Faster Randomness Testing with the NIST Statistical Test Suite

Marek Sýs and Zdeněk Říha

Masaryk University, Brno, Czech Republic
{syso,zriha}@fi.muni.cz

Abstract. Randomness testing plays an important role in cryptography. Randomness is typically examined by batteries of statistical tests. One of the most frequently used test batteries is the NIST Statistical Test Suite. The tests of randomness should be rather fast since they usually process large volumes of data. Unfortunately, this is not the case for the NIST STS, where a complete test can take hours. Alternative implementations do exist, but are not very efficient either or they do not focus on the most time-consuming tests. We reimplemented all NIST STS tests and achieved interesting speedups in most of the tests, including the tests with the highest time complexity. Overall, our implementation runs 30 times faster than the original code.

Keywords: Berlekamp-Massey algorithm, NIST STS, randomness statistical testing.

1 Introduction

Randomness is connected with many areas of computer science, in particular with cryptography. Well designed cryptographic primitives like hash functions, stream ciphers, etc., should produce pseudorandom data. Randomness testing therefore plays an important and fundamental role in cryptography. Randomness is typically examined by empirical tests of randomness. Each test examines data by looking at a specific feature (number of ones, m-bit blocks, etc.). Tests are usually grouped into test batteries (also called test suites) to provide more complex randomness analysis. When testing any new source of (pseudo)randomness, the statistical test suites are of crucial importance. New sources of (pseudo)randomness include pseudorandom generators and entropy collectors for various kinds of environments (including smartcard, wireless sensor nodes, mobile devices, servers, desktops, etc.) and quite many new ideas and/or environments are emerging, so the tests are run very frequently in practice. Passing statistical tests of randomness is an important step for a (pseudo)random data generator being recognized or approved by certification bodies or other authorities.

All tests measure how the observed statistics of the analysed feature fit the expected statistics. Empirical tests of randomness compare the expected and obtained characteristics by standard statistical methods. Thus randomness is characterized and described in the terms of probability. The result of each test is a P-value that represents the probability that the chosen test statistic will assume values that are equal to or worse than the observed test statistics. This concept allows one to evaluate randomness according to several examined features at once. Combination of several P-values increases the confidence about the randomness/non-randomness of given data. Confidence about the data randomness can be also increased by increasing the analysed data volume. In practice, the analysed data volume is usually in the order of GBs and therefore the speed of these tests should be high. Unfortunately, most batteries are not implemented efficiently.

There are five well-known batteries – NIST STS [3], Diehard [4], TestU01 [6], ENT [7] and CryptX [8]. Only the first three batteries are commonly used for the randomness analysis, since CryptX is a commercial software and ENT provides only a very basic randomness testing. Position of the NIST STS is special as it has been published as an official NIST document. Therefore NIST STS is often used in preparation of formal certifications or approvals. Diehard and its novel implementation Dieharder were proposed for testing randomness of numbers rather than bitstreams. The newest and most powerful battery TestU01 was introduced in 2007 by Lecleuyer and Simard. TestU01 [6] incorporates new tests and implements the current state of the art of randomness testing. Diehard/Dieharder and TestU01 also implement some of the NIST tests, but they do not implement all NIST tests (Diehard) or the tests are not efficient (TestU01).

From time to time, there appear news about optimised versions of NIST STS. However, no such optimisation with either full description of the changes or NIST ‘approval’ exists, at least to our best knowledge. The goal of our work is to rewrite the NIST STS battery into a new version, with the same tests, with much better time- and space-efficient implementation of empirical tests of randomness and to provide a full description of the optimisations in an open publication, together with the source code openly available.

This paper is organised as follows: Section 2 provides an overview of the NIST tests, alternative implementations and the performance of the original code. Section 3 briefly describes our improvements. Section 4 discusses how we evaluated our algorithms and Section 5 summarizes the results of the performance testing.

2 Statistical Test Suites

The NIST tests are defined in [3]. The NIST Statistical Test Suite (NIST STS) package implements all the NIST tests. Although some particular NIST tests are also implemented in other test batteries (Diehard, TestU01), we further focus on the reimplementation of the whole NIST STS package.

2.1 NIST Statistical Tests

The original NIST document [1] defined 16 empirical test of randomness. These tests were developed to test the hardware or software based cryptographic random or pseudorandom number generators. During the next two revisions [2,3], the Lempel-Ziv test was removed due to implementation problems identified by the NIST. The current set of the NIST tests consists of 15 tests. All tests are parameterised by a parameter n that denotes the length (in bits) of the processed bitstream. Although all the tests are proposed to detect deviations from randomness for the whole bitstream, only several tests can detect local non-randomness. These tests are also parameterised by a second parameter denoted by m or M [3]. Tests parameterised by m are developed to detect the presence of too many m-bit patterns in a sequence. Tests with the second parameter M examine distribution of the specific feature across n/M parts (of equal size M bits) of a given bitstream. All tests in NIST STS compute P-values using asymptotic reference distributions (χ^2 or normal) and therefore reasonable results are obtained only for appropriate settings of the parameters n, m and M. Overview of all the tests and meaningful settings of their parameters are summarized in Table 1.

Table 1. The recommended size n of the bitstream for the particular tests. Some tests are parameterised by a second parameter m, M, respectively. The table shows meaningful settings of the second parameter depending on n.

Test #	Test name	n	m or M
1.	Frequency (Monobit)	$n > 100$	-
2.	Frequency within a Block		$20 \leq M \leq n/100$
3.	Runs	$n \geq 100$	-
4.	Longest run of ones in a block		
5.	Binary Matrix Rank	$n > 38912$	-
6.	Discrete Fourier Transform (Spectral)	$n \geq 1000$	-
7.	Non-overlapping Template Matching		$2 \leq m \leq 21$
8.	Overlapping Template Matching		$1 \leq m \leq n$
9.	Maurer's Universal		$1 \leq m \leq n$
10.	Linear complexity	$n > 10^6$	$500 \leq M \leq 5000$
11.	Serial		$3 \leq m \leq \lfloor \log_2 n \rfloor - 3$
12.	Approximate Entropy		$m \leq \lfloor \log_2 n \rfloor - 6$
13.	Cumulative sums	$n > 100$	
14.	Random Excursions	$n \geq 10^6$	
15.	Random Excursions Variant	$n \geq 10^6$	

2.2 NIST STS

The NIST test suite implements various random number generators and the 15 empirical tests developed to test randomness of binary sequences. The whole package is written in ANSI C in order to obtain a platform independent code. The source code of NIST STS was ported to Windows XP and Ubuntu Linux, and with minor modifications it also may be ported to different platforms. The NIST STS transforms an input file (stored as ASCII characters '0' and '1' or as binary data) to a byte array, where each byte (value 0 or 1) represents a single bit of the analysed bitstream. Byte representation of data allows one to use the same implementation of tests on little- and big-endian systems. The code universality comes at the expense of memory and time inefficiency of tests. Some of the tests have a preprocessing phase, but it is negligible for large volumes of data. The time complexity of each test is linear according to data volume n. The performance of the Rank test described in Figure 2 illustrates the linearity of tests. Performance of the tests with the second parameter (m, M) depends on its particular value. Table 2 shows run times of tests (implemented in the NIST STS) obtained after processing 20 MB of pseudorandom data ($n = 167,772,160$) with minimum and maximum recommended values of m or M. Table also shows the percentage to identify the time critical tests.

Table 2. Run times of particular tests obtained for minimum and maximum values of their second parameters m or M

Test	m, M	Time(ms)	%	m, M	Time (ms)	%
Frequency (Monobit)	-	203	0.12	-	203	0.01
Frequency within a Block	$n/100$	46	0.03	20	63	0.00
Runs	-	1140	0.67	-	1140	0.08
Longest run of ones in a block	-	656	0.39	-	656	0.04
Binary Matrix Rank	-	3781	2.23	-	3781	0.25
Spectral	-	24625	14.50	-	24625	1.63
Non-overlapping Template	2	1750	1.03	21	140015	9.28
Overlapping Template	2	672	0.40	24	3343	0.22
Maurer's Universal	-	2843	1.67	-	2843	0.19
Linear complexity	500	122390	72.04	5000	1187453	78.69
Serial	2	3687	2.17	24	85297	5.65
Approximate Entropy	2	4422	2.60	24	55860	3.70
Cumulative sums	-	984	0.58	-	984	0.07
Random Excursions	-	562	0.33	-	562	0.04
Random Excursions Variant	-	2125	1.25	-	2125	0.14
Total		*169886*	*100*		*1508950*	*100*

2.3 NIST Reimplementations

There were some attempts to reimplement the NIST STS efficiently. In [9] authors rewrite the NIST STS package to a byte-oriented implementation. Byte-oriented code allows one to speed up most tests, since some precomputation (lookup tables) can be used. Authors also made other improvements to the source code and finally obtained 13.45 average speedup of tests. However, this average speedup says nothing about the overall time, since it does not reflects the fact that *durations of particular tests are quite different.* The most time-consuming test (Linear complexity) is only 3 times faster in the reimplemented version [9]. Thus the overall speedup is significantly smaller and the whole testing process is at most 4 times faster. In [10] authors also tried to reimplement the NIST STS, but they obtained mostly worse results than in [9]. Unfortunately, both implementations are not publicly available and therefore we use only the published textual results to compare the perfomance. We contacted the authors of both papers, and we got some response with authors not sharing the source codes.

3 Improvements

Although we probably use ideas and principles similar to those in [9] to speed up some of the tests, our optimizations have been proposed and implemented independently. Moreover, we were able to apply these ideas to speed up the most time-consuming tests, which are Linear complexity, Non-overlapping template matching and the Serial tests.

All our optimizations are based on three basic ideas. We use lookup tables (LUTs), fast extraction of an integer from the byte array and word-word operations instead of bit-bit operations. Each optimization is used for a different type of tests determined by its complexity. Although the run time of each NIST test is linear to the number of bits n, the run times of particular tests vary significantly as we can see in Table 2. Run times of fast (simple) tests are usually influenced by the second parameter m. Poor performance of the most time-consuming tests is caused by subroutines for complex algorithms like Berlekamp-Massey, Gauss elimination or Fast Fourier Transformation.

3.1 Classes of Tests

NIST STS tests can be divided (according to their complexity and used optimizations) into three classes as follows:

1. Fastest tests that process each bit of bitstream once – *Frequency*, *Block Frequency*, *Runs*, *Longest run*, *Cumulative sums*, *Random Excursion* and *Random Excursion Variant.*
2. Fast tests that process m-bit blocks – *Non-overlapping template matching*, *Overlapping template matching*, *Universal*, *Serial* and *Approximate entropy.* Run times of these tests are dependent on m since each bit of the m-bit block is compared with some pattern in the NIST STS implementation.
3. Slow and complicated tests – *Linear complexity*, *Spectral*, *Rank* – tests that use quadratic algorithms (Linear complexity, Rank) or sub-quadratic algorithm (Spectral).

3.2 Optimizations of Simple Tests

Tests from the class 1 are optimized by the LUTs since these tests compute single value characteristics of bits like proportion of '0' and '1' bits, frequency of bit change (runs), length of runs and cumulative sum of bits. Tests use LUTs that consist of precomputed values for all k-bit blocks indexed by a block interpreted as an integer value. In our implementations, we use $k = 8$ as an appropriate value since the LUTs have a reasonable number of entries ($2^k = 256$). To run the tests we need to divide the bitstream into 8-bit blocks and continuously compute bit characteristics using corresponding table values. Choosing $k = 8$ we have 8-bit blocks since the bitstream is stored as a byte array. The use of LUTs can be illustrated on the Frequency test that computes the number of ones in the bitstream. The frequency test uses a LUT with entries $LUT[i] = v_i$, where v_i represents the number of ones (Hamming weight) in the index i (8-bit block). To compute the number of ones in a bitstream, it is sufficient to sum the corresponding LUT values for all bytes in a byte array. It should be noted that for other tests we use several LUTs that describe the input, output and internal characteristics of 8-bit blocks. For a more detailed description of tests from the first class, look into the source code available at [11].

Tests from the class 2 process m-bit blocks of the bitstream. To speed up these tests, we implemented a fast function *get_nth_block* that can extract arbitrary m-bit block ($m \leq 25$) from a given bitstream (byte array). Upper bound 25 of the block size is sufficient for all the tests from this class since m is upper-bounded by $\log_2 n - 3$ (Serial test). For 20 MB of data this upper bound has the value $m = 24$ and therefore function *get_nth_block* can be used. Function *get_nth_block* is fast and it is able to return all m-bit blocks from a 100 MB bitstream within a second on a standard modern computer. More effective optimization is based on the observation that all these tests (except Universal) can be evaluated from a single histogram of m-bit blocks.

In these tests, we use the histogram represented by the array H of frequencies (integers) of m-bit blocks indexed by blocks themselves. The histogram is computed using the function *get_nth_block* that is used to extract overlapping m-bit blocks b_i for $i \in \{0, 1, \cdots, n - m\}$ from the bitstream. These blocks are used as indexes to H for incrementing the corresponding frequencies $H[b_i]$. Since access to the array H and the increment are very fast operations, the histogram can be obtained also within a second (on the standard modern computer for 100 MB of data). Figure 1 illustrates the typical dependency between parameter m and the performance of the test.

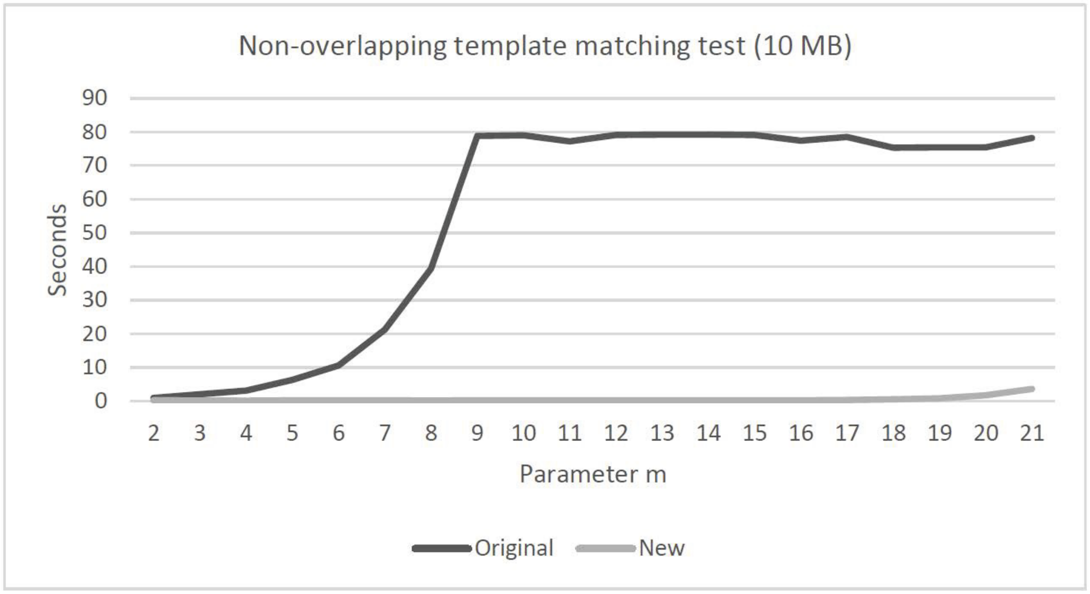

Fig. 1. Run times of the Non-overlapping test

The time complexity of the test depends on the number of searched templates (predefined k-bit patterns), which rise exponentially with the parameter m. Due to exponential nature of the number of templates, the original implementation imposes a practical limit on the maximum number of templates being tested (148). This limit is in effect for all $m > 9$ as we can see in the Figure 1, where the time complexity becomes constant.

Our implementation computes a histogram H for an arbitrary $m < 25$ in the same way and in a single pass. Therefore the complexity of our implementation is constant – independent on m (the real execution is influenced by processor cache management as you can see in Figure 1). Moreover, we compute H for all m-bit patterns and could easily provide complete statistics, yet for the reason of compatibility with the original version we stick to the limit of the maximum number of templates.

3.3 Spectral Test

The Spectral test is the only test that is not reimplemented in our battery. The Spectral test uses the Fast Fourier Transformation and therefore its run time is determined by the prime factors of n rather than by the value n itself. To speed up this test, it suffices to use n with small factors. The best choice is to take n of the form $n = 2^k$, for which the Spectral test run time is comparable to fast tests in the class 1.

3.4 Binary Matrix Rank Test

The Rank test uses the Gaussian elimination subroutine to examine whether the rank of the 32x32 boolean matrix is 32, 31 or less. Our implementation is based on the same idea as the Rank test in [9]. We use word-word operations instead of bit-bit operations. Since the square boolean matrix has the size 32, we represent it as an array of 32 unsigned integers, each of them representing a row of the matrix. Rank of the matrix is computed using fast bitwise operations XOR, AND and shift. The XOR operation realizes the row addition. Bitwise AND and shift are used for the pivot finding. Although our implementation of the Rank test is probably very similar to the implementation in [9], we improve it by adding the stop condition. We stop the computation if there are two columns with no pivot (thus the rank is less than 31).

3.5 Linear Complexity Test

The Linear complexity test is focused on determining the linear complexity L of a finite binary sequence. The linear complexity of a sequence equals to the length of the smallest linear feedback shift register (LFSR) that generates the given sequence. The Linear complexity test uses an efficient Berlekamp-Massey algorithm to compute this smallest LFSR. The Berlekamp-Massey (BM) algorithm for a binary sequence can be described by the following pseudocode:

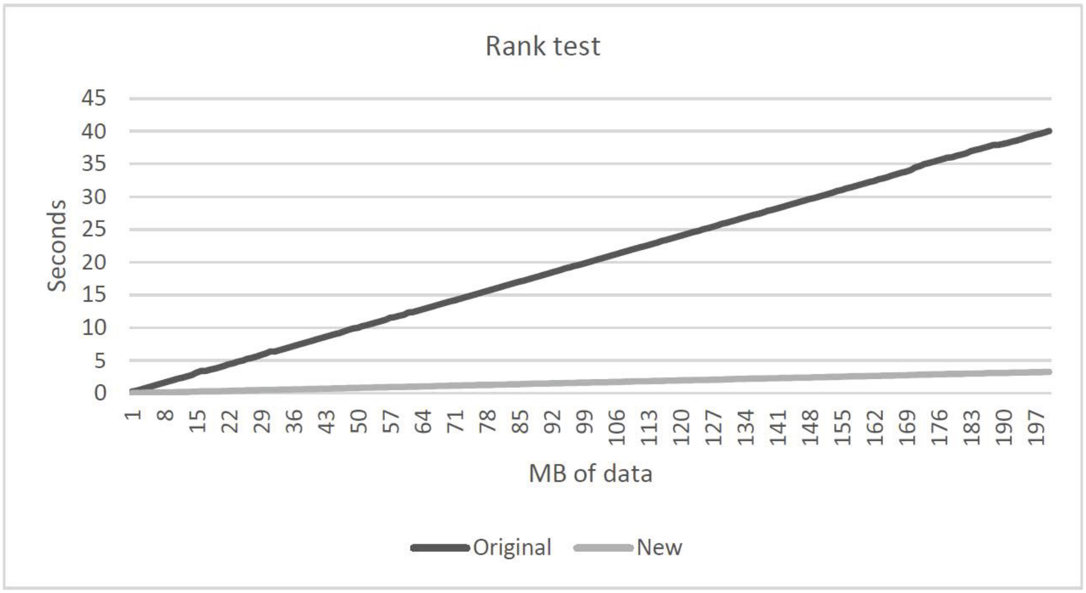

Fig. 2. Run time of the Rank test

```
Data: binary sequence $S = \{s_0, s_1, \cdots, s_{n-1}\}$ of the length n
Result: shortest LFSR generating S
Set arrays b, c to 1;
Set length $L$ of $c$ to 0;
$m \leftarrow -1$;
for $N \leftarrow 0$ to n-1 do
    Compute $d \leftarrow \sum_{i=0}^{L} c_i s_{N-i}$ ;
    if $d = 1 \pmod{2}$ then
        update $c$ by c ← c XOR ( b >> N-m) ;
        t ← c;
        if $L \leq n/2$ then
            t ← b;
            L ← N + 1 - L;
            m ← N ;
        end
    end
end
return c;
```

Algorithm 1. Pseudocode of the Berlekamp-Massey algorithm

The BM algorithm is an iterative method that constructs the smallest LFSR c_N generating the subsequence $S_N = \{s_0, s_1, \cdots, s_N\}$ of $S_n = \{s_0, s_1, \cdots, s_{n-1}\}$, $s_j \in \{0, 1\}$ in the N-th iteration. The BM tests whether the LFSR c_{N-1} that generates the subsequence S_{N-1} also generates the S_N sequence. The BM algorithm computes the discrepancy d that denotes the Hamming weight of the sequence S_N masked with the shifted LFSR c_{N-1} (stored as binary array). The BM algorithm uses another LFSR b_{N-1} (t is used to copy c to b) that represents the last LFSR (different to c_{N-1}) computed up to $N-1$ iteration. If the discrepancy d is even then $c_N = c_{N-1}$ and $b_N = b_{N-1}$. Table 3 illustrates how the discrepancy is computed from the LFSR. The table shows that the LFSR $b = x_2 + 1$ stored as a bit-array $b = 101$ forms a LFSR for S_4, i.e., $b_4 = 101$ since all discrepancies (sequence masked by shifted b) are even.

Table 3. Principles of the Berlekamp-Massey algorithm

index	0	1	2	3	4	5	6	7		
$S_{10} =$	0	1	0	1	1	0	1	1	1	
$b =$	1	0	1							d = 0
$b >> 1 =$		1	0	1						d = 2
$b >> 2 =$			1	0	1					d = 1

index =	0	1	2	3	4	5	6	7	8	
$S_{10} =$	0	1	0	1	1	0	1	1	1	
$b' = b >> 2 =$			1	0	1					d = 1
$c' = c >> 6 =$							1	1	1	d = 3
$c'' = b'$ XOR $c' =$			1	0	1		1	1	1	d = 4

In the case of an odd d the LFSR c_N is replaced by the right shifted c XOR-ed with $b_{N-1} >> (N-m)$ and b_N is set to c_{N-1}. The idea of combining of c_{N-1}, b_{N-1} to c_N is to get even discrepancy for the application of a new c_N. Table 3 shows the principle of the combination of LFSRs b_7, c_7 (represented by b', c') obtaining the new c_8 (represented by c'').

LFSR $b = 101$ forms the smallest LFSR for S_3, but not for the sequence S_4 and therefore the discrepancy for LFSR b and the subsequence S_4 is odd ($d = 1$). LFSR $c = 111$ forms the smallest LFSR for S_7, but not for the sequence S_8 ($d = 3$). The new smallest LFSR c for S_8 is constructed in the way that the discrepancy for a new $c = 1010111$ (represented by the bit array c'') is combined as XOR (in this case $d = 1+3 = 4$) of two odd discrepancies computed earlier for b', c'. It suffices to work with the shifted bit arrays b', c' instead of the original LFSRs b, c. Bit-arrays b', c' represent b, c shifted appropriately for the computation of d.
All improvements to the BM algorithm are based on the following observations:

1. the discrepancy d in the N-th iteration can be computed as the Hamming weight of the masked sequence S_n by bit array c',
2. the next discrepancy in $N+1$-th iteration is computed using $c' = c' >> 1$ that is shifted one bit to the right.

Our speedup of the BM algorithm is based on word (integer, long) representation of bit arrays S_n, c' and b'. This representation allows one to use fast bitwise operations. AND is used for masking of the word array S_N by the word array c'. The discrepancy d is computed from the masked S_n using a LUT storing the Hamming weights of bytes. XOR is used for the combination of b' and c' into a new $c'' = c'\ XOR\ b'$. In each iteration c' is shifted one bit to the right $c' = c' >> 1$. We made other improvements concerning elimination of processing zero words. For more details about the improvements look at the source code [11].

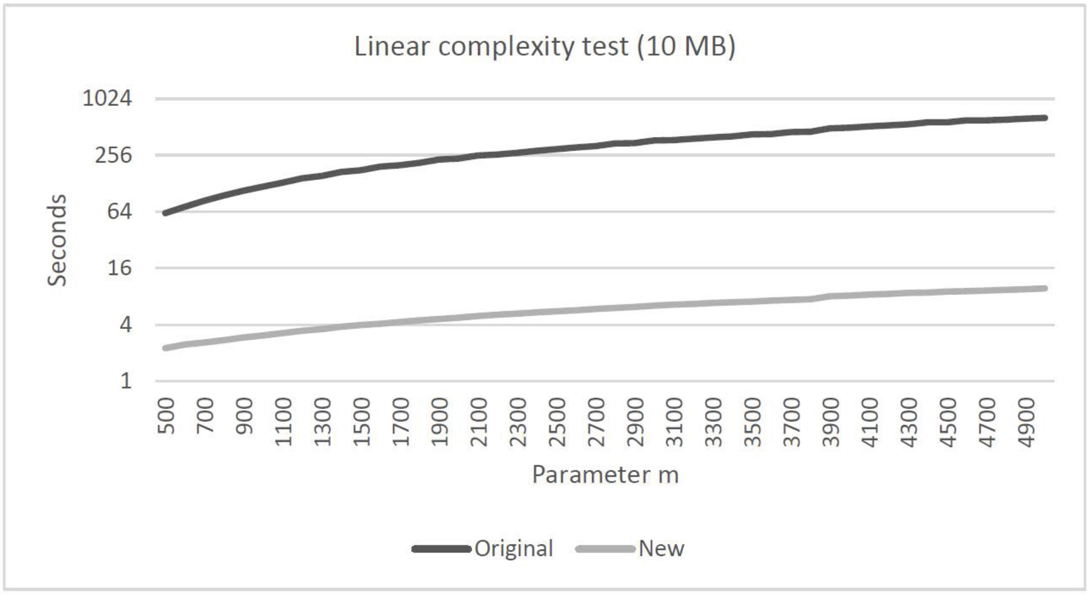

Fig. 3. Run time of the Linear complexity test (note the logarithmic scale of the y axis).

4 Implementation Testing

After the implementation of the faster variants of the algorithms we ran series of tests. The first tests were aimed at verification of the correctness. We verified the correctness of all values that are outputs from the tests (we ran both the original and new implementation and compared the results).

Many of the results are floating point numbers stored in the *double* type. Comparing floating point numbers can be tricky. We compiled the program with the improved consistency of floating-point operations. In majority of tests all the results stored as doubles matched perfectly in all the bits. In exceptional cases (Runs test) the results differed in a single (least significant) bit due to compiler computational optimizations. In these cases we had to allow for the differences in the least significant bit of the mantissa of the double type. Moreover we allow for the negligible difference of *FLT_EPSILON* for the same reasons in the Approximate entropy tests. We also had to consider special values of floating point numbers (e.g., the variable a storing the INDefinite value does not fullfil the $a == a$ condition) during comparison of the results.

We performed series of tests of all the algorithms with many different lengths of the bitsteams and with different parameters. We used pseudorandom bit sequences and also special values such as zeros, ones and alternating ones and zeros.

We followed the NIST parameter recommendations and typically performed the tests for all the bitstream lengths between 1 and 1,000,000 bits. We also performed the tests for randomly chosen bitstream lengths between 1 bit and 800,000,000 bits such that the length $n_{i+1} = n_i * 10 + rand()\%8$ was computed to catch possible errors caused by bitstream lengths not being a multiple of 8.

We were not able to perform all the above mentioned tests due to time complexity of the tests (Spectral, Non-overlapping template matching, Approximate entropy, Serial, Linear complexity) or high number of possible configurations (Block Frequency) for the time-consuming tests. Therefore we only performed a subset of the tests.

In a few configurations our implementation does provide a result while the original implementation is not able to compute a result (e.g., the Random Excursion test is limited in the number of cycles), in some situations our implementation does not support unusual parameters while the original implementation does (e.g., the Serial test with $m > 25$). In such situations, we could not compare the results. In all other situations the results do match. The limitations of our implementation with respect to the original implementation and the NIST recommendations (see Table 1) are:

- Overlapping template matching test: $m \leq 25$,
- Serial: $m \leq 25$.

The above mentioned limitation of the m above 25 can be easily shifted towards 32 as described in the Readme.txt file at [11]. The verification tests are time-consuming, but if you are interested you can run them on your own system as described in the source codes.

5 Performance Testing

We measured both the number of CPU cycles and the time consumed in milliseconds. As the results of both measurements are consistent, we present only the results in milliseconds (for very short tests the duration in milliseconds is recomputed from the number of CPU cycles). We performed all the tests ten times and we used the minimum value to avoid the noise introduced by the OS scheduling.

The source code, including the verification of the results and the speed measurement, can be compiled on Linux systems (tested with gcc 4.4.7 on RHEL 6.5), but we primarily used MS Windows for testing. The speed improvement was measured on a Windows 8.1 Fujitsu S792 notebook equipped with Intel Core i7 having 2 cores[1] running at 3 GHz and 8 GB of memory. The code was compiled using MS Visual Studio 2013. We produced a x64 binary in the Release mode with the default parameters.

Although the speed measurements were performed with a 64-bit binary on a 64-bit operating system, our implementation compiles also on a 32-bit system with a similar performance. The source code relies on the fact that the size of *int* is at least 32 bits and the processor works in the little-endian architecture.

The speed improvements are summarized in the Table 4. As you can see, the Linear complexity test significantly influences the overall numbers. We present the final results with the Linear complexity test configured to $m = 5000$. For the other extreme value of $m = 500$ the speedup factor of the test is 37x, which decreases the overall speedup to 10x.

[1] No multithreading is used in the application.

Table 4. Run times (for 20 MB of data) of the original implementation NIST STS, the implementation from [9] and our new implementation. Parameters m or M were chosen to be able to compare all three implementations.

Test	m, M	Original (ms)	New (ms)	Speedup Our vs. NIST	Speedup [9] vs. NIST
Frequency (Monobit)		203	15	13.5	9.82
Frequency within a Block	128	94	31	3.0	9.63
Runs		1140	31	36.8	5.84
Longest run of ones in a block		656	31	21.2	6.51
Binary Matrix Rank		3781	297	12.7	7.91
Spectral		24625	25062	0.98	-
Non-overlapping Template	9	139641	343	407.1	3.13
Overlapping Template	9	1359	406	3.3	15.15
Maurer's Universal		2843	156	18.2	12.8
Linear complexity	5000	1187453	18421	64.5	3.92
Serial	9	24078	313	76.9	48.73
Approximate Entropy	8	16484	312	52.8	54.16
Cumulative sums		984	31	31.7	3.31
Random Excursions		562	515	1.1	1.26
Random Excursions Variant		2125	515	4.3	6.09
Total		*1406028*	*46464*	*30.3*	-

6 Conclusion

We reimplemented the NIST STS with the focus on tests with the non-linear time complexity. Significant improvements were accomplished thanks to the byte oriented data storage, word-oriented data processing, the use of look up tables and other smart optimisations.

With the exception of the Spectral test, where the optimisations will be aimed at the parameter n, we achieved excellent speedup results for the three most time-consuming tests. The optimized Linear complexity test is 27.5x faster than original implementation for $m = 500$ and the speedup improves towards 64.5x for $m = 5000$. The speedup of the Non-overlapping template matching test is in the interval between 5.3x and 483x, where for the most usual parameters $m = 9$ and $m = 10$ the speedup of 407x is outstanding. Improvements of the Serial test relate to the use of a single pass calculation of block frequencies (instead of three independent calculations) and bring the speedup improvements in the range between 12x and 155x, in the dependence on m. The speedup is 155x for the default value of $m = 16$. Due to above mentioned improvements, we were able to achieve the overall speedup of about 30 times (compared to the NIST STS implementation). This means that the typical test setups that require hours to run can be executed within dozens of minutes now.

Acknowledgement. Credits go to Vashek Matyas for his crucial involvement in organizing the research and editing this paper. The first author was supported by the Ministry of Education, Youth, and Sport project CZ.1.07/2.3.00/30.0037 – Employment of Best Young Scientists for International Cooperation Empowerment. The second author was supported by the Czech Science Foundation, project GAP202/11/0422.

References

1. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S.: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST Special Publication 800-22 (May 2001), `http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22b.pdf`
2. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S.: A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST Special Publication 800-22rev1 (August 2008), `http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1.pdf`
3. Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S.: A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications, Version STS-2.1, NIST Special Publication 800-22rev1a (April 2010), `http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf`
4. Marsaglia, G.: The Marsaglia random number CDROM including the DIEHARD battery of tests of randomness (1996), `http://stat.fsu.edu/pub/diehard`
5. Brown, R.G.: Dieharder: A Random Number Test Suite, Version 3.31.1 (2004)
6. L'Ecuyer, P., Simard, R.: TestU01: A C library for empirical testing of random number generators. ACM Trans. Math. Softw. 33 (2007)
7. Walker, J.: ENT – A pseudorandom number sequence test program (1993), `http://www.fourmilab.ch/random/`
8. Caelli, W., et al.: Crypt X Package Documentation, Information Security Research Centre and School of Mathematics, Queensland University of Technology (1992), Crypt-X: `http://www.isrc.qut.edu.au/resource/cryptx/`
9. Suciu, A., Marton, K., Nagy, I., Pinca, I.: Byte-oriented efficient implementation of the NIST statistical test suite. In: Proceedings of the 2010 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR 2010), vol. 2, pp. 1–6. IEEE Computer Society, Washington, DC (2010)
10. Sadique Uz Zaman, J.K.M., Ghosh, R.: Review on fifteen Statistical Tests proposed by NIST. Journal of Theoretical Physics and Cryptography 1 (November 2012)
11. Sýs, M., Říha, Z.: Optimised implementation of NIST STS (2014), `https://github.com/sysox/NIST-STS-optimised`