Statistical Testing of PRNG: Generalized Gambler's Ruin Problem

Paweł Lorek[2], Marcin Słowik[1], and Filip Zagórski[1(✉)]

[1] Department of Computer Science, Faculty of Fundamental Problems of Technology, Wrocław University of Science and Technology, Wrocław, Poland
filip.zagorski@pwr.edu.pl

[2] Faculty of Mathematics and Computer Science, Mathematical Institute, Wrocław University, Wrocław, Poland

Abstract. We present a new statistical test (*GGRTest*) which is based on the generalized gambler's ruin problem (with arbitrary winning/losing probabilities). The test is able to detect non-uniformity of the outputs generated by the pseudo-random bit generators (PRNGs).

We also propose a new method, called *BitTracker*, of processing bits of a PRNG. In most of the statistical test-suites, bits are read in 31/32-bit groups. For many tests (*e.g.,* OPERM) only a few first bits of the group are taken into account. Instead of "wasting" bits (in some statistical tests), the method takes into account every single bit of the PRNG's output.

1 Introduction

Random numbers have lots of applications, *e.g.,* in physics, simulations, gaming, gambling, cryptography (*e.g.,* for generating a secret/public key). Pseudorandom number generators (PRNGs) are the algorithms outputting numbers (bits) which *should* be indistinguishable from truly random numbers (bits). We are unable to prove that a given PRNG returns numbers which are indistinguishable from truly random ones. All we can do is to invent/design some statistical tests and to state which PRNGs fail and which do not fail them. The better the PRNG is, the more tests it should pass.

Particularly, the cryptography requires high quality PRNGs, since the output bits are used *e.g.,* to generate secret and public keys. The straightforward application of PRNGs is especially visible in the case of stream ciphers. Here the use of a weak PRNG/stream cipher may immediately lead to breaking the secrecy [2]. To evaluate the aforementioned *quality* of PRNGs, a variety of tests can be applied to their output. They aim at checking if the sequence of numbers – or bits, depending on the actual implementation – produced by PRNG resembles a sequence of elements generated independently and uniformly at random. Because of the significance of the problem, lots of testing procedures were

Authors were supported by Polish National Science Centre contract number DEC-2013/10/E/ST1/00359.

J. Blömer et al. (Eds.): MACIS 2017, LNCS 10693, pp. 425–437, 2017.
https://doi.org/10.1007/978-3-319-72453-9_34

developed in recent years. Most of them are based on results from the probability theory: results on bin-and-boxes schemes, results on simple random walks (*e.g.,* law of iterated logarithm [19]), duration of gamblers' ruin game [11], or hitting time distributions [10]. One can then compare the realization obtained from a PRNG with what *should* happen.

A single statistical test checks only some specific property (or few of them, in the best case) that holds for truly random numbers. If a given PRNG passes the test, it does not imply that it produces truly random numbers. It only means that the tested sequence have this particular property. That is why batteries of tests were created. They run a series of tests on output of a given PRNG. We should interpret their output as follows: the more tests are passed, the better the PRNG is. There are some famous collections of tests, *e.g.,* NIST800-90A Test Suite [4], DieHard tests (considered deprecated), DieHarder [6], or TestU01 tests [12]. The last one by L'Ecuyer and Simard is considered as a state of the art in the field.

1.1 Related Work

RC4 family. Most of the experiments were conducted on variations of RC4 algorithm, the stream cipher designed by Ronald Rivest back in 1987. RC4 and its variations are known to have defects: the weaknesses were *e.g.,* shown in [8]; distributions of some biases were shown in [9]; a famous attack on RC4 implemented in WPA-TKIP was given in [18]; weaknesses of RC4+ (which was introduced in [16]) were given [3]; in [1] authors introduced the new bias, they were able to retrieve the RC4 secret key given the state table.

Gambler's ruin problem. The tests of Kim et al. [11] (titled "Tests of randomness by the gambler's ruin algorithm") were based on the expectation and the variation of the duration of a gambler's game. Assume initially a gambler has s dollars and a bank has $N - s$ dollars. At each step the gambler either wins a dollar with probability p or loses a dollar with the remaining probability. The game ends when the gambler is broke (reaches 0) or he wins (reaches N). The expectation and variance of the duration of the game is known in specific cases (authors' algorithms $A1$–$A5$ and $B1$–$B5$, for their algorithms $C1$–$C5$ it was not known earlier). Authors split the output of a PRNG into intervals of some prespecified number of bits, each sequence is then treated as a binary representation of a number from the interval $[0, 1]$. For the starting point s the PRNG is used until 0 or N is reached, the whole procedure is repeated 20000 times and a sample of the mean of the games' duration is calculated, finally the standard Z-test is performed. Authors do not rely on a single starting point s, but for $N = 300$ they start the gambler's game for each $s = 1, \ldots, N$ and they *judge* the PRNG depending on for how many starting points it fails. They claim "hidden defects" in some generators, including commonly used Mersenne Twister generator [14]. *However*, for each starting point s the same seed is used at each iteration. Thus, the games cannot be treated as independent. For example, this is one of the obvious dependencies: if the game started at say 10 finishes at 0 (i.e., losing),

then so do the games started at $1, \ldots, 9$. This also translates in obvious way to game's duration. The authors were aware that "the Z-values corresponding to different starting points are highly correlated", nevertheless they proceeded with "we can regard a Z-value to each starting point as a separate test result". The criticism was raised by Ekkehard and Grønvik in [7], where they baldly pointed out the mistakes. Moreover, Ekkehard and Grønvik [7] show that tests of Kim et al. [11] performed *properly* do not show any defects in Mersenne Twister (and some other) PRNG.

1.2 Our Contribution

We present two main contributions.

BitTracker. Many random-walk-based tests perform the following:

S1 Split the sequence of bits returned by PRNG into sequences of pre-defined length (usually 16 or 32-bit length)
S2 Treat each one as an integer or as a number from $[0, 1]$
S3 Perform some *walk* (e.g., go *left* or *right*) according to the number obtained in S2 *independently* of the current state.
S4 Iterate S1–S3, calculate some statistics and compare them to known facts about the statistic (distribution, expectation etc.).

The main drawback of the above procedure are steps S1 and S3. Imagine gambler's ruin problem where at each step we either go left or right with probability $1/2$. Then, treating consecutive 32 bits as a binary notation of a number, we actually test every 32-nd bit. Of course if we know that the probability is $1/2$ we can simply apply 1-bit long sequence. However, such a test can overlook some dependencies between the bits. We obviate the obstacle by applying as many bits as is needed (see Sect. 3 on `BitTracker`) and by using varying probabilities (which depend on the current state). The latter feature also eliminates the drawback of S3.

Generalized gambler's ruin test. The second contribution is the new statistical test. In spirit, our approach is similar to Kim et al. [11], we also use gambler's ruin based test. However, there are two main differences:

- we consider a generalized version (with arbitrary winning/losing probabilities);
- our test is based on winning probability (instead of a game duration).

The generalization is following: being at state $i : 0 < i < N$ (for some fixed N) we go right with probability $p(i)$, go left with probability $q(i)$ and stay with probability $1 - p(i) - q(i)$. Then, for each starting point s we calculate winning probabilities using recent results from [13]. We compare it to simulations' results.

Summarizing, we will introduce test (actually, a family of tests) *GGRTest* (Generalized Gambler's Ruin Test) which heavily exploits the results on winning probability in a generalization of gambler's ruin problem (Theorem 1). There is a large number of publications introducing a single test which usually tests just one or few aspects of PRNGs. The new test proposed in this paper shows its generality – by modifying parameters it can detect different weaknesses. This is achieved due to the fact that we use varying probabilities $p(i), q(i)$ and we use "as many bits as needed" via our `BitTracker` algorithm (see Algorithm 1).

Our family of tests while it is general it still is able to spot a weak cryptographic generators like Spritz or RC4-like generators.

2 Gambler's Ruin Problem: Explicit Formulas for Winning Probabilities

Fix N and sequences $\{p(i)\}_{i=1,\ldots,N-1}, \{q(i)\}_{i=1,\ldots,N-1}$ s.t. for $i \in \{1,\ldots,N-1\}$ we have $p(i) > 0, q(i) > 0$ and $p(i) + q(i) \leq 1$. Consider Markov chain $\mathbf{X} = \{X_k\}_{k\geq 0}$ on $\mathbb{E} = \{0, 1, \ldots, N\}$ with transition probabilities

$$\mathbf{P}_X(i,j) = \begin{cases} p(i) & \text{if } j = i+1, \\ q(i) & \text{if } j = i-1, \\ 1 - (p(i) + q(i)) & \text{if } j = i, \end{cases}$$

with convention $p(0) = q(0) = p(N) = q(N) = 0$. For $i \in \{0,\ldots,N\}$ define

$$\rho(i) = P(\tau_N < \tau_0 | X_0 = i),$$

where $\tau_k = \inf\{n \geq 0 : X_n = k\}$ (note that $\rho(N) = 1$ and $\rho(0) = 0$). This is an extension of classical Gambler's ruin problem, $1 - \rho(i)$ is the ruin probability of a gambler having initially capital i. We have

Theorem 1 ([13], version simplified to 1 dimension). *Consider generalized gambler's ruin problem: for fixed N the gambler starts with capital $0 \leq s \leq N$. Having s dollars he wins 1 with probability $p(s)$ or loses 1 with probability $q(s)$ for any $\{p(i)\}_{i=1,\ldots,N-1}, \{q(i)\}_{i=1,\ldots,N-1}$ such that $p(i) > 0, q(i) > 0, p(i)+q(i) \leq 1$. Then, the probability of winning is given by*

$$\rho(s) = \frac{\sum_{n=1}^{s} \prod_{r=1}^{n-1} \left(\frac{q(r)}{p(r)}\right)}{\sum_{n=1}^{N} \prod_{r=1}^{n-1} \left(\frac{q(r)}{p(r)}\right)}. \tag{1}$$

Remark. Note that for $p(r) = p, q(r) = q$ we recover the result for classical gambler's ruin problem (with possible ties)

$$\rho(s) = \frac{\sum_{n=1}^{s} \left(\frac{q}{p}\right)^{n-1}}{\sum_{n=1}^{N} \left(\frac{q}{p}\right)^{n-1}} = \begin{cases} \frac{1-\left(\frac{q}{p}\right)^{s}}{1-\left(\frac{q}{p}\right)^{N}} & \text{if } p \neq q, \\ \frac{s}{N} & \text{if } p = q. \end{cases} \tag{2}$$

3 BitTracker: Transforming Bit Strings into Intervals

It is a common practice to interpret a string of 0s and 1s as a binary representation of fractional part of a real number from the range $[0, 1)$. Such a representation has many flaws, for instance, the significance of bits is very biased. As already mentioned, the typical approach is to split bit string into strings of equal length and then to treat each one separately as a binary representation of a number from $[0, 1)$. This is especially critical when the representation is used to place the real number in one of several intervals. For instance, given a sequence of bits '01101100' and the interval $[0, 0.1_2)$, only the first bit is actually used to determine that the real number defined by the sequence (0.01101100_2) is within the given interval.

There are also other cases, when the finite, fixed representation has other flaws, like intervals with infinite binary representations of their end points (e.g., $[\frac{1}{3}, \frac{2}{3})$).

We propose a different method of *transforming* bit string into numbers from the range $[0, 1)$. Roughly speaking, the method requires *"as many bits as needed"* to determine in which interval it is. Let $\mathcal{P} = (x_0 = 0, x_1, x_2, \ldots, x_K = 1)$ be a partition of interval [0,1). The result of the method is not a number per se, but one of the intervals. If bit string is a string of truly random and independent bits, then the probability of each interval to be selected is equal to its length. The algorithm is given in Algorithm 1.

Algorithm 1. `BitTracker`

Require: Partition $\mathcal{P} = (x_0 = 0, x_1, x_2, \ldots, x_K = 1)$, bit string $\mathcal{B} = (b_0, b_1, \ldots)$.
Ensure: Interval $[x_{i-1}, x_i) \in \mathcal{P}$ determined by $\mathcal{B}$, number of consumed bits i.

1: $i = 0, l_0 = 0, r_0 = 1$
2: **if** $\exists\ \mathcal{A} \in \mathcal{P}$ s.t. $[l_i, r_i) \subseteq \mathcal{A}$ **then**
3: **return** $\mathcal{A}, i$ and **STOP**
4: **end if**
5: take a bit b_i from the bit stream
6: split the interval $[l_i, r_i)$ into two halves $[l_i, m_i)$, $[m_i, r_i)$ where $m_i = \frac{l_i + r_i}{2}$
7: **if** $b_i = 0$ **then** proceed with the left interval: $l_{i+1} = l_i, r_{i+1} = m_i$ **else** take the right interval: $l_{i+1} = m_i, r_{i+1} = r_i$.
8: $i := i + 1$
9: **goto 2**

Sample execution of `BitTracker` for partition $\mathcal{P} = \left(0, \frac{\sqrt{2}}{2}, 1\right)$ and bit string $\mathcal{B} = 1001110$ is given in Fig. 1.

3.1 BitTracker – Every Bit Counts

The *overlapping 5-permutations test* (OPERM5) takes five consecutive random (32-bit) numbers and check if orderings of these numbers occur with statistically

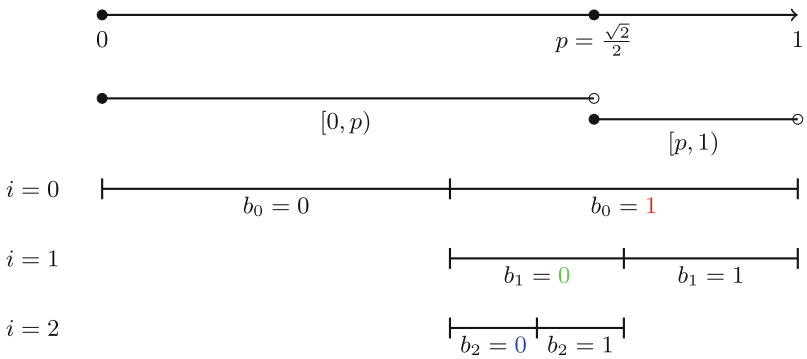

Fig. 1. `BitTracker`: Sample binary-to-interval conversion: Partition $\mathcal{P} = \left(0, \frac{\sqrt{2}}{2}, 1\right)$, bit string: $B = 1, 0, 0, 1, 1, 1, 0...$ (three first digits determine the interval)

equal probability. For the OPERM5 test only the most significant bits are taken into account. If one generates numbers in the form $x_{i,1}x_{i,2}\ldots x_{i,32}$ where $x_{i,j}$ are equal 0 and 1 with the same probability for $j = 1, \ldots, 6$ and $x_{i,j} = 0$ for $i > 6$ then the sequence of bits x_i would likely pass the test. This is because less significant bits play little role.

Now, if one applies BitTracker to the sequence and wants to apply the OPERM5 test, the situation changes.

The OPERMn takes n numbers and returns its relative order, which is then encoded into one of the possible $n!$ numbers. To perform OPERM5 with BitTracker on, one can do the following: Divide $[0, 1)$ into $[0, \frac{1}{n}), [\frac{1}{n}, \frac{2}{n}), \ldots, [\frac{n-1}{n}, 1)$ and check which interval is selected. – this is the position of the first number. To tell what is the relative position of the second number: divide $[0, 1)$ into $[0, \frac{1}{n-1}), [\frac{1}{n-1}, \frac{2}{n-1}), \ldots, [\frac{n-2}{n-1}, 1)$ and use BitTracker to tell which interval is chosen.

The position of j-th number is determined using the BitTracker on intervals: $[\frac{i}{n-j+1}, \frac{i+1}{n-j+1})$ for $i = 0, \ldots, n-j$.

4 Testing Procedure

4.1 General Approach

Fix N and sequences $\{p(i)\}_{i=1,\ldots,N-1}, \{q(i)\}_{i=1,\ldots,N-1}$ such that $p(i) > 0, q(i) > 0, p(i) + q(i) \leq 1$. From Theorem 1 we know the distribution of $Y^{(i)} = \mathbf{1}(\tau_N < \tau_0 | X_0 = i)$. Let $Y_j^{(i)}, j = 1, \ldots, n$ be *i.i.d.* samples form distribution of Y^i. The law of large numbers states that $\hat{\rho}_n(i) = \frac{1}{n}\sum_{j=1}^{n} Y_j^{(i)}$ (classical Crude Monte Carlo estimator) converges, as $n \to \infty$, to $EY^{(i)} = \rho(i)$. Moreover, from Central Limit Theorem we know that

$$Z^{(i)} = \frac{\hat{\rho}_n(i) - EY^{(i)}}{\sqrt{Var(Y^{(i)})}/\sqrt{n}} = \frac{\hat{\rho}_n(i) - \rho(i)}{\sqrt{\rho(i)(1-\rho(i))}}\sqrt{n}$$

has $N(0,1)$ distribution. $Z^{(i)}$ is often called **Z-statistics**. Thus, for a given starting point i we compare $Z^{(i)}$ with the critical value at 99% and 95%. If $|Z^{(i)}| > 2.58$ ($|Z^{(i)}| > 1.96$) we say that the point i "fails" the test at confidence level 99% (95%), *i.e.,* we mean that the numbers generated by a given PRNG and used for this point were not random. Finally, we count the number of points for which test fails and say that the PRNG fails if more than 1% (5%) of the points failed the test.

Our testing procedure $\mathsf{GGRTest}(\mathcal{G}, N, l, n, \{p(i)\}, \{\rho(s)\})$ is presented as Algorithm 2.

Algorithm 2. $\mathsf{GGRTest}(P, \mathcal{G}, N, M, l, n, \{p(i), q(i), \rho(i)\}_{i=1,\ldots,N-1})$

1: Generate $\mathcal{B} \leftarrow \mathsf{BitGeneration}(P, \mathcal{G}, M, l)$ as M l-bit long sequences from the generator $\mathcal{G}$.
2: **for all** $s \in \{1, \ldots, N-1\}$ **do**
3: Run n Gambler's ruin simulations starting at point s with the one-step probability of winning (losing) at point x is equal to $p(x)$ ($q(x)$), using bits of $\mathcal{B}$ to estimate the winning probability:

$$\hat{\rho}_n(s) = \frac{1}{n} \sum_{k=1}^{n} Y_{j,k}^{(s)},$$

where $Y_{j,k}^{(s)} = 1$ if the k-th simulation ended with winning and $Y_{j,k}^{(s)} = 0$ otherwise.
4: For obtained $\hat{\rho}_n(s)$ calculate Z-statistic:

$$Z^{(s)} = \frac{\hat{\rho}_n(s) - \rho(s)}{\sqrt{\rho(s)(1-\rho(s))}} \sqrt{n}.$$

5: Denote $\mathcal{F}_s = 1$ if $Z^{(s)} > 2.58$ and $\mathcal{F}_s = 0$ otherwise (for confidence level 99%, for confidence level 95% we set $\mathcal{F}_s = 1$ if $Z^{(s)} > 1.96$ and $\mathcal{F}_s = 0$ otherwise).
6: **end for**
7: Calculate $\mathcal{F} = \sum_{s=1}^{N-1} \mathcal{F}_s$.
8: **return** $\mathcal{F}$

4.2 Actual Parameters and Testing Procedure

We run $\mathsf{GGRTest}(P, \mathcal{G}, N, M, l, n, \{p(i)\}_{i=1,\ldots,N-1}, \{\rho(s)\}_{s=1,\ldots,N-1})$ for the following parameters:

- $N = 129$ – number of starting points,
- $M = 1$ – number of separate bit sequences,
- $l = 2^{35}$ – length of the sequence $\mathcal{B}$ of bits of a given PRNG $\mathcal{G}$ (4 GB),
- $n = 200000$ – number of simulations estimating the winning probability.
- $P =$ "GAMBLER_0001#" – per-run prefix for key derivation function.

The test was run for different generators $\mathcal{G}$ and for different gambler's ruin games. In fact we consider three different tests T1, T2, T3 which are defined

by the sequences of the winning probabilities $\{p(i)\}_{i=1,\ldots,N-1}$. In all cases we choose $p(i), q(i)$ such that $p(i) + q(i) = 1$ for all i. More precisely, the tests are defined as Algorithms 3, 4, 5.

Algorithm 3. TestT1$(P, \mathcal{G}, N, M, l, n)$

1: For $i = 1, \ldots, N-1$ set $p(i) = 1 - q(i) = 0.48$
2: Set the winning probability (the same as the winning probability of the classical Gambler's ruin problem) to:

$$\rho(s) = \frac{1 - \left(\frac{0.52}{0.48}\right)^s}{1 - \left(\frac{0.52}{0.48}\right)^N}$$

3: **return** GGRTest$(P, \mathcal{G}, N, M, l, n, \{p(i), q(i), \rho(i)\}_{i=1,\ldots,N-1})$

Algorithm 4. TestT2$(P, \mathcal{G}, N, M, l, n)$

1: For $p(i) = \frac{i}{2i+1}, q(i) = \frac{i+1}{2i+1}$,
2: Set

$$\rho(s) = \frac{\sum_{n=1}^{s} \prod_{i=1}^{n-1} \left(\frac{i+1}{i}\right)}{\sum_{n=1}^{N} \prod_{i=1}^{n-1} \left(\frac{i+1}{i}\right)} = \frac{\sum_{n=1}^{s} n}{\sum_{n=1}^{N} n} = \frac{s(s+1)}{N(N+1)}.$$

3: **return** GGRTest$(P, \mathcal{G}, N, M, l, n, \{p(i), q(i), \rho(i)\}_{i=1,\ldots,N-1})$

4.3 Bit Derivation

One more thing requires explanation. Namely, the way we "produce" bits $\mathcal{B}$ from a given PRNG $\mathcal{G}$. Recall $l = 2^{35}$ which corresponds to 4 GB.

Subsequently, the simulations are performed in the following way. Being at point s we apply `BitTracker` with partition $\mathcal{P} = (0, p(s), 1)$ (in our cases T1–T3 we have $p(s) + q(s) = 1$) with the *output* of the PRNG taken as the input. Tests reuse bits from the *output* only in case they run out of them (so-called *bit-wrapping*). We have found, that for the parameters described as above, the sequences wrap at most twice.

5 Experimental Results

We applied the procedure described in Sect. 4 to the following PRNGs:

- **RC4.** The above described RC4 with key of size 256 bits
- **RC4-64.** The version of RC4 using key of size 64 bits, where first 24 bits correspond to an IV.

- **RC4+** was introduced in 2008 in [16]. It is a modification of RC4 that introduced much more complex three-phase key schedule algorithm (taking about 3× as long as RC4) and slightly modified generation function.
- **RC4A** was introduced in 2004 in [17]. It is a modification of RC4 that is based on two independent internal permutations (states) interleaved in generation process.
- **VMPC** was introduced in 2004 in [20]. It is a modification of RC4 that uses a slightly modified KSA and a PRNG.
- **Spritz** is a recent (2014) sponge construction by Ronald Rivest that bears constructional similarities to RC4, see [15] for details. It can be used as random bit generator
- **Salsa20** which was introduced in 2008 in [5]. The stream cipher is based on pseudorandom function and ARX (add-rotate-xor) operations.
- **AES256-CTR.** AES with 256-bit key, counter mode.
- **Urandom.**
- **RandU LCG** known to be totally broken.

Algorithm 5. TestT3$(P, \mathcal{G}, N, M, l, n)$

1: For $i = 1, \ldots, N-1$ set $p(i) = \frac{i^3}{i^3+(i+1)^3}, q(i) = \frac{(i+1)^3}{i^3+(i+1)^3}$.
2: Set

$$\rho(s) = \frac{\sum_{n=1}^{s} n^3}{\sum_{n=1}^{N} n^3} = \frac{s^2(s+1)^2}{N^2(N+1)^2}.$$

3: **return** GGRTest$(P, \mathcal{G}, N, M, l, n, \{p(i), q(i), \rho(i)\}_{i=1,\ldots,N-1})$

Algorithm 6. BitGeneration$(P, \mathcal{G}, M, l)$

1: **for all** r in range $\{1, \ldots, M\}$ **do**
2: Calculate master key $K_r =$ SHA-256$(P||r)$
3: The key used for $\mathcal{G}$ is computed as λ-byte suffix of K_r, where λ denotes the length of the PRNG key.
3: Run $\mathcal{G}$ to obtain an l-bit long string $\mathcal{B}[r] = \mathcal{G}(K_r)_{|l}$.
4: **end for**
5: **return** $\mathcal{B}$

In Fig. 2 we present results of one experiment: number of points failing test means number of starting points for which the (absolute value of) z-statistic was larger then 2.58 (for confidence level 99%) or 1.96 (for confidence level 95%). Note that for 128 starting points we should have, on average 1.28 (for conf. level 99%) or 6.4 (for conf. level 95%) failing points.

The z-statistics for chosen PRNGs are presented in Fig. 3. The Z-statistics for RandU deviated significantly from the others, therefore we placed them in separate plot.

	Confidence Level 99%			Confidence level 95%		
	T1	T2	T3	T1	T2	T3
AES256CTR	2	3	3	8	10	10
RandU	115	123	110	119	123	112
RC4	1	1	3	7	3	6
RC4-64	0	2	2	4	7	10
RC4a	1	1	4	7	10	6
RC4+	0	1	1	5	3	4
Salsa20	2	1	1	5	6	8
Spritz	3	2	3	7	10	7
urandom	1	2	2	9	7	8
VMPC	0	2	1	0	8	3

Fig. 2. Simulations results for $N = 129$ and T1–T3. Number of points for which given test failed.

5.1 The χ^2 Test

We performed 31 times the experiment (whose results are presented in Fig. 2) for $T2$ and starting points 41–122 (we also excluded randu). Let O_i denotes number of times (out of 31) for the test failed for i starting points. Note that each point fails with probability 0.05 (conf. level 95%) or 0.01 (conf. level 99%), thus the number of failing points has $Bin(82, p)$ distribution ($p = 0.05$ for conf. level 95% and $p = 0.01$ for conf. level 99%). Thus, on average, there should be $E_i = 31\binom{82}{i}p^i(1-p)^{81-i}$ failing points observed. We can test it using $\tilde{\chi}^2$ statistics:

$$\tilde{\chi}^2 = \sum_{i=1}^{n_0} \frac{(O_i - E_i)^2}{E_i}.$$

In our experiments the maximal observed value was 12 (*i.e.,* in one of the simulations there were 12 (for conf. level 95%) failing points (6 for conf. level 99%) for some specific - Spritz - test), we truncate it to $n_0 = 13$ (conf. level 95%) or $n_0 = 7$ (conf. level 99%). Thus, the $\tilde{\chi}^2$ statistic has χ^2 distribution with 12 (or 6) degrees of freedom. The results are presented in Fig. 4. For χ^2 Test we excluded RandU, urandom and included Mersenne-Twister.

6 Summary

We presented a new, general technique called *BitTracker* – to handle bit sequences that are about to feed a statistical test. We also proposed and implemented a family of statistical tests which are based on properties of the generalized gambler's ruin problem. The preliminary results show that the new test-family has a potential to spot weaknesses in cryptographic bit generators (*e.g.,* Spritz).

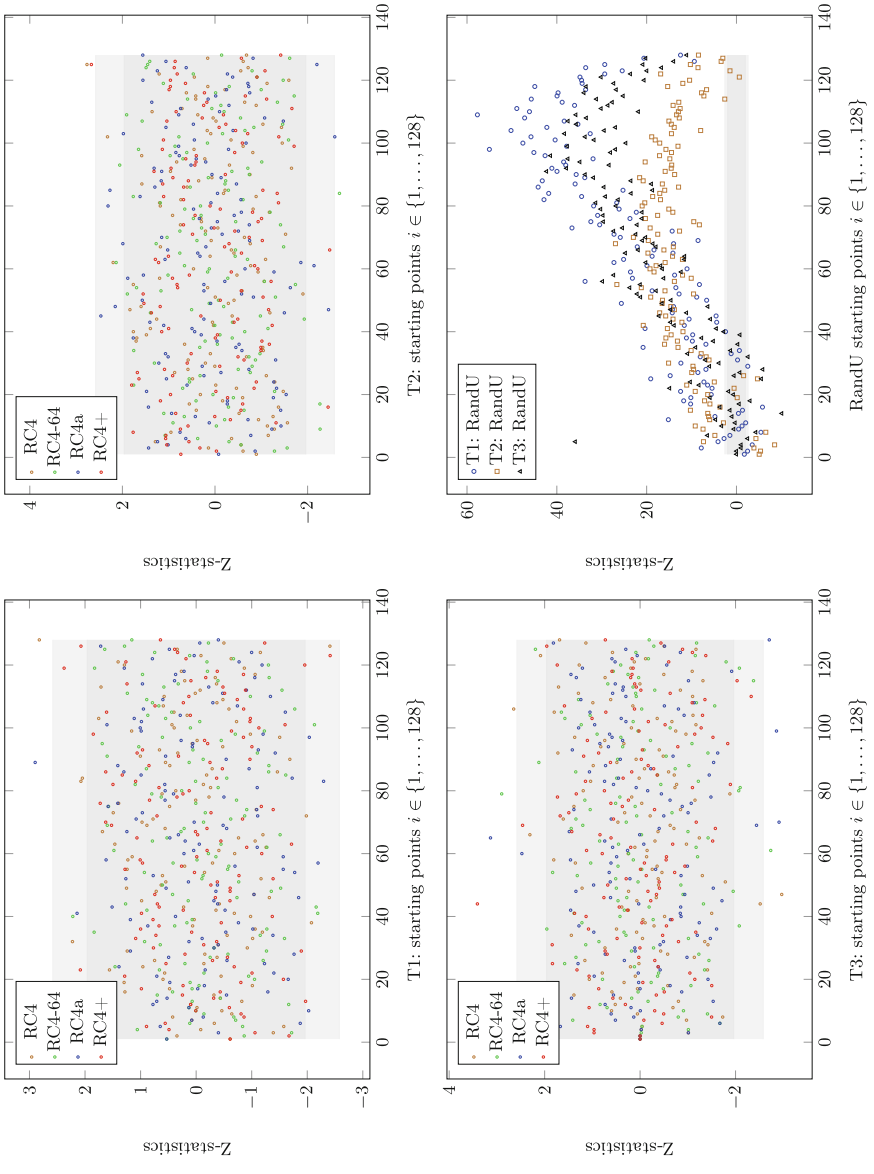

Fig. 3. Z-statistics obtained from simulations for tests T1–T3: RC4, RC4-64, RC4a, RC4+. RandU presented separately on last plot.

	Confidence Level 99%		Confidence level 95%	
	$\tilde{\chi}^2$	p-value	$\tilde{\chi}^2$	p-value
AES256CTR	5.6305	0.4658	23.1988	0.0260
Mersenne Twister	9.6979	0.13796	43.6676	1.7395e-05
RC4-64	30.9427	2.5994e-05	14.9915	0.24190
RC4	3.3817	0.75963	10.2043	0.5980
RC4a	9.6979	0.13796	21.5660	0.0426
RC4+	16.1583	0.0129	27.9784	0.0055
Salsa20	6.2032	0.4008	43.9593	1.5516e-05
Spritz	34.7307	4.8593e-06	70.1628	2.9859e-10
VMPC	31.8370	1.7533e-05	27.0109	0.0076

Fig. 4. χ^2 test for 31 experiments for $T2$, starting points 41–122.

References

1. Akgün, M., Kavak, P., Demirci, H.: New results on the key scheduling algorithm of RC4. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. LNCS, vol. 5365, pp. 40–52. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89754-5_4
2. AlFardan, N., Bernstein, D.J., Paterson, K.G., Poettering, B., Schuldt, J.C.N.: On the security of RC4 in TLS. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13), Washington, D.C., pp. 305–320. USENIX (2013)
3. Banik, S., Sarkar, S., Kacker, R.: Security analysis of the RC4+ stream cipher. In: Paul, G., Vaudenay, S. (eds.) INDOCRYPT 2013. LNCS, vol. 8250, pp. 297–307. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03515-4_20
4. Barker, E., Kelsey, J.: DRAFT NIST Special Publication 800-90A, Rev. 1 - Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical report, NIST (2014)
5. Bernstein, D.J.: The Salsa20 family of stream ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 84–97. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68351-3_8
6. Brown, R.G., Eddelbuettel, D., Bauer, D.: Dieharder: a random number test suite. www.phy.duke.edu/~rgb/General/dieharder.php
7. Ekkehard, H., Grønvik, A.: Re-seeding invalidates tests of random number generators. Appl. Math. Comput. **217**(1), 339–346 (2010)
8. Fluhrer, S., Mantin, I., Shamir, A.: Weaknesses in the key scheduling algorithm of RC4. In: Vaudenay, S., Youssef, A.M. (eds.) SAC 2001. LNCS, vol. 2259, pp. 1–24. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45537-X_1
9. Jha, S., Banik, S., Isobe, T., Ohigashi, T.: Some proofs of joint distributions of keystream biases in RC4. In: Dunkelman, O., Sanadhya, S.K. (eds.) INDOCRYPT 2016. LNCS, vol. 10095, pp. 305–321. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49890-4_17
10. Kang, M.: Efficiency test of pseudorandom number generators using random walks. J. Comput. Appl. Math. **174**(1), 165–177 (2005)
11. Kim, C., Choe, G.H., Kim, D.H.: Tests of randomness by the gambler's ruin algorithm. Appl. Math. Comput. **199**(1), 195–210 (2008)

12. L'Ecuyer, P., Simard, R.: TestU01: a C library for empirical testing of random number generators. ACM Trans. Math. Softw. **33**(4), 22-es (2007)
13. Lorek, P.: Generalized gambler's ruin problem: explicit formulas via Siegmund duality. Methodol. Comput. Appl. Prob. **19**(2), 603–613 (2017)
14. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (1998)
15. Schuldt, J.C.N., Rivest, R.L.: Spritz—a spongy RC4-like stream cipher and hash function. Technical report (2014)
16. Paul, S., Preneel, B.: A new weakness in the RC4 keystream generator and an approach to improve the security of the cipher. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 245–259. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25937-4_16
17. Maitra, S., Paul, G.: Analysis of RC4 and proposal of additional layers for better security margin. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. LNCS, vol. 5365, pp. 27–39. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89754-5_3
18. Vanhoef, M., Piessens, F.: All your biases belong to us: breaking RC4 in WPA-TKIP and TLS. In: USENIX Security Symposium (2015)
19. Wang, Y., Nicol, T.: On statistical distance based testing of pseudo random sequences and experiments with PHP and Debian OpenSSL. Comput. Secur. **53**, 44–64 (2015)
20. Zoltak, B.: VMPC one-way function and stream cipher. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 210–225. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25937-4_14