Algorithm 970: Optimizing the NIST Statistical Test Suite and the Berlekamp-Massey Algorithm

MAREK SÝS, ZDENĚK ŘÍHA, and VASHEK MATYÁŠ,
Masaryk University, Brno, Czech Republic

The NIST Statistical Test Suite (NIST STS) is one of the most popular tools for the analysis of randomness. This test battery is widely used, but its implementation is quite inefficient. A complete randomness analysis using the NIST STS can take hours on a standard computer when the tested data volume is on the order of GB. We improved the most time-consuming test (Linear Complexity) from the previous most efficient implementation of the NIST STS. We also optimized other tests and achieved an overall speedup of 50.6$\times$ compared with the reference implementation. This means that 20MB of data can be tested within a minute using our new optimized version of the NIST STS. To speed up the Linear Complexity test, we proposed a new version of the Berlekamp-Massey algorithm that computes only the linear complexity of a sequence. This new variant does not construct a linear feedback shift register and is approximately 187$\times$ faster than the original NIST implementation of the Berlekamp-Massey algorithm.

Categories and Subject Descriptors: G.2.1 [**Mathematical Software**]: Statistical Software, Mathematical Software Performance; I.0.3 [**Cryptography**]: Cryptanalysis and other Attacks

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Berlekamp-Massey algorithm, NIST STS, statistical randomness testing

ACM Reference Format:
Marek Sýs, Zdeněk Říha, and Vashek Matyáš. 2016. Algorithm 970: Optimizing the NIST statistical test suite and the Berlekamp-Massey algorithm. ACM Trans. Math. Softw. 43, 3, Article 27 (December 2016), 11 pages.
DOI: http://dx.doi.org/10.1145/2988228

1. INTRODUCTION

Randomness is related to many areas of computer science, particularly cryptography, in which it plays a fundamental role. Randomness is typically assessed using empirical tests. Each test examines data by looking at a specific feature (the number of ones, *m*-bit blocks, etc.). Multiple such tests are usually grouped into test batteries (also called test suites) to provide more complex randomness analyses. One of the most popular test batteries is the NIST Statistical Test Suite (NIST STS). The NIST STS is frequently used in the analysis and certification of generators of (pseudo)random sequences. It was used in the assessment of AES candidates [Soto and Bassham 2000], and it also plays an important role in the evaluation of outputs of cryptographic primitives such as hash functions or symmetric encryption functions.

Marek Sýs was originally supported by the Ministry of Education, Youth, and Sport through project CZ.1.07/2.3.00/30.0037 – Employment of Best Young Scientists for International Cooperation Empowerment. Zdeněk Říha was supported by the Czech Science Foundation through project GA16-08565S. Vashek Matyáš was supported by the Czech Science Foundation through project GA16-08565S.
Authors' addresses: Centre for Research on Cryptography and Security, Faculty of Informatics, Masaryk University, Botanická 68a, 602 00 Brno, Czech Republic; email: {syso,zriha,matyas}@fi.muni.cz.

DOI: http://dx.doi.org/10.1145/2988228

Although the NIST STS is widely used, its implementation is very inefficient. In practice, the amounts of analyzed data are usually on the order of GB. This makes the NIST STS quite impractical because several NIST tests typically take hours on a standard computer for such data volumes. Recently, we introduced a preliminary time- and memory-efficient implementation of the NIST STS in Sýs and Říha [2014]. However, this implementation was still insufficient for certain applications/hardware that require on-the-fly randomness analysis.

The goal of the work presented here was to improve on our implementation of the NIST STS from Sýs and Říha [2014]. We focus mainly on the re-implementation of the Linear Complexity test, which is one of the most time-consuming tests in the NIST STS. The Linear Complexity test uses the Berlekamp-Massey algorithm to construct the smallest Linear Feedback Shift Register (LFSR) that can generate a given binary sequence.

Many applications, including the Linear Complexity test, require only the length of the smallest LFSR rather than the LFSR itself. In this article, we introduce a new variant of the Berlekamp-Massey algorithm that is more efficient than the standard Berlekamp-Massey algorithm [Menezes et al. 2010] because it computes the linear complexity without constructing the corresponding LFSR. We also present a simple description and interpretation of the Berlekamp-Massey algorithm.

This article is organized as follows: Section 2 provides a brief overview of the NIST tests and their alternative implementations and a comparison of their performance. Section 3 briefly describes the ideas underlying our optimizations of simple NIST STS tests. Section 4 discusses the new variant of the Berlekamp-Massey algorithm. Section 5 summarizes the new optimizations and the results of performance testing.

2. PREVIOUS WORK AND THE NIST STS

The NIST STS package implements tests of randomness as defined by the relevant NIST document [Rukhin et al. 2001] and its revisions [Bassham et al. 2008, 2010]. All of the NIST tests are parametrized by a parameter n—the length (in bits) of the bitstream. Although all of the tests are designed to detect deviations from randomness for an entire bitstream, several tests can also detect local nonrandomness. These tests are parametrized by a second parameter, m or M [Bassham et al. 2010]. Tests parametrized by m are designed to detect the presence of too many m-bit patterns (words) in a sequence. Tests parametrized by the second parameter M examine the distribution of a specific feature across n/M parts (of an equal size of M bits) of a given bitstream.

The latest version of the package (NIST STS 2.1.2, released on 9 July 2014) implements 15 tests of randomness as defined in Bassham et al. [2010]. Although individual NIST tests are also implemented in other test batteries (Diehard [Marsaglia 1995], TestU01 [L'Ecuyer and Simard 2007]), to the best of our knowledge, only two re-implementations of the entire suite exist [Sýs and Říha 2014; Suciu et al. 2010].

TestU01 [L'Ecuyer and Simard 2007] is a software library (not an executable tool) that implements a large number of random number generators and statistical tests of randomness. It groups some of these tests into batteries (e.g., Crush, Rabbit, and Alphabit), but the NIST STS is not provided as a battery. The documentation provides readers with some hints regarding which TestU01 functions could be used to run the NIST STS tests. Although TestU01 is very flexible and general, certain NIST STS tests still are not implemented in TestU01 at all (e.g., the Overlapping Template Matching test), and some tests cannot be run with the parameters required for the NIST STS (e.g., the test for the Longest Run of Ones in a Block).

Dieharder [Brown et al. 2013] is a library and tool for testing the statistical properties of (pseudo)random number generators. It implements the original Diehard tests [Marsaglia 1995], several NIST STS tests, and a number of other tests. Although the

Table I. Runtimes (for 20MB of Data) of the Original Implementation of the NIST STS and the Speedups Achieved by Alternative Implementations

Test	m, M	NIST 2.1 (ms)	O-NIST Speedup	TestU01 Speedup	B-NIST Speedup	Dieharder Speedup
Frequency (Monobit)		137	**11.91**	1.81	9.82	0.26§
Frequency Within a Block	128	58	5.20	0.38	**9.63**	
Runs		1,138	**36.77**	1.68†	5.84	0.27
Longest Run of Ones in a Block		651	**21.16**	0.93‡	6.51	
Binary Matrix Rank		3,588	**12.7**3	4.53	7.91	
Spectral		18,700	0.98	*		
Nonoverlapping Template	9	125,862	**407.12**	5.62†	3.13	
Overlapping Template	9	1,355	3.35	*	**15.15**	
Maurer's Universal		2,758	**18.22**	6.75‡	12.8	
Linear Complexity	5,000	1,115,562	**64.46**	2.14‡	3.92	
Serial	9	28 811	**76.92**	56.56	48.73	3.54§, ¶
Approximate Entropy	8	16,406	52.83	**77.26**	54.16	
Cumulative Sums		287	**31.74**	*	3.31	
Random Excursions		548	1.09	*	**1.26**	
Random Excursions Variant		1,501	4.25	*	**6.09**	
Total		*1,317,357*	29.33			

*TestU01 does not implement this test.
†The parameters in TestU01 do not match the parameters in the NIST STS.
‡The parameters in TestU01 cannot be set exactly to the required values.
§The time also includes the time required for file reading.
¶The test is performed using many values of parameter m simultaneously.

(claimed) intent of Dieharder is to eventually implement all NIST STS tests, after more than 10 years of development, only three tests of the 15 have been implemented.

Neither TestU01 nor Dieharder offers a full implementation of the NIST statistical tests with the parameter flexibility of the NIST specification, and both libraries work with 32-bit integers instead of binary sequences.

In Suciu et al. [2010], the authors changed the original representation (in which each byte stores a single bit) of the analyzed bitstream to the natural one (with 8 bits per byte). This byte-oriented representation of the NIST STS (B-NIST STS) allows most of the tests to be sped up by means of precomputation, namely, through lookup tables (LUTs). A more efficient implementation of the NIST STS known as O-NIST (short for Optimized NIST STS) was described later in Sýs and Říha [2014]. Although some of the optimizations were novel, several of the ideas and principles applied in Sýs and Říha [2014] were similar to those used in Suciu et al. [2010] (LUTs, byte-byte representation); however, the implementation of Suciu et al. [2010] is not publicly available.

The performances of major implementations of the NIST tests are summarized in Table I. The table reports the runtime of the original implementation of each test in the NIST STS and the speedups achieved by alternative implementations when processing 20MB of pseudorandom data ($n = 167, 772, 160$) on a modern notebook. The performance of a test parametrized by a second parameter (m or M) depends on the particular value of that parameter. The values of the second parameter (m or M) were selected to allow comparison of the achieved speedups with Sýs and Říha [2014] and Suciu et al. [2010].

3. NEW IMPROVEMENTS

Our current work extends the initial implementation described in Sýs and Říha [2014]. All optimizations in this new implementation, O2-NIST, are based on three concepts:

LUTs, the fast extraction of a histogram of m-bit words, and the "parallelization" of bit-bit operations using word-word bitwise operations.

3.1. Types of Statistical Tests

Different approaches are used for different types of tests, depending on their complexity. The NIST tests can be divided (according to their complexity and the optimizations used) into three classes, as follows:

(1) The fastest tests, which process each bit of a bitstream once—*Frequency*, *Block Frequency*, *Runs*, *Longest Run*, *Cumulative Sums*, *Random Excursion*, and *Random Excursion Variant*.

To speed up these tests, we can use LUTs that have been precomputed for blocks larger than 8 bits. Because these tests are already very fast, the extraction of the blocks must be as fast as possible for other operations in the tests (the testing and summation of corresponding table values) to not be hindered by extraction slowdown. Thus, the block size should be a multiple of 8. Only values of 8 and 16 are reasonable because of the size of the corresponding LUTs (2^k), therefore we added LUTs for 16-bit blocks to tests in this class.

New LUTs were incorporated into the following tests: Frequency, Block Frequency, Longest Run, and Cumulative Sums. As a result, the performances of these tests were nearly doubled. In the Runs test, we significantly improved the algorithm such that it now runs $3\times$ faster. The idea underlying this improvement is based on the calculation of runs (sequences of either ones and zeros). We shift the sequence by 1 bit and then XOR it with the original sequence. This transforms the calculation of the run count into a computation of the Hamming weight of the resulting sequence, thereby enabling the use of LUTs, as in the tests mentioned previously.

(2) Fast tests, which process m-bit blocks—*Non-overlapping Template Matching*, *Overlapping Template Matching*, *Universal*, *Serial*, and *Approximate Entropy*. The runtimes of these tests depend on the value of m because each bit of the m-bit block is compared with a given pattern. We designed a new function that calculates a histogram of the m-bit blocks within a delimited sequence in an efficient way. This function reuses previously calculated values in overlapping sequences. This histogram function has proven useful in four tests: Approximate Entropy, Serial, Overlapping Template Matching, and Nonoverlapping Template Matching. The use of the histogram function speeds up these tests by nearly $3\times$. Now, the Nonoverlapping Template Matching test (the second most time-consuming test) runs more than $1000\times$ faster than in the original code.

(3) Slow and complicated tests—*Linear Complexity*, *Spectral*, and *Rank*. These tests use quadratic algorithms (Linear Complexity, Rank) or a subquadratic algorithm (Spectral).

The Spectral test is the only test that was not optimized in O-NIST. This test was also not implemented or tested in B-NIST. The Spectral test uses the Fast Fourier Transform (FFT), and therefore, its runtime is determined primarily by the prime factors of n (the length of the bitstream) and not by n itself. In O2-NIST, we incorporated the FFTW ("Fastest Fourier Transform in the West") to perform the Spectral test because this transform is fast for values of n with larger prime factors.

3.2. Improvements of the Interpretation of Results

In addition to the performance of the tests, we also improved the interpretation of their results. The most common way to analyze a large amount of data is to divide the data into several (k) sequences (typically, $k = 1{,}000$) and to apply each test to each sequence individually. In this case, the result of each NIST test is a set of p-values on the interval $(0, 1]$ (one p-value per analyzed sequence). For random data, the p-values should be uniformly distributed on the given interval. The NIST STS analyzes whether a computed set of p-values follows the given distribution. For each test, the NIST STS computes the following:

—the proportion of passing sequences—the proportion of sequences with p-values smaller than the significance level ($\alpha = 0.01$), and
—the p-value—computed by means of the statistical χ^2 test to analyze the uniformity of the p-values computed by the test.

The NIST STS checks whether the results (proportion and p-value) fall within the acceptable intervals ($0.99 \pm 3\sqrt{0.0099/k}$ and $(0.01,1]$, respectively) and marks suspicious values that fall outside of these intervals. In Sýs et al. [2015], we realized that the first interval is not accurate. Accordingly, we added the possibility to run the NIST STS post-processing using the new and more accurate interval. We also extended the analysis of the uniformity of the p-values. The χ^2 test (as implemented in the original NIST STS) divides the interval $(0, 1]$ into 10 intervals, $(0, 0.1], \ldots, (0.9, 1.0]$, and statistically checks whether the frequencies of the p-values in each interval are equal to $k/10$. This approach is not able to detect the nonuniformity of p-values within subintervals. Therefore, we added the Kolmogorov-Smirnov (KS) statistical test (also used by other batteries—TestU01, Dieharder) to the NIST STS because it is able to detect such defects in the uniformity of the p-values.

It should be noted that the p-values of a test are computed using an approximation of the distribution of the test statistic (typically normal or χ^2). This causes the p-values not to be (perfectly) uniformly distributed on the interval $(0, 1]$, which could lead (in extreme cases) to false rejection by the uniformity tests—either the original χ^2 test or the newly added KS test. An interesting discussion about uniformity tests can be found in L'Ecuyer et al. [2002] and L'Ecuyer and Simard [2005].

4. BERLEKAMP-MASSEY ALGORITHM

In this section, we discuss a new variant of the Berlekamp-Massey Algorithm (BMA) over the binary field. The standard BMA constructs the smallest LFSR that generates a given binary sequence $s^n = \{s_0, s_1, \ldots, s_{n-1}\}$. An LFSR that is represented as a vector of binary values $c_0, c_1, c_2, \ldots, c_L \in \{0, 1\}$ generates a sequence s^n if and only if $s^j = \sum_{i=1}^{L} c_i s_{j-i}$ for all $L \leq j \leq n-1$. The length L of the smallest LFSR that generates the sequence s^n represents its linear complexity ($L(s^n)$), which is used by the Linear Complexity test of the NIST STS.

The BMA is an iterative algorithm, and it constructs the smallest LFSR that generates a subsequence $s^N = \{s_0, s_1, \ldots, s_{N-1}\}$ of s^n ($N \leq n$) in its N-th ($N \in \{0, 1, \ldots, n-1\}$) iteration. The BMA checks whether the LFSR that generates s^N also generates the longer sequence s^{N+1}. To test this property, the BMA computes the discrepancy d_N between the $N+1$-st bit generated by the current LFSR and the $N+1$-st bit s_N of the sequence. This discrepancy for the LFSR defined by $c_0 = 1, c_1, \ldots, c_L$ is computed as follows:

$$d_N = s_N + \sum_{j=1}^{L} c_j s_{N-j} \pmod 2. \qquad (1)$$

If d_N is equal to zero, then the current LFSR also generates s_{N+1}. If d is equal to one, then the LFSR is updated using the latest LFSR (different from the current LFSR) computed up to the N-th iteration. The BMA resembles Euclid's algorithm, and it uses two LFSRs: C and B. The LFSR denoted by C represents the current minimal LFSR that generates s^N, and B represents the previous minimal LFSR for s^m (where m is the largest integer $<N$ for which the smallest LFSR for s^m is different from C). For a nonzero discrepancy, the new LFSR C' is computed using C and B, while the new B' is set to the old C. To describe operations (shift and sum) on the arrays used to compute C', the LFSRs are usually represented as polynomials $C(x)$ and $B(x)$ over the binary field. In polynomial notation, the standard BMA can be described as shown in Algorithm 1. The BMA returns the linear complexity L of the sequence s^n. During the computation, the BMA iteratively constructs the LFSRs that generate s^N for $N = \{0, 1, \dots, n-1\}$. Our new variant of the BMA omits the construction of the LFSRs.

ALGORITHM 1: Standard Berlekamp-Massey Algorithm for the Binary Field.

```
Data: Sequence $s^n = \{s_0, s_1, \dots, s_{n-1}\}$ of length $n$ over the binary field
Result: Length $L$ of the shortest LFSR that generates $s^n$
Initialize $C(x) \leftarrow 1$, $L \leftarrow 0$, $m \leftarrow -1$, $B(x) \leftarrow 1$;
for $N \leftarrow 0$ to $n-1$ do
    $d \leftarrow \sum_{j=0}^{L} c_j s_{N-j}$;
    if $d \neq 0$ then
        $T(x) \leftarrow C(x)$, $C(x) \leftarrow C(x) - x^{N-m} \cdot B(x)$;
        if $L \leq N/2$ then
            $L \leftarrow N+1-L$, $m \leftarrow N$, $B(x) \leftarrow T(x)$;
        end
    end
end
return $L$;
```

4.1. New Variant of the BMA

The new BMA does not construct LFSRs. It works with vectors of discrepancies D^C and D^B corresponding to the LFSRs C and B rather than with C and B themselves. The standard BMA uses the polynomial representation of the LFSRs C and B because shift and sum operations of arrays are used to update C. The new variant also uses these two operations. Whereas in the standard BMA, the size of C increases (right shift $\cdot x^{N-m}$), in the new BMA, the size of the vector of discrepancies (D^C) decreases. Using the polynomial representation of the discrepancies $D^C(x) = \sum_{i=0}^{L} d_i^C x^i$ and $D^B(x) = \sum_{i=0}^{k} d_i^B x^i$ (for some k) and with the left shift operation $<<$ defined for polynomials as $(a_0 + a_1 x + \dots) << 1 = a_1 + a_2 x + \dots$, the new variant of the BMA can be described as shown in Algorithm 2.

To prove the correctness of the new BMA algorithm, it suffices to show that the discrepancy d that is tested in the standard BMA is equal to the discrepancy $d = d_0^C$ that is tested in the new BMA for all iterations $0 \leq N < n$. We wish to show that at the beginning of each iteration ($N \in \{0, 1, \dots, n-1\}$), the vectors of discrepancies D^C and D^B represent the discrepancies corresponding to s^n and the LFSRs C and B.

We wish to show that in the N-th iteration, the discrepancies $D^C(x) = \sum_{i=0}^{n-N} d_i^C x^i$ and $D^B(x) = \sum_{i=0}^{n-m} d_i^B x^i$ represent the discrepancies corresponding to the LFSRs $C(x) = \sum_{i=0}^{L} c_i x^i$ and $B(x) = \sum_{i=0}^{k} b_i x^i$ (for some k). In fact, we wish to show the following equalities:

$$d_i^C = \sum_{j=0}^{L} c_j s_{i-j}, \quad d_i^B = \sum_{j=0}^{k} b_j s_{i-j}$$

for $i \in \{0, \dots, n-N\}$ or $i \in \{0, \dots, n-m\}$.

ALGORITHM 2: New Variant of the Berlekamp-Massey Algorithm for the Binary Field.

```
Data: Sequence $s^n = \{s_0, s_1, \ldots, s_{n-1}\}$ of length $n$ over the binary field
Result: Length of the shortest LFSR that generates $s^n$
Initialize $D^C(x) \leftarrow s^n(x)$, $L \leftarrow 0$ , $D^B(x) \leftarrow s^n(x)$;
for $N \leftarrow 0$ to $n - 1$ do
    $d = d_0^C$;
    $D^C(x) \leftarrow D^C(x) << 1$;
    if $d = 1$ then
        $T(x) \leftarrow D^C(x)$, $D^C(x) \leftarrow D^C(x) - D^B(x)$;
        if $L \leq N/2$ then
            $L \leftarrow N + 1 - L$, $B(x) \leftarrow T(x)$;
        end
    end
end
return $L$;
```

For the initialization step (iteration $N = 0$) of the standard BMA, we have the following LFSRs: $C(x) = B(x) = c_0 = b_0 = 1$. This immediately implies that $D^C(x) = D^B(x) = s^n(x)$, and therefore, the initial discrepancies $D^C(x)$ and $D^B(x)$ correspond to the initial LFSRs. It is also easy to see the correspondence between the LFSRs $C(x)$ and $B(x)$ and the discrepancies $D^C(x)$ and $D^B(x)$ for the case of $d_0^C = 0$ because the LFSRs C and B do not change (in the case of the standard BMA), nor do their corresponding discrepancies (D^C and D^B), and therefore, to obtain the next vector of discrepancies, D^C must be shifted ($D^C = D^C << 1$). Now, it suffices to show correctness for the case of $d_0^C = 1$. In the standard BMA, the new $C(x)$ is computed as $C(x) = C(x) + x^{N-m}B(x)$. The value $N - m$ represents the shift of $B(x)$ according to $C(x)$. Accordingly, $D^C(x)$ should be shifted to the left by $N - m$ positions, according to $D^B(x)$. This shift is performed by applying $N - m$ single left shifts. The correspondence between the standard BMA and the new BMA can be seen from the fact that in the standard BMA, the minimal value of $N - m$ is 1 and it is increased by one in each iteration. This corresponds to the single left shift ($D^C(x) = D^C(x) << 1$) applied in each iteration.

4.2. Implementation Details

The new variant of the BMA improves upon the standard BMA because the algorithm does not construct LFSRs (only the discrepancies are computed). The new BMA can be further simplified and optimized. To speed up the operations $<<$ and $+$ with polynomials over the binary field, we represent these polynomials as word (32-bit or 64-bit) arrays and implement three functions—*copy()*, *xor()*, and *lshift()*—that operate on word arrays. These functions are used to

—copy bit arrays—to copy $D^C(x)$ to $T(x)$ and to copy $T(x)$ to $D^B(x)$,
—XOR bit arrays—to compute $D^C(x) + D^B(x)$ (or, equivalently, $D^C(x) - D^B(x)$), and
—shift a bit array by (i) bits to the left ($D^C(x) << i$).

The function $xor(a, b, size)$ performs the XOR operation on word arrays a and b of size $size$, where b is added (XORed) to a using the bitwise *XOR* operation for words. The function $lshift(a, shift, size)$ performs a shift of array a by *shift* bits to the left. To speed up these operations, the word size should be as large as possible. We implemented variants of these functions for integer types of 32 and 64 bits.

In Algorithm 2, we do not shift D^C by one in each iteration. Instead, we find the first nonzero d_i^C coefficient and shift D^C by i positions, that is, $D^C = D^C << i$. To find the position of the first nonzero bit in a word, we use the *MultiplyDeBruijnBitPosition* function published on the webpage [Anderson 2005]. This function is very fast because it uses one integer multiplication, one bitwise shift, and a LUT.

5. IMPLEMENTATION AND TESTING

The O-NIST implementation allows the correctness of all values output by the tests to be verified. Therefore, we incorporated the new versions of the NIST tests into O-NIST to create O2-NIST. Compared with O-NIST, we re-implemented 11 of the 15 NIST tests.

After creating the implementation, we first focused on the verification of the results. We used a setting of O2-NIST that checks the correctness of the test results. We performed a series of tests in which all tests were applied to bitstreams of many different lengths and with different parameters. We tested the implementation using pseudorandom bit sequences and sequences of special values, such as zeros, ones and alternating ones and zeros.

We performed tests for all bitstream lengths between 1 and 1,000,000 bits and for randomly chosen bitstream lengths between 1 and 800,000,000 bits computed as $n_{i+1} = n_i * 10 + rand()\%8$ to catch possible errors caused by bitstream lengths that are not a multiple of 8.

In a few configurations, our implementation produces a result even when the original implementation is not able to compute a result (e.g., the Random Excursion test is limited in the number of cycles); additionally, in some situations, our implementation does not support unusual parameters, whereas 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 were consistent.

5.1. Performance Testing

We measured both the number of CPU cycles (using the *RDTSC* instruction) and the time consumed in milliseconds (using the *GetTickCount* function). Because the number of CPU cycles is a more accurate measure, we used these values in our calculations; however, for presentation purposes, we converted the number of CPU cycles into milliseconds. We performed all performance tests 15 times, and we used the minimum value to eliminate the noise introduced by Operating System (OS) scheduling. The source code, including that for result verification and 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 performance testing. The speed improvement was measured on a Windows 10 Fujitsu S792 notebook equipped with an Intel Core i7 processor with two cores (no multithreading was used) running at 3GHz and 8GB of memory. The code was compiled using MS Visual Studio 2015. We produced a x64 binary in Release mode using the default parameters. Although the speed measurements were performed using a 64-bit binary on a 64-bit operating system, our implementation also compiles on a 32-bit system with similar performance. The source code relies on the assumptions that the size of an *int* is at least 32 bits and the processor works in a little-endian architecture.

First, we measured the performance of the new BMA. We tested several variants of the BMA combining our functions *copy()*, *lshift()*, and *xor()* and the library functions *memset()* and *memcpy()*. The fastest version of the BMA was found to be that using the functions *memset()*, *memcpy()*, *xor()*, and *lshift()*. The algorithm now runs 3× faster than in O-NIST, which means that it is 187× faster than in the original NIST STS implementation.

Because the Linear Complexity test depends on a second parameter M, we also measured the performance of the test for the allowed range of the parameter M. The runtimes of the new and original Linear Complexity tests are summarized in Figure 1.

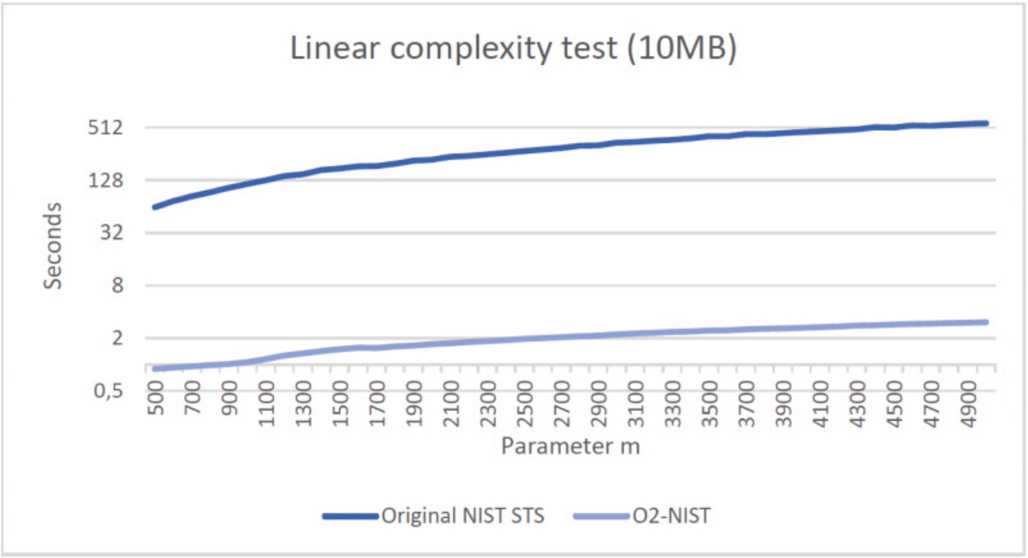

Fig. 1. Runtimes of the Linear Complexity test (note the logarithmic scale of the y axis).

Next, we measured the performance of the FFTW and the performance of the original FFT as implemented in the NIST STS. The performances of the FFTW and FFT strongly depend on the largest factor of n. The results of the performance measurements were highly variable. For n values with very small factors (such as 2 or 3), the original FFT is slightly faster. In all other cases, the FFTW is significantly faster than the FFT. To illustrate this behavior, we ran the test for $n = 2^{20} = 1,048,576$, $n = 10^6 =$ 1,000,000 (with small factors 2 and 5), and $n =$ 1,000,003 (a prime). For $n = 2^{20}$, the original FFT is slightly faster (approximately 1.12× faster; 110ms vs. 125ms); for $n = 10^6$, the FFTW is slightly faster (approximately 1.72× faster; 188ms vs. 109ms); and for $n =$ 1,000,003, the FFTW is 1,514× faster than the original FFT (923,312ms vs. 609ms). At compile time, the user can select either the original FFT implementation or the FFTW. In our performance test, we used $n = 5 \cdot 2^{25}$ (20MB), which has small factors, and therefore, we used the original implementation.

The performance of the battery is summarized in Table II. The overall speedup of O2-NIST is 50.6×. In practice, this means that 20MB of data can be tested in 26 seconds instead of 22 minutes.

6. CONCLUSION

We improved O-NIST [Sýs and Říha 2014] with a focus on the most time-consuming tests—the Linear Complexity and Nonoverlapping Template Matching tests. We proposed a new variant of the BMA that computes only the linear complexity of a sequence without constructing the corresponding LFSR. Using this new variant, the Linear Complexity test was sped up by 3×, making it now 187× faster than the original NIST implementation.

Using better lookup tables and including one trick used in the calculation of the number of runs in the Runs test, we improved the performance of five simple statistical tests. Four tests based on the calculation of m-bit block frequencies were significantly sped up using an efficient implementation of histograms.

The new O2-NIST implementation offers an overall speedup of 50.6× compared with the original NIST STS. In addition to these performance improvements, we also

Table II. Runtimes (for 20MB of Data) of the O2-NIST Implementation

Test	m, M	NIST 2.1.2 (ms)	O2-NIST (ms)	O2-NIST Speedup
Frequency (Monobit)		137	6	22.3
Frequency Within a Block	128	58	11	5.2
Runs		1,183	14	86.4
Longest Run of Ones in a Block		651	28	23.1
Binary Matrix Rank		3,588	307	11.7
Spectral		18,700	18,738	1.0
Nonoverlapping Template	9	125,862	107	1,178.7
Overlapping Template	9	1,355	61	22.1
Maurer's Universal		2,758	154	17.9
Linear Complexity	5,000	1,115,562	5,950	187.5
Serial	9	28,811	106	271.4
Approximate Entropy	8	16,406	106	154.5
Cumulative Sums		287	24	11.9
Random Excursions		548	212	2.6
Random Excursions Variant		1,501	211	7.1
Total		1,317,407	26,038	50.6

improved the interpretation of the results of the NIST STS. We incorporated the standard Kolmogorov-Smirnov test, which analyzes the uniformity of the p-values from different points of view (compared with the original χ^2 test), into the battery. We also improved the computation of the acceptable region for the "proportion of passing sequences" measure. Our new implementation is currently available as Sýs and Říha [2016].

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable comments, which led to a significant improvement of our paper.

REFERENCES

Sean E. Anderson. 2005. Bit Twiddling Hacks. Retrieved from http://graphics.stanford.edu/~seander/bithacks.html#IntegerLogDeBruijn.

Lawrence E. Bassham, III, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, Nathanael Alan Heckert, James F. Dray, and San Vo. 2008. *SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Technical Report. NIST, Gaithersburg, MD. Retrieved from http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1.pdf.

Lawrence E. Bassham, III, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, Nathanael Alan Heckert, James F. Dray, and San Vo. 2010. *SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Technical Report. NIST, Gaithersburg, MD. Retrieved from http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf.

Robert G. Brown, Dirk Eddelbuettel, and David Bauer. 2013. Dieharder: A Random Number Test Suite. Retrieved from http://www.phy.duke.edu/~rgb/General/dieharder.php.

Pierre L'Ecuyer and Richard Simard. 2007. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.* 33, 4, Article 22 (Aug. 2007). DOI:http://dx.doi.org/10.1145/1268776.1268777.

Pierre L 'Ecuyer and Richard Simard. 2005. TestU01: A software library in ANSI C for empirical testing of random number generators: User's guide, detailed version. Département d'Informatique et 'de Recherche Opérationnelle Université de Montréal (2005). http://simul.iro.umontreal.ca/testu01/tu01.html.

Pierre L'Ecuyer, Richard Simard, and Stefan Wegenkittl. 2002. Sparse serial tests of uniformity for random number generators. *SIAM J. Sci. Comput.* 24, 2 (2002), 652–668.

George Marsaglia. 1995. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. Department of Statistics, Florida State University, Tallahassee, FL, USA.

Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. 2010. *Handbook of Applied Cryptography*. CRC Press.

Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. 2001. A statistical test suite for random and pseudorandom number generators for cryptographic applications, NIST. Retrieved from http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22b.pdf.

Juan Soto and Lawrence Bassham. 2000. *Randomness Testing of the Advanced Encryption Standard Finalist Candidates*. Technical Report. Defense Technical Information Center. Retrieved from http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA393385.

Alin Suciu, Kinga Marton, Iszabella Nagy, and Ioana Pinca. 2010. Byte-oriented efficient implementation of the NIST statistical test suite. In *Proceedings of the 2010 IEEE International Conference on Automation, Quality and Testing, Robotics* 2 (2010), 1–6. DOI:http://dx.doi.org/10.1109/AQTR.2010.5520837.

Marek Sýs and Zdeněk Říha. 2014. Faster randomness testing with the NIST statistical test suite. In *Security, Privacy, and Applied Cryptography Engineering*, Rajat Subhra Chakraborty, Vashek Matyas, and Patrick Schaumont (Eds.). Lecture Notes in Computer Science, Vol. 8804. Springer International Publishing, 272–284. DOI:http://dx.doi.org/10.1007/978-3-319-12060-7_18.

Marek Sýs and Zdeněk Říha. 2014-2016. Optimised implementation of NIST STS. Retrieved from http://crcs.cz/projects/sts.

Marek Sýs, Zdeněk Říha, Vashek Matyáš, Kinga Márton, and Alin Suciu. 2015. On the interpretation of results from the NIST statistical test suite. *Romanian J. Inf. Sci. Technol.* 18, 1 (2015).

Received November 2014; revised May 2016; accepted August 2016