SALT - a Simple and Lightweight Testsuite for RNGs

Student Name:

Supervisor Name:

Submitted as part of the degree of MEng Computer Science to the

Board of Examiners in the Department of Computer Sciences, Durham University

I ABSTRACT

Random Number Generators (RNGs) are vital for securing communication and furthering research in the modern-day world; they are used across the globe to underpin the security of cryptosystems and throughout scientific investigations in Monte-Carlo based simulations. Without high-quality sources of random numbers, these simulations collapse into predetermined models and attackers can circumvent encryption by generating decryption keys to match the non-random private-keys, gaining access to confidential information without having to break the main cryptosystem. Unfortunately, true randomness is hard to generate quickly, so several Pseudo-RNGs (PRNGs) have been devised to quickly generate sufficiently random numbers. Subsequently, tests have been developed to certify the randomness of RNGs and these tests are combined into testsuites which aim to prove the randomness of a generator. A further issue persists, IoT devices can often lose entropy due to deterministic use, this results in even the best PRNG's output collapsing along with it. SALT provides a lightweight testsuite that can quickly run on minimal generator output (less than 1Kb) to prove that it is still working as expected. Thus, filling a much-needed gap in the literature, which currently focuses on proving the theoretical randomness of RNGs and validating their output using over 20Gb of data. Finally, a Python interface and website are provided to allow for efficient testing of extremely limited RNG output.

Keywords — Random Number Generators, PRNGs, CSPRNGs, Statistical Testing, Testsuites, Test batteries, Random number testing, Web-application

II INTRODUCTION

A Importance of RNGs

Random Number Generators (RNG) are relied on for almost all modern-day Computer Systems. All cryptographic algorithms rely on a random number source to achieve their full potential, it is a standard assumption prior to any proofs of security. Thus, modern-day cryptography is rendered useless without a random source of numbers; this often makes the RNG the best attack point in secure systems. For example, security vulnerabilities were introduced into millions of systems worldwide through flawed OpenSSL keys, as Debian OpenSSL had a poor pool of entropy for randomness (Strenzke, 2016). Furthermore, Wertheimer (2015) reported that the NSA allegedly introduced a flaw in the Dual_ECRB RNG to compromise secure systems undetected; however, the flaw was found due to a bias revealed through statistical testing. Additionally, random numbers are important to accurately model real-world physical systems; these rely on high-quality RNGs used by Monte-Carlo methods to produce meaningful results, they are particularly key to large universe simulations and turbulence models (Gentle, 2006).

Hardware entropy sources, such as nuclear decay and oscillator jitter (Sunar et al., 2006), can be used to create a perfect True-RNG (TRNG). However, TRNGs are extremely slow, one of the most famous (Haahr, 2010) utilises atmospheric noise and has an output of just 10^3 $bits/second$; whereas Pseudo-RNGs (PRNGs) are only limited by hardware and have been shown to produce

over 10^9 $bits/second$ (Saito and Matsumoto, 2008). Due to the speed increase and the difficulty in accessing a TRNG, almost all cryptography is based on PRNGs, which utilise and stretch the device's pool of entropy to create a random-looking output stream. Albeit, the resulting output stream is entirely deterministic and if the small starting seed were revealed through patterns in the output, the whole cryptographic system relying on the RNG is compromised. For that purpose, a sub-class of PRNGs called Cryptographically-Secure-PRNGs (CSPRNGs) have been developed, using cryptographic principles to ensure finding the starting seed is as hard as breaking the encryption itself. For instance, Blum et al. (1986) developed a CSPRNG based on the Prime-Integer-Factorisation problem (the trapdoor function used by RSA).

B Statistical Testing of RNGs

Due to the ubiquity of RNGs, it is vital that we are able to test the quality of their outputs. If the output of the PRNG is biased in any way, it reveals that a TRNG is not being used and may reveal to an attacker the exact type of PRNG, leading them closer to finding the initial seed. Thus, the ideal PRNG is indistinguishable from a TRNG and statistical tests can indicate this.

These tests attempt to ascertain the randomness of an output stream through a variety of methods and measures of entropy. The issue is, if a PRNG was truly random it could output a million 1s in a row, as if this wasn't an option the generator wouldn't be random. This is why no one test can fully measure randomness, instead, a variety of different methods must be deployed.

To further explain, the majority of tests result in a *p-value*, Bassham et al. (2010) explain this in great detail. In brief, this $p \in [0, 1]$, can be interpreted as the chance a TRNG would produce a less random stream; so *1.0* is perfectly random and *0.0* is perfectly non-random. Therefore, an RNG that produces only perfectly random numbers would actually be flawed; the ideal RNG would result in the uniform distribution: $X \sim U(0, 1)$, as we want all outputs to be equally likely. Therein lies the problem, it means little to state a stream results in X for its *p-value*, it is vastly more interesting to analyse the distribution of *p-values* taken via splitting the stream. We are seeking the most "uniform" RNG, not a continuously perfect one.

Consequently, this leads to the mammoth requirement for data in statistical testing. NIST's testsuite (Bassham et al., 2010) recommends 10^7 bits before any significant meaning can be drawn from the results, and Dieharder (Brown et al., 2013) needs over 10^{11} bits. Without this massive amount of data, it is hard to certify the uniformity of the generator output.

C SALT's Purpose

This project's main purpose is to develop a testsuite that does not require this humongous data collection. Existing test batteries almost exclusively focus on testing a known RNG with access to an unlimited amount of output, which is why they are rather liberal with the amount of data needed. However, there are several situations where it is important to test the current output quality of an RNG. For example, a user creating a new Bitcoin wallet needs to ensure the default RNG on their laptop is producing a sufficiently high-quality output; this may not be down to the generator alone, but also relies on the laptop's existing entropy pool. In this case, it is important a user can quickly test the RNG is working optimally, but with limited output and no knowledge of the algorithm used. This paper's proposed battery of tests could be used in such a situation. SALT also seeks to explore the minimum amount of data we need to begin to differentiate RNG output; this opens up a whole new area of research in the field, as existing literature only envisages the scenario of unlimited generator output and a largely unlimited amount of time to achieve the most

accurate results possible. This new battery, aimed at a data and time constrained environment, shows the surprising amount of information that can be extracted from much smaller samples. This paper seeks to address the following research questions:

> *"Can a lightweight Statistical Testsuite for RNGs with limited output be constructed? If so, how useful is it and to what degree can it differentiate RNG outputs?"*

D SALT's Aims and Achievements

This project advances the existing literature by demonstrating the following achievements:

- **Defining a Scoring Methodology -** For this, fourteen generators were implemented and ranked according to intuition about the data (i.e. it is from a TRNG) and analysis with the full NIST testsuite. NIST's was chosen as it is widely used and trusted by the community. After ranking the fourteen generators they were split into clearly defined groups and a scoring function was implemented assessing the test's ability to stratify the generators and differentiate within groups. Further, a basic scoring function was also crafted based on the widest groupings to determine the test's more basic abilities.
- **Identifying Appropriate Tests -** This paper's next endeavour was to analyse existing statistical tests, to determine which would be appropriate for SALT. Over sixty tests were found and considered from the literature, nineteen promising tests were then analysed further to assess their potential. Finally, fourteen of these tests were adapted and incorporated into one test battery, creating SALT, which is based on the NIST suite.
- **Comparing Test Batteries -** This showed that SALT outperformed all except one of the five tested batteries up to 10^7 bits; beyond which it was difficult to create enough data for experimentation. Moreover, it showed that from 250 bits the fourteen generators could be separated at a basic two-group level.
- **Creating a Simple Interface -** Finally, Python bindings were added to the C code, and a corresponding website was built allowing users to quickly and easily utilise SALT. This could easily be converted into a full-scale app or web-app which would allow users to get a basic understanding of their local generator, in a few seconds on a standard laptop.

III RELATED WORK

Existing literature in the field of RNGs is varied, for the purposes of this project we can broadly split them into three groups:

A RNG Analysis

Firstly, papers introducing PRNG's provide an analysis of the randomness outputted by the generator. This proved useful to construct SALT, as papers concerning the fourteen generators used by the scoring function helped inform the ground truth ranking. Albeit, these papers are not consistent, different papers utilise different sets of tests and employ their own test methodology. For instance, one of the first papers in the field, Rotenberg (1960), provides a statistical analysis of the PRNG using the simple metric of "runs" (the most 1s or 0s in a row), whereas Blum et al. (1986) provide only a theoretical proof of randomness for their BBS generator and Matsumoto and Nishimura (1998) use distribution tests to show the strength of the Mersenne Twister. Consequently, we can not simply directly compare generator performance; when publishing new ideas,

the authors are more likely to present theoretical concepts than to provide empirical analysis in comparison to previous generators. Papers such as Hellekalek (1998) and Hull and Dobell (1962) go on to investigate methods of designing PRNGs and present several statistical tests which can be used, which proved useful for gathering possible tests, but they refrain from demonstrating their use through any experimentation.

B Statistical Tests

Secondly, the next interesting grouping of papers are Statistical Tests. These papers, such as L'Ecuyer (1992) and Marsaglia and Zaman (1993) provide an in-depth exploration of the thinking behind each statistical test. This has led to the compilation of over sixty statistical tests for randomness to be considered for SALT. Analysing all of these by hand would be an impossible task, so utilising these existing papers has allowed the project to focus on the tests which look most promising. Despite these tests being published, few researchers go beyond the main batteries of tests when comparing RNGs. By analysing these papers, tests which have been encompassed by newer tests in the batteries have been revealed, allowing for simpler tests to be considered for the lightweight testsuite, which is vital to the performance of the suite. Tsang and Marsaglia (2002) and Rukhin (2000*b*) go further by providing in-depth mathematical backgrounds to why certain tests encompass others. For example, Rukhin (2000*b*) reveals a simpler version of the Approximate Entropy test used in the NIST testsuite, which would be more appropriate to consider for SALT.

C Existing Test Batteries

Finally, there are a group of papers which align much closer to this project's research question. They discuss the implementation and comparison of test batteries and form the main focus of literature in this area of research. As mentioned, no one test can accurately define randomness, therefore, test batteries are crafted in an attempt to capture the varying measures of randomness. The original suite was outlined by Knuth (1997), which focused on basic statistics such as frequency. Subsequently, several testsuites have been crafted, revised and critiqued.

Papers demonstrating new testsuites such as Rukhin (2000*a*) and Diehard (Marsaglia (1996)) often provide good overviews of statistical tests and they frequently give details of the minimum bit sizes needed for certain tests; Bassham et al. (2010) in particular gives this level of detail. However, no explanation is further given on why this minimum data size is at this level. Instead, this provided a starting point for the project to further investigate these threshold values researching what happens if we use smaller file sizes, and seeing what knock-on effect this has to the confidence levels produced by the tests.

Comparative papers such as Soto (1999), Larrondo et al. (2006), and Demirhan and Bitirim (2016) go on to investigate testsuites and outline the flaws and limitations of them. Often, these papers are purely investigating which testsuite is superior to the others, going into detail on the confidence level provided by each battery and the tests they contain.

Below, an overview is given to each suite studied and how it has informed the current project. Unfortunately, some of the common suites used in academia (such as Dieharder) have never been formally published, instead, their manuals have been referenced.

C.1 NIST

The most famous and widely used statistical testsuite is provided by NIST (Bassham et al., 2010). It is largely based on the work of Rukhin (2000*a*) and was originally used to compare the output of AES candidates. The paper provides recommendations on the number of bits to be used for each of the sixteen statistical tests, ranging from 10^2 to 10^6 bits, and goes on to explain that each test should be repeated one hundred times to obtain any meaningful results. Unfortunately for this project, no information is given leading to how the minimum bounds for each test were concluded. However, they provide a good starting point to identify which tests may be appropriate for SALT, and which would need heavy adaptation. Further work goes on to critique NIST's testsuite; the Lempel-Ziv test was removed from revised versions of the test due to the research of Kim et al. (2004). Kim explains that the test settings given in the original testsuite were incorrectly calculated, meaning testing on the AES algorithm is fundamentally flawed and should be repeated. Thereby, showing that testing for randomness is extremely hard and that many of the most prestigious institutes in the world struggle to accurately test for it. Zhu et al. (2016) and Georgescu et al. (2017) further explore NIST's tests and provide areas for improvement, particularly on the correlation between tests; however, these are beyond the scope of the project as they fix discrepancies that will only become relevant for large data samples.

C.2 Dieharder

Dieharder is another commonly used testsuite (Brown et al., 2013), it is based on the initial Diehard suite (Marsaglia, 1996) and attempts to prove beyond doubt if a generator source accurately mimics a TRNG. Consequently, the testsuite is phenomenally over-engineered and requires around 200Gbs of data to run in its entirety. However, within this suite are tests scarcely found elsewhere, such as the Parking Lot and Permutations tests; these have interesting concepts behind them and claim to cover areas of randomness not explored by NIST. Several of these tests were heavily adapted to work in a more lightweight testsuite and analysed for inclusion in SALT.

C.3 TestU01

TestU01 (L'Ecuyer and Simard, 2007) is similar to Dieharder but provides even more detailed analysis. The suite contains fourteen tests and several generators, unlike the others - three versions of their test battery are provided: smallcrush, crush, and bigcrush. However, smallcrush itself still needs over 20Mb of data, so is unsuitable as a ready-made SALT. The tests are once again aimed at detailed statistical analysis, this time by observing the data over different fields. For that reason, it is rather more complicated and no tests from it have been considered for SALT that aren't in other batteries. Nevertheless, smallcrush is used as a comparison point for SALT.

C.4 ENT

ENT (Walker, 1998) is an intriguing testsuite, it predates the batteries above, and as such is not focused on second-level p-value analysis, but is orientated towards providing float outputs for five different metrics. This results in a testsuite aligned with the goal of SALT, albeit, no final value is given, the user is left to further analyse the five floats returned. These tests were easily incorporated into the testing stage of SALT and were considered to complement the more rigorous p-value analysis of earlier tests, particularly for smaller bit sizes.

C.5 Crypt-XS

Crypt-XS (Gustafson et al., 1994) is the final testsuite considered. As with ENT, it was published before NIST's and has been largely forgotten by the current literature, since it has been superseded. Nevertheless, the suite proved useful for developing SALT, as the proposed tests are simpler and proved promising for working on a smaller dataset. For instance, the binary derivative test involves further analysis of a small stream by taking the modulo-two addition of the stream and repeating the standard frequency test; SALT utilised this to develop binary derivative versions of other tests. Unfortunately, there is no publicly available version of Crypt-XS so it could not be included in comparisons; however, interesting tests were implemented from scratch.

C.6 Current Limitations

Unfortunately, all the testsuites proposed in the literature have one oversight for the goal of SALT: there is no final value given to the stream's randomness. Instead, each test either returns multiple floats over differing ranges or provides an extremely discrete number of tests passed or failed. For SALT this is inadequate, particularly with Dieharder there is no differentiation available between a poor RNG and a mediocre one, instead, the majority of RNGs will fail all tests, with only the ones closest to mimicking TRNGs passing any at all. As previously mentioned, the size requirements for most of these suites are colossal, most requiring between 10Mb and 200Gb for what they deem statistically relevant results. Nevertheless, they all provide insight into the field and give a good starting point for developing a lightweight testsuite. NIST's suite was extended and used as the basic framework for SALT, and many of the tests from other batteries were repurposed into SALT after some modification, mostly in confidence values. The most challenging part of SALT was summing the results of the final fourteen tests up into one value of randomness; however, after carefully reading of the aforementioned papers, and studying the expected values, a function for randomness was developed.

IV METHODOLOGY

Sections IV through VII detail the solution, results and evaluation stages of constructing SALT. They are given in a chronological order to better understand the process leading to each decision. This section outlines the process used to compare the success of a test or test battery.

A Generators

Initially, it was important to collect a series of generators that could be used to provide the basis for future comparisons. The original NIST suite was used as the basis for all future experiments and SALT, this was because Bassham et al. (2010) provide an accessible guide for making additions to the testsuite. It is also the most commonly used battery by researchers, so there is a lot of support available, thus, making the final suite produced easy to understand and use. NIST includes nine RNGs in their testsuite, including two CSPRNGs. For this project, that was insufficient as several of the generators (i.e. *quadRes1* and *quadRes2*) were extremely similar to each other and we wouldn't expect to find a difference between them. Further, the default suite doesn't provide sufficient TRNG data to draw any meaningful comparisons with the PRNGs. To fix this, several generators and data sources were added to the suite.

Additional RNGs added included several purposefully poor RNGs, such as: *Alternating Bits*, *Square Numbers* and the *Fibonacci Sequence*. These were created to determine if SALT could

identify patterns, this was particularly difficult for the Fibonacci sequence which has no obvious pattern to it over small samples. Two further mid-tier RNGs were added. Firstly, Middle Square (Von Neumann, 1951) was implemented as it is the seminal RNG in the field; more importantly for this project, it is sufficiently random to outperform the sequences but should prove worse than all later PRNGs. Secondly, the Mersenne Twister (Matsumoto and Nishimura, 1998) was included; this is one of the most widely used PRNGs today, and is Python's default, it should outperform several PRNGs, but significantly, it performs worse than NIST's CSPRNGs. Finally, several TRNGs were sourced; since TRNGs need advanced hardware and are slow at producing output, the expansion of three irrational numbers was used, following the precedent set by literature. Specifically π, e and $\sqrt{2}$, generated by Nemiroff (1994) and Wayne (2016).

Precisely ten million bits were then collected from each generator for two reasons. Firstly, the TRNG data available for the expansion of irrationals was not easily sourced above ten million bits, and it was out of scope for this project to compute these numbers any further. Secondly, NIST's implementation of the CSPRNG developed by Blum et al. (1986) takes on average $73.53\ seconds$ to produce 10Kb of data. Meaning over 200 hours would be needed to produce the next level of data (10^8); again it was beyond the scope of SALT to spend time creating this quantity of data. Lastly, this project endeavours to see the level of information that can be gleaned from a relatively small number of bits, surpassing even a few thousand bits is beyond SALT's purpose and is only of interest as a comparison to existing test batteries.

Below, *Table 1* provides a summary of the generators used.

Table 1: Generators used by the Scoring Strategy to compare testsuites

Generator (s)	**Category**	**Included in NIST**
XORShift, G-SHA1	PRNGs	Yes
Linear and Quadratic Congruential Generators	Better PRNGs	Yes
Micali Schnorr, Blum Blum Shub	CSPRNGs	Yes
Alternating Bits, Square Numbers, Fibonacci	Sequences	No
Middle Square, Mersenne Twister	PRNGs	No
π, e, $\sqrt{2}$	TRNGs	No

B Scoring Strategy

A simple scoring strategy was vital to ensure a fair comparison amongst testsuites. Existing literature has done little to cover this area of research, instead focussing purely on the number of tests passed as a comparison. Since SALT outputs a float, a scoring strategy needs to be devised based on the rankings given to the generators. For tests such as Dieharder, this was particularly important, as they often result in no passed tests for weak RNGs and this lack of discrimination needed to be reflected in the score. Due to careful selection of the generators, as described above, it was relatively easy to rank the generators in their expected order of randomness (TRNGs at the top, sequences at the bottom, etc.). To confirm this ordering the full NIST testsuite was used to assess the entire ten million bit output stream for each data source, ensuring that the ranking was correct. Although this makes the incorrect assumption that the NIST suite itself is a perfect discriminator, this assumption is valid for the purposes of this project as we are looking for a comparison to the current best-known testsuite and NIST's is the leading, most prolific suite.

Algorithm 1 gives the scoring process utilised throughout this project and includes comments on the rankings discovered by the initial analysis, it returns an integer score out of 29, each point reflecting a correct part of the rankings identified by the testsuite. There is intentional bias in the function on testsuites which differentiate between the weaker RNGs more effectively, this is to ensure that a suite scoring highly would be suitable for the SALT criteria.

Algorithm 1 Advanced Scoring Function

Require: $rank \leftarrow$A dictionary s.t. $rank[ALG] = i \in [0, 1, ..., n]$ where 0 denotes the worst-ranked and n the best

There should be two extremely distinct groups

1: $1\ point$ for each of [*ALTBITS, G-SHA1, SQRNUM, XOR, MIDSQR, MTWIST, FIB*] ranking in the bottom half

2: $1\ point$ for each of [*LCG, QCG, MSG, BBS, PI, E, SQRT2*] ranking in the top half

These two should be the worst

3: $1\ point$ for $rank[ALTBITS] = 0$

4: $1\ point$ for each of [*ALTBITS, XOR*] ranking in the bottom two

A second grouping emerges just above the worst

5: $1\ point$ for each of [*G-SHA1, SQRNUM, MIDSQR*] ranking above the worst two, but below the rest

Interestingly, NIST scored the FIB generator rather well

6: $1\ point$ for each of [*MTWIST, FIB*] ranking above the worst five, but below the rest

Ensuring the TRNGs are ranked highest

7: $1\ point$ for each of [*PI, E, SQRT2*] ranking in the top three

8: $1\ point$ for any of [*PI, E, SQRT2*] ranking the highest

Differentiate the congruential generators from CSPRNGs and TRNGs

9: $1\ point$ for any of [*LCG, QCG*] ranking below [*PI, E, SQRT2*]

10: $1\ point$ for any of [*LCG, QCG*] ranking below [*MSG, BBS*]

Differentiate the CSPRNGs from the TRNGs

11: $1\ point$ for any of [*MSG, BBS*] ranking below [*PI, E, SQRT2*]

Algorithm 1 results in an extremely accurate measure for capturing the rankings of the generators. However, SALT is mainly interested in capturing the general trends using the minimal number of bits possible. Consequently, *Algorithm 2* describes a basic scoring function (out of 13) to determine how accurately the testsuite has captured basic groupings. In this function, the Mersenne Twist and Fibonacci generators are ignored, as NIST struggled to separate them from the stronger RNGs over smaller input sizes. Moreover, we only look for two groupings and a clear weakest RNG; demonstrating that the testsuite can discriminate RNGs at a basic level.

Algorithm 2 Basic Scoring Function

Require: $rank \leftarrow$A dictionary s.t. $rank[ALG] = i \in [0, 1, ..., n]$ where 0 denotes the worst-ranked and n the best

1: $1\ point$ for each of [*ALTBITS, G-SHA1, SQRNUM, XOR, MIDSQR*] ranking in the bottom half

2: $1\ point$ for each of [*LCG, QCG, MSG, BBS, PI, E, SQRT2*] ranking in the top half

3: $1\ point$ for $rank[ALTBITS] = 0$

V STATISTICAL TEST SELECTION

Selecting the statistical tests to include in SALT proved the most challenging aspect of this project. Detailed in this section is the process of whittling down the initial sixty tests to nineteen for further experimentation and finally, why fourteen tests were selected for SALT.

A Test Selection

As mentioned in *Section III.B*, consulting the literature resulted in over sixty statistical tests to be considered. It was infeasible for this project to implement them all, so further reading and analysis of how often each test was used was carried out. For instance, the standard *Frequency* test was used by almost all test batteries surveyed, demonstrating it was a staple test that should be considered for SALT. Furthermore, several tests superseded others and many were identical tests with different names. Through these methods, it was possible to narrow down the search to thirty-two commonly used tests that did not encompass any other tests themselves. Several of these tests were stated by Bassham et al. (2010) to be appropriate for the scale of testing SALT would compute, many more tests were described as needing a far greater quantity of bits than SALT was envisaged for. Other papers stated similar claims, which led to a rough grouping of fourteen appropriate tests and eighteen inappropriate ones.

A large proportion of these inappropriate tests had no chance of being adapted to work in a data constrained environment; for example, Diehard's (Marsaglia, 1996) *rank_32x32* test requires constructing 40,000 matrices each with 33Kb's worth of data. However, a minority of tests had intriguing concepts behind them that could possibly be adapted for smaller datasets; for that reason, three tests from Dieharder (Brown et al., 2013) were included for subsequent testing.

Additionally, the use of modulo-two addition by Gustafson et al. (1994) to increase the number of tests that could be performed on small datasets showed an exciting opportunity. Observing the code and performing some basic tests, two tests were identified that may work with this principle. So a *derivative Serial-Correlation* and a *derivative Entropy* test were added in the hope of identifying previously missed sequences; resulting in nineteen contenders.

B Test Descriptions

Below, each of the nineteen tests taken on for further consideration are described; also given, if available, is a test batteries' implementation; where multiple implementations existed the most appropriate one for SALT was selected. The number of suites takes into account the five major suites given above, as well as the less common SPRNG, Helsinki, and Knuth suites.

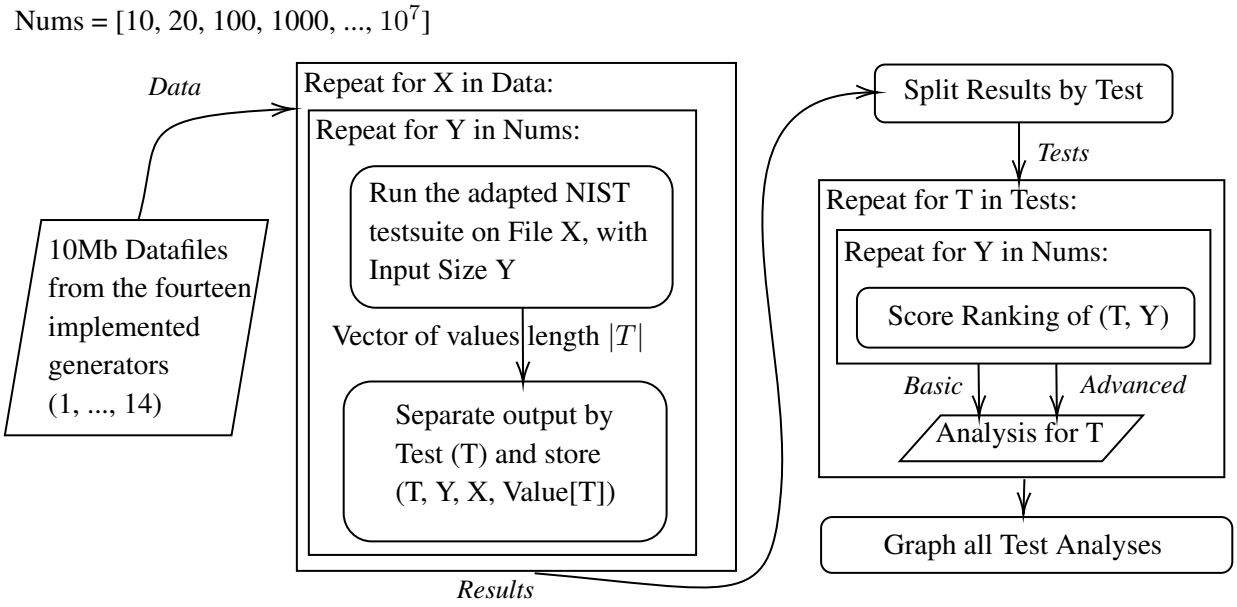
1. **Frequency (Mono-bit)** *5 Suites, NIST* - One of the simplest tests for randomness. The fraction of ones and zeroes is compared to the expected value of half each.
2. **Block Frequency** *4 Suites, NIST* - Similar to the standard Frequency test, except the test is run on multiple smaller blocks to check for clumping.
3. **Cumulative Sums** *1 Suite, NIST* - Partial sums of the sequence are taken, and the maximal excursion from zero is compared to the expected distribution for the stream length.
4. **Reverse Cumulative Sums** *1 Suite, NIST* - Same as Cumulative Sums, but in reverse.
5. **Wald–Wolfowitz Runs** *5 Suites, NIST* - Seeks to find the maximal increasing or decreasing sub-sequence of the stream, using a predetermined number of bits to define each float. NIST recommends at least 10^6 bits for this test; however other papers suggest it can be useful for smaller streams, so it has been included.

6. **Discrete Fourier Transform** *2 Suites, NIST* - A Fast Fourier Transform is applied to the stream, then the distance between peaks is measured to determine periodic features.
7. **Approximate Entropy** *1 Suite, NIST* - Compares the frequency of overlapping sub-blocks between two consecutive lengths of bits. NIST recommends this would be useful for small bit sizes, but the Serial Correlation test is similar and could prove better.
8. **Parking Lot** *1 Suite, Dieharder* - Assume you have a 100x100 grid, drop cars on it to park them at locations determined by the bitstream. The number of attempts compared to the number of crashes is expected to follow a normal distribution. The original version of this test uses a much larger parking lot so it will have to be adapted.
9. **Permutations** *2 Suites, Dieharder* - Model the bitstream as a series of integers, with each group of five integers representing a state. Transition around the state machine using the bitstream and check that the distribution of states follows the expected covariance matrix. To use with a smaller dataset we have used 6 rather than 32 bits to represent an integer.
10. **Birthday Spacings** *3 Suites, Dieharder* - Randomly drawing birthdays from integers encoded in the bitstream, the interval between them should follow a Poisson distribution. Again, the sample size and drawing needed to be adapted for use with smaller bitstreams.
11. **Serial Correlation** *3 Suites, ENT* - Measures how much the current byte is affected by the previous byte. Zero correlation would imply complete randomness, and full correlation would imply a sequence over every byte.
12. **Entropy** *1 Suite, ENT* - Tests the amount of compression that can be applied to the stream, but looks locally rather than across the whole stream to improve time efficiency.
13. **Monte-Carlo π Approximation** *1 Suite, ENT* - A square of length X is defined, bits are read in as coordinates within the square and are considered a hit if they fall within the circle of radius $\frac{x}{2}$ centred within the square. The proportion of hits compared to misses is used to approximate π, with the relative error returned. This test was altered to use a smaller square and fewer bits for each set of coordinates.
14. **Chi-Square** *1 Suite, ENT* - A simple Chi-Square test is applied to the entire data stream. Many other testsuites use this indirectly by applying the test to a previous test's distribution.
15. **Binary Derivative** *1 Suite, Crypt-XS* - A new stream of data is created by taking the modulo-two addition of the bitstring; this new stream then has a standard mono-bit frequency test applied. Thus, any patterns spread over alternating bits should be identified.
16. **2^{nd} Binary Derivative** *1 Suite, Crypt-XS* - Same as Binary Derivative, but modulo-two addition is applied again to create the next derivative; checking for a recurring pattern.
17. **Gap Test** *3 Suites, SPRNG* - Similar to the Birthday Spacings test, we are assessing the gaps between certain points of the bitstream. For this test, a gap is defined $[a, b] \in [0, 1]$ e.g. $[0.4, 0.6]$; the bitstream is then read in as float, x, and the gap between appearances of $x \in [a, b]$ is recorded. The distribution of the number of occurrences for each gap length is then compared to the expected Poisson distribution.
18. **Derivative Serial.Corr.** *0 Suites, None* - This is the Binary Derivative version of the Serial Correlation test, hopefully identifying lagged correlation
19. **Derivative Entropy** *0 Suites, None* - This is the Binary Derivative version of the Entropy test, searching for patterns that are interspersed throughout the bitstream.

C System Design

Following the selection of the above nineteen tests, this project needed to create a rigorous framework to ascertain the usefulness of each test. *Figure 1* (below) shows the process from start to finish for how the Scoring Strategy (explained in *Section IV.B*) was applied to each test.

Figure 1: Workflow for analysing Statistical Tests



D Testing and Issues

As shown in the System Design, before any testing could commence all tests had to be incorporated into the NIST testsuite. Otherwise, there would be comparison issues since the input and output of each test would not be consistent. In particular, Dieharder expected raw binary files as input, rather than the ASCII files expected by NIST. By using the same test framework we ensure a fair and valid test.

For the seven tests already included within NIST, nothing had to be done; however, the other twelve proved tricky to incorporate. Moreover, those already contained within testsuites used a variety of versions of C and C++, and had dozens of overhead auxiliary files managing the process. Standardising this into NIST's suite was extremely time-consuming; however, eventually, the eight tests worked within the NIST framework. Implementing the remaining four tests was relatively straightforward following the algorithm outlined by Gustafson et al. (1994).

Next, data files had to be produced from the generators previously incorporated into NIST. Without doing this, each new run of the testing harness would have produced different results, as not all tests have hardcoded seeds. Furthermore, several generators had a cold-start problem; therefore, eleven million bits were produced by each and the first million were discarded.

Finally, the settings for each test needed to be decided on. Most of these were changed when adapting the tests for smaller bit sizes; for example, the size of the Parking Lot. However, gathering the results for extremely minimal input would require not using the standard one hundred streams. Consequently, to get the necessary data-points for the lower end of the scale, tests were repeated using a smaller number of streams.

E Statistical Test Results

Given below are the results of the analysis of the nineteen selected statistical tests; both figures use logarithmic scales in the x-axis, as SALT focuses on showing trends with a minimal number of bits and it takes an exponential quantity of bits to reveal more information. To better visualise the data, *Figure 2* shows the five tests that were not selected and *Figure 3* shows the results of the final fourteen tests that were selected for SALT. Both figures include a comparison to the complete SALT suite as a reference point.

Regrettably, the curves produced are not smooth due to the nature of randomness, instead, only general trends can be seen. Returning to the initial criteria, the best RNG is one that produces values that uniformly fall over the entire spectrum of randomness. Specifically, any given subsection of the stream should produce a randomness value evenly drawn from $[0, 1]$. Necessarily, as we increase the amount of data tested, the ranking results of the generators will change as the data they output changes ranking. It is only with a colossal amount of data that this stabilises. Nevertheless, SALT is aimed at finding clear differences between RNGs for small amounts of data, so all fourteen tests selected show a general upwards trend, with the occasional anomaly.

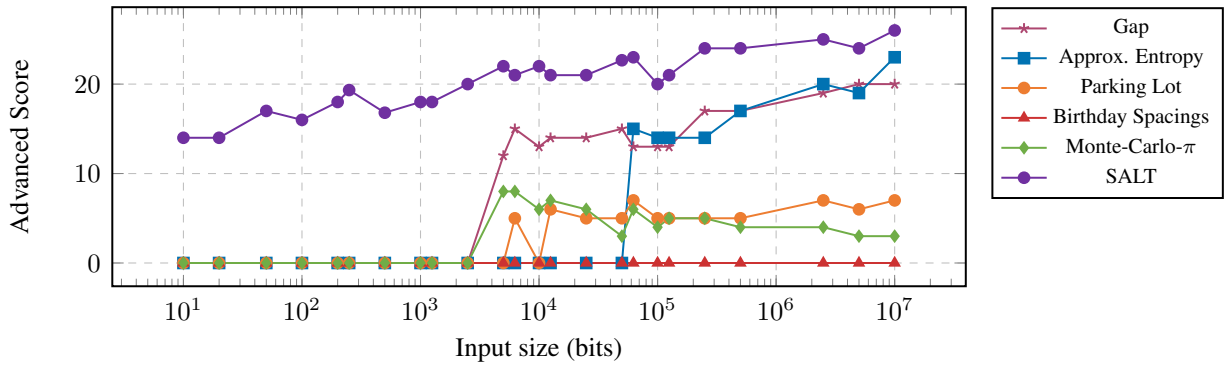


Figure 2: Results of the analysis of failed candidates for SALT

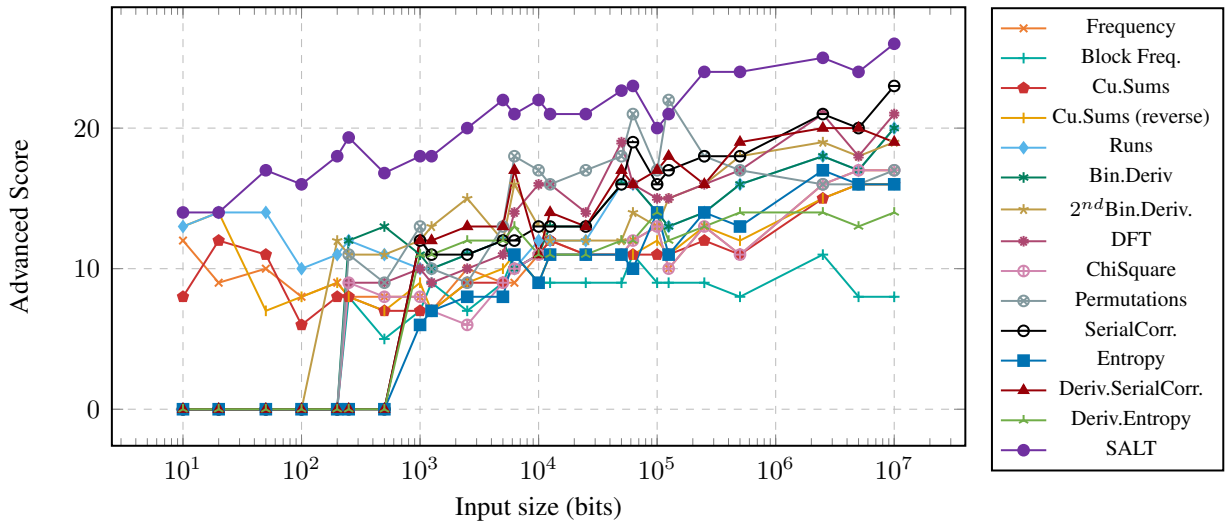


Figure 3: Results of the analysis of successful candidates for SALT

To begin with, *Figure 2* demonstrates why five tests were dropped from inclusion in SALT. Two of the tests from Dieharder did not translate well to such a small quantity of data, with

Birthday Spacings never managing to identify a correct trend, and Parking Lot failing to identify more than seven. Monte-Carlo π-Approx. suffered the same fate. Surprisingly, the Gap and Approximate Entropy tests did eventually start discriminating the generators extremely well. Nevertheless, they needed 5,000 and 62,500 bits, respectively, before any success was seen; this amount of data doesn't fit with the purpose of SALT and regardless, the tests don't seem to provide any additional discrimination beyond the Serial Correlation test, which was included.

Further investigating the trends from *Figure 3*, it is clear each test shows a positive correlation. Importantly for SALT, four of these tests show immediate discrimination abilities from just 10 bits; by 200 bits ten tests begin to work, and from 500 bits all fourteen tests work. Ideally, the entire suite would work from the minimum 10 bits. However, a testsuite of four tests would fail to adequately capture all measures of randomness. Using all fourteen tests doesn't hinder us at the start, but vastly improves the ability of SALT as each test begins to work. Moreover, 500 bits is still an incredibly minuscule number of bits for a testsuite; NIST recommends even the simplest tests will need at least 10^5 bits before they derive statistically relevant results. This experimentation shows that is not the case, upwards trends can be identified from 10^2 bits, a significant finding. Further, the graph reveals that the best discriminating tests include Permutations, Binary Derivative, Serial Correlation and Discrete Fourier Transform. Particularly poignant, is that Permutations was successfully adapted to work in a data-constrained environment, a major success during the development of SALT. Contrastingly, the block-frequency test performed extremely weakly by itself; initially, this was considered a failure, but when incorporated into SALT it did mildly improve SALT's rankings. This possibly shows that it detects a mode of randomness that is not a reliable discriminator in itself, but that can help holistically improve the picture. Finally, the cluttered graph is a positive sign for SALT; it shows that the majority of tests are independent. When a few tests peak others trough and taking the tests together reveals a continuous upwards trend. For that reason, the cluttered graph shows that SALT will be partially resilient to the changing nature of RNG outputs and that SALT as a whole will average out the tests fluctuations, providing a valid testsuite for even the smallest bitstream.

VI TEST BATTERY ANALYSIS

A Constructing SALT

SALT was then constructed around the fourteen tests deemed suitable in *Section V*. As mentioned, all these tests have been incorporated into NIST's testsuite, for ease of use and familiarity for researchers. After refactoring the testsuite to provide only the fourteen tests we are interested in, we are left with a problem - how do we output a single randomness value given fourteen different results? All existing testsuites ignore this question, instead, they merely highlight the tests that have passed, compared to the tests failed for a given confidence value; thus, leaving the researcher to interpret and compare results. For SALT this is inadequate, we are constructing a simple to use testsuite for the average user on their laptop or phone, not a researcher who is already extremely familiar with the field. How to do this is an open question that, so far, the literature has not provided an answer for.

In creating such a value, several measures were considered, namely the product, average and RMSE of the fourteen results. Through brief experimentation, the product of the fourteen test results proved most useful. Intuitively, each test should be validating a different aspect of randomness within the bitstream, *Figure 3* shows this is the case as each test peaks and troughs in different areas. Consequently, if one of the fourteen tests performs poorly, the overall result

should reflect this; albeit, taking the product of fourteen values can quickly drop to zero. For ten of these tests, this was not an issue, as for a TRNG we would expect them to return a uniform distribution over $[0, 1]$ for each stream; resulting in the average of the *p-values* tending towards 0.5. To alleviate the issue of one test exerting too much influence over the final value, we simply take the test's absolute difference from 0.5 and subtract it from 1, giving a range of $[0.5, 1]$. Clearly, this is imprecise compared to advanced Chi-Square analysis of the *p-value* distribution; however, for these tiny bitsizes it is the best option as more intricate analysis is fruitless. Moreover, it is extremely rare for a weak RNG to return an unweighted distribution, so it would be difficult to achieve the 0.5 average (Hellekalek, 1998). The final four tests (based on ENT) do not expect a uniform distribution and have a bias to return higher numbers, so their product is taken and multiplied with the first product. Creating the final randomness function, given by *Eq. 1*.

Further experimentation was undertaken to investigate the effect of weighting certain values, or performing distribution analysis; these experiments showed that any tweaking to the basic formula worsened the performance of SALT. Moreover, this is expected, as SALT is intended for minimal bitstreams. Larger bitstreams could benefit from a more precise formula, but in the data-constrained environment SALT operates in, it is best to keep it simple.

$$salt\_value(\tau) = \prod_{T \in \tau[1,..,10]} (1 - |0.5 - T|) * \prod_{T' \in \tau[11,..,14]} T' \tag{1}$$

However, *Eq. 1* was imperfect for extremely small bitstreams - below 100 bits; as the expected variation in the output stream caused the average *p-value* to lie too far away from 0.5 for each test, resulting in close to 0 returned for all generators. Subsequently, a new formula was devised, *Eq. 2*. This uses a weighted average for scoring the initial ten tests and then the product for the final four, whereby the minimal result is later incorporated to further influence the score. Whilst useful for returning a result below 100 bits, this scoring methodology was not utilised in the below comparison of testsuites, instead, the first product equation was used.

$$salt\_avg(\tau) = \frac{\min\limits_{T \in \tau[1,..,10]} (1 - |0.5 - T|) + \sum\limits_{T \in \tau[1,..,10]} \frac{(1 - |0.5 - T|)}{10}}{2} * \prod_{T' \in \tau[11,..,14]} T' \tag{2}$$

In practice, it is up to the user to determine the best scoring function to compare on. The SALT Website utilised the average function, as bitstreams below 100 bits are expected. Whereas, the C-library utilises the product function by default, as it is expected more than 100 bits will be available, even in a data-constrained environment.

B Comparative Testsuites

To assess the performance of SALT, several testsuites needed to be adapted to output rankings. This was achieved through modifying each suite to save the number of passed tests to a file, rather than a several hundred line analysis. Moreover, many testsuites had three classifications: "PASSED", "WEAK" and "FAILED"; for this, two points were given for passed, one for weak and none for failed. Utilising these outputs, the fourteen generators could then be ranked and the scoring methodology applied (outlined in *Algorithm 1*).

Additionally to NIST, Dieharder, and TestU01, the comparisons included "SALT-esque" modifications to suites; namely, ENT and NIST. For these, only tests that were included in SALT

were kept and *Eq. 1* was utilised for scoring, as well as any slight modifications to the test to allow them to work over small data ranges. Importantly, this could provide a basis showing the limits of modifications to existing suites, helping show the advancements made by SALT.

C Testing and Issues

Using the three testsuites and two adapted testsuites we can adequately assess if SALT has achieved what it set out to do. Specifically, be able to provide a basic differentiation to RNGs that wasn't available before; also of interest is how well SALT achieves advanced ranking of the RNGs. Subsequently, all six testsuites were analysed using the previously outlined methodology.

Testing was carried out as depicted in *Section V.C*; although, this time the results were not split by Test and instead all generated files were run against all six testsuites, for all sizes.

Again, several issues arose. For Dieharder, it proved impossible to set the size parameter for all tests, as the test needed an extremely specific number of bits to run properly, magnitudes higher than even the largest size parameter. Consequently, Dieharder had to be adapted for this comparison by using seven of the seventeen tests available. This was unfortunate, however, the remaining ten tests could not run on 10^7 bits without completely restructuring them, so this was a valid comparison to assess the capabilities of existing testsuites in data-constrained environments. Similarly, for TestU01, three testsuites are available (smallcrush, crush and bigcrush), only smallcrush could be tested due to the data requirements of the other suites.

Other issues included the lack of a unified output from testsuites and their lack of one single output. These issues have already been addressed above; whereby, the output has been adapted and a count of each "pass" has been taken. Finally, the test range proved problematic. As discussed, the NIST implemented generators are unreasonably slow; further, TRNG data is extremely hard to source. This resulted in a maximum of 10^7 bits being available for each generator. Whilst this was certainly sufficient for SALT, and above the minimum required for NIST, it fell magnitudes short of the other suite's requirements. Resulting in incomplete curves for the final graph since the current testsuites do not have a chance to reach their full potential. However, this was not overly problematic, the main goal of the comparison was to see SALT's performance compared to other testsuites in a data-constrained environment; it was unnecessary to further show a comparison of SALT in an unlimited data scenario. Furthermore, 10^7 bits is a rather large amount of data and already more than surpasses the range SALT is intended for.

D Test Battery Results

Presented below is the final analysis of SALT in comparison to other batteries. Again, it is important to note that smooth curves are not observed due to the nature of RNG output. Further, the graphs are presented using logarithmic scales as the focus of SALT is in the sub-10^4 bit range, where analysis will focus. Data-points up to 10^7 are provided to inform the reader where other testsuites begin to work and to show the gains made by SALT compared to the existing literature. Moreover, not every testsuite has a full set of data-points, this was due to quirks of each suite and not lack of testing. For example, many of TestU01's tests utilise an exact power of two.

Figure 4, below, shows the results of analysis using the basic scoring function (*Algorithm 2*). This scoring method is out of 13 and assesses the testsuite's ability to separate the weak PRNGs from the stronger ones, CSPRNGs and TRNGs. Crucially, the purpose of this project is to design a testsuite that can show when the entropy reduces so far that a strong PRNG no longer outputs sufficiently random numbers; thus, this basic differentiation is vital for meeting that goal.

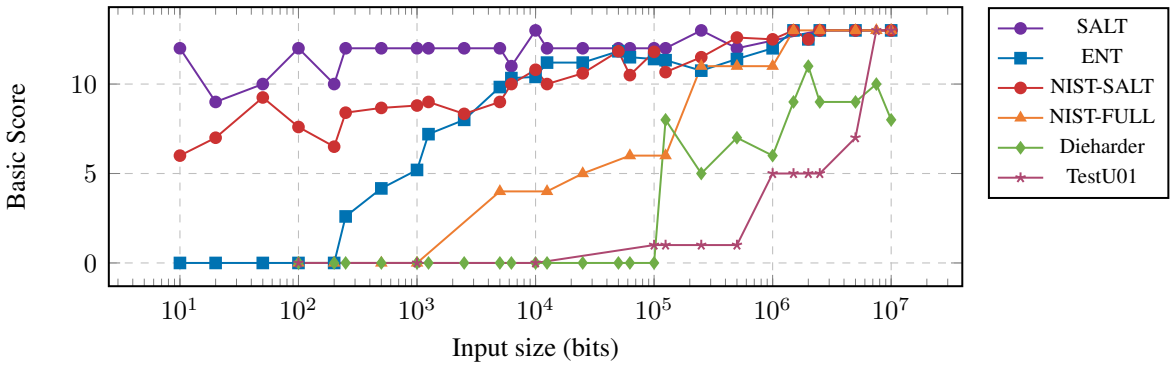


Figure 4: Basic Scoring results for Testsuite Analysis

Graphically, it is clear that SALT performs more than adequately for this task. *Figure 4* shows that after initial uncertainty, SALT can correctly classify all but one PRNG from 250 bits; however, it takes over 10^6 bits to consistently return a full score. Nevertheless, this is a remarkable achievement for SALT and shows that it works as desired. Moreover, no ready-made testsuites began to show any form of discrimination until 10^5 bits, over three orders higher than SALT. The two suites partially adapted for smaller datasets did outperform the other three; however, they did not match SALT and needed at least 10^4 bits to adequately work. By 10^6 bits, all but one testsuite successfully discriminates the fourteen RNGs, with Dieharder still struggling to meet this basic goal. Thus, we have constructed a testsuite which consistently works from just 250 bits, compared to the one and a half million bits previous testsuites would have needed.

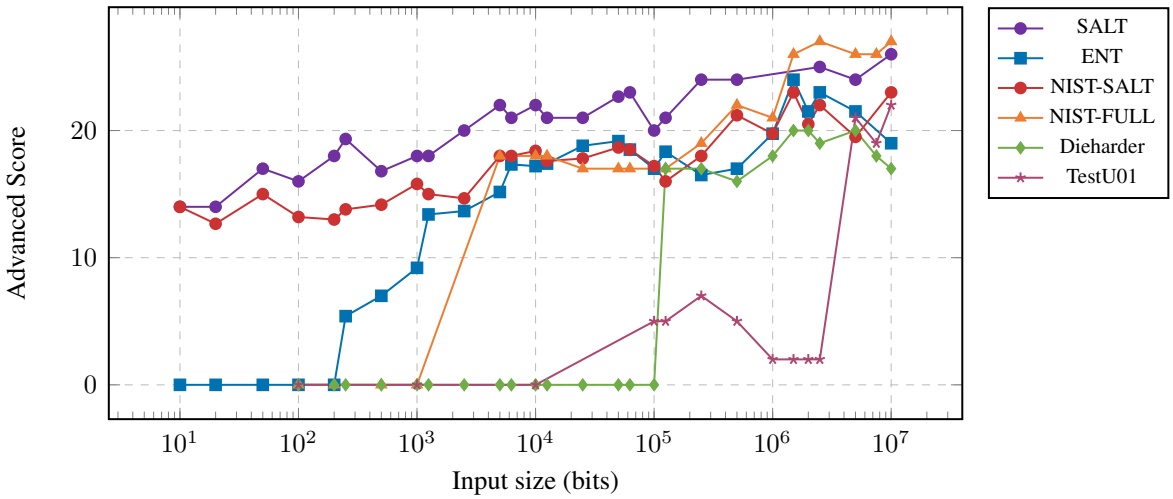


Figure 5: Advanced Scoring results for Testsuite Analysis

Continuing the analysis, *Figure 5* shows the result of applying the advanced scoring function (*Algorithm 1*). This was not necessarily the focus of SALT, as it assesses a testsuite's ability to determine small differences between generators. Specifically, it is looking at a suite's ability to separate sequences from PRNGs, PRNGs from CSPRNGs and the best CSPRNGs from TRNGs, as well as several minor differences within each group. As NIST was used to inform the ground truth rankings for this scoring function, it is no surprise that above 10^6 bits it outperforms the other testsuites. Nevertheless, SALT outperforms the rest and surpasses NIST up to this threshold. To dive deeper into the trends seen, from 200 bits SALT performs the same level of discrimination shown by NIST at 5,000, Dieharder by 250,000, and TestU01 by 5,000,000. Moreover, SALT outperforms the simplified versions of NIST and ENT for all data-points.

Finally, *Figure 6* displays the timings for the different testsuites; these were taken using an Intel(R) Xeon(R) CPU E7-8860 v3 @ 2.20GHz. The figure shows that SALT takes significantly less time than the other testsuites, but that as the bitlength increases it gradually approaches the same time as TestU01. However, it is still significantly faster than NIST, the only testsuite that surpassed SALT in the advanced scoring test. Although not included for testing, the full seventeen test version of Dieharder took over three hours to run on 10^7 bits. Further, the two simplified testsuites are not included as they simply follow the same growth as SALT.

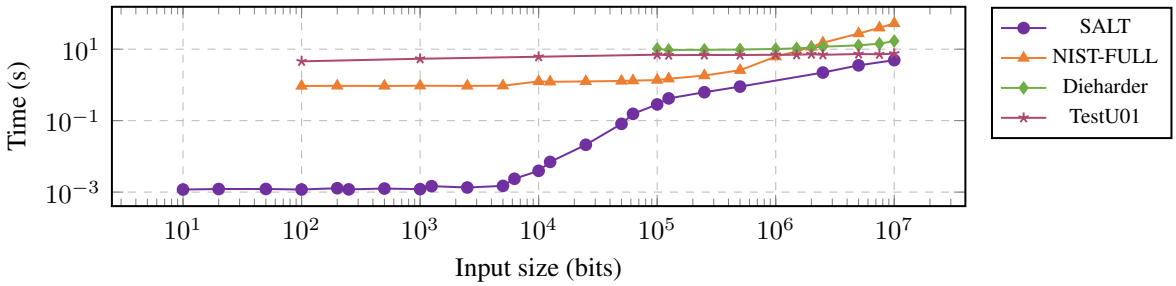


Figure 6: Timing results for Testsuites Analysis

These results shows that SALT has fulfilled all of its original goals. SALT can complete basic differentiation between tests from 250 bits, and for up to a million bits it outperforms all other assessed testsuites in the advanced ranking of generators. It completes all this far quicker than existing testsuites, showing that SALT embodies the principle of a simple, fast and lightweight testsuite. Moreover, we have shown it is possible to adapt tests for data-constrained environments and that the existing lower bounds for statistically relevant tests stated in the literature can be dramatically improved upon.

VII SALT

A C Library

This project's main goal was to construct a new Simple and Lightweight Testsuite (SALT). This was created and fine-tuned following the testing stages outlined above. The final code is structured identically to NIST's testsuite (Bassham et al., 2010). Unlike NIST, this testsuite is specifically designed for smaller input sizes and can run on just ten bits, without segfaulting (which was a particular issue with NIST's existing implementation).

SALT builds upon the NIST framework in various ways, the original sixteen tests have been reduced to a subset of six. Additionally, one test from Dieharder (Brown et al., 2013), three from ENT (Walker, 1998) and four new twists on existing tests have been implemented; resulting in fourteen tests. Further, a few of these tests have been heavily adapted to notice trends over much smaller streams than originally intended; for instance, the Dieharder Permutations test now uses 6 rather than 32 bits to represent each number. Furthermore, the NIST suite has been embellished with four new generators and three new sets of TRNG data, as originally NIST had few weak generators to assess. Thus, enabling us to better reveal the differences between testsuites.

More importantly, SALT outputs two floats describing the randomness of the generator stream. As previously described, existing tests merely list "passed" or "failed" for each test and leave any further interpretation up to the user. Since SALT is designed to be simple to use we perform basic analysis on the *p-values* generated, to produce a product value and a weighted average value. The product function multiplies the test results absolute difference from 0.5, thus resulting in a higher number for the "most average" RNGs; the product was intentionally picked as each test

analyses a different aspect of randomness, so if one test output is poor the overall value should reflect this. Yet, the average value is also returned; as for extremely small inputs (10 - 250 bits), the product returns values approaching zero, meaning the average value is a better description. Above 250 bits the product function becomes a far superior discriminator between RNGs, so it is left for the user to compare based on their input requirements.

Finally, SALT brings a new output stream option to the testuite - the CLI. By allowing users to enter bits directly on the command line, without the hassle of saving raw binary data, SALT is making a step in the right direction to quick and simple testing. Subsequently, the finished product is a C library that any users who are familiar with NIST's testsuite will be able to utilise with ease. This was intentional as several existing test batteries have a steep learning curve to install and use. By making SALT an evolution of NIST's suite, this testsuite should be more accessible to other researchers in the field, and in turn, be easier for others to build on.

B Python bindings

Problematically, a C library which has to be compiled by each user may be the best scenario for the research community, but it is by no means simple for the wider public. For that reason, Python bindings have been added to SALT to allow for easier manipulation of the testsuite. A boolean vector of tests and a binary string are passed via the bindings to the testsuite and the two salt values are returned. These Python bindings are then themselves hidden behind a file which can be imported and used elsewhere. Whilst SALT is not yet a free-standing library on Pip, these initial steps lay the groundwork, and make the suite more accessible to other programmers.

C Website

The final piece of SALT is a website, run via Flask and the Python bindings. Allowing users to graphically select which tests to run, and input a bitstring; the average salt value is then returned to the interface. This is the capstone to the SALT project and if the Flask web-server were hosted, it would allow anyone to use the website to quickly determine the randomness of their local RNG; meaning this project has met its original goal. The average result is returned, rather than the product, as this proved better for the expected 10 - 250 bit range. Results are returned in milliseconds via the Python bindings, compared to the minutes and often hours of existing testsuites, demonstrating SALT is indeed a quick, simple and lightweight testsuite.

VIII CONCLUSION

This project has delved into a gap in the literature, resulting in the construction of SALT - a simple and lightweight testsuite. Thus, successfully answering the initial research questions:

> *"Can a lightweight Statistical Testsuite for RNGs with limited output be constructed? If so, how useful is it and to what degree can it differentiate RNG outputs?"*

Through extensive reading of the literature and analysis of existing implementations, fourteen tests have been incorporated, adapted and created to form the testsuite. With one test, Permutations, successfully adapted to work with Kbs of data instead of its intended Gbs. A scoring methodology was then devised to compare SALT to other existing test batteries, namely NIST's, ENT, Dieharder and TestU01. The results were definitive, showing that SALT was able to better differentiate the fourteen generators considered between 10^2 and 10^6 bits and that NIST's was the only suite to outperform it, needing 10^7 bits to do so. Furthermore, from just 250 bits SALT was able to stratify the tests on a basic level, a feat which it took the other suites over 10^6 bits

to achieve. Throughout this paper, we have demonstrated that a lightweight testsuite works far better than anticipated and that we can differentiate RNGs using several orders of magnitude fewer bits than previously thought. SALT also demonstrates that hours of computing time are not needed to produce these results, as it runs in seconds compared to other batteries' hours. Finally, we have wrapped SALT in an intuitive website, fulfilling the original goal of producing a testsuite anyone can quickly and easily use to perform a spot check on their device's RNG.

Thus, whereas the focus on other batteries is to prove beyond doubt that a PRNG accurately mimics a TRNG using gigabytes of data - SALT has demonstrated that in a constrained environment it is feasible to perform a basic analysis and assessment of RNGs, focusing on distinguishing poor-performing generators from better ones, rather than proving it is not a TRNG.

A Limitations

Conversely, SALT and this project have not been without their limitations. SALT is in no way intended as a replacement for the existing test batteries; the trend in results show that with over 10Mbs of data NIST's suite can outperform SALT and that given several Gbs the other test batteries will as well. Instead, SALT fills the gap at the other end of the scale; showing that we can still perform informative testing when only limited output and time is available

Moreover, the scoring methodology used by this project may prove inaccurate, hence the results show general correlations rather than smooth curves. This is due to the nature of RNGs. As discussed, they are expected to perform extremely poorly for a percentage of the time, which means given limited output we can accurately rank the bitstrings. However, these, in turn, do not result in an accurate ranking of the RNGs. Scoring also used NIST's suite as a benchmark, making the incorrect assumption that it is perfect. Repeating this project, further analysis could provide a more neutral benchmark point, utilising several Gbs more data. Unfortunately, due to time and workstation constraints, this was not possible to generate for this paper.

B Future Work

SALT has explored an entirely new area of the field compared to the existing literature. As such, the potential for future work in this area is enticing. Firstly, the work to make SALT accessible to the public should be continued, by providing a pip install library for Python programmers and permanently hosting the web service already developed. Using this web service, or possibly an app, users could easily check their device's current generator to ensure it was performing close to its expectation as calculated by the slower, more rigorous test batteries. Secondly, repeating this project - using further generators and a larger amount of data - would prove useful. This would result in a more accurate scoring function and may show us the absolute limit that SALT can differentiate between RNGs. Currently, the RNGs assessed can only be split into five broad groups, there may be others between these. Moreover, attempts to produce other lightweight testsuites would be welcomed. This paper has demonstrated the need for such a testsuite and shown that it is feasible to produce one. SALT could easily be extended or adapted to create further testsuites; by adding additional tests or tweaking the existing ones, it is possible to produce a more accurate testsuite. As the ubiquity of encrypted communication and cryptocurrencies continues to grow (Vandenberg, 2017), the need for such a testsuite grows with it. Knowing that a PRNG can theoretically produce TRNG-esque numbers is useless if, in practice, the device has insufficient entropy to generate them. With future work developing an even further streamlined SALT, it can be envisaged that before every new cryptographic key generation SALT would be deployed to ensure the ensuing numbers are "random enough" for the key generation's purpose.

REFERENCES

Bassham, L. E., Rukhin, A. L., Soto, J., Nechvatal, J. R., Smid, M. E., Barker, E. B., Leigh, S. D., Levenson, M., Vangel, M., Banks, D. L., Heckert, N. A., Dray, J. F. and Vo, S. (2010), SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications, Technical report.

Blum, L., Blum, M. and Shub, M. (1986), 'A Simple Unpredictable Pseudo-Random Number Generator', *SIAM J. Comput.* **15**, 364–383.

Brown, R. G., Eddelbuettel, D. and Bauer, D. (2013), 'Dieharder: A Random Number Test Suite', *Open Source software, URL http://www.phy.duke.edu/ rgb/General/dieharder.php* . Accessed 05/04/20.

Demirhan, H. and Bitirim, N. (2016), 'Statistical Testing of Cryptographic Randomness', *İstatistikçiler Dergisi:İstatistik ve Aktüerya* **9**, 1 – 11.

Gentle, J. (2006), 'Random number generation and Monte Carlo methods', *Metrika* **64**, 251–252.

Georgescu, C., Simion, E., Nita, A.-P. and Toma, A. (2017), A view on NIST randomness tests (in) dependence, *in* '2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)', IEEE, pp. 1–4.

Gustafson, H., Dawson, E., Nielsen, L. and Caelli, W. (1994), 'A computer package for measuring the strength of encryption algorithms', *Computers & Security* **13**(8), 687–697.

Haahr, M. (2010), 'Random. org: True random number service', *School of Computer Science and Statistics, Trinity College, Dublin, Ireland. Website (http://www. random. org). .* Accessed 01/04/20.

Hellekalek, P. (1998), 'Good random number generators are (not so) easy to find', *IMACS* **46**(5), 485 – 505.

Hull, T. E. and Dobell, A. R. (1962), 'Random Number Generators', *SIAM Review* **4**, 230–254.

Kim, S.-J., Umeno, K. and Hasegawa, A. (2004), 'Corrections of the NIST statistical test suite for randomness', *IACR Cryptology* **2004**, 18.

Knuth, D. E. (1997), *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Larrondo, H., Martín, M., González, C., Plastino, A. and Rosso, O. (2006), 'Random number generators and causality', *Physics Letters A* **352**(4), 421 – 425.

L'Ecuyer, P. (1992), Testing Random Number Generators, *in* 'Proceedings of the 24th Conference on Winter Simulation', WSC '92, ACM, New York, NY, USA, pp. 305–313.

L'Ecuyer, P. and Simard, R. (2007), 'TestU01: A C Library for Empirical Testing of Random Number Generators', *ACM Trans. Math. Softw.* **33**(4), 22:1–22:40.

Marsaglia, G. (1996), 'Diehard: a battery of tests of randomness', *Self-distributed from Florida State University* .

Marsaglia, G. and Zaman, A. (1993), 'Monkey tests for random number generators', *Computers & Mathematics with Applications* **26**(9), 1 – 10.

Matsumoto, M. and Nishimura, T. (1998), 'Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator', *ACM Trans. Model. Comput. Simul.* **8**(1), 3–30.

Nemiroff, R. (1994), 'RJN's More Digits of Irrational Numbers Page'. Accessed 17/01/20.
URL: *https://apod.nasa.gov/htmltest/rjn_dig.html*

Rotenberg, A. (1960), 'A New Pseudo-Random Number Generator', *J. ACM* **7**(1), 75–77.

Rukhin, A. (2000*a*), 'Testing Randomness: A Suite Of Statistical Procedures', *TVP* **45**.

Rukhin, A. L. (2000*b*), 'Approximate entropy for testing randomness', *Journal of Applied Probability* **37**(1), 88–100.

Saito, M. and Matsumoto, M. (2008), SIMD-oriented fast Mersenne Twister: a 128-bit pseudorandom number generator, *in* 'Monte Carlo and Quasi-Monte Carlo Methods 2006', Springer, pp. 607–622.

Soto, J. (1999), Statistical Testing of Random Number Generators, *in* '22nd NISSC'.

Strenzke, F. (2016), An Analysis of OpenSSL's Random Number Generator, *in* M. Fischlin and J.-S. Coron, eds, 'Advances in Cryptology – EUROCRYPT 2016', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 644–669.

Sunar, B., Martin, W. J. and Stinson, D. R. (2006), 'A provably secure true random number generator with built-in tolerance to active attacks', *IEEE Transactions on computers* **56**(1), 109–119.

Tsang, W. and Marsaglia, G. (2002), 'Some Difficult-to-Pass Tests of Randomness', *J; of Statistical Software* **7**.

Vandenberg, D. T. (2017), 'Encryption served three ways: Disruptiveness as the key to exceptional access', *Berkeley Tech. LJ* **32**, 531.

Von Neumann, J. (1951), 'Various techniques used in connection with random digits', *Appl. Math Ser* **12**(36-38), 5.

Walker, J. (1998), 'ENT - A Pseudorandom Number Sequence Test Program'. Accessed 05/04/20.
URL: *https://www.fourmilab.ch/random/*

Wayne, K. (2016), '10 Million Digits of PI'. Accessed 17/01/20.
URL: *https://introcs.cs.princeton.edu/java/data/pi-10million.txt*

Wertheimer, M. (2015), 'Encryption and the NSA Role in International Standards', *Notices of the AMS* pp. 165–167.

Zhu, S., Ma, Y., Lin, J., Zhuang, J. and Jing, J. (2016), More powerful and reliable second-level statistical randomness tests for NIST SP 800-22, *in* 'ASIACRYPT 2016', Springer, pp. 307–329.