This pdf is a compilation of the six texts available under “AHS-RNG Explanations” on sicap.lu

Author: Alain Schumacher   © 2024

History of AHS-RNG

After a few attempts with the classic principles of the well-known PRNGs, I quickly realized that only a radically different approach could enable the generation of true random numbers. The smallest unit of random numbers is the individual bit. Ergo, one possible solution would be to try the much-vaunted coin toss as a basis. If it is possible to generate the individual bits with the best possible 50%/50% distribution and complete independence from the previous and subsequent bits, then the generation of genuine random numbers should be possible. The solution found must be described as creative programming and not as an arithmetic algorithm.

The main development work was carried out between 2004 and 2006. The first part was the definition of the "bit fishing table / BFT". In my first opinion the largest possible table seems desirable. I started to experiment with tables of 64 KB and 256 KB. The table had to contain a random distribution of 50% "0" and 50% "1". The possible variants through permutation are in unimaginable orders of magnitude, e.g. for a BFT of 64 KB there are more than 10exp157823 variants. At the moment we are only testing the applications with BFT 8 KB, where there are already more than 6*10exp19725 variations. Every single variation will generate a completely different random number series.

If we want to "fish" a single bit from this table, we still need a fishing rod, so to speak, in this case in the form of an address of a single bit. The solution to this task presented in the AHS-RNG demo is the 16th version of various programs that have slowly, improvement by improvement, led to the current structure. In addition to the logical structure to achieve the goal, the speed of the program execution had to be considered and optimized.

In the beginning, the speed on Pentium 3 and 4 computers with one core was a modest 124 megabits per second per computer, but through further optimization, e.g. through the two 32-bit calculations running in parallel in 64-bit registers, the speed could be increased for modern cores: to over 650 megabits per second and above, depending on the current speed of the core. Compared to good PRNGs (MT19937 or XOshiro256**), the speed is a considerable disadvantage. That is why we insist on recognizing it as a true random number generator, and anyone who does not want to recognize this should prove the deviation from true random numbers. According to our tests with the ID-Quantique (16 megabits/second), AHS-RNG has also a quality advantage over physical random number generators. We also disagree with the claim that computer-generated random numbers lack entropy and we are ready to carry out comparisons in this regard.

The best motivation for intensive research and programming until the breakthrough was a sentence in "Discrete event simulation in C, Kevin Watkins, 1993" page 67: "It is not a good idea, however, to try and design your own from scratch. The design of random number generators is full of nasty surprises for the unwary and is best left to established experts". Anyone who is familiar with me knows that such a statement is a challenge for me that I will try to refute.

Deterministic true random numbers?

It may seem rather heretical to some if we claim that deterministic true random numbers exist. To answer this question, we need to analyze the constituent properties of true random numbers. Why the good PRGNs like MT19937 or XOshiro256** don't generate true random numbers? These PRNGs are algorithmic mathematical procedures that generate quite reasonable random numbers. However, they involve running through large periods, in the case of MT19937 even a very large period. Nonetheless, both inventors claim that these are equally distributed random numbers and that all possible values occur equally often after passing through the period. This fact alone indicates that they are not genuine random generators. True random numbers would have to follow the laws of probability even with the huge quantities of random numbers and the number per pattern would have to follow the normal distribution. In addition, statistically relevant deviations from probability can be found when analyzed closely.

The question of why deterministic random numbers can be true random numbers becomes interesting. Let's go back to the beginnings of "high-speed computation", to the contribution of John von Neumann at the symposium "Monte Carlo Method" from June 29, 30 and July 1, 1949. The section before the famous quote: "Any one who considers arithmetical methods ..." reads:

"We see then that we could build a physical instrument to feed random digits directly into a high-speed computing machine and could have control call for these numbers as needed. The objection to this procedure is the practical need for checking computations. If we suspect that a calculation is wrong, almost any reasonable check involves repeating something done before. At that point the introduction of new random numbers would be intolerable. I think that the direct use of a physical supply of random digits is absolutely inacceptable for this reason and for this reason alone. The next best thing would be to produce random digits by some physical mechanism and record them, letting the machine read them as needed. At this point we have maneuvered ourselves into using the weakest portion of presently designed machines - the reading organ. Whether or not this difficulty is an absolute one will depend on how clumsy the competing processes turn out to be."

We are firmly convinced that this statement is still valid today. The possibilities of storing and re-accessing physically generated random numbers have grown to an unimaginable extent. Nevertheless, there are still limits in this respect today. The following example proves the untenability of the statement "Deterministic random numbers are automatically to be regarded as pseudo-random numbers": Using ID-Quantique's generator, during many weeks in 2008, we recorded two data files of 2000 gigabytes each. Since then, these files have been read out for various tests, as this is the only way to make repeatable calculations (in line with John von Neumann's statements) and, on the other hand, direct processing would be ineffective due to the low production rate.

It is indisputable that the random bits read from the physical random number generator deserve to be called "true random numbers". We now use these random numbers to perform a calculation that we call a "simulation generated with true random numbers". We save the random numbers in a file for later checking. According to the generally accepted theory, the random numbers saved in this way have now degenerated into pseudo-random numbers, since the saved "true random numbers" have now mutated into "deterministic random numbers" if we were to insist on applying the false rule that all deterministic random numbers are automatically pseudo-random numbers! There is no doubt that random numbers read from a file are deterministic random numbers. We would now have to call the first simulation "with true random numbers", while the second simulation is called "with pseudo-random numbers", even though the same sets of numbers were used twice.

This distinction contradicts the principles of science, because one must not automatically and incoherently make a statement of properties that are different in essence just to promote one's own marketing ideas. We would therefore make it clear that, in our opinion, there are "true random numbers" that have been generated using appropriate procedures in accordance with John von Neumann's maxim. These can exist both as non-deterministic or deterministic random numbers, the latter as stored values or as latently existing true random numbers.

Is the deterministic AHS-RNG a TRNG ?

Without any scientifically proven basis, it is repeatedly and reflexively claimed that a computer as a deterministic machine cannot generate true random numbers. The bogus arguments put forward relate exclusively to the typical PRNG structure, i.e. arithmetic procedures for calculating a series of random numbers using algorithms, starting from a seed. The random numbers generated in this way vary from abysmally bad for the older algorithms (partly due to the limited technology of the time) to quite useful newer variants, such as MT19937 and Xoshiro256**.

What they all have in common, however, is the fact that the mathematical function does not make it possible to generate true random numbers. The classic statement by John von Neumann applies here: "Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin." But the statement then goes on: "For, as has been pointed out several times, there is no such thing as a random number - there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method."

Of course, it is in principle impossible to prove directly that the random numbers generated by AHS-RNG are true random numbers, but we can only substantiate the assessment "true random numbers" through very extensive tests, and demonstrate the non-mathematical principle by showing exactly how it works. To paraphrase John von Neumann, the method of generating random numbers used in AHS-RNG must be considered a correct method of generating true random numbers, the underlying principle being the generation of every single bit with a 50/50 chance. The motto we gave the method is: "Born to be wild or tame, or anything between".

In this evaluation, we want to limit ourselves to the deterministic AHS-RNG, which can deliver a minimum number of 281,474 billion bits, corresponding to a file of over 35 terabytes, without additional entropy, using only the loaded parameters BFT, FAAP and LCG. To run a large simulation on 1000 cores, you would therefore need only 17 megabytes of AHS-RNG parameters, compared against up to 35 petabytes of stored random numbers in case of physical RNGs. If the generated AHS-RNG random numbers really have the properties that are essential for true random numbers, this would certainly be a major improvement. We have created a pool of one million variants and we need only 17 gigabytes. For a pool of a billion different parameter tables, the memory requirement would be 17 terabytes. This can be compared with the memory requirement of 35,000,000 petabytes for the same amount of real, physically generated random numbers.

How this large amount of true random numbers can be generated from 17 kilobytes of initial data? Due to the way AHS-RNG works, more than 10exp19800 completely independent variants of random number sequences are possible with the limited initial values, and are thus stored virtually in the initial data. For example, if we create a program for multiplication, there are no results in the program. But by calling the program with the help of two indexes, we can calculate a multiplication table up to 12 digits each, i.e. over 500,000 trillion results. We can therefore justifiably claim that, if our program can calculate results up to 24 digits, then this number of results is virtually available. The deterministic AHS-RNG can be viewed in a similar way. The only decisive question is: Are the virtually available random numbers, which are materialized by the execution of the program, true random numbers? We are convinced that the answer to this question is "yes".

How the non-deterministic AHS-RNG works

While the deterministic AHS-RNG generates the true random numbers with a specific program from the initial parameters, comparable to reading out stored physical random numbers, the non-deterministic AHS-RNG uses a physical random component. A physical random event does not always have to be a radioactive decay process; time is also a physical parameter that can be used advantageously in a Linux operating system. The biggest advantage is of course the fact that this physical random generator is free of charge and available with a few lines of C code.

Anyone who has worked with applications under a Linux operating system, probably on a

NUMA (Non-Uniform Memory Architecture) machine with L1, L2 and L3 caches and fewer memory channels than processor cores, knows that the execution time of identical jobs can sometimes vary greatly, which can easily be shown by using the time command. While the time command works only with milliseconds, it is possible to query the time in nanoseconds. However, one restriction must be made: The random numbers determined with the physically help of the run of the time are not statistically correct random numbers. Fortunately, this is not necessary, the values need only to be roughly different. With these values used as a timer, "one progression event" (FBM3 to FBM4, FBM2 to FBM3, FBM1 to FBM2 and a new FBM1) of the sequence control after 32 bits (or after 64 bits in the parallel version) is randomly replaced by "two progression events", always one after each 16 bits. In addition, the increment value of the LCG is increased by 2, which causes an additional change in the sequence.

The easiest way to generate a random value for this timer is to calculate the result of the nanosecond-query modulo 997, and to add a constant of 313. The latter to ensure that a minimum number of cycles is calculated before the next "two progression events" occurs. The random value calculated in this way is simply counted down after each 32-bit generation cycle. At the end of the timer countdown a new "two progression events" cycle is executed, and a new calculation of the timer random value, with the help of the nanosecond clock. If we use the same parameters of the AHS-RNG to generate a large to very large number of non-deterministic random number series, we need to run the generator in a "warm up" phase of 2 million cycles to get the registers into individually different states. With different parameters per AHS-RNG instance, the "warm up" is also recommended if we want completely different random numbers right from the start. The "warm up" takes about 1/5 of a second.

This is how the non-deterministic AHS-RNG works in "secret" mode. This means that nobody can recalculate the random numbers generated, even if he knows the parameter tables used. According to the first tests, the true random numbers generated in this way are superior to those generated with physical RNG. A further advantage is described in the article on the four different Families of AHS-RNG.

The use of the internal time data as supplementary entropy was first described in our white paper from 2006 (p. 25) in the Glossary as "Run-time randomness". We designate this principle as "mini-entropy". In combination with the function of the program the resulting entropy in the produced random numbers is perfect, and these random numbers are completely different from all the others. The term mini-entropy comes from the fact that these time derived variables are only around 1/1700 of the random numbers produced.

The four Families of AHS-RNG

1. Family **AHS-RNG-determ** or simply **AHS-RNG**

There are two different basic systems of the AHS-RNG program. The first is the deterministic system, abbreviated as "AHS-RNG-determ" or simply "AHS-RNG". At the initialization, the bit-fishing-table (BFT), the final address assembly parameters (FAAP table) and the parameters a, c and seed for one or two LCGs are loaded as parameters. These three pieces of information then determine the generation of the true random numbers, similar to reading true physical random numbers from a file. We would like to mention that there are different program variants, e.g. a 32-bit and a 64-bit variant working in parallel. The loading of the parameter table can also vary. However, the random numbers produced differ only for the 32-bit and 64-bit variants. All deterministic versions form the first family.

2. Family **AHS-RNG-secret**

We call the second family the program variants that work in a non-deterministic way with the physical mini-entropy "time", but without storing the random values of the mini-entropy used. This makes it impossible to recalculate the random numbers generated, even if all the parameters are known. This is why we refer to this family as AHS-RNG-secret.

3. Family **AHS-RNG-record**

The third family works in the same way as the second, with the important difference that the mini-entropies used are stored when the non-deterministic random numbers are generated. The mode of operation is therefore the same as with a purely physical random number generator, in which all random numbers generated are stored. This family is therefore called AHS-RNG-record. The big difference, however, is that the data to be stored is around 1700 times smaller than with a purely physical random number generator. The name of the write file must be specified as an additional parameter. This ensures that identical real random numbers are available later, for example to carry out a corrected simulation with identical random numbers. This corresponds to John von Neumann's requirement that purely physical random data must also be available again in order to be able to carry out new calculations with identical random numbers. AHS-RNG-record makes it possible to carry out very large simulations with non-deterministic random numbers, which would fail with purely physical random number generators due to the required storage capacity.

4. Family **AHS-RNG-replay**

The fourth family is the counterpart of the third family. By reading out the mini-entropy file created with the third family, the generation of identical random numbers is guaranteed. This is why we refer to this family as AHS-RNG-replay. It therefore corresponds to reading out the stored random numbers of a physical RNG, with the great advantage of only having to process a data volume that is 1700 times smaller. The name of the file with the mini-entropy saved by AHS-RNG-record is the additional parameter here. Of course, the same parameters BFT, FAAP and LCG must be used as for the original AHS-RNG-record.

Why is AHS-RNG cryptographically secure?

Cryptographically secure random number generators, whether deterministic or non-deterministic, must first and foremost ensure that anyone who somehow learns a subset of the random numbers generated cannot guess or calculate any previous or subsequent random numbers. Pseudorandom number generators that have this property (BlumBlumShub, for example) are also known as CSPRNGs.

The AHS-RNG, both in the deterministic variant and in the non-deterministic variant, fulfils this requirement perfectly. As each individual bit is the result of a complex calculation of a bit address of the bit-fishing-table and is selected from a set of 32768 identical bits (each 0 or 1), it is impossible to trace the origin of the individual bit. The bit-fishing-table is in reality a static internal "state" which has a specific value from over 10exp19725 possible values. A second important static secret is the FAAP. In addition, there is the static value of the multiplier (once or twice 64 bits, a prime number) and the static (in the deterministic variant) or dynamic value (in the non-deterministic variant) of the increment of the LCG, any odd value. The starting value of the LCG with once or twice 64 bit must also be known, as the first values from the LCG influence the calculation over the entire further course. These form half of the basic modifiers, while the other half is determined using the BFT. This means an additional 256 bits of internal static code, which can no longer be calculated subsequently, as this information is one of the bases for indirectly creating the address of the bit to be selected. Before this address is completely finished, it is compiled from four different basic randomness values with the help of FAAP. We recommend the AHS-RNG demo for demonstration purposes.

If, contrary to all expectations, someone comes up with an initial promising plan for a possible attack, we are happy to make our HPC of 30 TFLOPS temporarily available if necessary. For the time being, however, we are convinced that all the secrets used must be known, i.e. the BFT, the FAAP and the parameters of the LCG as well as their seed, in order to make an attack possible.