CROWN STERLING CRYPTOGRAPHIC SECURITY PROTOCOL

Freedom Lies in the Sovereignty of the Digital Domain

CrownRNG™, CrownEncrypt™, Crown SovereignOTP™, CrownEncryptOTP™

for Quantum Resistant Blockchains and Messaging

2021

Contents

Crown Sterling Limited LLC
Newport Beach, CA
USA
P: 949.260.1702
F: 949.260.1705
www.CrownSterling.io

In this paper, we lay out the software architecture of Crown Sterling encryption products, which include CrownRNG, CrownEncrypt, CrownSovereignOTP for quantum-resistant state transition functions of the Crown Sterling blockchain, and CrownEncryptOTP for quantum-resistant secure messaging.

CrownRNG is a novel cryptographically secure random number generator (RNG). It exploits the proven randomness of irrational numbers to produce highly randomized strings of numbers. CrownEncrypt is an encryption platform designed to encrypt and secure the handling of data. It can be used as a stand-alone, or it can be incorporated within existing encryption platforms to provide more robust and reliable data handling. CrownRNG feeds into a 512 bit Elliptic-curve Diffie-Hellman key exchange protocol, coupled to an AES-based encryption algorithm, to deliver a highly secured data handling environment.

CrownRNG also feeds into CrownSovereignOTP and CrownEncryptOTP, which utilize the one-time pad encryption protocol that is proven to be quantum-computing resistant. CrownSovereignOTP is used to secure the state transition function of the Crown Sterling blockchains, while CrownEncryptOTP is used to secure messaging with multi-factor authentication and partial key transport for optimum security.

I. CrownRNG™

CrownRNG exploits the *by-default* randomness of irrational numbers. Mathematically speaking, irrational numbers are defined as numbers that can't be expressed in terms of ratios of two integers. They are proven to have digital sequences, also known as mantissas, extending to infinity without ever repeating. Therefore, they are excellent sources for true randomness[1,2]. Mathematical functions known to generate irrational numbers include the square roots of non-perfect square numbers (NPSN), e.g., $\sqrt{20}$, $\sqrt{35}$, square roots of all prime numbers, etc., and also trigonometric functions having natural numbers for their arguments, among many others. (Please refer to Appendix A for a partial list of functions proven to generate irrational numbers).

CrownRNG uses the mantissas of irrational square root values. Irrational numbers can be produced by appending 2, 3, 7, or 8 to any integer to ensure that it is not a perfect square. As a result, this non-perfect square number will have an irrational square root. Therefore, it is sufficient to prove that any integer ending in 2, 3, 7, or 8 is not a perfect square and any integer that is not a perfect square has an irrational square root.

Proof: Any integer ending in 2, 3, 7, or 8 is not a perfect square:

We can easily prove by contradiction that no integer can be squared to produce an integer ending in 2, 3, 7, and 8. Assume that some integer exists such that squaring it produces an integer ending in 2. Assume the same for 3, 7, and 8.

- If an integer ends in 1, its square will also end with a 1.
- If an integer ends in 2, its square will always end in 4
- If an integer ends in 3, its square will always end in 9
- If an integer ends in 4, its square will always end in 6
- If an integer ends in 5, its square will always end in 5
- If an integer ends in 6, its square will always end in 6
- If an integer ends in 7, its square will always end in 9

- If an integer ends in 8, its square will always end in 4
- If an integer ends in 9, its square will always end in 1
- If an integer ends in 0, its square will always end in 0

Therefore, squaring an integer will always produce an integer ending in 1, 4, 5, 6, 9, or 0. This excludes 2, 3, 7, and 8. This contradicts our assumption that some integer exists such that squaring it produces an integer ending in 2, 3, 7, or 8. Therefore, no perfect square integer can end in 2, 3, 7, or 8.

Proof: Any integer that is not a perfect square has an irrational square root:

Consider the polynomial $f(x) = x^2 - n$, where n is a positive integer. Then $\sqrt{n}$ is one of the roots of $f(x)$. Suppose $\sqrt{n} = p/q$, where p and q are coprime positive integers, so that their largest common factor is 1. We note that p^2 and q must also be coprime. We have $p^2/q^2 = n$, or $p^2 = n \times q^2 = q \times (n \times q)$. This means that p^2 is divisible by q. Since q is clearly divisible by q too, we conclude that q is a common factor of p^2 and q. But p^2 and q are coprime, so q must be 1. This implies that $\sqrt{n} = p$. We have just proved that if $\sqrt{n}$ is rational, then it must be an integer. Clearly, $\sqrt{n}$ is an integer only when n is a perfect square. Consequently, if n is not a perfect square, then I is neither an integer nor a rational number, concluding that it must be an irrational number[3,4].

As discussed in Dr. Johnson's and Dr. Leeming's paper[2], the mantissas of irrational values performed exceptionally well on various entropy tests, distinguishing the CrownRNG from pseudo-random number generators.

CrownRNG is made of four main components:

1- Entropy gathering Daemon

2- Xeno unit.

3- Functions Table.

4- Random Bits Generator (RBG).

1- The Daemon

The Daemon gathers entropy from many random system processes, including the pc metrics, such as the Heap, Memory, and stack, along with mouse movements and clicks, keyboard strokes, etc. The Daemon ensures 2048 bits of random data where the random processes are hashed and rehashed into three binary outputs that work as input features. The three outputs are then passed on to the Xeno unit, which generates another set of three random numbers.

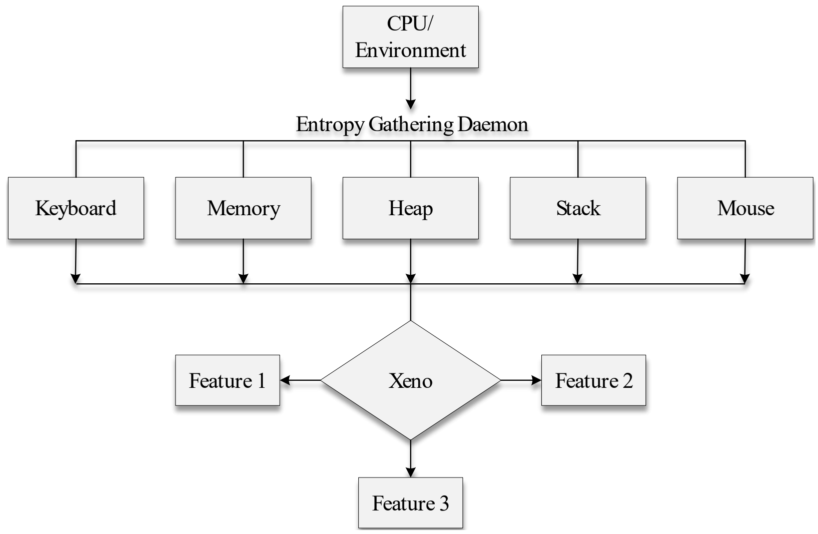

Figure 1

Figure 1: A schematic representation of the Daemon workflow.

2- The Xeno Unit: A Non-Sequential Randomizer

This unit generates the randomized parameters needed by the system. The unit is initialized by the Daemon, using its metrics as initial features to predict new labels via linear regression estimator and then captures the randomized bits of the predictions' mantissas. One sub-unit of Xeno (MusicSU) will transform the predicted numeric values into a set of three numbers labeled octave, note, and tempo. These three values are then converted, via digital root arithmetic, into specific ranges such that they can be utilized by the Functions Table. The other sub-unit (MathSU) creates random NPSNs. The square roots of these numbers create irrational numbers with infinite mantissas. These mantissas are truncated to specific bit-lengths and then passed on to the RBG as seeds.

a- The Music Sub-Unit (MusicSU):

The main workflow of the MusicSU can be summarized as follows: First, *M* random metrics are collected by the Daemon. These metrics will be collected in intervals of 1 millisecond for a total of 5 seconds. This will generate 5000 data points for each value. Next, each metric will be divided into three parts, with two parts used to predict the third. This process is repeated two times for a total of three sets of predicted values. One predicted value would be allocated to the *note* variable and hence be truncated to mod(8), in other words, eight values from 0 to 7. The other one will transform into the *tempo* using mod(7), and finally, the third will transform into the *octave*, using mod(13). These three values will then pass on to the function table, as will be explained later. Other numeric variables can be obtained by using different mods as well, such as for the last digit and the range variables.

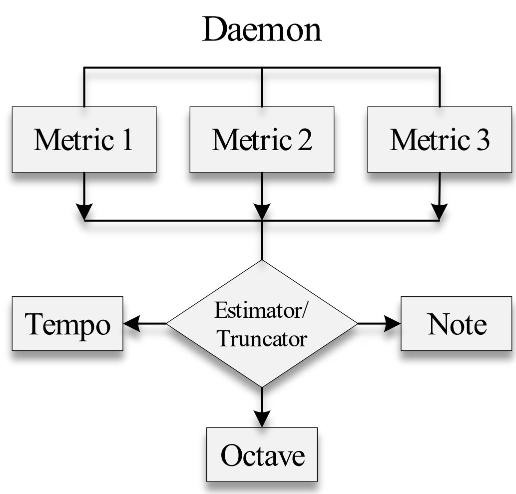

Figure 2: A schematic representation of the MusicSU workflow.

b- The Math Sub-Unit (MathSU):

The MathSU shares the same supervised machine learning algorithm with MusicSU. However, for MathSU, the three predicted values are truncated using mod(10). The operation is repeated, and the values are concatenated to form one single number of a specific length designated by the programmer. When needed, a single digit of either [2, 3, 7, 8] is randomly chosen and added to the end of the concatenated number to ensure that the number is not a perfect square, as explained above. The final step is to apply the square-root function to the number with the result passed on to the next element.

In summary, the Xeno unit outputs the following parameters:

- The irrational seed: an infinite irrational number truncated to a specific length.
- The note, tempo, and octave parameters, in the ranges of (0-7), (0-6), and (0-12), respectively.
- Last digit number: this is a list of four numbers [2, 3, 7, 8] where one of them will be randomly added to the end of the privately shared key to make sure it becomes a non-perfect square number (NPSN).
- Range number: this is a number, from 1 to 1 million (minus 1), that determines the starting index in the mantissa of the square root of the NPSN number.

Below is a schematic rendering of the workflow of the Xeno unit.

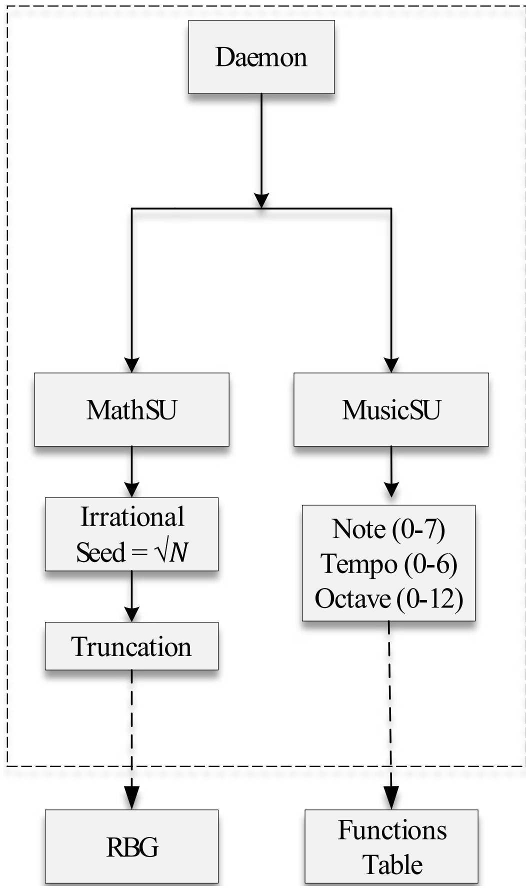

Figure 3: A schematic representation of the Xeno unit general workflow.

3- The Functions Table

The Functions Table is defined by a set of horizontal and vertical variables that are mathematical functions proven to always produce perfect irrational numbers. The arguments of these functions are not fixed, determined by the random internal states, mainly the timestamp of the current time, as well as the tempo variable. The tempo, note, and octave parameters coming out of the Xeno unit will be used to determine which two cells on the vertical and horizontal axis will be utilized for the current run. The output of these cells (the irrational mantissas) are truncated accordingly and used to compute the arithmetic mode through which the RBG will operate. The current model uses square root functions on the horizontal axis of the table and trigonometric ones on the vertical axis.

There are seven cells on the horizontal axis (Figure 3), with the argument of the square roots being the product of the tempo value, the timestamp (TS), and a non-square number (A) as follows: $\sqrt{TS \times (Tempo + 1) \times A}$. (This non-square number A is passed from the MathSU; however, it is not the same as the N number used to generate the seed.) The horizontal scale is made of 104 cells corresponding to 13 octaves, with each octave divided into eight notes. The octave parameter selects one of the 13 octaves, and the note parameter selects which note of this specific octave will be used. Each note corresponds to a trigonometric function having an argument made of the time stamp divided by a specific frequency value, TS/fr.

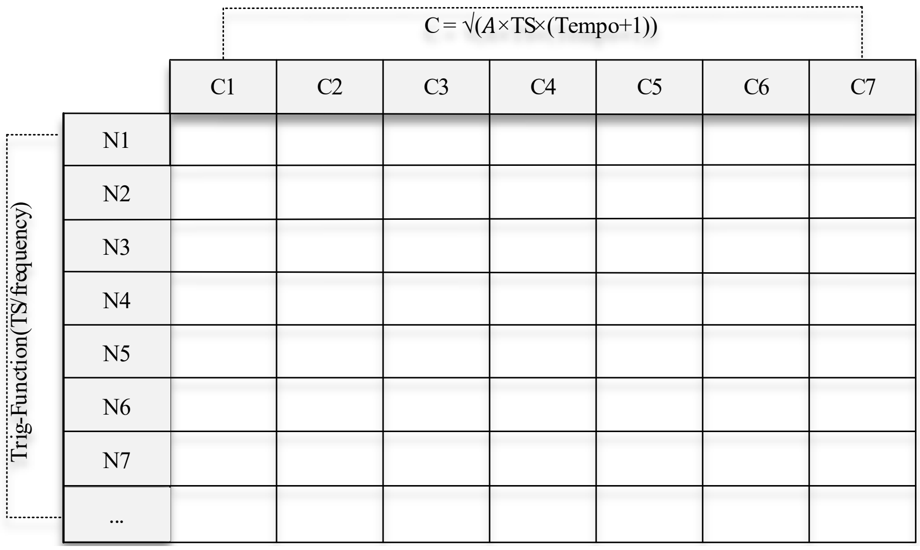

Figure 4: A schematic representation of a small portion of the Functions Table.

The trigonometric functions, along with the frequencies of the notes, are listed in the table below.

Function	**Frequencies**												
Sin	432	450	468	252	270	288	306	324	342	360	378	396	414
Cos	864	900	936	504	540	576	612	648	684	720	756	792	828
Tan	1728	1800	1872	1008	1080	1152	1224	1296	1368	1440	1512	1584	1656
Ctan	3456	3600	3744	2016	2160	2304	2448	2592	2736	2880	3024	3168	3312
Sec	6912	7200	7488	4032	4320	4608	4896	5184	5472	5760	6048	6336	6624
Csc	13824	14400	14976	8064	8640	9216	9792	10368	10944	11520	12096	12672	13248
Sin	27648	28800	29952	16128	17280	18432	19584	20736	21888	23040	24192	25344	26496
Cos	55296	57600	59904	32256	34560	36864	39168	41472	43776	46080	48384	50688	52992

Table 1: A list of the trigonometric functions used along with the music frequencies.

When the two irrational values of the horizontal and the vertical cells (the square root and trig function) are calculated, they will be truncated to specific lengths and then passed on to the RBG as variables I_1 and I_2, along with the seed (the truncated number N).

4- The Random Bit Generator (RBG)

The RBG utilizes a specific mathematical function that takes the seed output of the Xeno unit as its initial argument and the two truncated irrational numbers of the Functions Table (I_1 and I_2) as the arithmetic mod parameters. The RBG then iterates on each calculated value to calculate new ones that are concatenated to create a randomized sequence of bits.

The RBG general design is based on the cryptographically secure Blum-Blum-Shub (BBS) generator[5]. The primary difference between the original BBS and CrownRNG relates to the numerical basis for the arithmetic mod calculation. In the original BBS, the mod is computed from the product of two prime numbers, whereas CrownRNG uses the truncated irrational numbers coming from the Functions Table.

The general mathematical flow of the modified BBS generator works as follows:

1- Two truncated irrational numbers, I_1 and I_2 of specific bit-length are chosen such that each is congruent to 3 modulo 4: $p \equiv q \equiv 3\ mod(4)$.
2- The two truncated irrational numbers are multiplied to generate *n,* the arithmetical mode by which the generator will perform its calculations.
3- A random integer $\boldsymbol{s}$ (the seed) is generated from the Xeno unit.
4- The seed will initiate the generation process through the operation $x_0 = s^2 mod(n)$.
5- The function $x_{i+1} = x_i^2 mod(n)$ is then used to iterate on each previously calculated value, generating new values for every iteration and outputting a string of numbers: $x_1, x_2, x_3, \ldots, x_k$.
6- These output values are converted into a string of binary bits.
7- The bit-parity of each binary number is determined depending on the type of parity, even or odd (0 or 1).
8- Finally, the parity digits are concatenated to form the desired CSPRN, depending on the required bit-length of the key, which also determines the level of security: $Y = y_1 y_2 y_3 \ldots y_k$.

As mentioned above, the only modification the RBG introduces to the original BBS is replacing prime numbers with irrational ones. The usage of prime numbers in the original BBS is necessary if we want to have the ability to reverse the direction of the generator, as in the case when the BBS system is used as an encryption/decryption algorithm. However, as we do not want to reverse the operation in our system, there is no problem with using numbers that are not prime. In fact, this introduces additional security to the system because when we compare the limited amount of prime numbers having specific bit-length to the infinite amount of potential irrational numbers of the same bit-lengths, the infinity factor introduces an extra layer of security to the RBG against cyber-attacks that try to predict these values.

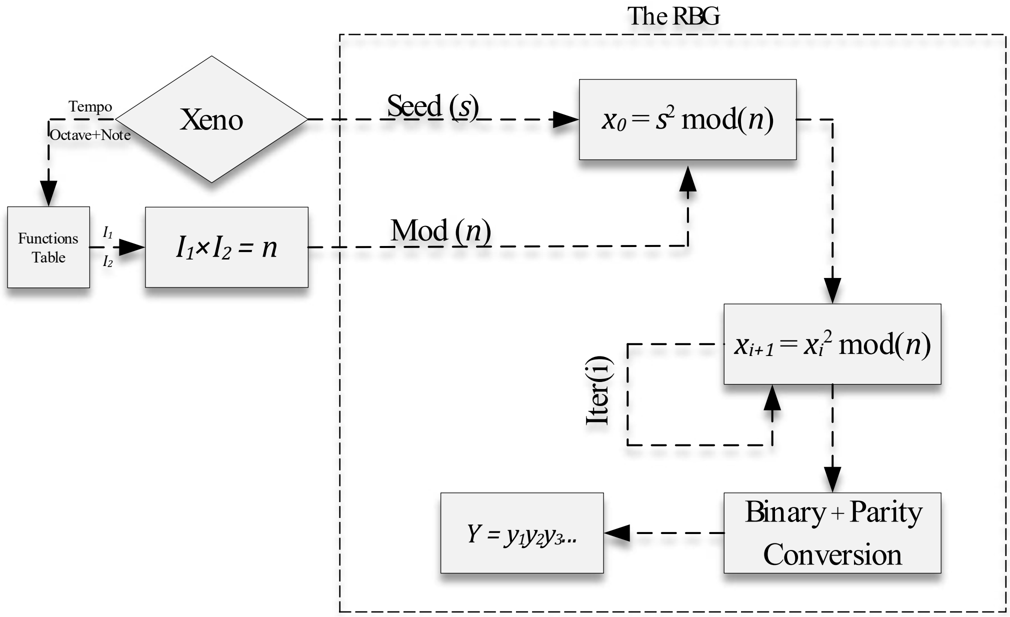

Figure 5: A schematic representation of the RBG workflow.

CrownRNG Randomness Tests Results

Many statistical testing suites were designed to test the randomness level of random number generators. The most important of these tests are TestU01, NIST, PractRand, and DIEHARDER. The tests are evaluated depending on a specific statistical value called the *p-value*. A p-value extremely close to 0 or 1 indicates a failure, while more moderate values are considered a pass. If a theoretically ideal source of randomness were given to the tests, the p-values would be uniformly distributed in the interval [0, 1], and so values extremely close to either limit would be very unlikely. P-values in the range of 0.001 to 0.999 are considered unremarkable. Values in the range of 0.00001 to 0.99999 would also not be terribly surprising given that we are running hundreds of tests. However, much more extreme values such as p-values less than 10^{-9} or greater than $1 - 10^{-9}$ would be suspicious and would suggest that the RNG is failing to emulate some aspect of random behavior.

NIST tests suite is made of 15 different tests designed to check the randomness level of numbers generated from the RNG[6]. (The Cumulative Sum Test generates two values, Forward and Backward, increasing the total number of p-values to 16). To pass the Test, the random numbers should generate a statistical *p*-value that is greater than a specific threshold, usually chosen to equal 0.01. Furthermore, for all the numbers tested, the percentage criteria of a successful pass for each of NIST's tests should not be less than 98% of all tested numbers. (The irrational functions used in the CrownRNG unit are also tested for their randomness by the NIST tests suite and passed the threshold value (0.01) for all the tests. Refer to Appendix B for NIST test results for one such function.)

Below we list the NIST tests results for 1000 numbers generated by the CrownRNG system, with each number having a length of one million binary bits. The tests were conducted locally by Crown Sterling Staff and implemented using the Python programming language.

Test Name	P-Value	Variance	Success %
Frequency	0.503087	0.084981	99.1
Block Frequency	0.504021	0.084437	98.9
Run	0.497848	0.084397	99.2
Longest Run	0.490239	0.082727	99.3
Matrix	0.499644	0.083488	99.0
Spectral	0.499322	0.086307	98.8
Non-overlapping Template	0.492563	0.082768	99.1
Overlapping Template	0.492579	0.08619	99.1
Universal	0.481775	0.078071	98.6
Complexity	0.519357	0.082535	99.0
Serial	0.49152	0.07987	99.4
Entropy	0.502573	0.083398	98.7
Cumulative Sum Forward	0.497417	0.08383	99.0
Cumulative Sum Backward	0.508077	0.084272	98.9
Random Excursion	0.509958	0.016384	100
Random Excursion Variant	0.506634	0.022137	100

Table 2: NIST test results for the CrownRNG.

As evident from the above table, the average p-values for the 1000 tested numbers are around 0.5, right in the middle of the range [0, 1], as expected from a well-designed PRNG. Additionally, the success rate for all the tests is above 98%, as demanded by NIST.

Dieharder tests on CrownRNG's outcome were performed by Crown Sterling. When running the randomness testing, the Dieharder test suite recommends having a minimum dataset size of 15GB for each analysis. Datasets from CrownRNG were created in the 15GB – 20GB range by concatenating individual 100MB entropy files from separate runs of the tool. These individual 100MB data files were created on multiple Ubuntu virtual machines running the CrownRNG Docker container V1.0.3 and written to the local file system. A total of 10 datasets of ~20GB each, representing a cumulative total of ~200GB of data, were tested. Below is the result of the testing. As obvious from the results, CrownRNG performed well for all the tests. For the full report, please refer to Appendix C.

Installed dieharder tests:

Test Number	Test Name	Test Reliability
-d 0	Diehard Birthdays Test	Good
-d 1	Diehard OPERM5 Test	Suspect
-d 2	Diehard 32x32 Binary Rank Test	Good
-d 3	Diehard 6x8 Binary Rank Test	Good
-d 4	Diehard Bitstream Test	Good
-d 5	Diehard OPSO	Good
-d 6	Diehard OQSO Test	Good
-d 7	Diehard DNA Test	Good
-d 8	Diehard Count the 1s (stream) Test	Good
-d 9	Diehard Count the 1s Test (byte)	Good
-d 10	Diehard Parking Lot Test	Good
-d 11	Diehard Minimum Distance (2d Circle) Test	Good
-d 12	Diehard 3d Sphere (Minimum Distance) Test	Good
-d 13	Diehard Squeeze Test	Good
-d 14	Diehard Sums Test	Do Not Use
-d 15	Diehard Runs Test	Good
-d 16	Diehard Craps Test	Good
-d 17	Marsaglia and Tsang GCD Test	Good
-d 100	STS Monobit Test	Good
-d 101	STS Runs Test	Good
-d 102	STS Serial Test (Generalized)	Good
-d 200	RGB Bit Distribution Test	Good
-d 201	RGB Generalized Minimum Distance Test	Good
-d 202	RGB Permutations Test	Good
-d 203	RGB Lagged Sum Test	Good
-d 204	RGB Kolmogorov-Smirnov Test Test	Good

Table 3: Dieharder battery test results for CrownRNG, performed by Crown Sterling.

Additionally, a smaller data size of CrownRNG output was tested by Dr. John Cook and for all the three types of randomness tests, U01, Dieharder, and PractRand.

1- U01 Test

TestU01 is the most academically respected RNG test suite at this time[7]. The suite comes in three versions: small crush, crush, and big crush. The small crush uses on the order of a gigabyte of data, and in that sense, it is not small. For our case, this Test was run by John D. Cook, Ph.D., and he reported that all tests were passed. Below are the U01 Test full results.

- **Smarsa_BirthdaySpacings test:**

N = 1, n = 5000000, r = 0, d = 1073741824, t = 2, p = 1
Number of cells = d^t = 1152921504606846976
Lambda = Poisson mean = 27.1051
Total expected number = N*Lambda : 27.11
Total observed number : 27
p-value of test : 0.53
CPU time used : 00:00:01.21

- **Test sknuth_Collision calling smultin_Multinomial**

HOST = Silver, Linux
32-bit stdin
smultin_Multinomial test:
N = 1, n = 5000000, r = 0, d = 65536, t = 2,
Sparse = TRUE
GenerCell = smultin_GenerCellSerial
Number of cells = d^t = 4294967296
Expected number per cell = 1 / 858.99346
EColl = n^2 / (2k) = 2910.383046
Hashing = TRUE
Collision test, Mu = 2909.2534, Sigma = 53.8957
Test Results for Collisions
Expected number of collisions = Mu : 2909.25
Observed number of collisions : 2961
p-value of test : 0.17
Total number of cells containing j balls
j = 0 : 4289970257
j = 1 : 4994082
j = 2 : 2953
j = 3 : 4
j = 4 : 0
j = 5 : 0

- **Sknuth_Gap test:**

N = 1, n = 200000, r = 22, Alpha = 0, Beta = 0.00390625
Number of degrees of freedom : 1114
Chi-square statistic : 1063.19
p-value of test : 0.86

- **Sknuth_SimpPoker test:**

N = 1, n = 400000, r = 24, d = 64, k = 64
Number of degrees of freedom : 19
Chi-square statistic : 27.58

p-value of test : 0.09

- **Sknuth_CouponCollector test:**

N = 1, n = 500000, r = 26, d = 16
Number of degrees of freedom : 44
Chi-square statistic : 50.60
p-value of test : 0.23

- **Sknuth_MaxOft test:**

N = 1, n = 2000000, r = 0, d = 100000, t = 6
Number of categories = 100000
Expected number per category = 20.00
Number of degrees of freedom : 99999
Chi-square statistic : 1.01e+5
p-value of test : 0.05
Anderson-Darling statistic : 0.065
p-value of test : 0.93

- **Svaria_WeightDistrib test:**

N = 1, n = 200000, r = 27, k = 256, Alpha = 0, Beta = 0.125
Number of degrees of freedom : 41
Chi-square statistic : 40.53
p-value of test : 0.49

- **Smarsa_MatrixRank test:**

Number of degrees of freedom : 3
Chi-square statistic : 3.71
p-value of test : 0.29
CPU time used : 00:00:00.53
Generator state:

- **Sstring_HammingIndep test:**

N = 1, n = 500000, r = 20, s = 10, L = 300, d = 0
Counters with expected numbers >= 10
Number of degrees of freedom : 2209
Chi-square statistic : 2173.19
p-value of test : 0.70
Swalk_RandomWalk1 test:
N = 1, n = 1000000, r = 0, s = 30, L0 = 150, L1 = 150
Test on the values of the Statistic H
Number of degrees of freedom : 52
ChiSquare statistic : 64.15
p-value of test : 0.12
Test on the values of the Statistic M
Number of degrees of freedom : 52
ChiSquare statistic : 34.02
p-value of test : 0.97

- **Test on the values of the Statistic J**

Number of degrees of freedom : 75
ChiSquare statistic : 89.70
p-value of test : 0.12

- **Test on the values of the Statistic R**

Number of degrees of freedom : 44
ChiSquare statistic : 29.80
p-value of test : 0.95

- **Test on the values of the Statistic C**

Number of degrees of freedom : 26
ChiSquare statistic : 27.09
p-value of test : 0.40

```
========= Summary results of SmallCrush =========
Version:          TestU01 1.2.3
Generator:        32-bit stdin
Number of statistics:  15
Total CPU time:   00:00:10.36
All tests were passed
```

Table 4: U01 Test results for CrownRNG data.

2- DIEHARDER Test

George Marsaglia's DIEHARD "battery" was the first widely used RNG test suite. The suite has been maintained and extended by Robert Brown and others under the name DIEHARDER[8]. This suite is commonly run because it is so well known, even though TestU01 is more highly regarded in the academic community. The DIEHARDER test suite was run by John D. Cook, using version 3.31.1, using all the default options, by giving it 1 GB of random bits generated by the CrownRNG.

All tests passed. However, three tests, one instance of rgb permutations and two instances of rgb lagged sum, passed with a *weak* pass, generating p-values of 0.99837, 0.00149, and 0.00068. These are not such extreme values and are to be expected when running a large number of tests. Below are the full results of the Test.

Test_name	ntup	tsamples	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.3726455	PASSED
diehard_operm5	0	1000000	100	0.9507281	PASSED
diehard_rank_32x32	0	40000	100	0.2128847	PASSED
The file file_input_raw was rewound 1 times					
diehard_rank_6x8	0	100000	100	0.7168254	PASSED
The file file_input_raw was rewound 1 times					
diehard_bitstream	0	2097152	100	0.8986398	PASSED
The file file_input_raw was rewound 2 times					
diehard_opso	0	2097152	100	0.8855053	PASSED
The file file_input_raw was rewound 2 times					
diehard_oqso	0	2097152	100	0.6678726	PASSED
The file file_input_raw was rewound 2 times					
diehard_dna	0	2097152	100	0.9865553	PASSED
The file file_input_raw was rewound 2 times					
diehard_count_1s_str	0	256000	100	0.7042336	PASSED
The file file_input_raw was rewound 3 times					
diehard_count_1s_byt	0	256000	100	0.5736274	PASSED
The file file_input_raw was rewound 3 times					
diehard_parking_lot	0	12000	100	0.8022002	PASSED
The file file_input_raw was rewound 3 times					
diehard_2dsphere	2	8000	100	0.3493633	PASSED
The file file_input_raw was rewound 3 times					
diehard_3dsphere	3	4000	100	0.8124341	PASSED

The file file_input_raw was rewound 4 times					
diehard_squeeze	0	100000	100	0.6349244	PASSED
The file file_input_raw was rewound 4 times					
diehard_sums	0	100	100	0.0268951	PASSED
The file file_input_raw was rewound 4 times					
diehard_runs	0	100000	100	0.2016708	PASSED
diehard_runs	0	100000	100	0.4288304	PASSED
The file file_input_raw was rewound 4 times					
diehard_craps	0	200000	100	0.948407	PASSED
diehard_craps	0	200000	100	0.0408707	PASSED
The file file_input_raw was rewound 12 times					
marsaglia_tsang_gcd	0	10000000	100	0.2599033	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.4730578	PASSED
The file file_input_raw was rewound 12 times					
sts_monobit	1	100000	100	0.5136588	PASSED
The file file_input_raw was rewound 12 times					
sts_runs	2	100000	100	0.9639255	PASSED
The file file_input_raw was rewound 12 times					
sts_serial	1	100000	100	0.5473606	PASSED
sts_serial	2	100000	100	0.1930377	PASSED
sts_serial	3	100000	100	0.2471209	PASSED
sts_serial	3	100000	100	0.9262692	PASSED
sts_serial	4	100000	100	0.7346105	PASSED
sts_serial	4	100000	100	0.7899912	PASSED
sts_serial	5	100000	100	0.5861151	PASSED
sts_serial	5	100000	100	0.569177	PASSED
sts_serial	6	100000	100	0.3839097	PASSED
sts_serial	6	100000	100	0.6381691	PASSED
sts_serial	7	100000	100	0.7448842	PASSED
sts_serial	7	100000	100	0.804561	PASSED
sts_serial	8	100000	100	0.8183399	PASSED
sts_serial	8	100000	100	0.8295544	PASSED
sts_serial	9	100000	100	0.7499303	PASSED
sts_serial	9	100000	100	0.809182	PASSED
sts_serial	10	100000	100	0.1525207	PASSED
sts_serial	10	100000	100	0.0328435	PASSED
sts_serial	11	100000	100	0.1174684	PASSED
sts_serial	11	100000	100	0.4339114	PASSED
sts_serial	12	100000	100	0.3079395	PASSED
sts_serial	12	100000	100	0.3618067	PASSED
sts_serial	13	100000	100	0.8066997	PASSED
sts_serial	13	100000	100	0.7378393	PASSED
sts_serial	14	100000	100	0.0899083	PASSED

sts_serial	14	100000	100	0.2961106	PASSED
sts_serial	15	100000	100	0.7296562	PASSED
sts_serial	15	100000	100	0.1508948	PASSED
sts_serial	16	100000	100	0.4669664	PASSED
sts_serial	16	100000	100	0.0671334	PASSED
The file file_input_raw was rewound 12 times					
rgb_bitdist	1	100000	100	0.8573912	PASSED
The file file_input_raw was rewound 12 times					
rgb_bitdist	2	100000	100	0.6990537	PASSED
The file file_input_raw was rewound 12 times					
rgb_bitdist	3	100000	100	0.7451708	PASSED
The file file_input_raw was rewound 12 times					
rgb_bitdist	4	100000	100	0.1124013	PASSED
The file file_input_raw was rewound 13 times					
rgb_bitdist	5	100000	100	0.3132411	PASSED
The file file_input_raw was rewound 13 times					
rgb_bitdist	6	100000	100	0.9467392	PASSED
The file file_input_raw was rewound 14 times					
rgb_bitdist	7	100000	100	0.9594866	PASSED
The file file_input_raw was rewound 14 times					
rgb_bitdist	8	100000	100	0.7924018	PASSED
The file file_input_raw was rewound 15 times					
rgb_bitdist	9	100000	100	0.9157191	PASSED
The file file_input_raw was rewound 16 times					
rgb_bitdist	10	100000	100	0.5395454	PASSED
The file file_input_raw was rewound 17 times					
rgb_bitdist	11	100000	100	0.2418238	PASSED
The file file_input_raw was rewound 18 times					
rgb_bitdist	12	100000	100	0.576704	PASSED
The file file_input_raw was rewound 18 times					
rgb_minimum_distance	2	10000	1000	0.3693933	PASSED
The file file_input_raw was rewound 18 times					
rgb_minimum_distance	3	10000	1000	0.6119019	PASSED
The file file_input_raw was rewound 18 times					
rgb_minimum_distance	4	10000	1000	0.3693909	PASSED
The file file_input_raw was rewound 18 times					
rgb_minimum_distance	5	10000	1000	0.0792296	PASSED
The file file_input_raw was rewound 18 times					
rgb_permutations	2	100000	100	0.6973623	PASSED
The file file_input_raw was rewound 18 times					
rgb_permutations	3	100000	100	0.9983674	WEAK
The file file_input_raw was rewound 18 times					
rgb_permutations	4	100000	100	0.0961262	PASSED

The file file_input_raw was rewound 19 times					
rgb_permutations	5	100000	100	0.3519567	PASSED
The file file_input_raw was rewound 19 times					
rgb_lagged_sum	0	1000000	100	0.292929	PASSED
The file file_input_raw was rewound 20 times					
rgb_lagged_sum	1	1000000	100	0.1262539	PASSED
The file file_input_raw was rewound 21 times					
rgb_lagged_sum	2	1000000	100	0.8598643	PASSED
The file file_input_raw was rewound 22 times					
rgb_lagged_sum	3	1000000	100	0.0144612	PASSED
The file file_input_raw was rewound 24 times					
rgb_lagged_sum	4	1000000	100	0.7765675	PASSED
The file file_input_raw was rewound 26 times					
rgb_lagged_sum	5	1000000	100	0.4394164	PASSED
The file file_input_raw was rewound 29 times					
rgb_lagged_sum	6	1000000	100	0.4212405	PASSED
# The file file_input_raw was rewound 32 times					
rgb_lagged_sum	7	1000000	100	0.0014851	WEAK
The file file_input_raw was rewound 35 times					
rgb_lagged_sum	8	1000000	100	0.7676263	PASSED
The file file_input_raw was rewound 39 times					
rgb_lagged_sum	9	1000000	100	0.6756103	PASSED
The file file_input_raw was rewound 43 times					
rgb_lagged_sum	10	1000000	100	0.8372619	PASSED
The file file_input_raw was rewound 48 times					
rgb_lagged_sum	11	1000000	100	0.3911788	PASSED
The file file_input_raw was rewound 53 times					
rgb_lagged_sum	12	1000000	100	0.4920459	PASSED
The file file_input_raw was rewound 58 times					
rgb_lagged_sum	13	1000000	100	0.3267564	PASSED
The file file_input_raw was rewound 63 times					
rgb_lagged_sum	14	1000000	100	0.9464163	PASSED
The file file_input_raw was rewound 69 times					
rgb_lagged_sum	15	1000000	100	0.0006816	WEAK
The file file_input_raw was rewound 76 times					
rgb_lagged_sum	16	1000000	100	0.48635	PASSED
The file file_input_raw was rewound 82 times					
rgb_lagged_sum	17	1000000	100	0.0300186	PASSED
The file file_input_raw was rewound 89 times					
rgb_lagged_sum	18	1000000	100	0.9675033	PASSED
The file file_input_raw was rewound 97 times					
rgb_lagged_sum	19	1000000	100	0.0505593	PASSED
The file file_input_raw was rewound 105 times					

rgb_lagged_sum	20	1000000	100	0.8654232	PASSED
The file file_input_raw was rewound 113 times					
rgb_lagged_sum	21	1000000	100	0.4340315	PASSED
The file file_input_raw was rewound 121 times					
rgb_lagged_sum	22	1000000	100	0.1940784	PASSED
The file file_input_raw was rewound 130 times					
rgb_lagged_sum	23	1000000	100	0.1102014	PASSED
The file file_input_raw was rewound 140 times					
rgb_lagged_sum	24	1000000	100	0.9884602	PASSED
The file file_input_raw was rewound 149 times					
rgb_lagged_sum	25	1000000	100	0.6760483	PASSED
The file file_input_raw was rewound 159 times					
rgb_lagged_sum	26	1000000	100	0.5673114	PASSED
The file file_input_raw was rewound 170 times					
rgb_lagged_sum	27	1000000	100	0.0156134	PASSED
The file file_input_raw was rewound 181 times					
rgb_lagged_sum	28	1000000	100	0.7224232	PASSED
The file file_input_raw was rewound 192 times					
rgb_lagged_sum	29	1000000	100	0.1861631	PASSED
The file file_input_raw was rewound 203 times					
rgb_lagged_sum	30	1000000	100	0.8136724	PASSED
The file file_input_raw was rewound 215 times					
rgb_lagged_sum	31	1000000	100	0.7300451	PASSED
The file file_input_raw was rewound 228 times					
rgb_lagged_sum	32	1000000	100	0.5328815	PASSED
The file file_input_raw was rewound 228 times					
rgb_kstest_test	0	10000	1000	0.8888545	PASSED
The file file_input_raw was rewound 228 times					
dab_bytedistrib	0	51200000	1	0.805461	PASSED
The file file_input_raw was rewound 228 times					
dab_dct	256	50000	1	0.7330565	PASSED
Preparing to run test 207. ntuple = 0					
The file file_input_raw was rewound 229 times					
dab_filltree	32	15000000	1	0.6329858	PASSED
dab_filltree	32	15000000	1	0.4053801	PASSED
Preparing to run test 208. ntuple = 0					
The file file_input_raw was rewound 229 times					
dab_filltree2	0	5000000	1	0.8434582	PASSED
dab_filltree2	1	5000000	1	0.9402829	PASSED
Preparing to run test 209. ntuple = 0					
The file file_input_raw was rewound 229 times					
dab_monobit2	12	65000000	1	0.4629416	PASSED

Table 5: Diehardér test results for CrownRNG data.

3- PractRand Test

John D. Cook tested the Crown Sterling CrownRNG using the same data described above using the PractRand test suite[9], version 0.94, and using all the default options. The PractRand suite starts by testing 1 kilobyte of data. It then doubles the amount of data at each iteration and will eventually use as much data as it is given. When the tests ran on 16 megabytes of data, the tests passed, but the results were reported as *unusual* with a p-value of 0.99946. This is, as reported, an unusual p-value, but is not a cause for alarm as later stages of testing are more rigorous, and the tests ran on up to the full gigabyte of data provided without reporting any anomalies.

RNG_test using PractRand version 0.94	
RNG = RNG_stdin	seed = unknown
test set = core	
rng=RNG_stdin	seed=unknown
length= 1 kilobyte (2^10 bytes)	time= 0.2 seconds
no anomalies in 6 test result(s)	
rng=RNG_stdin	seed=unknown
length= 2 kilobytes (2^11 bytes)	time= 0.3 seconds
no anomalies in 8 test result(s)	
rng=RNG_stdin	seed=unknown
length= 4 kilobytes (2^12 bytes)	time= 0.4 seconds
no anomalies in 12 test result(s)	
rng=RNG_stdin	seed=unknown
length= 8 kilobytes (2^13 bytes)	time= 0.5 seconds
no anomalies in 25 test result(s)	
rng=RNG_stdin	seed=unknown
length= 16 kilobytes (2^14 bytes)	time= 0.8 seconds
no anomalies in 30 test result(s)	
rng=RNG_stdin	seed=unknown
length= 32 kilobytes (2^15 bytes)	time= 1.0 seconds
no anomalies in 45 test result(s)	
rng=RNG_stdin	seed=unknown
length= 64 kilobytes (2^16 bytes)	time= 1.3 seconds
no anomalies in 54 test result(s)	
rng=RNG_stdin	seed=unknown
length= 128 kilobytes (2^17 bytes)	time= 1.7 seconds
no anomalies in 63 test result(s)	
rng=RNG_stdin	seed=unknown
length= 256 kilobytes (2^18 bytes)	time= 2.1 seconds
no anomalies in 69 test result(s)	
rng=RNG_stdin	seed=unknown
length= 512 kilobytes (2^19 bytes)	time= 2.5 seconds
no anomalies in 84 test result(s)	

rng=RNG_stdin	seed=unknown
length= 1 megabyte (2^20 bytes)	time= 2.9 seconds
no anomalies in 94 test result(s)	
rng=RNG_stdin	seed=unknown
length= 2 megabytes (2^21 bytes)	time= 3.3 seconds
no anomalies in 109 test result(s)	
rng=RNG_stdin	seed=unknown
length= 4 megabytes (2^22 bytes)	time= 3.7 seconds
no anomalies in 124 test result(s)	
rng=RNG_stdin	seed=unknown
length= 8 megabytes (2^23 bytes)	time= 4.2 seconds
no anomalies in 135 test result(s)	
rng=RNG_stdin	seed=unknown
length= 16 megabytes (2^24 bytes)	time= 4.7 seconds
[Low1/8]BCFN(2+0	13-Jun
...and 150 test result(s) without anomalies	
rng=RNG_stdin	seed=unknown
length= 32 megabytes (2^25 bytes)	time= 5.4 seconds
no anomalies in 167 test result(s)	
rng=RNG_stdin	seed=unknown
length= 64 megabytes (2^26 bytes)	time= 6.4 seconds
no anomalies in 179 test result(s)	
rng=RNG_stdin	seed=unknown
length= 128 megabytes (2^27 bytes)	time= 8.1 seconds
no anomalies in 196 test result(s)	
rng=RNG_stdin	seed=unknown
length= 256 megabytes (2^28 bytes)	time= 10.8 seconds
no anomalies in 213 test result(s)	
rng=RNG_stdin	seed=unknown
length= 512 megabytes (2^29 bytes)	time= 15.7 seconds
no anomalies in 229 test result(s)	
rng=RNG_stdin	seed=unknown
length= 1 gigabyte (2^30 bytes)	time= 25.2 seconds
no anomalies in 248 test result(s)	

Table 6: PractRand test results for CrownRNG data.

II. CrownEncrypt™

CrownEncrypt utilizes the keys generated from CrownRNG to encrypt data and securely exchange it, along with the keys needed for decryption. CrownRNG does not depend on CrownEncrypt to operate, while the latter takes its input from CrownRNG (or any key generating unit), which is essential to its operation.

CrownEncrypt is basically made of two main units:

i. The CrownRNG Unit
ii. The Encryption Unit

As explained above, CrownRNG delivers highly randomized binary bits suitable to be used as private keys required by the Encryption unit, which is made of two main elements:

1- Key Exchange Protocol: This element generates and controls the secure exchange of encryption keys through which the data is encrypted, whether it is a password, a confidential message, credit card information, etc.
2- Encryption Algorithm: this element encrypts the data and then locks it with the keys generated by the Key Exchange Protocol.

1- The Key Exchange Protocol.

CrownEncrypt implements the *Diffie-Hellman*[10,11] public-key exchange protocol, which is built on the principle of trapdoor functions, being mathematical functions that can be easily calculated in one direction; however, reversing the calculation is very difficult and requires an enormous amount of time and computing power. One such function is the product of two prime numbers; for large prime numbers, computing the product is an easy and fast operation; however, factorizing the product to find the two prime numbers is very difficult and resource-expensive. This method is primarily used in the RSA encryption[12]. Nevertheless, prime factorization is becoming increasingly vulnerable due to the advancement in the processing power of computers, especially with the advent of quantum computers, as well as novel discoveries in prime number patterns, which enable faster factorizing algorithms[13].

Another trapdoor function utilizes the algebraic properties of *elliptic curves*[14,15], where adding a point on the curve to itself *k* times is very easy; however, figuring out *k* from the result is very complicated. Elliptic-curve Cryptography (ECC) is more secure than RSA and requires smaller encryption keys for the same level of security. When combined, ECC-DH becomes a key agreement protocol that allows two parties, each utilizing the same elliptic curve, to establish a shared secret key over an insecure channel.

The ECC-DH protocol works as follows:

1- Both communicating parties generate their own private keys: α, β. (These private keys are generated by the random number generator, which, in our case, is CrownRNG.)
2- Next, they generate their own public keys, $\alpha \times G$ and $\beta \times G$, by utilizing the algebraic rules of elliptic curves, where the generator point G is a point on the curve. (This is the trap door of elliptic curves, as calculating $\alpha \times G$ is easy. However, even when G is known, figuring out α from the product is very difficult.)
3- These two public keys, $\alpha \times G$ and $\beta \times G$, are exchanged between the two parties insecurely.

4- Finally, each party multiplies the other public key by its own private key to create the new private encryption/decryption key: α×β×G, shared only by the two communicating parties

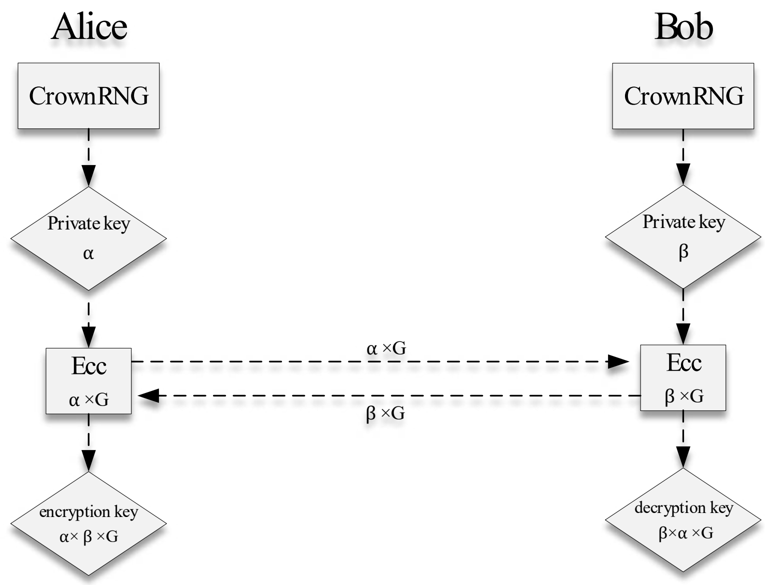

Figure 6: A schematic representation of the ECC-DH workflow.

\

2- <u>The Encryption Algorithm</u>

This unit utilizes the AES encryption algorithm to encrypt the message using the key obtained from the previous unit. AES stands for *Advanced Encryption Standard*. It is an encryption algorithm developed in 2001 to satisfy the NIST specification for the encryption of electronic data. It is based on a design principle known as a *Substitution–Permutation Network* and is efficient in both software and hardware requirements[16].

The first step of the cipher is to put the data into an array. Next, specific cipher transformations are repeated over multiple encryption rounds. The first transformation is the substitution of data using a substitution table; the second transformation shifts data rows; the third mixes columns. The last transformation is performed on each column using a different part of the encryption key.

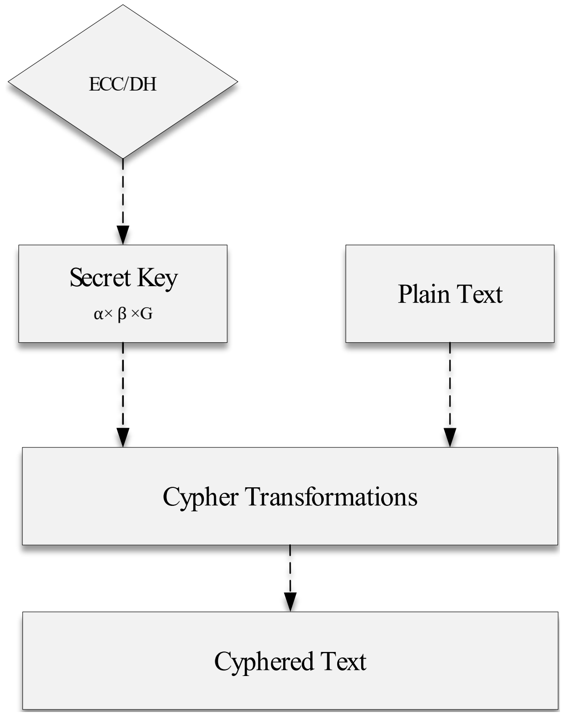

Figure 7: A schematic representation of the AES encryption workflow.

The full operational flow of CrownEncrypt can be summarized as follows:

1- The Xeno unit generates an irrational seed along with the other random parameters required for the operation of the Functions Table.
2- The random parameters will feed into the Functions Table to determine the working cells along the X and Y axis, as well as the arguments of these functions, which will lead to the generation of two new irrational numbers, which are then truncated to a specific length.
3- The three truncated random numbers will feed into the RBG, which will create the required key.
4- The key will feed into the ECC-DH protocol, from which it will create a public key shared with the other party to whom the encrypted message will be sent.
5- Both parties will create from each other's public key a new private key that is known only to both communicating parties.
6- These keys will be used by the AES system to securely encrypt/decrypt the message.

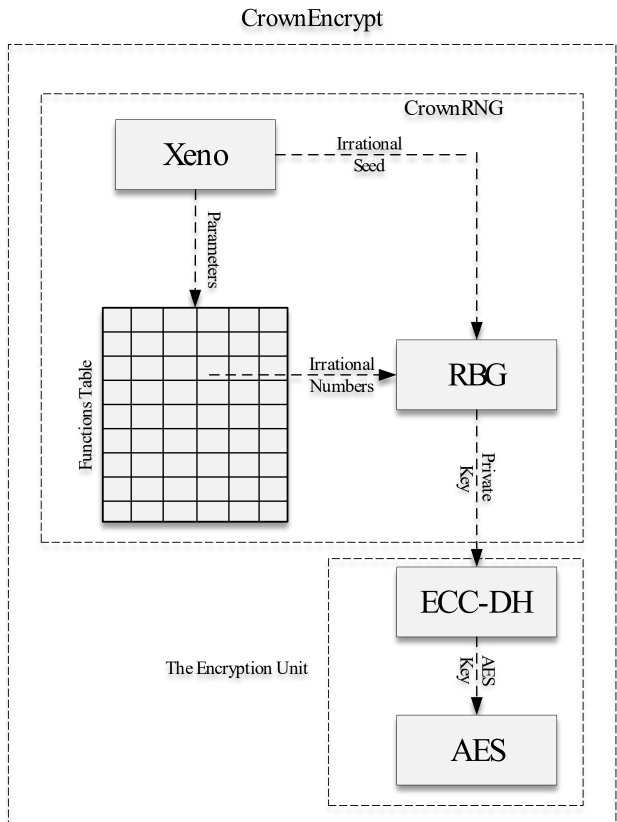

Figure 8: A schematic representation of the CrownEncrypt general workflow.

The Security Layers of the CrownEncrypt Architecture

CrownEncrypt incorporates five different layers of security, three in CrownRNG and two in CrownEncrypt. This multi-layering design renders it very secure and robust against determined cyber-attacks. These five layers are as follows:

1- The 1st layer is that of the Xeno unit. This is the innermost layer where the algorithm makes sure that its output variables, e.g., the seed, are not only highly random but also resilient to any determined attack.

2- The 2nd layer is that of the Functions Table, which receives its parameters from Xeno to produce two truncated irrational numbers from specific functions. The arguments of these functions are also randomly determined by the Xeno unit as well as local time stamps. This makes predicting these truncated irrational numbers a very challenging endeavor in both software and hardware resources. And similar to the 1st layer, the Functions Table system is designed such that compromising one specific state will not automatically jeopardize past and future ones.

3- The 3rd layer is that of the RBG, which exploits the mathematically proven properties of the BBS generator in deterring determined and engineered attacks.

4- The 4th layer is that of the ECC-DH system, where we use the most secure and NIST recommended elliptic curves, using 512-bit encryption keys, along with the Diffie-Hellman protocol, to deliver a highly secure and reliable key-exchange system.

5- Finally, the 5th layer is that of the AES encryption, which ensures perfect encryption of the messages, locked by the keys generated by the ECC-DH system.

These five nested layers create a secured hierarchy that is guaranteed to deliver superb security protection for the encrypted message as well as for the randomly generated keys.

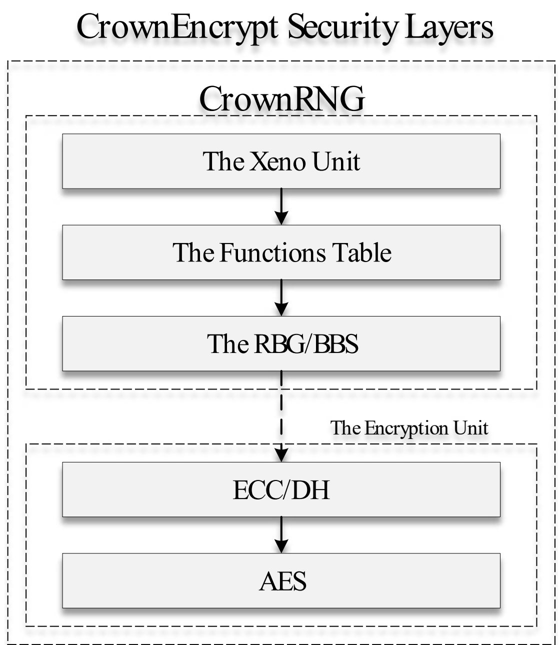

Figure 9: A schematic representation of CrownEncrypt's five layers of security.

III- Crown Sterling One-Time Pad Cryptographic Solution

One-Time Pad Cryptography (OTP) is encryption that cannot be cracked[17,18]. It requires the use of a one-time pre-shared key/pad having the same size as, or longer than, the message being sent (hence the name one-time pad). It is first described by Frank Miller[19], dating back to the late 1800s.

The message to be encrypted is paired with the secret pad/key such that each bit of the message is combined with a corresponding bit from the pad/key using modular addition (the XOR function in our case). The resulting ciphertext will be impossible to decrypt or break given the following four conditions are all met:

1- The key must be truly random.
2- The key must be at least as long as the plaintext.
3- The key must never be reused in whole or in part.
4- The key must be kept completely secret.

All the above conditions are met in the Crown Sterling OTP solutions, where the keys are generated from CrownRNG, which, as we illustrated above, produces highly randomized streams of numbers. Additionally, the ECC-DH key exchange protocol used is the standard in secured key-sharing.

The main reason why OTP cryptography is not in wide usage, even though it offers unbreakable encryption, is due to the difficulty arising from sharing the pad/key, which is as large as or larger than the message itself. Crown Sterling solved this problem by generating keys using the square root function where the problem of sharing the whole key is reduced to simply sharing the number that generates it instead, the NPSN, which is much smaller than the whole message and can be securely and easily exchanged using the usual ECC-DH protocol.

There is a misconception that OTP is a stream cipher which arises from the fact that stream ciphers, in many ways, mimic OTP. Note that the deviations stream ciphers have from OTP are what compromise their security. OTP requires a random key that is equal in length to the data being encrypted. The key contains random digits, and any given string of digits cannot be used more than once, which ensures the highest level of security. The digits in the key come from the mantissas of NPSNs. These mantissas are proven to not contain repeating strings and have been shown to perform very well in various statistical tests for randomness. The CrownRNG random number generator produces 2.1472 billion bits (netting 870 MB) of random key material. Multiple NPSNs can be used to derive square root values that can be combined to achieve longer data transfers. In contrast, stream ciphers use a 128 or 256-bit key, therefore generating a pseudorandom keystream that may contain repeating strings, distinguishing them from a true one-time pad.

Crown Sterling OTP solution is made of three basic units:

1- CrownRNG.
2- Key exchange protocol unit.
3- Message encryption unit.

1. <u>CrownRNG</u>

This is the same RNG explained above. It supplies the next unit with a key of highly random bits.

2. **Key Exchange Protocol Unit**

This is mainly an ECC-DH unit responsible for securing the sharing of the required metrics, coming mainly from CrownRNG, such as the index numbers and the NSPN.

3. **Encryption Unit**

Instead of encrypting the message using the AES algorithm, as in CrownEncrypt, the key will undergo mathematical operations first and then be passed on to an XOR-based algorithm instead. First, the key is converted into a 10-base numeric system. The random, last digit provided by CrownRNG will be attached to its end to ensure it is converted into an NPSN. Next, the square root of this number is calculated. Therefore, in our case, the pad/key length is equal to that of the message. The message and the key are then converted into binary forms before they are added together using the XOR-based function. In its simplest form, the XOR logical function adds the zeroes and ones of the binary format as follows:

		XOR
1	1	0
1	0	1
0	1	1
0	0	0

Table 7: The logical outcome of the XOR function.

Crown Sterling OTP solution is utilized in two different versions. One version is CrownSovereignOTP, which provides a quantum-secured environment for the state transition functions of blockchains, while the other, CrownEncryptOTP, provides the same level of quantum security for messaging exchange.

IV- CrownSovereignOTP for Quantum Resistant State Transition Functions (STF) of the Blockchain

There is a threat to the security of blockchains as all blockchains use authentication algorithms for enabling participants to secure transactions. All the authentication algorithms currently rely on modern non-quantum-resistant cryptography protocols, such as Bitcoin's Pay to Public Key (P2PK) algorithm, which is an Elliptic Curve Digital Signature Algorithm. These authentication mechanisms are becoming more and more vulnerable due to the looming threat of quantum computers because of their ability to perform, using Shor's algorithm[20], large number factorization required to decrypt message text. Shor's algorithm represents a material risk to current blockchain cryptographic protocols and their STFs. The STF is the logic of the blockchain that determines how the state changes when a block is processed (here, 'state' refers to data that persists between blocks). The term STF is often used synonymously to blockchain runtime.

Below, we present the Crown Sterling One-Time Pad Blockchain technology, CrownSovereignOTP. We also describe the Pay to One-Time Pad Key (P2OTPK), a quantum-resistant authentication protocol, and other integral components of CrownSovereignOTP.

Pay To One-Time Pad Key (P2OTPK)

The security of P2OTPK is based on irrational numbers. P2OTPK relies on cryptographically secure random number generation. The components of P2OTPK are:

1- NPSN: The non-perfect square number.
2- INX: The index of the mantissa.
3- LEN: the length of the one-time pad key.
4- OTPK: the one-time pad key.
5- ID: The id of the receiver.

The protocol of P2OTPK consists of two main processes:

1- Locking/Sending: The process of locking a transaction.
2- Unlocking / Receiving: The process of unlocking a transaction.

The process of locking/sending a transaction goes as follows:

1) Generate the required components: NPSN, INX, LEN.
2) Square root the NPSN. Let the result be SRNPSN.
3) Derive the OTPK by indexing into the mantissa of SRNPSN using INX as the starting point and ending at INX + LEN. This is the OTPK.
4) Send the desired value (e.g., token amount) as a transaction with the ID and OTPK to the One-Time Pad blockchain
5) Offline transfer the NPSN, LEN, and INX to the owner of ID.

The process of unlocking/receiving a transaction goes as follows:

1- Use NPSN, INX and LEN received from the sending party offline.

2- Send a transaction with the desired value to unlock (e.g., token amount) from the locked transaction with the NPSN, INX, and LEN.
3- When the One-Time Pad blockchain receives the transaction, it extracts the NPSN, INX, and LEN then initiates the authentication process to unlock the value in the locked transaction as follows:
   a) Square root the NPSN. Let the result be SRNPSN.
   b) Derive the OTPK by indexing into the mantissa of SRNPSN starting at and ending at INX + LEN.
   c) Compare the derived OTPK with the OTPK in the locked transaction.
   d) If equal, the locked transaction gets unlocked, and the user transactions execute successfully, thus accessing the value of the locked transaction (e.g., token amount).
   e) e. If not equal, the transaction gets rejected.

Below is a schematic representation of the workflow of CrownSovereignOTP.

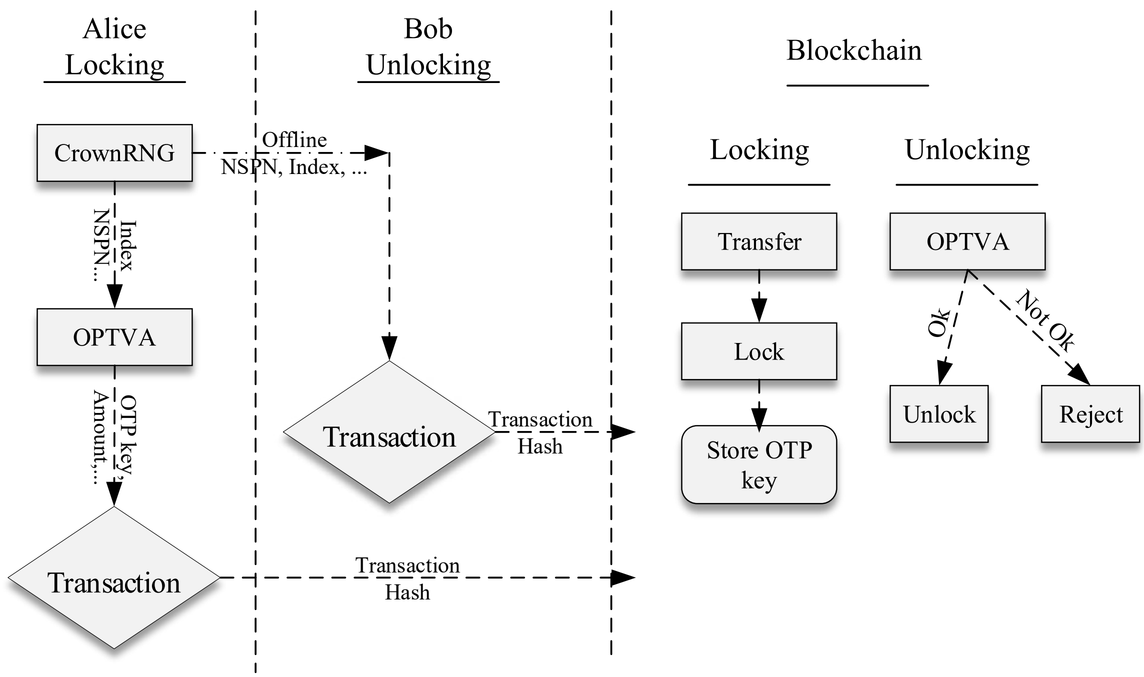

Figure 10: The workflow of CrownSovereignOTP.

V- CrownEncryptOTP™ for Quantum Secure Messaging

CrownEncryptOTP utilizes all the elements of the Crown Sterling OTP solution. Additionally, and for maximum security, a multi-factor authentication and partial key transport method is implemented. In this method, CrownRNG creates two NPSN; one is used directly (online) through ECC-DH to create a private key, shared by the two parties (Alice and Bob), and through which the index is encrypted. The other NPSN is transformed into a QR code and transferred indirectly (offline) from one party (Alice) to the other (Bob) using a multi-factor authentication method. Thus Bob receives the index using an online ECC-DH, while he receives the NPSN through an offline ECC-DH. When the two partial keys are combined, the index and the NSPN, the full key is generated, and the encrypted message can be decrypted.

Below is a schematic drawing for the CrownEncryptOTP workflow.

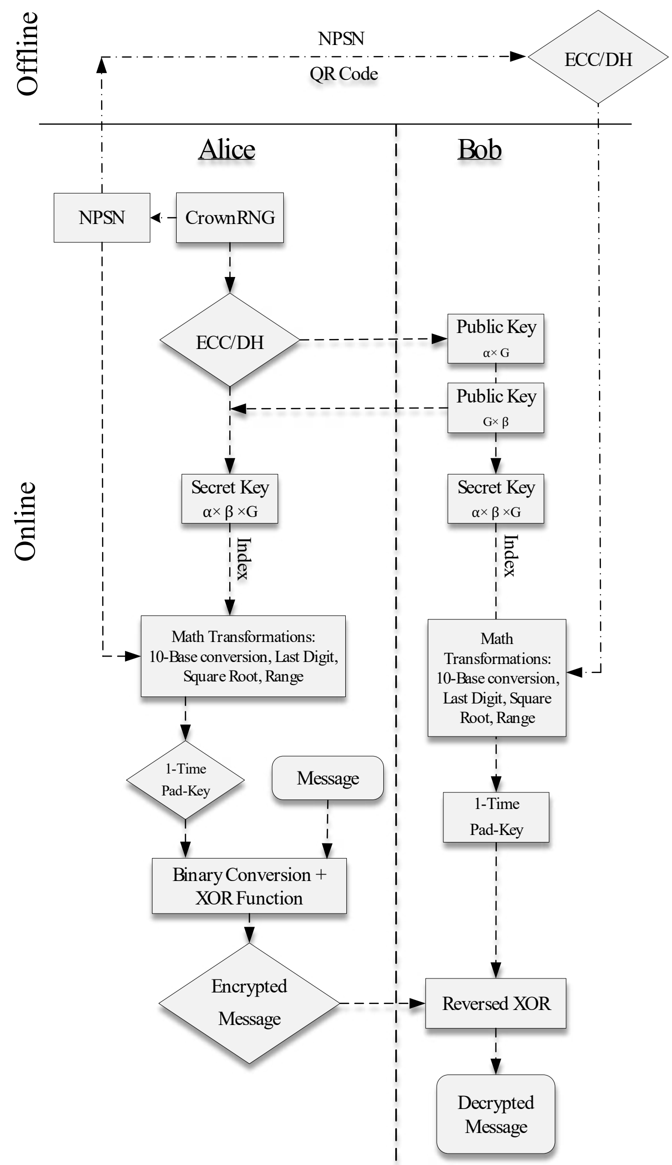

Figure 11: A schematic representation of the workflow of the CrownEncryptOTP with a partial-key distribution.

Appendix

A- Partial List of Functions that Generate Irrational Numbers:

The Square Root Function:

Taking the square root of a non-square integer creates an irrational number, such as $\sqrt{2}, \sqrt{3}, etc.$ This is guaranteed when prime numbers or non-sqaure numbers are used.

The Logarithm Function:

This method uses the natural log of integers: $log_n x$. One required condition is that *n*, the order of the log, has one prime factor, at least, that is not also a factor of *x*.

The Power Function:

This method rests on the fact that raising any algebraic number *y* to the power of another irrational algebraic number *x* (y^x) is sure to generate an irrational number.

The Inverse-Power Function:

Where $y^{\frac{1}{x}}$ is irrational for both integers *y* and *x* except when *y* is the x^{th} power of some integer, of course.

The Trigonometric Function:

Based on Niven's theorem, the tan(*x*), cos(*x*), and every other trigonometric function of any rational number *x* that is not equal to 0, is irrational.

Polynomial Function:

Where the solution of a polynomial equation of order *n* is either a natural number or irrational.

$$x^n + c_{n-1}x^{n-1} + \cdots + c_0 = 0$$

B- NIST Tests Results for the Cosine Function:

Below are the NIST tests' results for the cos(1450) trigonometric function and for a single number with a mantissa length of 1660957 binary digits. Notice how the random number generates a *p*-value larger than 0.01 for all the tests.

Test Name	**P-Value**
Frequency	0.099978
Block Frequency	0.184093
Run	0.260697
Longest Run	0.388854
Matrix	0.185961
Spectral	0.693646
Non-overlapping Template	0.532113
Overlapping Template	0.090185
Universal	0.933074
Complexity	0.180485

Serial	0.706031
Entropy	0.606385
Cumulative Sum Forward	0.249214
Cumulative Sum Backward	0.543305
Random Excursion	0.279346
Random Excursion Variant	0.661555

C- Dieharder Full Testing Report

When running the randomness testing, the Dieharder test suite recommends having a minimum dataset size of 15GB for each analysis. Datasets from Archimedes were created in the 15GB – 20GB range by concatenating individual 100MB entropy files from separate runs of the tool. These individual 100MB data files were created on multiple Ubuntu virtual machines running the Archimedes Docker container V1.0.3 and written to the local file system. A total of 10 datasets of ~20GB each, representing a cumulative total of ~200GB of data, were tested.

The Dieharder test flags used were as follows:

• dieharder -a -g 201 -k 2 -Y 1 -f <Input File>

- -a = run all tests
- -g 201 = use external data file for entropy
- -k 2 = use high precision on p-samples
- -Y 1 = Resolve Ambiguity (RA) – this flag will rerun any tests that return an initial weak result. RA mode adds p-samples (usually in blocks of 100) until the test result ends up solidly not weak or proceeds to unambiguous failure. Any initial or subsequent failure of any individual test constituted an immediate failure of that individual Test with no RA rerun initiated.
- -f = Filename of input datafile

The ENT test flags were as follows:

• ent -c -t <Input File>

- -c = create a table of value occurrences from 0-255 with % distribution
- -t = terse mode with output written in CSV format

For each Dieharder and ENT test run, the output results were piped to a text file and saved to the local file system.

Dieharder Testing Results

All datasets were processed through Dieharder suite representing thousands of individual tests.
The following tests were performed:

- diehard_birthdays marsaglia_tsang_gcd
- diehard_operm5 sts_monobit
- diehard_rank_32x32 sts_runs
- diehard_rank_6x8 sts_serial

- diehard_bitstream rgb_bitdist
- diehard_opso rgb_minimum_distance
- diehard_dna rgb_permutations
- diehard_count_1s_str rgb_lagged_sum
- diehard_count_1s_byt rgb_kstest_test
- diehard_parking_lot dab_bytedistrib
- diehard_2dsphere dab_dct
- diehard_3dsphere dab_filltree
- diehard_squeeze dab_filltree
- diehard_sums dab_filltree2
- diehard_runs dab_filltree2
- diehard_craps dab_monobit2

Note: Tests in yellow have been flagged by the tool's developer as having suspect or unreliable results. Results for these tests were included for completeness.

Installed dieharder tests:

Test Number	Test Name	Test Reliability
-d 0	Diehard Birthdays Test	Good
-d 1	Diehard OPERM5 Test	Suspect
-d 2	Diehard 32x32 Binary Rank Test	Good
-d 3	Diehard 6x8 Binary Rank Test	Good
-d 4	Diehard Bitstream Test	Good
-d 5	Diehard OPSO	Good
-d 6	Diehard OQSO Test	Good
-d 7	Diehard DNA Test	Good
-d 8	Diehard Count the 1s (stream) Test	Good
-d 9	Diehard Count the 1s Test (byte)	Good
-d 10	Diehard Parking Lot Test	Good
-d 11	Diehard Minimum Distance (2d Circle) Test	Good
-d 12	Diehard 3d Sphere (Minimum Distance) Test	Good
-d 13	Diehard Squeeze Test	Good
-d 14	Diehard Sums Test	Do Not Use
-d 15	Diehard Runs Test	Good
-d 16	Diehard Craps Test	Good
-d 17	Marsaglia and Tsang GCD Test	Good
-d 100	STS Monobit Test	Good
-d 101	STS Runs Test	Good
-d 102	STS Serial Test (Generalized)	Good
-d 200	RGB Bit Distribution Test	Good
-d 201	RGB Generalized Minimum Distance Test	Good
-d 202	RGB Permutations Test	Good
-d 203	RGB Lagged Sum Test	Good
-d 204	RGB Kolmogorov-Smirnov Test Test	Good

Table 8: Dieharder battery test results for CrownRNG, performed by Crown Sterling.

The following table represents the results for the first 4 Archimedes datasets. Note that when running the sts_serial tests, if a single test reported weak results, the entire set of 30 sts_serial tests are rerun during the RA process. All 30 individual sts_serial tests must pass or fail for the process to move to the next test in the sequence.

Command to generate the data = diehard -a -g 201 -k 2 -Y 1 -f "datafile name" > Entropy##Results-DIE.txt

DIEHARDER v3.31.1

YELLOW CELL = Resolve Ambiguity - Passed

			EntropySet01.bin			EntropySet02.bin			EntropySet03.bin			EntropySet04.bin		
TEST NAME	ntup	tsamples	psamples	p-value	Assessment	psamples	p-value	Assessment	psamples	p-value	Assessment	psamples	p-value	Assessment
diehard_birthdays	0	100	100	0.71685576	PASSED	100	0.60537677	PASSED	100	0.36842528	PASSED	100	0.58325289	PASSED
diehard_operm5	0	1000000	100	0.5792205	PASSED	100	0.76692354	PASSED	100	0.96987557	PASSED	100	0.95432053	PASSED
diehard_rank_32x32	0	40000	100	0.93099737	PASSED	100	0.50356511	PASSED	100	0.7825877	PASSED	100	0.10405138	PASSED
diehard_rank_6x8	0	100000	100	0.5302821	PASSED	100	0.223979	PASSED	100	0.8840799	PASSED	100	0.56812365	PASSED
diehard_bitstream	0	2097152	100	0.10069532	PASSED	100	0.15979312	PASSED	200	0.55030535	PASSED	100	0.32916713	PASSED
diehard_opso	0	2097152	200	0.68522913	PASSED	100	0.45712577	PASSED	100	0.16841943	PASSED	100	0.90879155	PASSED
diehard_oqso	0	2097152	100	0.1045786	PASSED	100	0.96605293	PASSED	100	0.99351691	PASSED	100	0.89655682	PASSED
diehard_dna	0	2097152	100	0.70061882	PASSED	100	0.66424424	PASSED	100	0.91271732	PASSED	100	0.3677377	PASSED
diehard_count_1s_str	0	256000	100	0.82990879	PASSED	100	0.61434153	PASSED	100	0.81068258	PASSED	100	0.12291007	PASSED
diehard_count_1s_byt	0	256000	100	0.78203148	PASSED	100	0.19447805	PASSED	100	0.01000195	PASSED	100	0.3548228	PASSED
diehard_parking_lot	0	12000	100	0.07906014	PASSED	100	0.90162649	PASSED	100	0.08126248	PASSED	100	0.03660255	PASSED
diehard_2dsphere	2	8000	100	0.89730762	PASSED	100	0.77510958	PASSED	100	0.15903098	PASSED	100	0.12261519	PASSED
diehard_3dsphere	3	4000	100	0.323029	PASSED	100	0.51813741	PASSED	100	0.24262982	PASSED	100	0.86526076	PASSED
diehard_squeeze	0	100000	100	0.05418857	PASSED	100	0.18971518	PASSED	100	0.35927275	PASSED	100	0.95090579	PASSED
diehard_sums	0	100	100	0.49112947	PASSED	100	0.02631941	PASSED	100	0.48132626	PASSED	100	0.06075829	PASSED
diehard_runs	0	100000	100	0.95204128	PASSED	100	0.44643272	PASSED	100	0.00664701	PASSED	100	0.01815349	PASSED
diehard_runs	0	100000	100	0.75692122	PASSED	100	0.9921304	PASSED	100	0.29206046	PASSED	100	0.95996994	PASSED
diehard_craps	0	200000	100	0.63156512	PASSED	100	0.48219676	PASSED	100	0.27295377	PASSED	100	0.97798712	PASSED
diehard_craps	0	200000	100	0.14098989	PASSED	100	0.12246671	PASSED	100	0.96315494	PASSED	100	0.9255788	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.9919605	PASSED	100	0.26747079	PASSED	100	0.9494302	PASSED	100	0.77520028	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.56260623	PASSED	100	0.95900092	PASSED	100	0.41807999	PASSED	100	0.41220702	PASSED
sts_monobit	1	100000	100	0.69111635	PASSED	100	0.8011193	PASSED	100	0.13125362	PASSED	100	0.06726843	PASSED
sts_runs	2	100000	100	0.07567638	PASSED	100	0.92333734	PASSED	100	0.23287764	PASSED	100	0.26034508	PASSED
sts_serial	1	100000	100	0.7047525	PASSED	300	0.04136523	PASSED	100	0.02502924	PASSED	200	0.13891262	PASSED
sts_serial	2	100000	100	0.17760135	PASSED	300	0.16317266	PASSED	100	0.06228787	PASSED	200	0.85009961	PASSED
sts_serial	3	100000	100	0.5945386	PASSED	300	0.93125563	PASSED	100	0.48527087	PASSED	200	0.05679462	PASSED
sts_serial	3	100000	100	0.91213017	PASSED	300	0.22295868	PASSED	100	0.74379596	PASSED	200	0.13278586	PASSED
sts_serial	4	100000	100	0.95871859	PASSED	300	0.4286502	PASSED	100	0.46447208	PASSED	200	0.69191747	PASSED
sts_serial	4	100000	100	0.22325645	PASSED	300	0.93852095	PASSED	100	0.88099854	PASSED	200	0.72422918	PASSED
sts_serial	5	100000	100	0.61198869	PASSED	300	0.98544468	PASSED	100	0.66148701	PASSED	200	0.54564861	PASSED
sts_serial	5	100000	100	0.78261568	PASSED	300	0.45067938	PASSED	100	0.84537409	PASSED	200	0.80546466	PASSED
sts_serial	6	100000	100	0.65182379	PASSED	300	0.02300516	PASSED	100	0.88835287	PASSED	200	0.7752131	PASSED
sts_serial	6	100000	100	0.96845467	PASSED	300	0.79342563	PASSED	100	0.93711666	PASSED	200	0.96066714	PASSED
sts_serial	7	100000	100	0.48329619	PASSED	300	0.38952267	PASSED	100	0.59657784	PASSED	200	0.41736604	PASSED
sts_serial	7	100000	100	0.81349434	PASSED	300	0.57297946	PASSED	100	0.52719765	PASSED	200	0.23561475	PASSED
sts_serial	8	100000	100	0.89732543	PASSED	300	0.80188623	PASSED	100	0.61777902	PASSED	200	0.27749453	PASSED
sts_serial	8	100000	100	0.78651174	PASSED	300	0.15668977	PASSED	100	0.98350474	PASSED	200	0.31251522	PASSED
sts_serial	9	100000	100	0.88958356	PASSED	300	0.586994	PASSED	100	0.86402319	PASSED	200	0.55843995	PASSED
sts_serial	9	100000	100	0.72668123	PASSED	300	0.8429107	PASSED	100	0.63572672	PASSED	200	0.51420533	PASSED
sts_serial	10	100000	100	0.72859225	PASSED	300	0.49195724	PASSED	100	0.8664598	PASSED	200	0.54122624	PASSED
sts_serial	10	100000	100	0.4020343	PASSED	300	0.88114239	PASSED	100	0.62872409	PASSED	200	0.34742613	PASSED
sts_serial	11	100000	100	0.67746792	PASSED	300	0.52328047	PASSED	100	0.72776658	PASSED	200	0.64147485	PASSED
sts_serial	11	100000	100	0.24309987	PASSED	300	0.85847756	PASSED	100	0.65562401	PASSED	200	0.46672655	PASSED
sts_serial	12	100000	100	0.35820997	PASSED	300	0.4734149	PASSED	100	0.04907307	PASSED	200	0.82718156	PASSED
sts_serial	12	100000	100	0.40166975	PASSED	300	0.76768472	PASSED	100	0.02730176	PASSED	200	0.68901543	PASSED
sts_serial	13	100000	100	0.60524144	PASSED	300	0.56835703	PASSED	100	0.49760852	PASSED	200	0.63436396	PASSED
sts_serial	13	100000	100	0.96407621	PASSED	300	0.63560329	PASSED	100	0.24714651	PASSED	200	0.23750082	PASSED
sts_serial	14	100000	100	0.51971174	PASSED	300	0.81242699	PASSED	100	0.10639105	PASSED	200	0.99454207	PASSED
sts_serial	14	100000	100	0.30604386	PASSED	300	0.3394023	PASSED	100	0.02361182	PASSED	200	0.99408855	PASSED
sts_serial	15	100000	100	0.16198215	PASSED	300	0.75897456	PASSED	100	0.76796944	PASSED	200	0.58899147	PASSED
sts_serial	15	100000	100	0.83841164	PASSED	300	0.98039692	PASSED	100	0.17804237	PASSED	200	0.79008483	PASSED
sts_serial	16	100000	100	0.29647402	PASSED	300	0.36431747	PASSED	100	0.9527724	PASSED	200	0.62740437	PASSED
sts_serial	16	100000	100	0.8597187	PASSED	300	0.97707496	PASSED	100	0.55712739	PASSED	200	0.37509087	PASSED
rgb_bitdist	1	100000	100	0.81469815	PASSED	100	0.18258382	PASSED	100	0.75015101	PASSED	100	0.05561871	PASSED
rgb_bitdist	2	100000	100	0.50952618	PASSED	100	0.69881122	PASSED	100	0.92840234	PASSED	100	0.54465966	PASSED
rgb_bitdist	3	100000	100	0.84134091	PASSED	200	0.79706323	PASSED	100	0.82785905	PASSED	100	0.90789783	PASSED
rgb_bitdist	4	100000	100	0.34387694	PASSED	100	0.56107955	PASSED	100	0.88046374	PASSED	100	0.32921824	PASSED
rgb_bitdist	5	100000	100	0.81701916	PASSED	100	0.84364538	PASSED	100	0.76299848	PASSED	100	0.83626876	PASSED
rgb_bitdist	6	100000	100	0.90355289	PASSED	100	0.28942127	PASSED	100	0.02736031	PASSED	100	0.37944054	PASSED
rgb_bitdist	7	100000	100	0.08266363	PASSED	100	0.05116329	PASSED	100	0.92237438	PASSED	100	0.01766868	PASSED
rgb_bitdist	8	100000	100	0.58756192	PASSED	100	0.72253155	PASSED	100	0.33495541	PASSED	100	0.89550341	PASSED
rgb_bitdist	9	100000	100	0.80503141	PASSED	100	0.37198844	PASSED	200	0.35992296	PASSED	100	0.60659977	PASSED
rgb_bitdist	10	100000	100	0.9880369	PASSED	100	0.8532545	PASSED	100	0.60660483	PASSED	100	0.87033102	PASSED
rgb_bitdist	11	100000	100	0.94870399	PASSED	100	0.11074833	PASSED	100	0.04354972	PASSED	100	0.95643405	PASSED
rgb_bitdist	12	100000	100	0.28802247	PASSED	100	0.9803487	PASSED	100	0.39044962	PASSED	100	0.51274853	PASSED
rgb_minimum_distance	2	10000	1000	0.90868453	PASSED	1000	0.02022633	PASSED	1000	0.20420056	PASSED	1000	0.87168571	PASSED
rgb_minimum_distance	3	10000	1000	0.23290135	PASSED	1000	0.35864541	PASSED	1000	0.45928531	PASSED	1000	0.69013417	PASSED
rgb_minimum_distance	4	10000	1000	0.82295884	PASSED	1000	0.33503913	PASSED	1000	0.75343406	PASSED	1000	0.49368635	PASSED
rgb_minimum_distance	5	10000	1000	0.44221128	PASSED	1000	0.86110925	PASSED	1000	0.4059959	PASSED	1000	0.46209913	PASSED
rgb_permutations	2	100000	100	0.69689809	PASSED	100	0.73546984	PASSED	100	0.80781758	PASSED	100	0.37014857	PASSED
rgb_permutations	3	100000	100	0.95384999	PASSED	100	0.54786804	PASSED	100	0.68905844	PASSED	100	0.86782578	PASSED
rgb_permutations	4	100000	100	0.69091694	PASSED	100	0.64332397	PASSED	100	0.80682049	PASSED	100	0.49591174	PASSED
rgb_permutations	5	100000	100	0.54486475	PASSED	100	0.93242666	PASSED	100	0.67870627	PASSED	100	0.76821983	PASSED
rgb_lagged_sum	0	1000000	100	0.80251177	PASSED	100	0.85167751	PASSED	100	0.90948058	PASSED	100	0.15928811	PASSED
rgb_lagged_sum	1	1000000	100	0.41626131	PASSED	200	0.97520711	PASSED	100	0.68388816	PASSED	100	0.08214551	PASSED
rgb_lagged_sum	2	1000000	100	0.28753389	PASSED	100	0.65130546	PASSED	100	0.68004444	PASSED	100	0.59604819	PASSED
rgb_lagged_sum	3	1000000	100	0.09888694	PASSED	100	0.89448078	PASSED	100	0.92996069	PASSED	100	0.63474933	PASSED
rgb_lagged_sum	4	1000000	100	0.95978206	PASSED	100	0.41530882	PASSED	100	0.92260674	PASSED	100	0.87316685	PASSED
rgb_lagged_sum	5	1000000	100	0.97950247	PASSED	100	0.41423702	PASSED	100	0.6919032	PASSED	100	0.8612622	PASSED
rgb_lagged_sum	6	1000000	100	0.09385282	PASSED	100	0.09516962	PASSED	100	0.08027648	PASSED	100	0.41169229	PASSED
rgb_lagged_sum	7	1000000	100	0.71340553	PASSED	100	0.81963827	PASSED	100	0.34655986	PASSED	100	0.61823383	PASSED
rgb_lagged_sum	8	1000000	100	0.4864485	PASSED	100	0.80415356	PASSED	100	0.52802698	PASSED	200	0.592901	PASSED
rgb_lagged_sum	9	1000000	100	0.71966417	PASSED	100	0.10644656	PASSED	100	0.22052925	PASSED	100	0.46085014	PASSED
rgb_lagged_sum	10	1000000	100	0.60712033	PASSED	100	0.75886123	PASSED	100	0.77322015	PASSED	100	0.97538064	PASSED
rgb_lagged_sum	11	1000000	100	0.05957404	PASSED	100	0.72364609	PASSED	100	0.07811603	PASSED	100	0.81743124	PASSED
rgb_lagged_sum	12	1000000	100	0.1819899	PASSED	100	0.43156707	PASSED	100	0.57740552	PASSED	100	0.63389743	PASSED
rgb_lagged_sum	13	1000000	100	0.53544413	PASSED	100	0.51253489	PASSED	100	0.97243465	PASSED	100	0.99242397	PASSED
rgb_lagged_sum	14	1000000	100	0.91373371	PASSED	100	0.46489384	PASSED	100	0.54960746	PASSED	100	0.13596265	PASSED
rgb_lagged_sum	15	1000000	100	0.70150318	PASSED	100	0.08612775	PASSED	100	0.01089496	PASSED	100	0.26600852	PASSED
rgb_lagged_sum	16	1000000	100	0.79431569	PASSED	100	0.54653929	PASSED	100	0.68560234	PASSED	100	0.22676009	PASSED
rgb_lagged_sum	17	1000000	100	0.54064313	PASSED	100	0.17083459	PASSED	100	0.27996205	PASSED	100	0.61133056	PASSED
rgb_lagged_sum	18	1000000	100	0.53040019	PASSED	100	0.55107525	PASSED	100	0.48451517	PASSED	200	0.84373556	PASSED
rgb_lagged_sum	19	1000000	100	0.59777043	PASSED	100	0.02509667	PASSED	100	0.22877754	PASSED	100	0.53819268	PASSED
rgb_lagged_sum	20	1000000	100	0.57230024	PASSED	100	0.42132707	PASSED	100	0.71738613	PASSED	100	0.92032879	PASSED
rgb_lagged_sum	21	1000000	100	0.22365989	PASSED	100	0.45639716	PASSED	100	0.23612556	PASSED	100	0.5497902	PASSED
rgb_lagged_sum	22	1000000	100	0.21845417	PASSED	100	0.62271571	PASSED	100	0.53010231	PASSED	100	0.07337567	PASSED
rgb_lagged_sum	23	1000000	100	0.06650593	PASSED	100	0.77854179	PASSED	100	0.20911675	PASSED	100	0.39431224	PASSED
rgb_lagged_sum	24	1000000	100	0.18380305	PASSED	100	0.1493375	PASSED	100	0.60487154	PASSED	100	0.90805244	PASSED
rgb_lagged_sum	25	1000000	100	0.90557516	PASSED	100	0.05992653	PASSED	100	0.52678138	PASSED	100	0.88292659	PASSED
rgb_lagged_sum	26	1000000	100	0.98776738	PASSED	100	0.05931362	PASSED	100	0.36629044	PASSED	100	0.03827676	PASSED
rgb_lagged_sum	27	1000000	100	0.01576437	PASSED	100	0.83615229	PASSED	100	0.22054541	PASSED	100	0.59648443	PASSED
rgb_lagged_sum	28	1000000	100	0.09900715	PASSED	100	0.08600062	PASSED	100	0.31445477	PASSED	100	0.540884	PASSED
rgb_lagged_sum	29	1000000	100	0.61676755	PASSED	100	0.1000821	PASSED	200	0.45077377	PASSED	100	0.97707311	PASSED
rgb_lagged_sum	30	1000000	200	0.30259938	PASSED	100	0.4895755	PASSED	100	0.1882678	PASSED	100	0.2696096	PASSED
rgb_lagged_sum	31	1000000	100	0.98502982	PASSED	100	0.51104972	PASSED	100	0.55951485	PASSED	100	0.93869399	PASSED
rgb_lagged_sum	32	1000000	100	0.50181708	PASSED	100	0.55678178	PASSED	100	0.88400493	PASSED	100	0.02912613	PASSED
rgb_kstest_test	0	10000	1000	0.74848435	PASSED	1000	0.2718663	PASSED	1000	0.72478362	PASSED	1000	0.59770779	PASSED
dab_bytedistrib	0	51200000	1	0.39521864	PASSED	1	0.98378034	PASSED	1	0.80240791	PASSED	1	0.64540673	PASSED
dab_dct	256	50000	1	0.37839355	PASSED	1	0.16606725	PASSED	1	0.00745721	PASSED	1	0.9149572	PASSED
dab_filltree	32	15000000	1	0.66086514	PASSED	1	0.25150052	PASSED	1	0.23603564	PASSED	1	0.65510269	PASSED
dab_filltree	32	15000000	1	0.76608163	PASSED	1	0.56793111	PASSED	1	0.09752038	PASSED	1	0.92261503	PASSED
dab_filltree2	0	5000000	1	0.95373421	PASSED	1	0.69639142	PASSED	1	0.68390862	PASSED	1	0.98105107	PASSED
dab_filltree2	1	5000000	1	0.03211056	PASSED	1	0.36405214	PASSED	1	0.70216697	PASSED	1	0.60663441	PASSED
dab_monobit2	12	65000000	1	0.569971	PASSED	1	0.87352395	PASSED	1	0.8443833	PASSED	1	0.51151241	PASSED

Table 9: Dieharder battery test results for CrownRNG, performed on four different data sets.

All datasets were processed through ENT test tool. All the individual tests (Entropy, Chi-square, Mean, Monte Carlo Pi, and Serial Correlation) passed. The distribution of the bit values of 0 and 1 were 50%/50% with a general range of +/- 0.0002%. The distribution of byte values from 00-FF also showed linear distribution with ~0.3906 – 0.3607% per value. The following table represents the results for each of the first four datasets.

	EntropySet01	EntropySet02	EntropySet03	EntropySet04
File Size	19.8GB	19.0GB	20GB	20.1GB
Entropy	8	8	8	8
Chi-Square	242.3568	284.1494	266.0102	218.0463
Mean	127.5001	127.5002	127.4996	127.4993
Monte Carlo Pi	3.141578	3.141587	3.141627	3.141602
Serial Corr.	-4E-06	0	0.000005	0.000003

Table 10: The entropy results of four different data sets of CrownRNG

ENT Testing Results

All datasets were processed through ENT test tool. All the individual tests (Entropy, Chi-square, Mean, Monte Carlo Pi, and Serial Correlation) passed. The distribution of the bit values of 0 and 1 were 50%/50% with a general range of +/- 0.0002%. The distribution of byte values from 00-FF also showed linear distribution with ~0.3906 – 0.3607% per value. The following table represents the results for each of the first four datasets:

Test Result Files

Each of the datasets and the individual Dieharder and ENT results for each dataset are available from Crown Sterling.

References

[1] Luka Milinković , Marija Antić , and Zoran Čiča. Pseudo-random number generator based on irrational numbers. 10th International Conference on Telecommunication in Modern Satellite Cable and Broadcasting Services (TELSIKS) (2011).

[2] Johnson, B. R., Leeming, D. J., A Study of the Digits of π, e, and Certain Other Irrational Numbers. University of Victoria, Canada (1990).

[3] Hardy, G. H. Quadratic Surds. 10th ed. Cambridge University Press (1967).

[4] Hughes, C. R.. Irrational Roots. The Mathematical Gazette, 83(498), 502–503. (1999)

[5] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. SIAM J. Comput. Vol. 15, No. 2 (1986).

[6] Andrew Rukhin et al. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST Special Publication 800-22 (2010).

[7] P. L'Ecuyer and R. Simard, TestU01: A C Library for Empirical Testing of Random Number Generators ACM Transactions on Mathematical Software, Vol. 33, article 22, 2007.

[8] Practically Random. http://pracrand.sourceforge.net.

[9] Robert G. Brown et al. Dieharder: A Random Number Test Suite. http: //webhome.phy.duke.edu/~rgb/General/dieharder.php Based on earlier work by George Marsaglia on the DIEHARD test suite.

[10] Diffie, W., and Hellman, M. New directions in cryptography. IEEE Trans. Inform. Theory IT-22 (1976).

[11] Diffie, W., and Hellman, M. Exhaustive cryptanalysis of the NBS data encryption standard. Computer 10 (1977).

[12] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM, Vol 21 (1978).

[13] Robert E. Grant and Talal Ghannam. Accurate and Infinite Prime Prediction from Novel Quasi-Prime Analytical Methodology. Arxiv.org, arXiv:1903.08570 (2019).

[14] Jeremy Wohlwend. Elliptic curve cryptography: pre and post-quantum. Conference Proceedings (2016).

[15] Sheetal Kalra and Sandeep K. Sood. Elliptic Curve Cryptography: Current Status and Research Challenges. Springer, Communications in Computer and Information Science, vol 169 (2011).

[16] Bruce Schneier; John Kelsey; Doug Whiting; David Wagner; Chris Hall; Niels Ferguson; Tadayoshi Kohno; et al. The Twofish Team's Final Comments on AES Selection (2000).

[17] Schneier, Bruce. Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley and Sons, Inc. (1996).

[18] Menezes, Alfred J., Paul C. van Oorschot, and Scott A. Vanstone. Handbook of Applied Cryptography, CRC Press (1997).

[19] Miller, Frank. Telegraphic code to ensure privacy and secrecy in the transmission of telegrams. C.M. Cornwell (1882).

[20] Shor, P.W. Algorithms for quantum computation: discrete logarithms and factoring. Proceedings 35th Annual Symposium on Foundations of Computer Science. IEEE Comput. Soc. Press: 124–134, (1994).