable, c is the counter).

Now that all of the key bits have been permuted through the internal state, a next-state function is iterated 4 times (the number of rounds) to shuffle the bits throughout the state. The next-state function is highly non-linear. The first step of the next-state function is to increment the counters. This is where the counter carry bit comes in. Each counter is incremented in sequence (i.e. c_0, then c_1, …, then c_7), and the counter carry bit is used in each increment. Thus, the value of the counter carry for the c_1 increment is dependent on the result of the c_0 increment, and so on. On the next increment, the value of the counter carry for the c_0 increment is determined by the result from the previous round’s c_7 increment. The initial starting value for the counter carry is 0. The counter increment is performed following the equations in figure 2.

$$
\begin{aligned}
c_{0,i+1} &= c_{0,i} + a_0 + \mathrm{b} \\
c_{1,i+1} &= c_{1,i} + a_1 + \mathrm{b} \\
c_{2,i+1} &= c_{2,i} + a_2 + \mathrm{b} \\
c_{3,i+1} &= c_{3,i} + a_3 + \mathrm{b} \\
c_{4,i+1} &= c_{4,i} + a_4 + \mathrm{b} \\
c_{5,i+1} &= c_{5,i} + a_5 + \mathrm{b} \\
c_{6,i+1} &= c_{6,i} + a_6 + \mathrm{b} \\
c_{7,i+1} &= c_{7,i} + a_7 + \mathrm{b}
\end{aligned}
\qquad
\begin{aligned}
a_0 &= \mathrm{0x4D34D34D} & a_1 &= \mathrm{0xD34D34D3} \\
a_2 &= \mathrm{0x34D34D34} & a_3 &= \mathrm{0x4D34D34D} \\
a_4 &= \mathrm{0xD34D34D3} & a_5 &= \mathrm{0x34D34D34} \\
a_6 &= \mathrm{0x4D34D34D} & a_7 &= \mathrm{0xD34D34D3}
\end{aligned}
$$

$$
\mathrm{b} = \begin{cases} 1 & \text{if } c_{j,i} + a_j + \mathrm{b} \geq 2^{32} \wedge j > 0 \\ 0 & \text{otherwise.} \end{cases}
$$

Figure 2 Equations for the counter increment operation, where b is the counter carry bit

The incremented value of a counter is equal to the current value plus a defined value plus the counter carry bit. The values for the array ‘a’ are shown in figure 2 as well. These were specified in the original specification for Rabbit. The addition is performed modulo 2^{32}. If the result of a particular increment is greater than or equal to 2^{32}, then the counter carry bit for the next increment is 1. If the result is less than 2^{32}, then the counter carry bit is 0. Once all counters have been incremented, the rest of the next-state function can continue. The original specification provides a diagram showing the working of the next-state function. This can be seen in figure 3. However, the equations in figure 4 are much easier to understand than the circular diagram.

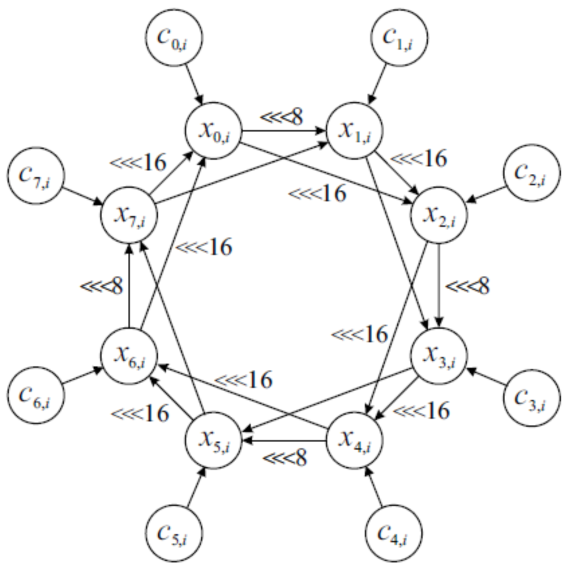


Figure 3 Circular diagram representing the next-state function

$$\begin{aligned}
x_{0,i+1} &= g_{0,i} + (g_{7,i} \lll 16) + (g_{6,i} \lll 16) \\
x_{1,i+1} &= g_{1,i} + (g_{0,i} \lll 8) + g_{7,i} \\
x_{2,i+1} &= g_{2,i} + (g_{1,i} \lll 16) + (g_{0,i} \lll 16) \\
x_{3,i+1} &= g_{3,i} + (g_{2,i} \lll 8) + g_{1,i} \\
x_{4,i+1} &= g_{4,i} + (g_{3,i} \lll 16) + (g_{2,i} \lll 16) \\
x_{5,i+1} &= g_{5,i} + (g_{4,i} \lll 8) + g_{3,i} \\
x_{6,i+1} &= g_{6,i} + (g_{5,i} \lll 16) + (g_{4,i} \lll 16) \\
x_{7,i+1} &= g_{7,i} + (g_{6,i} \lll 8) + g_{5,i}
\end{aligned}$$

$$g_{j,i} = \left((x_{j,i} + c_{j,i+1})^2 \oplus ((x_{j,i} + c_{j,i+1})^2 \gg 32)\right) \bmod 2^{32}$$

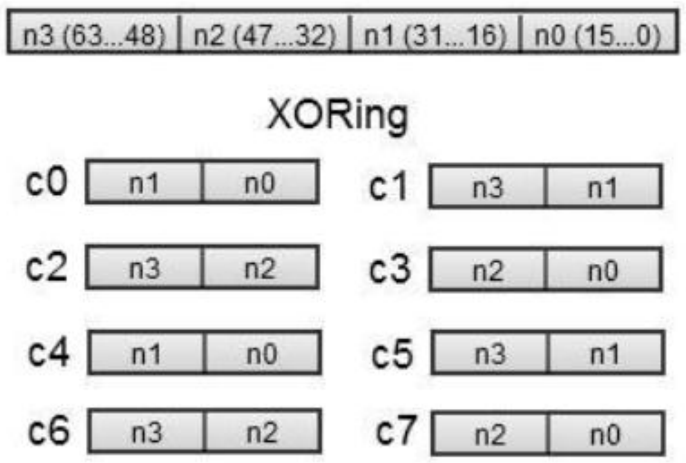
Figure 4 The equations representing the next-state function

Since each particular value of ‘g’ is used more than once, it makes more sense to calculate out all the values of ‘g’ (the final line in figure 4) before iterating over the start variables. The ‘g’ value is determined by adding the current state of the state variable with the incremented counter (hence the reason for performing the increment first) mod 2^{32}, then squaring this value, and XORing the low order 32 bits with the high order 32 bits, and only keeping the resulting low order 32 bits. Once all ‘g’ values have been computed, the state variables are updated by adding together a combination of ‘g’ values, with some being left shifted 8 or 16 bits. This addition is also performed mod 2^{32}. Once all state variables have been updated, the next state function has been completed. Again, this is performed 4 times after the initial key permutation. After all 4 iterations, a final XOR, as shown in figure 5, is performed on all of the counters so that the counter system cannot be inverted to recover the key.

$$c_{j,4} = c_{j,4} \oplus x_{(j+4 \bmod 8),4}$$

Figure 5 The final counter XOR ensures that the counter system cannot be inverted to recover the key. This is performed on all counters (j = 0 to 7)

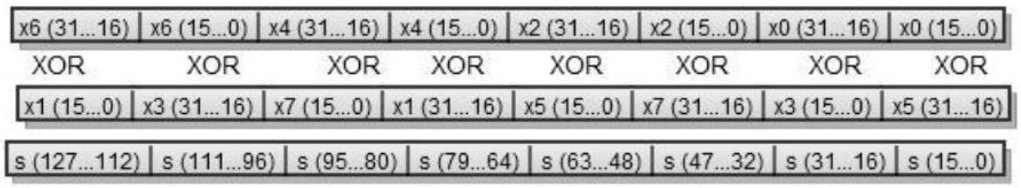
At this point, the internal state is said to be in the master state. If a nonce has been supplied, the nonce setup is now performed on the master state. The 64 bit nonce is split into four 16 bit chunks, with n_0 = bits 15…0 (lowest order bits), and so on with n_3 = bits 63…48 (highest order bits). The chunks are then XORed into the counters as shown by the diagram and equations in figure 6. After the XORs are performed, the next-state function is performed 4 more times (number of rounds) to ensure that the nonce bits have affected the entire internal state. After these 4 iterations, the internal state is in its final state, and keystream extraction can begin.



$$
\begin{aligned}
c_{0,4} &= c_{0,4} \oplus IV^{[31..0]} & c_{1,4} &= c_{1,4} \oplus (IV^{[63..48]} \diamond IV^{[31..16]}) \\
c_{2,4} &= c_{2,4} \oplus IV^{[63..32]} & c_{3,4} &= c_{3,4} \oplus (IV^{[47..32]} \diamond IV^{[15..0]}) \\
c_{4,4} &= c_{4,4} \oplus IV^{[31..0]} & c_{5,4} &= c_{5,4} \oplus (IV^{[63..48]} \diamond IV^{[31..16]}) \\
c_{6,4} &= c_{6,4} \oplus IV^{[63..32]} & c_{7,4} &= c_{7,4} \oplus (IV^{[47..32]} \diamond IV^{[15..0]})
\end{aligned}
$$

Figure 6 The chunks of the nonce (IV) are concatenated and XORed into the counters

The keystream is extracted 128 bits at a time, computed by eight individual 16 bit chunks. Each chunk is the result of XORing 16 bits from one particular state variable with 16 bits from another state variable. For example, the lowest order 16 bits of the keystream is the result of XORing the low order 16 bits of state variable 0 with the high order 16 bits of state variable 5. The entire key extraction is shown in the diagram and equations in figure 7.



$$
\begin{aligned}
s_i^{[15..0]} &= x_{0,i}^{[15..0]} \oplus x_{5,i}^{[31..16]} & s_i^{[31..16]} &= x_{0,i}^{[31..16]} \oplus x_{3,i}^{[15..0]} \\
s_i^{[47..32]} &= x_{2,i}^{[15..0]} \oplus x_{7,i}^{[31..16]} & s_i^{[63..48]} &= x_{2,i}^{[31..16]} \oplus x_{5,i}^{[15..0]} \\
s_i^{[79..64]} &= x_{4,i}^{[15..0]} \oplus x_{1,i}^{[31..16]} & s_i^{[95..80]} &= x_{4,i}^{[31..16]} \oplus x_{7,i}^{[15..0]} \\
s_i^{[111..96]} &= x_{6,i}^{[15..0]} \oplus x_{3,i}^{[31..16]} & s_i^{[127..112]} &= x_{6,i}^{[31..16]} \oplus x_{1,i}^{[15..0]}
\end{aligned}
$$

Figure 7 Key extraction scheme for the 128 bit keystream

The system must call the next-state function in between keystream extractions. Encryption and decryption are performed by XORing the 128 bit keystream with 128 bits of data.

II. Test Suite Description

The NIST test suite [2] was chosen to test the randomness of the keystream produced by our implementation of the Rabbit stream cipher. There were three main reasons for choosing this test suite over the others.

Firstly, there is a wide range of tests in the suite. The NIST test suite includes a battery of 15 randomness tests including both simple and complex tests. For example, the most basic test, which tests the frequency of ones in the sequence, can be used to determine whether more complicated tests, such as cumulative sums test, should even be performed. If a cipher cannot pass the most basic test, then it is likely to fail the more complex tests.

Second, the suite has incredibly comprehensive documentation. The NIST website provides in-depth documentation on the tests used in the suite, with detailed descriptions of each individual test. Moreover, there are presentation slides and papers from several conferences that provide additional detail about the test suite. Altogether, the amount of documentation provides deep insight into the contents of various tests.

Lastly, the suite is developed and maintained by a large and well respected organization that sets industry standards. This means that the suite is mature and already used by many industry developers. The suite was developed in 2009 and uses some tests from other suites (i.e. Diehard), as well as tests specific only to the NIST test suite.

Each test in the suite can be configured with some input parameters and outputs a series of P-values for the set of keystream sequences. The P-value is defined as the probability that a truly random number generator would output a sequence less random than the one being tested. There are a total of 15 tests in the NIST statistical package that can test arbitrarily long binary sequences produced by cryptographic ciphers or pseudorandom number generators for randomness.

Frequency (Monobit) Test

This is the most basic test, upon which the other tests depend. It tests whether the number of ones and zeros found in the sequence are similar to those that would be produced by a truly random number generator. In other words, this test assesses whether the number of ones constitutes close to half of the sequence. The minimum recommended sequence length is 100 bits. If the P-value is less than 0.01, then the sequence is considered to be nonrandom.

Frequency Test within a Block

This test checks the fraction of ones within a block of size M, which should be approximately half of M if the sequence is truly random. If the block size is set to 1, the test becomes equivalent to the Frequency (Monobit) test. The minimum recommended sequence length is 100 bits, and the minimum recommended block size is 20.

Runs Test

This test concentrates on the number of runs in the sequence. A run is defined as a subsequence of ones or zeros of length k. The test tries to find out whether the number of runs in the sequence is similar to one produced by a truly random number generator. Again, if the P-value is less than 0.01, the sequence is considered nonrandom. The minimum recommended sequence length is 100 bits.

Test for the Longest Run of Ones in a Block

The test checks whether the length of the longest run of ones in a block of size M is similar to the length of the longest run of ones in a block of a truly random sequence. The sequence is considered non-random if the P-value is less than 0.01. The minimum recommended sizes for sequence (n) and the size of the block (M) are:

Minimum n	**M**
128	8
6272	128
750,000	10,000

Binary Matrix Rank Test

This test splits the sequence into disjoint matrices of size M x Q and checks its ranks to determine linear dependence in the string constituting the matrix. If the P-value is less than 0.01, the sequence is considered nonrandom. By default, M and Q are set to be 32, and the minimum recommended sequence length is 38,912 bits (38*M*Q).

Discrete Fourier Transform (Spectral) Test

This test computes the discrete Fourier transform of a sequence and checks whether the number of peaks exceeding a 95% threshold is significantly different than the number of peaks exceeding a 5% threshold. If the P-value is less than 0.01, the sequence is considered nonrandom. The minimum recommended sequence length is 1000 bits.

Non-overlapping Template Matching Test

This test runs a sliding window of size m through the sequence to detect a pre-defined aperiodic string pattern of size m. If the pattern is found in the current position of the window, it jumps to the bit following the end of the found pattern. If the P-value is less than 0.01 then the sequence is considered nonrandom. Although m values from 2 to 10 are supported, the recommended size of the window is 9 or 10. No minimum length of the sequence is specified.

Overlapping Template Matching Test

This is the same as the previous except that when the pattern is found, the window jumped only one bit to the right (rather than jumping over the rest of the pattern). If the P-value is less than 0.01, the sequence is considered nonrandom. The minimum recommended sequence length is 1,000,000 bits.

Maurer's "Universal Statistical" Test

This test checks the number of bits between matching patterns in order to determine how much it can be compressed. A highly compressible sequence is considered nonrandom. The test operates on blocks of size L and is initialized with Q such blocks. If the P-value is less than 0.01, the sequence is considered nonrandom. The minimum recommended sequence length is dependent on the block size and initial number of blocks:

n	L	$Q = 10 * 2^L$
≥ 387,840	6	640
≥ 904,960	7	1,280
≥ 2,068,480	8	2,560
≥ 4,654,080	9	5,120
≥ 10,342,400	10	10,240
≥ 22,753,280	11	20,480
≥ 49,643,520	12	40,960
≥ 107,560,960	13	81,920
≥ 231,669,760	14	163,840
≥ 496,435,200	15	327,680
≥ 1,059,061,760	16	655,360

Linear Complexity Test

This test splits the sequence into blocks of size M and finds the length of the LFSR that generates each block. The minimum-length of all of the LFSRs characterizes randomness of the sequence: the shorter the LFSR, the less random the sequence is. If the P-value is less than 0.01, then the sequence is considered nonrandom. The minimum recommended sequence length is 1,000,000 bits, with $500 \leq M \leq 5,000$.

Serial Test

This test checks the uniformity of frequency of all possible overlapping m-bit patterns across the sequence. In the case of m = 1 the test is equivalent to the Frequency (Monobit) test. If the P-value less than 0.01, the sequence is considered nonrandom. The size of m and n should satisfy the following condition:

$$m < \lfloor \log_2 n \rfloor - 2$$

Approximate Entropy Test

This test is almost the same as the previous except it tests the frequency of overlapping blocks of two consecutive lengths (m and m+1), and compares the result against that which a truly random sequence would produce. If the P-value is less than 0.01, the sequence is considered nonrandom. The size of m and n should satisfy the following condition:

$$m < \lfloor \log_2 n \rfloor - 5$$

Cumulative Sums (Cusum) Test

This test concentrates on computing the cumulative sum of adjusted (-1, +1) digits in the random walk over the sequence. If the sum is far from the expected sum of a truly random sequence, the sequence is considered nonrandom. If the P-value is less than 0.01, the sequence is considered nonrandom. The minimum recommended sequence length is 100 bits.

Random Excursions Test

This test is a modified version of previous test. In addition to computing a cumulative sum over a random walk, the test aims to find the number of cycles with exactly K visits in walk. A cycle is defined as a sequence of steps taken at random that begin at return to the origin. If the P-value is less than 0.01, the sequence is considered nonrandom. The minimum recommended sequence length is 1,000,000 bits.

Random Excursions Variant Test

This test tries to find deviations in the number of times that a specific state is visited in a cumulative sum random walk in the range [-9,+9]. If the P-value is less than 0.01, the sequence is considered nonrandom. The minimum recommended sequence length is 1,000,000 bits.

III. Random Input Data Program

The program that was created to generate a keystream sequence is rather straightforward. It takes 5 arguments as input to produce a sequence. It is capable of generating multiple sequences of a specified length using the Rabbit stream cipher with a specified number of rounds and key. Providing a key is optional, and if none is provided then a key of all 0s is used. The parameters of the program are as follows:

Parameter name	**Description**
R	The number of initialization rounds
N	The sequence length (in bytes)
M	The number of keystream sequences to generate
filename	Path to the output file
K	48 Hex characters representing a key. All zeros by default.

After parsing these command line arguments, the program sets the number of rounds as well as the key, and then initializes the Rabbit stream cipher. Next, M sequences of N bytes are generated by encrypting zeroes as the plaintext. The generation is performed byte-by-byte, and after a desired number of bytes (N) are generated, the sequence is written to the output file as an ASCII string of zeros and ones on one line of the file. Until M is reached, a new sequence is generated. At the end, the program closes the file and exits. If an exception occurs, the stack trace is printed to the default error stream.

After the program successfully finishes, the output file contains exactly M keystream sequences of N bytes written as ASCII zeros and ones and separated with new line characters. This output file can now be fed into the NIST Test suite program.

IV. Test Suite Results

1 Round

After running the NIST test suite on 100 keystreams generated by the Rabbit stream cipher reduced to 1 round, it turned out that none of the P-values in any test fell under the 0.01 P-value threshold. To recall, P-value shows the probability that the tested sequence would produce as random a sequence as one generated by a truly random number generator. Therefore, a low P-value would indicate a small chance that the tested sequence is truly random. Thus, our results indicate strong evidence that the Rabbit stream cipher is still random even when reduced to 1 round.

<u>2-6 Rounds</u>

We ran the test suite for keystreams generated by reduced round, default round, and increased round versions of the Rabbit stream cipher and have not found any evidence of non-randomness, as all of the P-values were above the 0.01 threshold. The analysis of uniformity of P-values across all the tests for each round shows that the distribution of averaged P-values is close to uniform (see section below). The results support the claim that the Rabbit stream cipher produces truly random keystreams.

Due to large size of the output tables, we have chosen to only include a small snippet of one of the NIST test suite output files. The rest can be seen in the results files found in our source code archive. The data shown below are some results for the Rabbit stream cipher reduced to 1 round.

```
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
 13   8  10  10   7  12   9  11   6  14  0.739918     99/100     Frequency
  5   7  10  14  11   6  13  11  10  13  0.474986     99/100     BlockFrequency
 10   8  12  12  11   9  11   6  10  11  0.955835     99/100     CumulativeSums
 12   7  15   9   9   6   8  11  11  12  0.678686    100/100     CumulativeSums
 11   9   9   6  12   8  17  10   6  12  0.383827     99/100     Runs
 10   9  10   6   6  16   7  11  10  15  0.319084     99/100     LongestRun
 16  12  10  12   5   9  10   6   5  15  0.137282     99/100     Rank
 13  14   9   8  10  11   4  10  10  11  0.657933    100/100      FFT
```

V. Analysis of Randomness

Based on the results of the NIST statistical test suite, it seems as though reducing the number of rounds has almost negligible effect on the randomness of the keystream. The tests were passed with a high enough proportion (for 100 sequences, at least 95 must pass to not mark an issue) regardless of the number of rounds. Likewise, increasing the number of rounds to 5 and 6 made very little difference in the randomness of the generated keystreams.

The NIST test suite documentation suggests interpreting the test results with a uniformity analysis of the P-value distribution. The review of uniformity consists of evaluating the frequency of P-values from the tests over 10 ranges [0 – 0.1), [0.1 – 0.2), …, [0.9 – 1). The average frequency was calculated and below is the analysis of the distribution for each of the six tests.

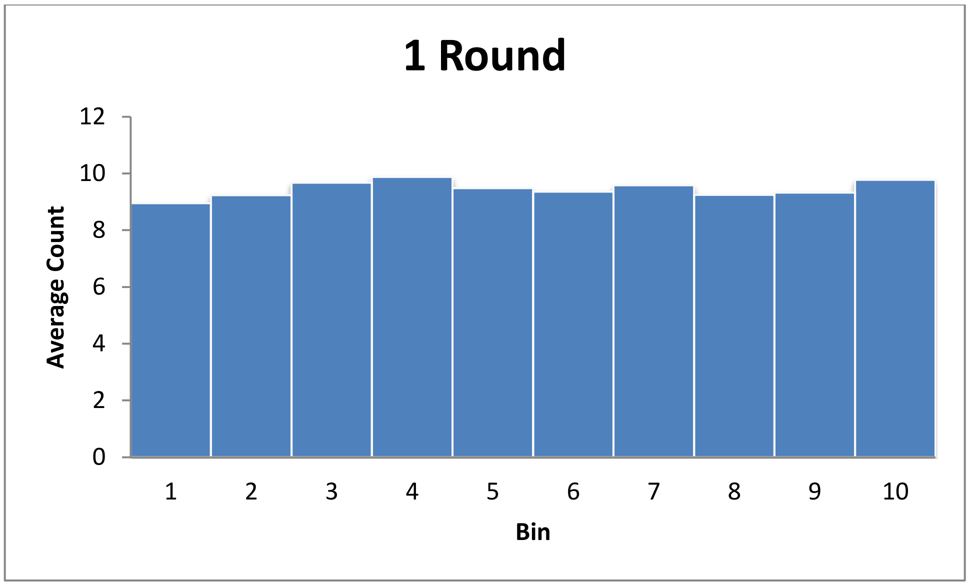


Figure 8 Average number of P-values by their range for Rabbit cipher with 1 round

Figure 8 shows the average number of P-values falling in one of 10 bins for the 1-round Rabbit cipher. The range of n-th bin is [n * 0.1 – 0.1; n * 0.1). For example, there were 9.87 out of 100 P-values falling in bin 4 with range [0.3; 0.4) which is the largest number of P-values over all 10 bins. The smallest average number of P-values, 8.94 out of 100, is in the first bin with range [0.0; 0.1). Overall, the mean number of P-values over all bins is 9.46, with standard deviation being 0.28. It seems to be very close to a uniform distribution.

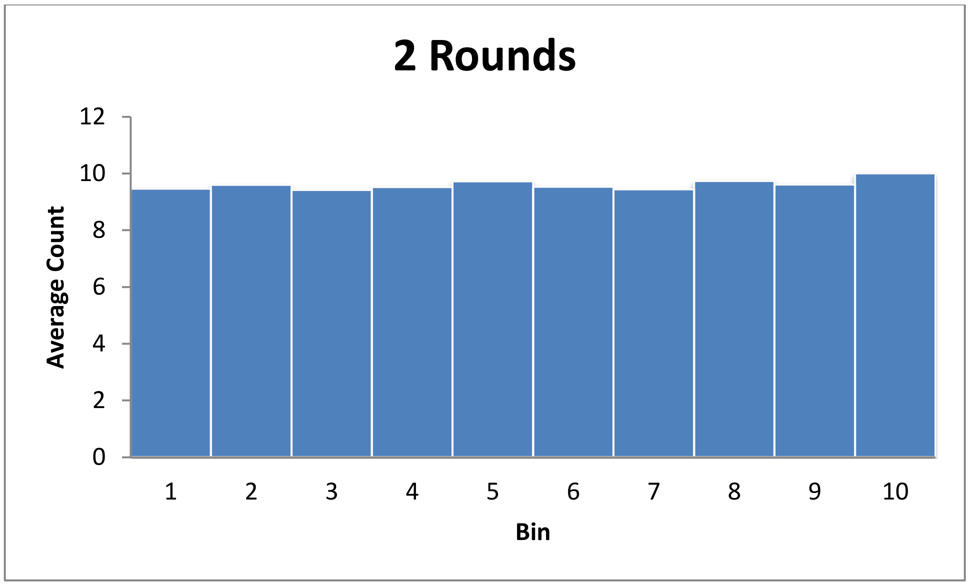


Figure 9 Average number of P-values by their range for Rabbit cipher with 2 rounds

Figure 9 shows the frequency of P-values by their range for a 2-round Rabbit cipher. The number of P-values appears to be slightly more uniform than in 1-round version of Rabbit stream cipher, with higher mean of 9.6 and a standard deviation of 0.18. It is notable that there were 10

P-values for the range [0.9 – 1.0). This result indicates a very slight increase in randomness in the keystreams compared to the 1-round version of the cipher.

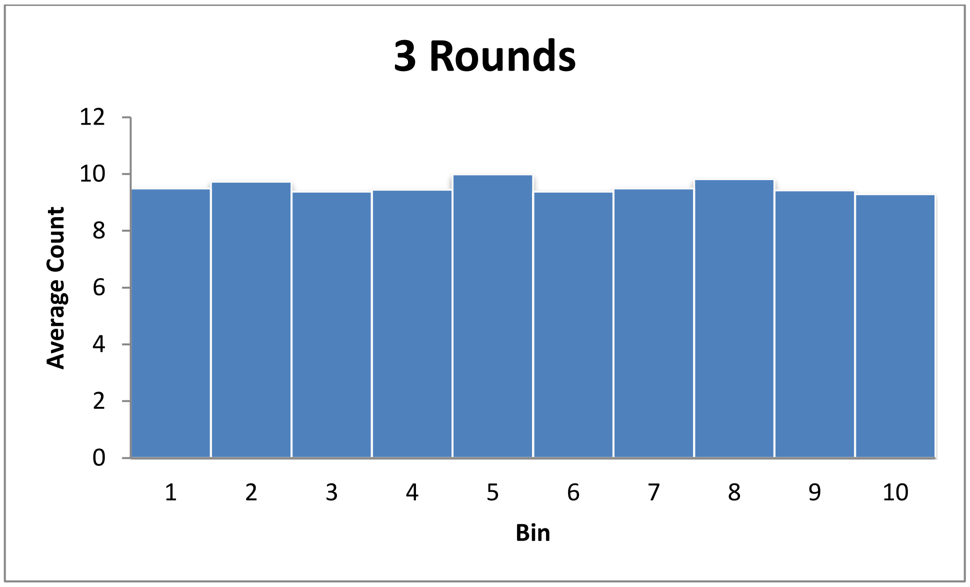


Figure 10 Average number of P-values by their range for Rabbit cipher with 3 rounds

Figure 10 shows the average number of P-values obtained from the NIST test suite for 100 keystream sequences of the 3-round version of the Rabbit stream cipher. None of the bins have less than 9 P-values on average, with mean frequency of 9.54 and standard deviation 0.22. The distribution looks uniform.

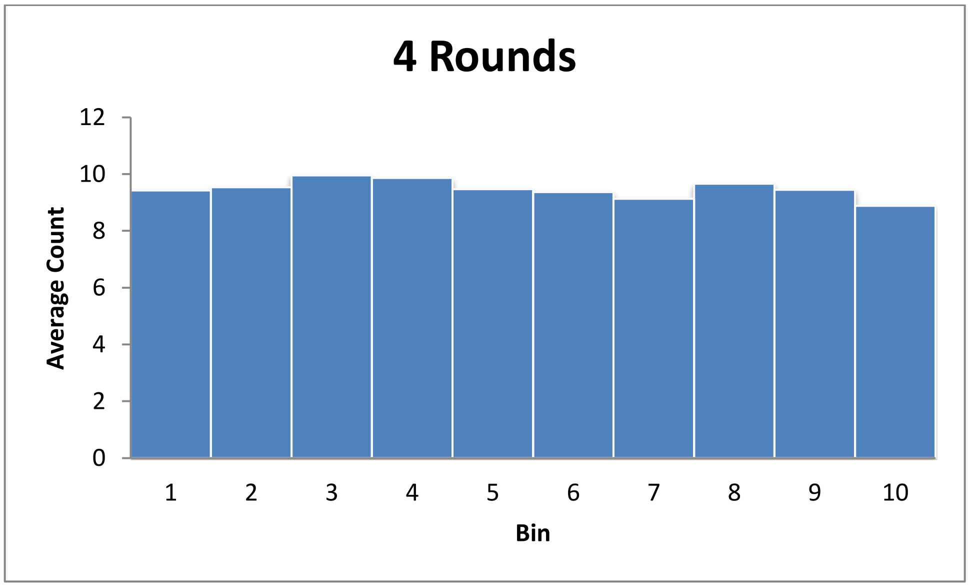


Figure 11 Average number of P-values by their range for Rabbit cipher with 4 rounds

Figure 11 plots P-value counts for the default 4 round Rabbit cipher. Surprisingly, the average number of values looks slightly less uniform than for the 2-round and 3-round versions of the cipher, however, it is still close to uniform. The mean frequency of P-values in this distribution is 9.47 with a standard deviation of 0.32. As it easily seen from the chart, the smallest average number of P-values is in the range [0.9 ; 1.0) and is 8.88. Still, the count of values over the 10 bins looks close to uniform.

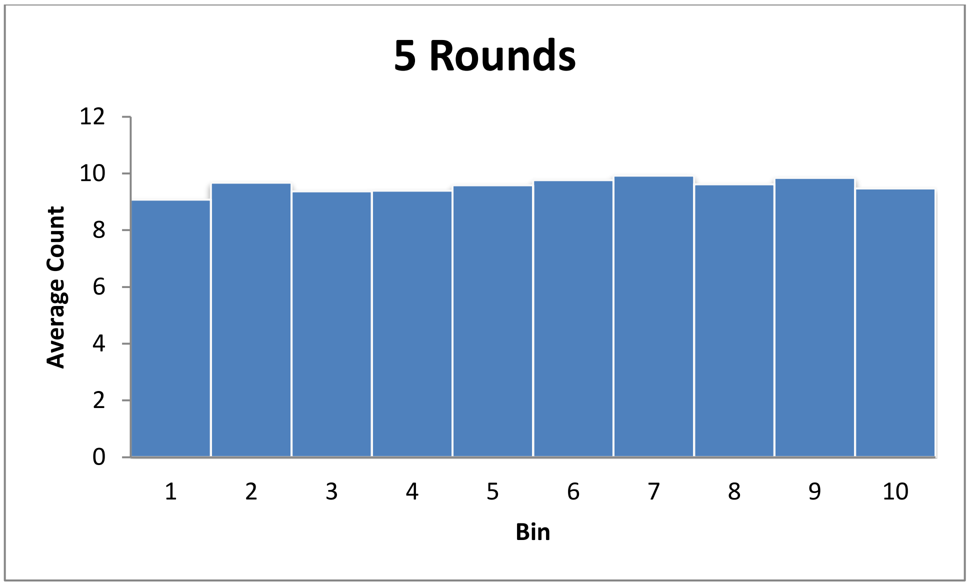


Figure 12 Average number of P-values by their range for Rabbit cipher with 5 rounds

Figure 12 shows the average frequencies of P-values falling in 10 different bins of width 0.1 for a 5-round Rabbit cipher. The chart shows that none of the bins have less than 9 P-values on average. The mean frequency is 9.57 with a standard deviation of 0.25. These results look slightly more uniform than results for 4 rounds of stream cipher. It is possible that increasing the number of rounds helped to eliminate non-uniform patterns from the generated keystreams.

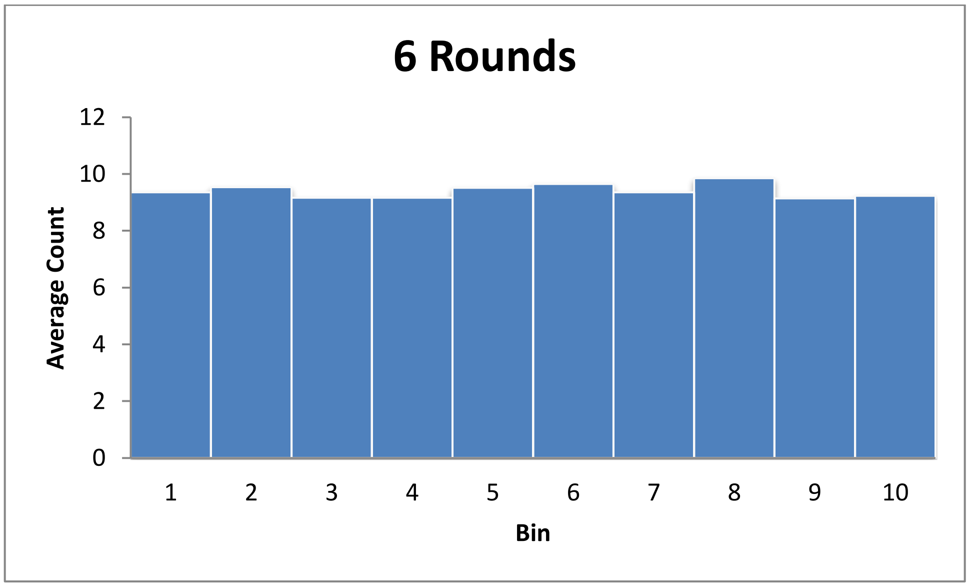


Figure 13 Average number of P-values by their range for Rabbit cipher with 6 rounds

Figure 13 shows the average count of P-values falling in 10 ranges of width 0.1 for a 6-round Rabbit cipher. It can be seen that the distribution is close to uniform, with none of the bins having less than 9 P-values. The distribution has a mean of 9.4 and a standard deviation of 0.24. The results support the randomness of the Rabbit stream cipher keystream sequences.

In conclusion, changing the number of rounds in the Rabbit stream cipher has a negligible effect on the randomness of the generated keystreams. Thus, the Rabbit stream cipher can be claimed as being a truly random cipher.

VI. Analysis of Literature Search

The literature search only turned up one instance of statistical tests being run on the Rabbit stream cipher, and that was found in the original specification paper [1]. The authors performed a variety of tests, including the NIST test suite, the Diehard tests, and the ENT test suite. They ran it both on keystreams and on the internal state, both in its full implementation as well as a version where the internal state and counter variables were reduced to only 8 bits each. Although they did not include any specific details of the test results, they stated that they found no weaknesses in any of their tests. This lines up with our results from the NIST test suite, as none of the tests were failed, regardless of the number of rounds used in the setup. All of the other results of our literature search were attempts at attacking the Rabbit stream cipher, but not in regards to any statistical test suite being performed.

VII. Developer's Manual

To compile our software, both the CSCL library and the Parallel Java library must be included in your classpath. Then, simply compile our three java files (StreamCipher.java, Rabbit.java, and KeystreamGenerator.java) using the javac command. Again, they will not compile if either of professor Kaminsky's libraries are missing from the classpath.

To compile the NIST statistical test suite, start by downloading the test suite archive located on the NIST website[1]. Once downloaded, extract the archive to a folder of your choosing. Browse to the extracted folder and run the 'make' command. This will compile the NIST statistical test suite, at which point it can be used to assess the randomness of our generated keystreams.

VIII. User's Manual

Once compiled, the first step is to generate a file containing a sequence of keystreams. This file will be accessed by the test suite to run statistical tests on. The program that generates a keystream file is KeystreamGenerator. The arguments required to generate a keystream are the number of rounds to perform (Rabbit's default it 4), the length of each keystream in bytes, the number of sequences to generate, and the name of the output file. The user may also provide a key nonce pair to be used if desired, but a key nonce pair of all 0's is the default. The key nonce pair must be 24 bytes (entered as 48 hex characters). The program is run by the following command:

[1] http://csrc.nist.gov/groups/ST/toolkit/rng/documents/sts-2.1.1.zip

```
java KeystreamGenerator <R> <N> <M> <filename> [K]
```

Here, R is the number of rounds to perform, N is the length of each sequence (in bytes), M is the number of sequences to generate, filename is the name of the output file, and K (if provided) is the 24 byte key nonce pair to use. In our tests, we set N to 125000 (producing 1 million bit length sequences) and M to 100.

Once a keystream file has been created, the test suite can be run on it. To start the test suite, browse to the folder where the test suite archive was extracted and compiled. Start the program by running the command:

```
./assess <N>
```

Here, N is the bit length of each sequence (note: this will be 8 times the N provided to the keystream generator). It will then ask which dataset to run tests on. Enter “0” to select an input file, and then enter the path to the keystream file that has just been generated. It will then ask which tests to perform on the file. In our tests, we always entered “0” which will run all tests. If “1” is entered, then it will allow you to choose specifically which tests to run, by entering a “1” in each column of a test that is desired, and “0” in each column of a test that is not desired. It will then ask how many bit streams it will be testing. This will be the value of M that was entered in the keystream generator (in our tests, we generated 100 keystreams). Lastly, it will ask for the format of the input file. The keystream generator outputs a file format of ASCII 0’s and 1’s, so enter “0” to indicate that format. Testing will now commence, and depending on the size of the keystream file, can take some time to complete. Once finished, it will indicate completion, and the resultant file “finalAnalysisReport” can be found in the “/experiments/AlgorithmTesting/” directory. The “experiments” directory will be in the root directory of where the test suite archive was extracted. This file will contain the results of all of the statistical tests. To produce the keystream sequence files we used in our tests, run the batch file found in the source code archive after the code has been compiled.

IX. Lessons Learned

Likely the most useful information we learned was more of the low-level bitwise operations and tricks. Especially with Java, care had to be taken with a lot of the bitwise operations, such as rotations, shifts, casts from one data type to another, and modulus operations, among others. One thing that had held us up for a few days was the fact that, when casting an int to a long, Java will perform sign extension. Thus, instead of getting a long with 32 0s followed by the bit pattern of the int, if the int had a high order bit of 1, the high order 32 bits of the long would also be 1s. This was leading us to incorrect results at first, and learning that we needed to mask off the high order 32 bits after the cast was important, and a valuable lesson. The method for rotating bits was also useful, since Java does not provide an operation to perform rotations.

Using a statistical test suite was also entirely new to us. Downloading, compiling, and running the test suite was all a learning experience. Interpretation of the results, although somewhat intuitive, was also slightly more involved than looking at the surface values.

Another important thing that caught us off guard was a debugging paper by the authors of the cipher [4]. This paper showed examples of keys and nonces, and the internal states at particular times that should result from using them. In one of the examples, they made a small typo in the key (swapping an E and a 3), which resulted in incorrect internal states. However, since the very first state is just the key pieces iterated throughout the system, it did not take too long to realize what the issue was. Changing that one hex character of the key lead us to the correct internal states, and so we concluded that they had made a typo. Thus, we learned to be wary of possible incorrect statements in technical papers.

X. Future Work

For future work, we would like to test the Rabbit stream cipher using other statistical test suites. In this investigation, we only assessed the cipher using the NIST statistical test suite. It would be interesting to find the results of running it through some other test suites, such as Diehard and TestU01, to see if it would pass their battery of tests as well.

We would also like to run the test suite on keystreams that were generated with differing keys. In our research, we chose to use only a key of all 0s. By doing this, we were testing how the actual algorithm was randomizing the keystream, without any randomness coming from our input. Intuitively, it is likely that if the cipher can pass all the tests with a key of all 0s, then it would be likely to pass all the tests with a truly random key.

It would also be interesting to try and optimize it further than we already have. However, considering the optimization efforts we put into the Java implementation, perhaps it would be a good idea to start by trying a C implementation, and checking the performance of that. One way to possibly optimize the run time of the test suite would be to have our Java program output a binary data file instead of a file of ASCII 0s and 1s. Perhaps the test suite can run more efficiently on this data type.

XI. Individual Contributions

Michael:

- Implemented incrementCounters(), updateGs(), nextState(), getKeystream(), encrypt(), and the nonce setup
- Implemented parts of the keystream generator
- Focused on the cipher algorithm parts of the presentation/paper
- Sections I, VI, VII, VIII, IX, X, XI, and XII of the report
- Generated and ran the test suite on some of the keystreams
- Debugging and testing

Dmitry:

- Implemented the key setup
- Implemented parts of the keystream generator
- Researched the bulk of the NIST statistical test suite documentation and interpretation
- Focused on the test suite parts of the presentation/paper
- Sections II, III, IV, and V of the report
- Generated and ran the test suite on some of the keystreams
- Debugging and testing

XII. References

[1] M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen and O. Scavenius, *The Stream Cipher Rabbit*, Presented at Fast Software Encryption Conference, 2003. http://www.ecrypt.eu.org/stream/p3ciphers/rabbit/rabbit_p3.pdf

[2] Rukhin, Et. Al, "*A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications", NIST Special Publication 800-22-1a, April 2010. http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf*

[3] J. Soto, *"Statistical Testing of Random Number Generators,"* Proceedings of the 22nd National Information Systems Security Conference, 10/99. http://csrc.nist.gov/groups/ST/toolkit/rng/documents/nissc-paper.pdf

[4] M. Boesgaard, M. Vesterager, E. Zenner, "*A Description of the Rabbit Stream Cipher Algorithm"*, RFC 4503, May 2006. http://www.ietf.org/rfc/rfc4503.txt?number=4503