Evolving Cryptographic Pseudorandom Number Generators *

Stjepan Picek[1], Dominik Sisejkovic[2], Vladimir Rozic[1], Bohan Yang[1], Nele Mentens[1], and Domagoj Jakobovic[2]

[1] KU Leuven, ESAT/COSIC and iMinds, Kasteelpark Arenberg 10, bus 2452, B-3001 Leuven-Heverlee, Belgium
[2] Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia

Abstract. Random number generators (RNGs) play an important role in many real-world applications. Besides true hardware RNGs, one important class are deterministic random number generators. Such generators do not possess the unpredictability of true RNGs, but still have a widespread usage. For a deterministic RNG to be used in cryptography, it needs to fulfill a number of conditions related to the speed, the security, and the ease of implementation. In this paper, we investigate how to evolve deterministic RNGs with Cartesian Genetic Programming. Our results show that such evolved generators easily pass all randomness tests and are extremely fast/small in hardware.

Keywords: Random number generators, Pseudorandomness, Cryptography, Cartesian Genetic Programming, Statistical tests

1 Introduction

Random number generators are used in a range of applications spanning from producing simple values and adding randomness to programs, over online betting to various cryptographic applications. Accordingly, they are important components in many real world scenarios. In cryptographic applications one relies on a source of randomness that can produce truly random numbers when generating seeds, nonces, initialization vectors (IVs), etc. However, for many of today's applications like generating masks or padding messages, true randomness is not needed, only statistical quality is required [**?**]. There, it is sufficient to use PRNGs that produce good results and yet are realized with deterministic methods. One example is the Blum Blum Shub generator [**?**] that produces numbers that are indistinguishable from true random numbers by means of standard statistical testing. This example shows that it is possible to obtain numbers of a sufficient statistical quality with a deterministic method.

* This work has been supported in part by Croatian Science Foundation under the project IP-2014-09-4882. In addition, this work was supported in part by the Research Council KU Leuven (C16/15/058) and IOF project EDA-DSE (HB/13/020).

In this paper, we investigate the efficiency of Cartesian Genetic Programming (CGP) when evolving deterministic PRNGs. In order to do so, we present a new framework capable of generating many PRNGS that pass statistical tests and are fast and small when implemented in hardware.

Motivation and Contributions

One real-world application of PRNGs in cryptography is to use them for masking [**?**] as a countermeasure against side channel attacks. When used for such a purpose, we want those generators to be extremely fast and small when implemented in hardware. To obtain a generator with such characteristics, we cannot use "expensive" operations like multiplication or addition.

Therefore, in this paper we evolve PRNGs that pass all statistical tests and use only cheap operations, which is also an important difference between our approach and related work, since there multiplication and addition operations appear without constraints [**?**, **?**]. We emphasize that if we can use the multiplication operation, then there exist much smaller PRNGs than those presented in related work and it is a trivial task to design a PRNG that passes all statistical tests. Next, we consider some of the fitness functions that are used in related work actually inappropriate since they do not mimic the inner working of a PRNG. Therefore, in this paper we use a fitness function that we believe describes the PRNG behavior better. Finally, we are the first to apply CGP to this problem, since that paradigm is the most natural for PRNG structures because of its multiple input – multiple output configuration.

In Section 2, we give the necessary introduction to random numbers, PRNGs, and testing methods. Then, in Section 3, we give an overview of related work. In Section 4, we discuss the model of the PRNG we design and the obtained results. Section 5 offers a short discussion on the results, their applicability and some possible future research avenues. Finally, Section 6 offers a short conclusion.

2 (Pseudo) Random Number Generators

In this paper, we follow the terminology as given in the AIS 20/31 proposals [**?**]. An **ideal random number generator** is a mathematical construct that generates sequences of independent and uniformly distributed random numbers. A **random number generator** (RNG) is any group of components or an algorithm that outputs sequences of discrete values. RNGs can be divided into true random number generators and pseudorandom number generators.

A **true random number generator** (TRNG) is a device for which the output values depend on some unpredictable source that produces entropy.
Pseudorandom number generators (PRNGs) or **deterministic random number generators** (DRNGs) represent mechanisms that produce random numbers by performing a deterministic algorithm on a randomly selected seed.
Cryptographically secure pseudorandom number generators are PRNGs with properties suitable for use in cryptography.

A **seed** is a random value used to initialize the internal state of the generator. A **state** is an instantiation of a random number generator. Note that PRNGs can accept additional input data besides the seed value. In Figure 2, we give a model of a PRNG.

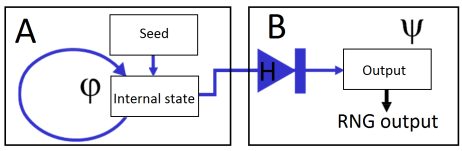

Fig. 1. Model of a PRNG.

In the rectangular A, we depict the input value for the generator (seed) and the PRNG (some function φ) and in the rectangular B the one-way function (H) with the RNG output. Here, ψ is the output function, φ is the state transition function, and $x_{n+1} = \varphi(x_n)$. In this work, we concentrate only on part A and we assume that the one-way function H does not exist. This diagram conforms to the model called DRNG.2 [**?**], but we note that here we are not interested in forward and backward secrecy requirements. Forward and backward secrecy ensure that it will not be possible to determine the successor and predecessor values from a known subsequence of output values.

2.1 Testing Randomness

The quality of PRNGs is evaluated using statistical tests which follow the same procedure as any other hypothesis testing. The hypothesis under test is that the PRNG produces a perfectly random output. Tests are applied on the output bit sequences. Each test defines a metric called *the test statistic* which can be computed from the sequence under test. A simple example of the test statistic is the bias of the sequence defined as:

$$\varepsilon = \left| \frac{N_{ones}}{N_{ones} + N_{zeroes}} - 0.5 \right|, \tag{1}$$

with N_{ones} the number of ones and N_{zeros} the number of zeros in the sequence. The next step is to compute the P_{value} which is the probability that an ideal RNG produces a sequence which is more extreme with respect to the defined metric than the sequence under test. For example, if the sequence under test contains 70 zeros and 30 ones, then the P_{value} is equal to the probability that an ideal RNG produces a 100-bit sequence with bias higher than 0.2. The final step is comparing the P_{value} with the predetermined constant, for example $\alpha = 0.01$. If the P_{value} is higher than the cut-off value α, than the sequence passes the test, otherwise it fails.

Each statistical test checks for a different statistical defect, therefore more extensive testing results in a more reliable outcome. Several batteries of statistical tests have been proposed, where the most famous ones are the NIST [**?**] and DIEHARD [**?**] test suites. In general, the tests applied on longer sequences are more likely to detect statistical weaknesses. In this work, we use the NIST battery of 15 tests applied on sequences of 10^6 bits. Statistical tests can produce both false-positive and false-negative errors. The probability of a false-positive error (truly random sequence fails the test) is equal to the chosen parameter α. Here, we use the value $\alpha = 0.01$ as recommended by the NIST standard [**?**].

3 Related Work

John Koza used genetic programming (GP) to evolve programs that output random numbers [**?**]. As a fitness function he used the notion of information entropy as defined by Shannon and the end result was a program that was able to accept a sequence of consecutive integer values and transform it into random binary digits. Hernandez, Seznec, and Isasi used GP to evolve random number generators where they used the strict avalanche criterion (SAC) as a fitness function [**?**]. Martinez et al. designed a pseudorandom number generator suitable for cryptographic usage by means of GP [**?**]. The obtained generator – Lamar was tested with a number of tests where the input values were obtained via a counter function. We consider this work the most serious attempt on evolving PRNGs for cryptographic usage although in our opinion this work has some potential drawbacks since it does not follow the structure a PRNG should have as discussed in Sect 5. Lopez et al. focused on the evolution of PRNGs that could be used in low cost RFID tags [**?**]. They followed an approach similar to previous work where the fitness function was based on the notion of the SAC and the testing on values obtained via a counter function.

4 The Proposed Model of a PRNG and Results

As a design choice, we decided to work with PRNGs that have four input terminals. Each terminal is represented with a 32-bit unsigned integer variable. This means that the state of our generator has 128 bits. Since we assume that the input and output sizes of the variables are of the same size, it means that our PRNG should output 128 bits of random data in every iteration.

We use the function set (inner nodes) that consists of binary Boolean primitives: rotate right/left for one position (RR/RL), shift right/left for one position (SR/SL), AND, NOT, XOR, and P(x). The function P(x) is a basic perfect outer shuffle where the bits are interleaved into two halves of a word and the outer (end) bits remain in the outer positions [**?**]. Note that the hardware implementations of RR, RL, SR, SL, and P consist only of rearranging the signal wires and therefore come without the cost of additional logic gates. Such an architecture is suitable for both hardware and software implementations because it utilizes a simple FIFO buffer. To handle the cases where the input value to a PRNG is

all zeros or all ones, we use one constant that we select randomly and it equals $4E2D93A6_{16}$. Although we work with generators that have four variables and a 128-bit state, we note that we could have chosen any number of variables and any variable size that is available in ANSI C (for the results to be platform independent).

4.1 Cartesian Genetic Programming Approach

In Cartesian Genetic Programming (CGP) a program is represented as an indexed graph. The terminal set (inputs) and node outputs are numbered sequentially. Node functions are also numbered separately. CGP has three parameters that are chosen by the user; number of rows n_r, number of columns n_c, and levels-back l [**?**]. In our experiments, for the number of rows we use a value of one and for the levels-back parameter we use the same value as for the number of columns. The number of node input connections n_n is two, the number of node output connections n_o is one, and the population size equals five in all our experiments. To obtain 128 bits of random values we set the number of output nodes to 4, so we need only one iteration to obtain the full generator state. For the CGP individual selection, we use a (1 + 4)-ES strategy in which offspring are favored over parents when they have a fitness less than or equal to the fitness of the parent. The mutation type is probabilistic.

To evaluate the performance of the evolved PRNGs, we use the following procedure. In every generation we randomly create four 32-bit input values and assign them to the terminal set. Next, we run the PRNG with those input values and we obtain output values (which we informally call "initial output"). Then, we check how a small change at the input propagates to the output. To do so, we XOR the original input values with all vectors of 128 bits and Hamming weight equal to one. For each of those modified input values, we again run the PRNG and save the output values (called "modified outputs"). Then, we do a pairwise XOR between the "initial output" and the "modified output" values and we send the result to the Test function (see Eq. (2)).

The goal of this part of the evaluation process is to check the impact of a single bit change. One way to do this is by checking the avalanche criterion (AC), which indicates how many output bits change when a single input bit changes. The ideal generator would have on average 50% of output bits changed for every input bit change. It is possible to enforce an even stricter criterion – SAC, where the demand is that *exactly* 50% of output bits are changed for every input bit change. However, SAC does not necessarily imply better statistical quality. It is possible to construct a simple array of XOR gates which satisfies the SAC and yet has very poor performance as a PRNG. Instead, we choose to evaluate the entropy of the change caused by a single input bit flip.

We developed a test function based on the NIST approximate entropy test [**?**]. This choice was guided by two facts. First, the statistic of this test is the function that always results in a value between 0 and 1, with a higher value corresponding to better randomness. This is very suitable for computing the fitness function. Second, the approximate entropy test is applicable to very short bit sequences.

Note that when computing the fitness function we apply the test function on sequences of 128 bits.

The approximate entropy test uses the estimation of the entropy-per-bit as the test statistic. First, the relative frequencies of all 4-bit and 3-bit patterns are estimated. Based on these frequencies, the entropy is estimated as:

$$Test(output\_value) = \sum_{i(m=3)} \frac{\nu_i}{n} log \frac{\nu_i}{n} - \sum_{i(m=4)} \frac{\nu_i}{n} log \frac{\nu_i}{n}, \quad (2)$$

where ν_i is the number of occurrences of each bit pattern and n is the length of the sequence. Since the higher the value of the Test function the better, we aim to *maximize* the fitness value, where the maximal result equals 128 (scoring 1 on all 128 tests).

Finally, to check how a generator deals with the all-zeros and all-ones input vector, we run it for those values and the result is again sent to the Test function (now there are no pairs of output values to XOR before invoking the Test function). Finally, the resulting fitness function equals:

$$fitness = \sum_{1}^{130} Test(output\_value) - missing * 130, \quad (3)$$

where *output_value* is either an XOR between two output values (as is the case for the first 128 tests) or is a single output value (as is the case for the all-zeros and all-ones input vectors). The variable *missing* represents the number of missing terminals in the generator. The parameter 130 is selected so to enforce that every solution that is missing a terminal is worse than any correct solution (i.e. with all terminals).

Finally, we run our evaluation procedure for n "rounds". The round process is very simple and it just repeats the whole procedure described above, but instead of randomly selecting input variables for every round, we use the output variables from the previous round. By this technique we aim to mimic the mechanism of a PRNG since there the input of iteration $t + 1$ is the output from the iteration t. When using a mechanism with multiple rounds, the cumulative fitness equals the *smallest fitness value* over all rounds. With this criterion, we ensure that the generator behaves at least as good as for the worst evaluation round. Indeed, when we work with only one round, it is hard to predict how the generator behaves when it takes the previously generated values as an input. We experimented with several round number values, but we did not observe any improvement when the number of rounds is greater than two. Consequently, in all our experiments we use two rounds.

4.2 Experimental Results

As one of our goals is to evolve generators that are as small and fast as possible in hardware, that intuitively means we want to restrict the size of the graph we have. Therefore, here we investigate what is the necessary population size and

graph size to evolve PRNGs that pass statistical tests. In all the experiments we set the termination criterion to 20 000 generations. We emphasize that our experiments showed that it is already possible to evolve good PRNGs with a significantly lower number of generations.

The results in Table 1 show the obtained mean fitness values of the best individuals out of 30 runs for every combination of parameters. However, although the average value can help us to deduce which parameter combination works the best, it can also be misleading. Therefore, we additionally give the best obtained values for every combination of parameters.

Table 1. Average/max results for CGP.

Genotype/p_m	1	4	7	10
15	60.8925/74.8561	66.8327/73.576	69.4915/77.7739	69.284/73.6348
30	74.2445/80.5798	72.8384/80.4259	72.5254/76.4385	71.0447/77.7706
50	72.0271/75.471	74.8202/81.6425	76.9252/82.7421	75.1236/80.4513
100	70.9798/76.7845	76.7837/82.3453	78.4326/83.7166	76.057/**84.7544**
200	74.7316/82.4207	77.4329/83.157	79.2718/84.6614	78.9486/82.1012
500	75.3872/82.5726	80.1348/84.3552	80.0269/83.7311	80.3652/83.4072
1 000	78.3797/83.895	79.9121/84.458	**80.6124**/84.1686	79.1203/83.0524

On the basis of the results, we select a genotype size equal to 100 and a mutation probability of 10% as the best performing parameters.

4.3 Evaluation of the Results

After we obtain the results from CGP, we need to test whether they actually pass the statistical tests. In order to do so, we first use a parser that takes as an input the CGP encoding of a solution and produces a C source code as given below. Note that this example of PRNG passes all statistical test, but otherwise we do not impose any other criterion in the choice of PRNG (i.e. we did not select it on the basis of the size or specific operations). Here, *uint* represents the unsigned int variable type and *const* is the value we chose as a constant ($4E2D93A6_{16}$). It is important to note that the full set of NIST statistical tests are applied only in this phase, and not during the evolution since they are relatively slow.

```
void CGP (uint x0,uint x1,uint x2,uint x3,uint*z0,uint*z1,uint*z2,uint*z3)
{ uint y4 = x0 & x1; uint y5 = x2 ^ x3; uint y6 = (y5 >> 1) | (y5 << 31);
uint y7 = p1(y6); uint y8 = x3 ^ y7; uint y9 = p1(y8); uint y10 = y6 ^ y9;
uint y11 = (y9 << 1) | (y9 >> 31); uint y12 = const; uint y13 = p1(y10);
uint y14 = y12 ^ y11; uint y15 = y12 ^ y13;
uint y16 = (y15 >> 1) | (y15 << 31); uint y17 = y10 ^ y16;
uint y18 = p1(y17); uint y19 = y18 >> 1; uint y20 = y18 ^ y4;
uint y21 = p1(y20); uint y22 = y18 ^ y21; uint y23 = p1(y18);
uint y24 = y19 ^ y18; uint y25 = y23 ^ y19; uint y26 = y22 ^ y14;
*z0 = y18; *z1 = y25; *z2 = y26; *z3 = y24; }
```

The source code is then automatically run until it outputs a string of bits of length n, with n equal to 1 000 000. That string serves as an input for the NIST

statistical test suite [**?**]. Only if a generator passes all the tests, we consider it good enough to be used in practice. Our results showed that on average 80% of evolved PRNGs pass statistical tests.

Finally, we implemented our CGP example solution and Lamar in Verilog HDL and then we compared them with the Mersenne Twister generator which is a widely used general-purpose PRNG [**?**]. These algorithms were synthesized using Xilinx ISE14.7 on a Virtex4 xc4vfx100-10ff1152 to draw a fair comparison with the reference implementation of the Mersenne Twister. The implementation result is given in Table 2. With the utilization of 188 slices, our algorithm achieves a maximum working frequency of $286MHz$. The Lamar can reach a working frequency of $43MHz$ with 645 slices. Designs can be parallelized to obtain a higher throughput. Therefore we use throughput per slice as the metric for implementation efficiency. As shown in the table, given the same footprint on FPGA, our CGP implementation could be **90 times faster** than the Lamar and **3 times faster** than the Mersenne Twister design [**?**].

Table 2. Comparison of the hardware implementation results

	Slices	LUTs/FFs/BRAMs	Throughput/slice
CGP	188	317/128/0	**195Mbps/slice**
Lamar [**?**]	645	1045/238/0	2.16 Mbps/slice
Mersenne twister [**?**]	128	213/193/4	65.7Mbps/slice

5 Discussion and Future Work

If comparing our approach with the one followed for the Lamar PRNG [**?**], we see there is a number of important differences. In Lamar, the authors run independent tests (rounds) for a number of times (usually repeated 16 384 times with an explanation that it is experimentally proven to be enough). Our first objection is that the number of repetitions is an additional parameter one needs to tune and there is no background knowledge one can use. Second, since they use independent input values to create new output values, this does not mimic the working of a PRNG, but rather resembles the procedure one would use when testing a number of Boolean functions. Although this does not necessarily lead to bad results, we find it potentially problematic since in general we do not aim to evolve PRNGs that output an extremely short sequence before needed to be reseeded with a new value. Next, Lamar uses operations like addition and multiplication that we believe are not suitable for small and fast PRNGs to be implemented in hardware. Besides those operations, there are also rotations and shift operations where the number of shift positions is huge and therefore results in zero values for shift operations and a number of unnecessary rotations in rotation operations. Considering the hardware implementation on both ASIC and

FPGA, the fixed point multiplier usually has a larger footprint than other logical functions and elementary arithmetic functions, as addition and subtraction. The on-chip DSP slices on FPGAs can be used to implement the multiplication without occupying reconfigurable fabric. However, the number of these dedicated DSPs is relatively small.

Finally, the authors of Lamar consider using it as a stream cipher [**?**], which we believe is unrealistic. Since it is not possible to automatically test all the properties a stream cipher should have (since it would mean a fully automatic cryptanalysis, which is not possible) it is also not possible to write an appropriate fitness function. Therefore, although we do not categorically claim it is not possible to successfully evolve a stream cipher, we state that such a cipher would be good by accident since we cannot evolve it specifically for that purpose.

In future work, we plan to work with a variable number of rounds, where in the beginning we would use a smaller number of rounds and as the evolution progresses we would increase the number of rounds to increase the selection pressure and ensure our PRNGs have higher chances passing a posteriori testing. Next, we could use longer sequences as inputs for the fitness function since then the fitness function will be able to better discriminate PRNGs and consequently, the success rate of PRNGs after the statistical tests will be higher. To obtain such longer sequences, we could use a concatenation of results for several rounds. Additionally, recall Fig. 2 where we said that in this paper we disregard the rectangular B that includes the one-way function. The simplest solution in adding a one-way function would be to simply combine several bits of the output string via an XOR function. If one decides to go with the EC approach, then he could evolve one or more Boolean functions with high nonlinearity [**?**].

We believe more experiments are necessary to determine the limits of the evolutionary approach. Since we aim to find generators that not only pass all the statistical tests, but are also fast and small when implemented in hardware, we could improve the fitness function in an effort to reduce the number of nodes in CGP. Finally, we propose a setting where we believe that the evolutionary approach would display its full benefits. Consider an FPGA board that also has an ARM processor (e.g. Zynq). Then one can put on the ARM the CGP that evolves PRNGs. Such evolved PRNGs can be sent to the FPGA to be partially reconfigured. Therefore, with this approach we would effectively use evolvable hardware [**?**] to increase the security of a system.

6 Conclusions

In this paper, we address the issue of evolving pseudorandom number generators that are suitable for cryptography. The results obtained show that CGP can be used as a viable choice to evolve PRNGs. To define a real-world application for such generators, we discuss the limitations of PRNGs and where they could be used and consequently what properties they need to have. Furthermore, we present a fitness function that in our opinion ensures better results than those used before. We emphasize that we aimed to evolve PRNGs that are extremely

small and fast in hardware and therefore do not rely on expensive operations like multiplication or addition.