Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2017) 25: 4818 – 4828

doi:10.3906/elk-1703-361

A random number generator for lightweight authentication protocols: xorshiftR+

Umut Can ÇABUK, Ömer AYDIN*, Gökhan DALKILIÇ
Department of Computer Engineering, Faculty of Engineering, Dokuz Eylül University, İzmir, Turkey

Received: 30.03.2017 • **Accepted/Published Online:** 05.09.2017 • **Final Version:** 03.12.2017

Abstract: This paper presents the results of research that aims to find a suitable, reliable, and lightweight pseudorandom number generator for constrained devices used in the Internet of things. Within the study, three reduced versions of the xorshift+ generator are built. They are tested using the TestU01 suite as well as the NIST suite to measure their ability to produce randomness and performance values along with some other existing generators. The best of our reduced variations according to our tests, called the xorshiftR+, demonstrated great suitability for lightweight devices considering its randomness, performance, and resource usage.

Key words: TestU01, xorshift, lightweight cryptography, Internet of things

1. Introduction

The rapidly emerging concept of the Internet of things (IoT) brings new approaches to everyday problems as well as industrial applications. These approaches rely on bundles of cheap, efficient, and dedicated networked devices that work and communicate continuously. These so-called lightweight devices of the IoT have limited power, space, and computation resources; hence, there is a huge need for developing suitable security protocols and methodologies tailored for those. Furthermore, most of the contemporary security protocols are not optimized for lightweight environments and require more sources than IoT devices may efficiently provide. For example, almost all authentication protocols use random number generators to function and improvements on these may boost the usage of IoT devices in areas where security concerns exist. In this paper, we propose a lightweight random number generator recommendation for security applications in constrained devices.

While working with random number generators (RNGs), the most important thing to take into consideration is that there is no perfect generator that fits all conditions [1]. This is mostly because true randomness is a nondeterministic process, which cannot be synthesized using mathematical methods in a software environment, and this leads to the concept of pseudorandomness. On the other hand, even by using hardware sources, it is practically very hard to produce a series of numbers that have the anticipated characteristics of true randomness. In any case, the quality of a generator is correlated with its proximity to true randomness and its computational requirements. True or high-degree randomness can be very expensive or inefficient or just unnecessary in some cases. Hence, the best option should be chosen according to the needs of the application under development.

The main work that leads to this study is to develop a radio frequency identification (RFID) authentication protocol, which runs on RFID tags without intervention of the reader or another computer, for particularly, but not limited to, a wireless sensing and identification platform (WISP). A very accurate use case scenario

*Correspondence: omer.aydin@deu.edu.tr

was given in another study [2]. In this context, there is a need for a lightweight and reliable RNG. Besides, there are hardware sensors like thermometers, accelerometers, etc. on the WISP and many other types of tags that may be used to feed or seed the RNG algorithms. However, these hardware inputs cannot be used solely to produce random numbers, since in certain stable environmental conditions, the outcomes will usually follow certain patterns. Thus, we cannot trust hardware sources, but a pseudo-RNG seeded with hardware sources or external inputs may provide satisfactory results if suitable RNG algorithms are used.

2. Related works

RNGs were mostly studied from the point of view of their proximity to true randomness and reliability until the last decade since they were in use in relatively powerful computers. When smart mobile devices gained popularity and eventually the IoT concept was introduced, the idea of moving RNGs to mobile devices with (sometimes very) limited capabilities brought performance issues into sight. For example, the standardization document of randomness recommendations for security, RFC 1750, suggests a data encryption standard (DES), cryptographic hash functions, and some other (not so simple) methods to provide randomness; however, there is no concern for lightweight devices [3].

Before proceeding to the RNG algorithms, it would be helpful to take a glance at the review criteria for these RNGs. There are several popular randomness tests (or more accurately, test suites) used to examine generator functions that are supposed to have random outcomes. The ones we refer to are the Diehard battery of Marsaglia [4]; TestU01 suite of L'ecuyer and Simard [5], which consists of 6 test batteries; and the NIST test suite having 15 tests [6]. The more recent publications of NIST, namely SP800-90 (a, b, c), are also taken into account [7–9]. The generators we propose and the TestU01 suite comply with most (if not all) of the instructions given in these papers. For example, SP800-90b [9] implies the need for a dataset containing at least 1,000,000 values, yet TestU01 produces around 50,000,000 numbers to test the generators [5]. Our ultimate generator can be used in the frameworks mentioned by NIST [7], with no or very little modification depending on the scenario.

The TestU01 suite contains Small Crush, Crush, Big Crush, Alphabit, and Rabbit batteries and a pseudo-NIST battery. This NIST battery contains 13 original tests out of the 15 included in the real NIST suite, and the remaining 2 are replaced by more advanced variations [5]. Therefore, it will be the main measure of our experiments.

Marsaglia in his paper [10] showed a class of RNGs, called "xorshift RNGs", consisting of consecutive bitwise xor and shift operations using seeds. He claimed that the algorithms are extremely fast and reliable in terms of randomness. Since the speed of this algorithm family is proven, they will be in our scope in this research. However, that reliability claim was invalidated in some later studies [11]. Figure 1 shows the main members of the xorshift family of RNGs according to the development timeline.

Apart from the TestU01 suite (and the NIST suite), there is another measure of randomness for some (especially RFID-compatible) lightweight RNGs, defined in the EPC UHF RFID Generation-2 standardization document. It is used to test the suitability of our ultimate generator for UHF RFID devices.

One of the parallel researches by Vigna [12] revealed another xorshift-based RNG, called xorshift*. The RNG performed well when compared to its predecessors, but was worse than the successors. However, it showed that the xorshift family is flexible and is very open to further development.

After the famous and widespread "Mersenne Twister" [13], in a follow-up study, Saito and Matsumoto came up with another similar algorithm, "XSadd", which is claimed to be more reliable. This claim is verified

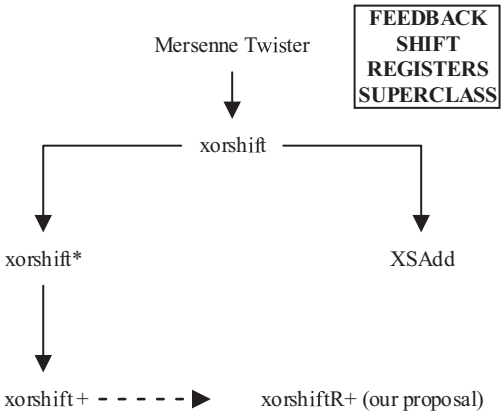

Figure 1. Development lapse of (some) xorshift based RNGs.

according to the results of the tests of the TestU01 suite [14]; however, it is reported that the inverse of this algorithm fails some of the tests of the same suite.

A later and yet very recent study of Vigna [14] again introduced another xorshift-based RNG, "xorshift+", claimed to pass all tests of the TestU01 suite and even inverse, which is not an invalidated claim in any newer publication that we could analyze. For this obvious reason, this RNG is chosen as a starting point in our study. In later sections of this paper, it is referred as the original xorshift+.

3. Candidate RNGs

Since the computational limitations may be very stringent on lightweight devices, many of the complex algorithms considered as successful in related works are not considered as eligible for this purpose. Xorshift+, a recent algorithm of the known xorshift family, is in focus with its simple structure [14]. It is also a main part of the study to force this algorithm to run on a variety of input and output sizes. Moreover, the original xorshift+ algorithm is manipulated by removing some last steps to make it even more compact. O'Neill also mentioned this reduction idea [11], though not tested systematically.

Within this work, we have made random scramblings (based on our predictions) of the original xorshift+ and produced many generators with slight differences in order to derive a better xorshift RNG. Nevertheless, we have eliminated many of these, which are considerably weak or do not sufficiently mitigate the complexity. Our efforts to find a suitable reduced xorshift+ version (which is to be named as xorshiftR+, where R stands for reduced) resulted in these 3 candidate algorithms (later called variations or shortly var), given as follows (in C notation);

```
uint64_t s[2]; // seeds
uint64_t xorshift128plus(void) {
      uint64_t x = s[0];
      uint64_t const y = s[1];
      s[0] = y;
      x ^= x << 23; // a, shift & xor
      x ^= x >> 17; // b, shift & xor
```

```
      //- - - - - - - -
Var1: x ^= y  ~~^ (y >> 26)~~ ; // c, xor
      s[1] = x;
      return x + y;
      //- - - - - - - -
Var2: x ^= y  ~~^ (y >> 26)~~ ; // c, xor
      s[1] = x + y;
      return x + y;
      //- - - - - - - -
Var3: x ^= y  ~~^ (y >> 26)~~ ; // c, xor
      s[1] = x + y;
      return x ~~+ y~~ ; }
```

The body of the code above (from the beginning until the first dashed line) is untouched and common for all variations (in fact, step c is the same for all, too), while the struck code snippets are removals from and the black-highlighted parts are additions to the original code of the xorshift+. The variations given under three consequent tags are applied singly to the body of the code so that a full RNG algorithm is obtained.

Ending variation 1 excludes 26 right shifts and 1 xor operation per generated random number. Ending variation 2 excludes 26 right shifts and 1 xor but adds 1 addition. Lastly, variation 3 excludes 26 right shifts and 1 xor, removes 1 addition, and adds 1 addition, which makes it identical to variation 1 in terms of operation count. These manipulations supposedly produce a big difference in applications where many numbers are generated sequentially. However, from the codes themselves, it is not possible to accurately estimate the time gain without experiments. That is because the run-time highly depends on the processor type [15]. Thus, we make use of our test completion times to measure the algorithms' performance on a lightweight IoT device, given in Section 5.

Memory gain, on the other hand, is easier to calculate or estimate. The memory usage of generators is mostly dependent on the period of their output and the number of their input (seed) states. These two factors are determinative of the number and the size of the variables used in their source code. To make a standard, we fixed the integer types to 32 bits, as mentioned above. Please note that the original xorshift+ (and our reduced variations) uses two seed inputs, which makes it occupy 64 bits of input memory, while most RNGs use only one, so it is implanted using 64 bits of seed state. Nevertheless, because of its very poor performance, we did not use the 32-bit seed/input version of the linear feedback shift register (LFSR), but the 128-bit version instead. The output was kept as 32 bits to fix the output sizes of all subjects. In order to see the effect of the output or the period of the RNGs on their randomness scores, please refer to another paper [16].

Other than our xorshift+ variations, we also tested an earlier xorshift-based algorithm, namely xorshift64* [13], a simple linear congruential generator (LCG), and an LFSR implementation to compare the results and estimate the benefits of using each RNG. XSadd is intentionally excluded since it was already superseded by xorshift+. A comparative breakdown is provided for a brand new xorshift-based RNG, called xoroshiro128+, that was published at the time of the writing of this paper. The LCG is included because of its popularity,

even though it is not recommended as a reliable RNG anymore [1,11]. It is based on the one implemented in the TestU01 suite as "CreateLCG()". The LFSR is included because of its different bit-level structure and high-speed nature. It is also based on the one introduced in the suite with the name "CreateLFSR113()".

We had momentous efforts to functionalize the DES with such a purpose as recommended in RFC 1750 [3] with relevant modifications. Nevertheless, later we discarded it during our preevaluation phase because of its extremely poor performance results. It was not even close to being "lightweight" when compared to the other RNGs presented here. Information that explains why and by how much the DES, advanced encryption standard (AES), and derivatives are slower than these lightweight algorithms can be found elsewhere [17].

4. Test methodology

The RNGs are tested using the Big Crush battery of tests, which is the most comprehensive test battery in the TestU01 suite. It contains 106 different tests and makes a total of 160 runs (some tests are repeated with different parameters) [5,18].

L'ecuyer and Simard defined 3 failures out of 15 tests of the Small Crush battery and 7 failures out of 38 tests of the Rabbit battery as "clearly unacceptable", which make 80% and 81.6%, respectively. In order to be called successful, any RNG must obtain a clearly higher score than these, preferably 100% depending on the scenario. Nevertheless, the rate of passed tests is not solely enough to decide; the result values of each failed test should also be taken into account. TestU01 checks p-values to determine if RNGs pass certain tests or not, for all tests, with a pass interval of 0.001 to 0.999. It also introduces two epsilon values, namely eps and eps1, which represent very small numbers (the latter being much smaller) and are practically equal to 0 [5,18]. Hence, if an RNG fails at least one test with a value of eps (or eps1), then that RNG will most likely fail this test in each run and possibly for all seeds, which means a systematic fail. However, if the p-value is very close to the limits (while the term "very close" is essentially vague), for instance $0.999 < p < 0.9999$, then the result is not clear. Even though L'ecuyer and Simard proposed repeating suspicious tests as the best practice to get rid of the suspicious results, our experiments show that in the case when a p-value falls in the range given above, the test can be accepted as a pass without repeating it, especially if it is the only fail or there are very few, since a repeat with a different seed will most likely result in a pass. However, if the p-value is not so close to the limits, though not eps, then repeating the tests might be necessary. Note that the same seed most likely will cause exactly the same result in further repeats. If repeating the tests is not feasible, these tests should be taken as fails. Remember, the number of total test runs is more than the number of different tests, since some of the tests are repeated with various parameters in all batteries [18].

Other than the randomness levels, since we also have lightness concerns, run time, seed, and output sizes of each algorithm are taken into account to determine their memory usage. Memory usage is not at the center of focus since no tests are seen as required besides the register size calculations, which can be done using function definitions.

For evaluating the time performance, two measures are considered: total time lapsed during the tests and time period required to generate 100 million numbers (not bits). The total duration of the tests is a good indicator of speed of the generators because it represents a comprehensive usage scenario. However, it is not always very consistent as the number of tests applied may differ for some RNGs depending on their algorithmic structure. Thus, we also recorded the time required to create 100 million numbers to be sure about their pure speed. This was done using the native timer function of the TestU01 suite, with the default method GetU01. In order to improve the solidity of our tests and recommendations, we have run most of the algorithms, namely

the xorshift-based ones and LFSR113, also on a WISP device 10,000 times sequentially and recorded the total creation time likewise. This additional step made us sure that our claims verified on a PC are also valid for a real lightweight IoT device and vice versa.

When an RNG is forced to use different seed numbers without completing its regular run for its full period, i.e. the seed is changed after each random number output, then the RNG will most probably lose its ability to produce randomness (except for some, i.e. LFSR derivative scramblers). In this case, the degree of the randomness of the output will depend on the randomness of the seed string, because every seed input will cause the RNG to produce the same set of random numbers in each of its runs. This prevented us from using hardware-generated so-called true random numbers as a seed chain. Instead, we used only a single seed (duplicated when the RNG required multiple seeds) for each subject generator. A seed of an RNG is only changed to renew a test if there are suspicious results.

5. Empirical results

In order to make the final evaluation, all subject generators are tested with the Big Crush battery of the TestU01 suite, which is the biggest and toughest battery in the suite, while other batteries (like Small Crush and NIST) are used to make preliminary tests. Big Crush contains 106 tests and makes a total of 160 runs, since some of the tests are repeated with different parameters. Furthermore, in order the find out the average time required for generation of a single random number for each algorithm, 100 million numbers on the PC and 10,000 numbers on the WISP are generated. Hence, the average time required for 1 number is calculated and shown in the Table.

Table. Comparative data of empirical test results from Big Crush for all subject RNGs.

Empirical results	Big Crush score (# of runs)	Big Crush score (# of tests)	Run time of Big Crush	Time for 1 number on PC	Time for 1 number on WISP
xorshift+	159/160 99.3%	105/106 99.1%	7h:35:03	0.02 µs	0.5515 ms
xorshift*	156/160 97.5%	102/106 96.2%	7h:28:10	0.0202 µs	0.2703 ms
Reduced xorshift+var1	155/160 96.9%	103/106 97.2%	7h:27:02	0.0191 µs	0.1955 ms
Reduced xorshift+var2	160/160 100%	106/106 100%	7h:32:40	0.0192 µs	0.1954 ms
Reduced xorshift+var3 [xorshiftR+]	160/160 100%	106/106 100%	7h:30:07	0.0191 µs	0.1953 ms
Simple LCG	91/153 59.5%	57/106 53.8%	7h:25:45	0.0177 µs	N/A
LFSR113	154/160 96.3%	100/106 94.3%	6h:47:57	0.0175 µs	0.1641 ms
C rand()	0/160 0%	0/106 0%	N/A	N/A	N/A

5.1. Testbeds

Our PC testbed was a generic server computer with an Intel Xeon E5540 processor @2.53 GHz running on Windows utilizing 4 GB of RAM. Nevertheless, we observed that the software of TestU01 only occupies a single core during all test runs. This leads to a more linear base time comparison and prevented us from using our

bed at full rate, but this might also be positive since a lightweight system most probably will only have a single core dedicated to random number generation. Our IoT testbed is a standalone WISP5 device that includes a built-in MSP 430 processor. Results of the tests are as follows.

5.2. xorshift+

The original xorshift+, not surprisingly, demonstrated excellent randomness according to the Big Crush tests as Vigna claimed [14]. It passed 105 of the 106 different tests and 159 of the 160 test runs in our initial attempt.

The p-value for this test is very close to the limit and, as explained in Section 4, it can be marked as a false positive and be ignored. In fact, repeated attempts show that this fail was not systematic, but a coincidence. The ultimate scores might also be taken as 100% for both measures. We still kept this fail in our records for the sake of temporal consistency. It was, however, somewhat slow in terms of speed on the PC and the slowest on the WISP when compared to other RNGs, as can be seen in the Table, which makes our efforts more significant.

5.3. xorshift*

The original xorshift* obtained, not surprisingly, slightly lower randomness scores on the Big Crush suite, though not bad. It failed four tests systematically. On the PC, its speed was unsatisfactory, actually slightly the worst among all. However, on the WISP device it was (only) better than xorshift+. Please see the Table. Hence, it cannot be recommended as a lightweight IoT solution, but it might be used on PC platforms depending on the scenario.

5.4. Reduced xorshift+ var1 (s[1] = x, return x + y)

Our first variation obtained test scores very close to the original xorshift*; they also share the 81st LinearComp test as a fail. On the other hand, our variation was super-fast. On the PC, it was the fastest (along with variation 3) among other xorshift-based subjects. It failed five tests systematically.

From the p-values of the failed tests, we can say that these are clear fails, because they are far outside the limits. However, considering its speed, this might still be a useful generator since its score is much higher than the (bad) examples given in Section 4 (which were 80% and 81.6%).

5.5. Reduced xorshift+ var2 (s[1] = x + y, return x + y)

Our second variation got excellent scores of 100% from the Big Crush suite by passing all the tests in all runs with no exception. Even though it was not the fastest, it was still clearly faster than the original xorshift+ and xorshift*. This agility difference is much more obvious on the WISP than the PC. On the PC, all our variations showed very similar results.

5.6. Reduced xorshift+ var3 (s[1] = x + y, return x)

This third variation, which we have named xorshiftR+, demonstrated excellent scores (again 100%) by passing all the tests in all runs without any exception, too. In fact, variation 3 was slightly faster than variation 2 on both the PC and WISP. It was also slightly faster than variation 1 on the WISP, while having the same speed on the PC. Hence, it is the fastest of its kind. Considering its perfect and consistent randomness scores and very good time performance, it can safely and properly replace the original xorshift+. There are no statistics in which the original xorshift+ is better than our variation 3, as can be seen in the Table. A bitmap graph

using TestU01's PlotUnif function is seen in Figure 2. It shows how the bits are randomly distributed under the function's drawing rules.

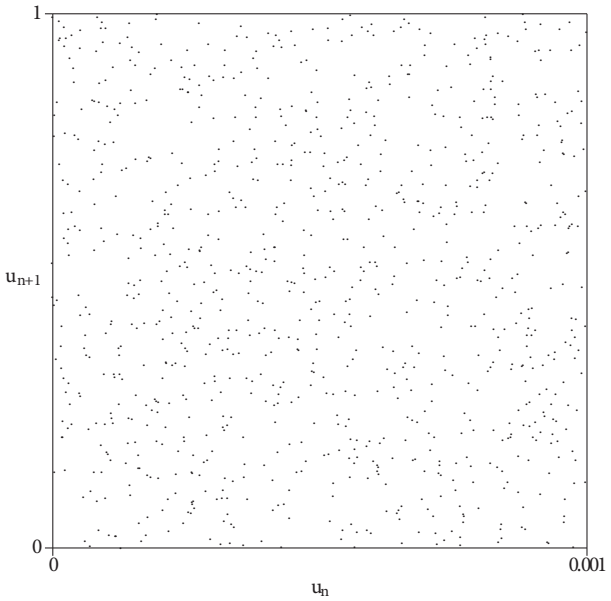

Figure 2. Bitmap of xorshiftR+; 1000 points plotted using scatter in TestU01 (u_n and u_{n+1} stand for sequential random numbers; see definitions in [18]).

Even though the TestU01 suite is already superior to the well-known NIST suite, because of its widespread use and fame, we have tested xorshiftR+ using the NIST suite, too. The result was a complete pass for all instances of all tests in the suite, as expected. The test was made with 1,000,000 generated 64-bit numbers. Detailed documentation of the test results is given online at http://srg.cs.deu.edu.tr/publications/2017/xor/.

To assure the algorithm's compliance with the RFID tags' security standards, three conditions mentioned in the EPC Gen-2 Class 1 document were tested, too. The first implies that the probability of a single 16-bit random number should be $0.8/2^{16} < \mathrm{P(RN16)} < 1.25/2^{16}$ for 2^{30} numbers. xorshiftR+ satisfies this condition; moreover, even for 2^{26} numbers, the result was $0.852/2^{16} < \mathrm{P(RN16)} < 1.18/2^{16}$. The second condition is that the probability of simultaneously identical sequences for 10,000 tags should be $< 0.1\%$. With xorshiftR+, since there are two 64-bit seeds, this probability is calculated as $(1/2^{64}) \times (1/2^{64}) = 1/2^{128}$ and then the result is $[10{,}000 \times (1/2^{128})] \times 100 = 2.94\% \times 10^{-34} < 0.1\%$. Lastly, the third implies that an RN16 drawn from a tag's RNG shall not be predictable with a probability of greater than 0.025%. This is proven via the ENT suite, another old yet popular test suite. There, we have also used the original xorshift+ for the sake of a consistent comparison, and xorshiftR+ demonstrated a very high performance that is practically equivalent to that of the original xorshift+. Detailed information regarding these conditions as well as the test results can also be found via the link given above.

Overall, for xorshiftR+, NIST, and EPC Gen-2 Class 1 (incl. ENT), examinations were successful, too, in addition to TestU01.

5.7. Simple LCG

The simple LCG, implemented with parameters (2147483647, 16807, 0, 12345), was very weak in terms of randomness, since it can only pass a little more than the half of the tests and test runs, which is far below

the unacceptable examples given in Section 4 and the lowest among our subjects. Figure 3, generated by the scatter of TestU01, reveals a repeating pattern and thus clearly shows that the generated bits are not random. The total number of test runs was lower because some tests were not repeated as a result of the RNG's poor performance as described in Section 4. Names and parameters of the failed tests are not presented because of their huge count. All fails were clear considering their p-values. Note that the run time for the suite given in the Table (which is very short) might be misleading since the number of test runs was fewer than the others (because of its poor randomness performance). The number creation times show that this RNG is very fast in scale (not tested on the WISP). Despite its speed, the LCG as implemented here is not recommended in any scenario and should not be used in applications where a high level of randomness is required.

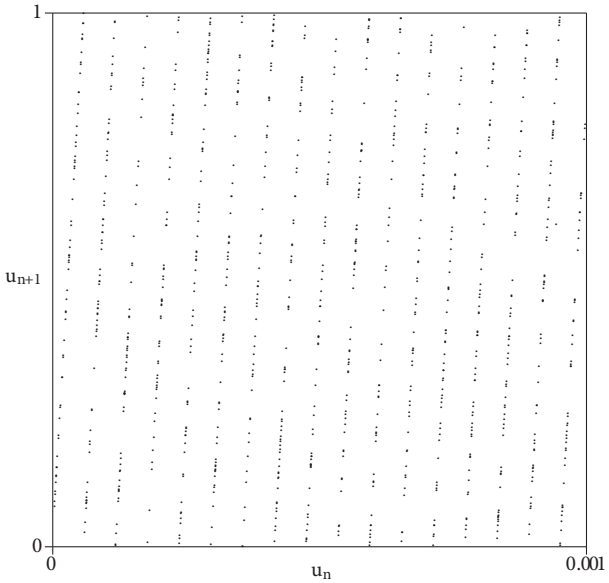

Figure 3. Bitmap of simple LCG; 1000 points plotted using scatter in TestU01 (u_n and u_{n+1} stand for sequential random numbers; see definitions in [18]).

5.8. LFSR113

LFSR113, with parameters (12345, 12345, 12345, 12345), was the fastest RNG on the PC among our candidates with a decent difference when compared to the xorshift-based subjects including our variations. It was approximately 8% faster than all our variations on the PC. Nevertheless, it is remarkable that it demonstrated even better performance on the WISP. Conversely, LFSR has indecisive test scores, which can be found acceptable depending on the scenario, but surely not better than the xorshift family. The generator failed six tests systematically. This RNG is recommendable on PCs and in a cases where the randomness requirements are loose but time limitation is a more dominant concern.

5.9. xoroshiro+

After we had concluded our tests and most of this study as well, a brand new xorshift-based generator, namely xoroshiro128+, was announced by Vigna. It is claimed that this is the fastest of all. Hence, we could not ignore it and made a comparative analysis, although we could not run a new TestU01.

xoroshiro128+ contains the following operations: 105 left shifts, 37 right shifts, 1 addition (64 bit number), 2 or (64 bit numbers), and 2 xor (64 bit numbers), for a total of 462 bitwise operations. Our xorshiftR+ only

contains 23 left shifts, 17 right shifts, 2 xor (64 bit numbers), and 1 addition (64 bit number), which makes 232 bitwise operations. Since the types of the operations required are the same for both and the number of operations for xorshiftR+ are almost half that for xoroshiro+, xorshiftR+ is clearly lighter and expected to run faster. Thus, xorshiftR+ is apparently superior to xoroshiro+, too.

5.10. Further notes

As further information, test results of the given xorshift-based generators represent their 32-bit output versions, instead of the contemporarily used 64-bit (or more) versions, where the integers and the outputs are 64-bit numbers. This is because, in the lightweight family of devices, 32-bit is a common upper limit. Presumably, 64-bit versions get the same or better results from the tests. In contrast, 16-bit integer-sized versions of the generators (including the original xorshift+) fail almost all tests, even in the Small Crush battery. This is because many tests of TestU01 assume an output of at least 30 bits and fail the subject generator otherwise [18]. The test results of these 16-bit versions are still very weak (they fail most of the tests in the suite) and are not statistically valuable. This also explains why the standard rand() function of the C language (initiated by the srand(time(NULL)) code) fails all the tests in the Big Crush suite (obtaining a score of 0% in both measures). Because of their very poor performance, rand() and simple LCG are not tested on the WISP.

Please note that the runtime or number generation time difference between the RNGs may look small or even negligible, but it surely is not. When slower machines are used as hosts or extensive repeats will be made, these differences will become more obvious. All three of our reduced variations were faster than the xorshift+ itself, but our variation 1 could not pass all the tests and so was discarded during the final evaluation. Even though variation 2 also passed all the tests, variation 3 was the fastest among all our variations and consequently selected as the xorshiftR+. It can be used wherever xorshift+ can be used and it can safely replace the xorshift+ in further studies and applications.

Additionally, xorshift* can be considered as deprecated since xorshift+ (and xorshiftR+) was introduced, as was also mentioned in [14]. The simple LCG was expectedly very weak (in terms of randomness) and should not be used in any scenario without further modifications. LFSR was very fast but weaker than all the xorshift family members given here. Lastly, bitmap graphs were not drawn for other subject generators, since it was not possible for the human eye to detect a pattern from them as well. Only the best and the worst generators' graphs are shown (see Figures 2 and 3).

6. Conclusion

According to our experiments, the xorshiftR+ (also mentioned as the reduced xorshift+ variation 3; R stands for reduced; pronounced xorshifter plus) has demonstrated outstanding results when compared to other candidates in terms of both randomness and speed performance. It passes all the tests in the Big Crush battery of TestU01 in all runs and is slightly (but noticeably) superior to its ancestors, especially the xorshift+, with its higher speed. This superiority becomes even more obvious in lightweight environments. It also occupies less memory due to its compacted nature, and it passes all tests of the NIST and ENT suites and complies with EPC-Gen 2 Class 1 security standards. This RNG, xorshiftR+, is strongly recommended for use in both powerful computer environments and lightweight devices, where high levels of randomness and temporal performance are desired. As an addition to the conclusion given in [14], our experiments showed that, even if integer sizes are set to 32 bits (this makes 64-bit input/seed and 32-bit output), xorshift+ and xorshiftR+ can still pass all TestU01 tests in all runs systematically.

LFSR113, on the other hand, was the fastest on the PC, although it was not able to pass all tests. Plus, it occupies vast memory. As its original form (used here), it can only be recommended when randomness requirements are loose but time limitations are dominant. If it is possible to make it fail-free hereafter, then it might be a good alternative to xorshiftR+. Such improvements on the LFSR could definitely be a worthy future work.

Acknowledgment

We sincerely thank Hakan Altaş for his valuable efforts during our experiments.

References

[1] Hellekalek P. Good random number generators are (not so) easy to find. Math Comput Simulat 1998; 46: 485-505.

[2] Özcanhan MH, Dalkılıç G. Mersenne twister-based RFID authentication protocol. Turk J Elec Eng & Comp Sci 2015; 23: 231-254.

[3] Eastlake D 3rd, Crocker S, Schiller J. Randomness Recommendations for Security. RFC 1750. Fremont, CA, USA: IETF, 1994.

[4] Marsaglia G. The DIEHARD Battery of Tests of Randomness. Technical Report. Tallahassee, FL, USA: Florida State University, 1995.

[5] L'Ecuyer P, Simard R. TestU01: A C library for empirical testing of random number generators. ACM T Math Software 2007; 33: 22.

[6] Bassham LE 3rd. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST Special Publication 800-22rev1a. Gaithersburg, MD, USA: NIST, 2010.

[7] Barker E, Kelsey J. Recommendation for Random Bit Generator (RBG) Constructions. NIST Special Publication 800-90C Second Draft. Gaithersburg, MD, USA: NIST, 2016.

[8] Barker E, Kelsey J. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. NIST Special Publication 800-90A. Gaithersburg, MD, USA: NIST, 2012.

[9] Turan MS, Barker E, Kelsey J, McKay KA, Baish ML, Boyle M. Recommendation for the Entropy Sources Used for Random Bit Generation. NIST Special Publication 800-90B Second Draft. Gaithersburg, MD, USA: NIST, 2016.

[10] Marsaglia G. Xorshift RNGs. J Stat Softw 2003; 8: 1-6.

[11] O'Neill ME. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. Available online at http://www.pcg-random.org/pdf/toms-oneill-pcg-family-v1.02.pdf.

[12] Vigna S. An experimental exploration of Marsaglia's xorshift generators, scrambled. ArXiv e-print 2014. Available online at https://arxiv.org/abs/1402.6246.

[13] Matsumoto M, Nishimura T. Mersenne twister: a 623-dimensionally equi-distributed uniform pseudo-random number generator. ACM T Model Comput S 1998; 8: 3-30.

[14] Vigna S. Further scramblings of Marsaglia's xorshift generators, J Comput Appl Math 2016; 315: 175-181.

[15] Singh KP, Kumar D. Performance evaluation of low power MIPS crypto processor based on cryptography algorithms. International Journal of Engineering Research and Applications 2012; 2: 1625-1634.

[16] Brent RP. Some long-period random number generators using shifts and xors. ANZIAM J 2007; 48: 188-201.

[17] Thakur J, Kumar N. DES, AES and Blowfish: Symmetric key cryptography algorithms simulation based performance analysis. International Journal of Emerging Technology and Advanced Engineering 2011; 1: 2250-2459.

[18] L'Ecuyer P, Simard R. TestU01: A Software Library in ANSI C for Empirical Testing of Random Number Generators User's Guide Document. Montreal, Canada: University of Montreal, 2014.