Design of a Cryptographically Secure Pseudo Random Number Generator with Grammatical Evolution

Conor Ryan
The Science Foundation Ireland Research Centre for Software, University of Limerick
Meghana Kshirsagar (meghana.kshirsagar@ul.ie)
The Science Foundation Ireland Research Centre for Software, University of Limerick
Gauri Vaidya
The Science Foundation Ireland Research Centre for Software, University of Limerick
Andrew Cunningham
Intel Research and Development Ireland Limited
R Sivaraman
Shanmugha Arts, Science, Technology and Research Academy, Deemed to be University

Research Article

Keywords: Cryptographically Secure, Pseudo Random, Grammatical Evolution

Posted Date: December 14th, 2021

DOI: https://doi.org/10.21203/rs.3.rs-1148385/v1

Design of a Cryptographically Secure Pseudo Random Number Generator with Grammatical Evolution

Conor Ryan[1,*,+], **Meghana Kshirsagar**[1,*,+], **Gauri Vaidya**[1,*,+], **Andrew Cunningham**[2], **and Sivaraman R**[3]

[1]Biocomputing and Developmental Systems Group, Lero, The Science Foundation Ireland Research Centre for Software, Computer Science and Information System Department, University of Limerick, Limerick, V94 T9PX, Ireland

[2]Intel Research and Development Ireland Limited, Leixlip, W23 CX68, Ireland

[3]Department of Electronics Communication Engineering, School of Electrical and Electronics Engineering, Shanmugha Arts, Science, Technology and Research Academy, Deemed to be University, Thanjavur, 613401, India

[*]conor.ryan@ul.ie, meghana.kshirsagar@ul.ie, gauri.vaidya@ul.ie

[+]these authors contributed equally to this work

ABSTRACT

This work investigates the potential of evolving an initial seed with Grammatical Evolution (GE), for the construction of cryptographically secure (CS) pseudo-random number generator (PRNG). We harness the flexibility of GE as an entropy source for returning initial seeds. The initial seeds returned by GE demonstrate an average entropy value of **7.920261600000001** which is extremely close to the ideal value of 8. The initial seed combined with our proposed approach, *control_flow_incrementor,* is used to construct both, GE-PRNG and GE-CSPRNG.

The random numbers generated with CSPRNG meet the prescribed National Institute of Standards and Technology (NIST) SP800-22 requirements. Monte Carlo simulations established the efficacy of the PRNG. The experimental setup was designed to estimate the value for *pi*, in which 100,000,000 random numbers were generated by our system and which resulted in returning the value of *pi* to ***3.146564000***, with a precision up to six decimal digits. The random numbers by GE-PRNG were compared against those generated by Python's rand() function for sampling. The sampling results, when measured for accuracy against twenty-nine real world regression datasets, showed that GE-PRNG had less error when compared to Python's rand() against the ground truths in seventeen of those, while there was no discernible difference in the remaining twelve.

Introduction

Random number generators[1] are classified into two categories, true random number generators (TRNG)[2] and pseudo random number generators (PRNG). TRNGs are able to generate randomness by relying on some physical source, such noise from thermal, atmospheric or radioactive decay sources. They are highly secure due to their reliance on such sources for strong entropy, but suffer due to their reliance on additional devices. On the other hand, PRNGs generate random numbers deterministically based on an initial seed[3] that, ideally, should be hard to predict and secure - if one gets hold of the seed or can influence the generation of the seed, one can predict the PRNG output and the whole system collapses. The main advantages are the rapidity and repeatability of output sequences[4] with relatively small memory requirements. Several approaches[5, 6, 7] have been explored and investigated in the design of efficient PRNGs.

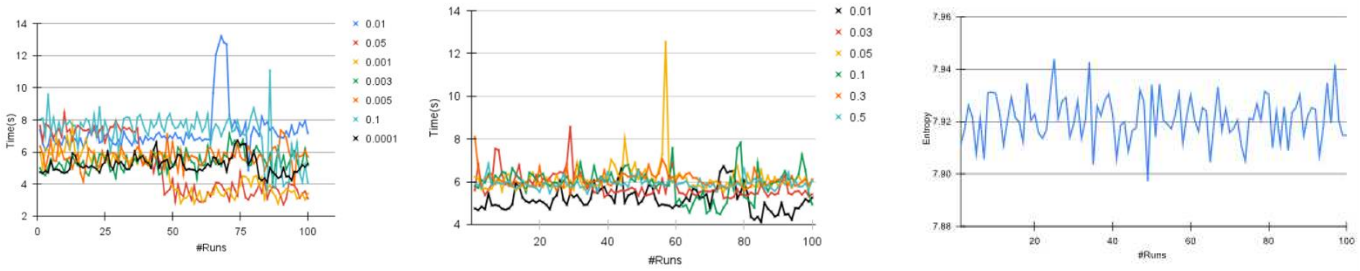
Most existing PRNGs are based on complex mathematical operations[8] such as non-linear congruences[9], linear feedback shift registers[10] and quadratic residuosity, non-quadratic variants[11] and cellular automata[12], amongst others. PRNGs can be divided into two broad categories[13], namely, basic PRNG and CSPRNG. Basic PRNGs are designed for simulations while CSPRNG are designed for cryptography. CSPRNG requirements fall into two groups: first, that they pass statistical randomness tests; and secondly, that they hold up well under serious attack, even when part of their initial or running state becomes available to an attacker. The random sequences are uniquely identified by the following three characteristics: (a) high entropy; (b) no repetition in strings generated; (c) zero or negative correlation.

Grammatical Evolution (GE)[14,15,16,17,18], is a bio-inspired population-based Machine Learning (ML) tool that makes use of Backus-Naur Form (BNF) grammars[19] to generate legal structures for various problem domains. We investigate GE as a potential entropy source, as every evolutionary run is capable of producing a different seed, and hence propose a novel GE-based PRNG and GE-based CSPRNG. It is found that the both GE-based PRNG and CSPRNG can generate random sequences which adheres to meeting all standards of an excellent pseudo randomness.

Results

Obtaining initial seed with GE

To extract a high entropy initial seed, the evolutionary parameters {population size: 3, generations: 5, crossover: 0.0001, mutation: 0.01} for our proposed approach, tuning over 30 runs resulted in an average entropy greater than ***7.904753208160.*** This initial seed is subsequently used to generate random sequences with the *control_flow_incrementor* approach. These parameters have been obtained through experimentation on a variety of different BNF grammars. We obtained an entropy of 7.31 in version 1 with the evolutionary parameters {population: 10, generations: 15, crossover: 0.9, mutation: 0.04} and subsequent modifications to the production rules and evolutionary parameters led to the optimal parameters for our fourteenth version [Supplementary Table S1]. This strongly supports the use of GE as a high-quality entropy source for initial seeds. The BNF grammar serves as a high-quality entropy source, independent of hardware or software requirements. The production rules in grammar can be used to yield arbitrarily large output random sequences, thus making it easily adaptable across diverse applications. These features of GE make the proposed PRNG widely suitable for cryptographic applications.



(a) (b) (c)

Figure 1. Optimisation of evolutionary parameters (a) Variable crossover with mutation rate 0.01; (b) Variable mutation with crossover rate 0.0001; (c) Entropy over 100 runs with constant mutation 0.01 and crossover 0.0001.

The evolutionary parameters were derived after performing a series of experiments over 30 runs. Fig. 1 (a) and Fig. 1 (b) show the tuning of crossover and mutation rates, while Fig. 1 (c) shows the mean entropy of the PRNG with the fourteenth version of grammar. The values for the genetic operators of crossover and mutation were varied in the range [0,1] and experimented on all the fourteen versions of the BNF grammar which resulted in the final values of crossover fixed at 0.0001 and mutation at 0.01. This strongly validates the potential of GE-PRNG as being computationally efficient in terms of speedup and complexity [Supplementary Table S2, S3 and S4].

A PRNG is defined by three key characteristics: (a) high entropy, (b) no repetition in strings generated, and (c) zero or negative

Fitness Function	**Range**	**Objective Function**
F1: Shannon's Entropy	[0-8]	Maximise
F2: Hamming Distance	[0-512]	Maximise
F3: Autocorrelation	[0-1]	Minimise

Table 1. Details of fitness functions used.

correlation. To cover all these characteristics, we designed multi-objective fitness functions to satisfy each of these characteristics. We use the Shannon's entropy as a fitness function F1 to obtain high entropy initial seeds. The fitness function Hamming Distance, *F2*, ensures the maximum number of different bits in the subsequent output sequences. Similarly, the fitness function, *F3*, Autocorrelation, ensures no repetition of input patterns within the same sequence. For two objectives, we use NSGA-II in two setups: (a) *F1* and *F2* and (b) *F1* and *F3*. For three objectives, we use NSGA-II with pymoo[20] as a single setup combining *F1*, *F2* and *F3*. When GE is configured according to Table (1) the output PRNG code will be the one with the maximum amount of entropy, the maximum amount of hamming distance and the minimum amount of autocorrelation within the same random sequence generated as output.

Fig. 2 (a) illustrates the fitness versus generation graph for single-objective fitness function. Fig. 2 (b) and Fig. 2 (c) illustrate the results obtained for two objectives after performing 30 revolutionary runs. The optimal solutions lie in the

region of [7.80, 7.85] for *F1* and for *F2*, the region is [240, 280]. Similarly, the region for *F1* and *F3* is [7.80, 7.85] and [0, 0.09]

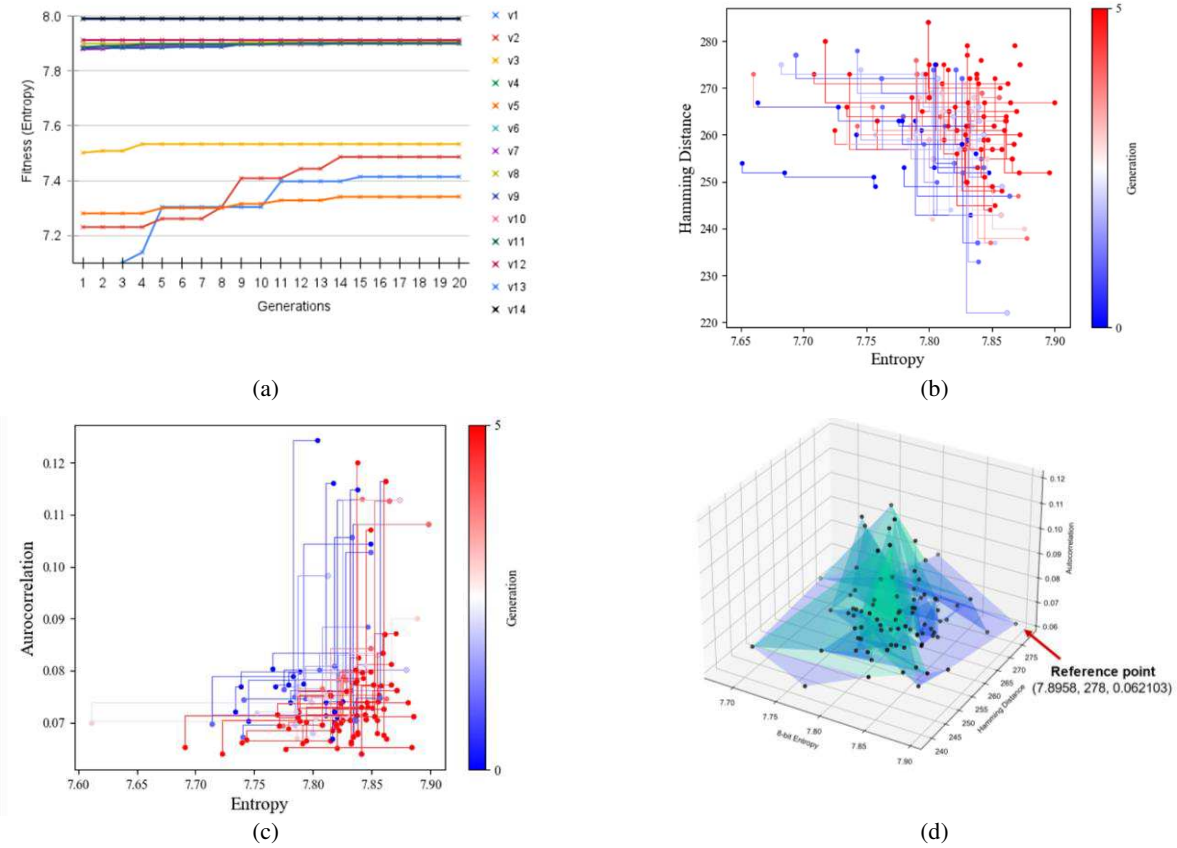


Figure 2. Results from fitness evaluations (a) Single objective: *F1*; (b) Pareto fronts for multi-objective: *F1* and *F2*; (c) Pareto fronts for multi-objective: *F1* and *F3*; (d) Hypervolume for many-objective: *F1, F2, F3*.

respectively. Similarly, the region for the optimal solutions when combining *F1*, *F2* and *F3* was found out to be the same as for *F1* and *F2*.

The optimal solution obtained for three objectives was evaluated with a hypervolume indicator from a reference point [7.8958, 278, 0.06]. The hypervolume indicator[21] is a measure for many-objective optimization where the volume enclosed between a reference point and a Pareto front is calculated. A reference point[22] is the maximum/minimum point obtained in the series or runs for an evolutionary experiment. The smaller the volume, the more optimal are the solutions obtained from the runs. Fig 2 (d) illustrates the hypervolume obtained after combining *F1*, *F2* and *F3* over 30 runs. Similar to multi-objective experiments, the individuals here too lie in the average region {[7.80, 7.85], [240, 280], [0, 0.09]}.

Random sequences with *control_flow_incrementor*

The approach for generating random numbers for constructing CSPRNG is analogous to an incremental counter, where we define a *for* loop to act as an incrementor counter and hence we name this approach *control_flow_incrementor*. This method uses a variable initialized to 0. For each new sequence, the current value of the variable is incremented and appended to the least significant bit (LSB) position of the initial seed. The resulting string is processed using a cryptographically secure hash algorithm such as SHA-256 or SHA-512[23], depending on user requirement to make it secure against brute-force attacks. We can easily extend the size of the strings making it suitable for different applications with simple concatenation functions. For example, if we need a 1024-bit key, two 512-bit strings can be concatenated. As statistically inferred from our NIST experiments, we reseed our CSPRNG after generating every 4000 sequences for our 4096-bit CSPRNG to comply with the CSPRNG properties.

GE has the capability to generate *n* number of useful initial seeds depending upon the choices in production rules and the definition of production rule for pattern formation in the BNF grammar. We store all best individuals obtained during each run of GE in a repository [Supplementary Table S5]. This can be useful in scenarios where we may not be able to obtain the best individuals in the first generation itself, given the inherent randomness of evolutionary computation. In such situations, we employ the mechanism of last in first out (LIFO) for seed extraction from the repository. We conducted an experiment to

verify the potential of updating our entropy pool with unique seeds. In this, 1000 evolutionary runs resulted in 576 unique seeds in a time of 584.02 seconds where each evolutionary run took an average time of 0.5 seconds [Supplementary Table S6].

Security Analysis of GE-CSPRNG

As the proposed BNF grammar is flexible, the production rules can be extended by merely adding additional choices for symbols, thus having the potential to generate $\sum 2^n$ number of unique seeds, where n is the total number of choices for each of the production rules. The BNF grammar proposed in the article, has the potential to generate ***1,514,240*** (as explained in equation (1)) unique seeds with the permutations of the production rules and available choices. Hashing the initial seed with SHA-512 ensures it against session replay attack and brute force attacks.

$$Permutation(BNFGrammar) = ch(1) \times ch(2) \ldots \times ch(8) \tag{1}$$

where ch(1) is the number of choices for rule 1, ch(2) is the number of choices for rule 2, etc.
The number of choices for each of the eight rules in this grammar are 1, 1, 1, 8, 26, 26, 28 and 10. This gives the possible permutations for the grammar as 1,514,240, which equates to the number of unique seeds for our grammar.
Similarly, with this logic, we can generate a larger number of unique initial seeds simply by adding more combinations of choices in the BNF grammar by the inclusion of additional production rules. This feature shows the flexibility of GE-PRNG to be scalable for applications that cater to a large number of users needing unique initial seeds for generating random numbers.
The length of random numbers generated by a PRNG after which the sequences start to repeat themselves is called the period of the PRNG[23]. The period of the proposed PRNG depends upon the limit of variables for the incrementor loop in the system program. For example, if we use **long long int** in C, the period for the PRNG would be **9,223,372,036,854,775,807** as that is the limit for a **long long int** in C.
The experiments were performed by employing Linux version 16.04, libGE version 0.32, using C and Python 3.7 with the following configurations: Intel(R) Core (TM), i3-5005U CPU 2.00GHz 4 GB RAM. The visuals were created in Libreoffice and Matlab R2020a. The mean time taken by the experiment to evolve an individual is **0.02051258087158203s.** The data rate for 4096-bit strings is **120 Mbps**, while the corresponding throughput is **40.98360 Mbps**.

Statistical Complexity

The statistical complexity of the random sequences can be measured with Martın-Platino-Rosso (MPR) statistical complexity[24], which is the product of the randomness of the sequence (entropy) and the distance of the probability distribution of the sequence to the equiprobable distribution. For calculating MPR complexity, we plotted normalized entropy and statistical complexity. For truly random sequences, the value of statistical complexity tends to zero, while the value of normalised entropy tends to one. The equations for the normalised entropy and MPR statistical complexity are as follows:

$$H_{norm} = -\sum_{i=1}^{N} \frac{p_i log(p_i)}{log(N)} \tag{2}$$

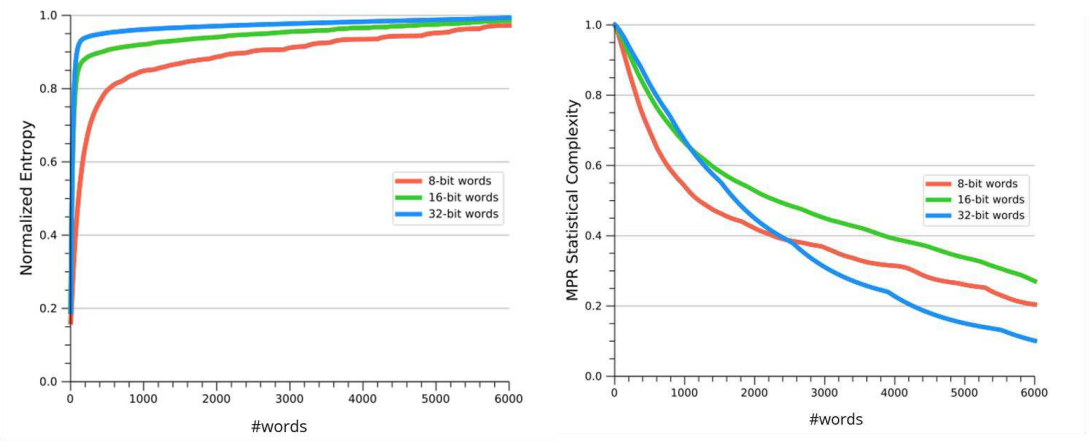
$$Q = Q_0 cos^{-1} \sum_{i-1}^{N} \sqrt{p_i} \times \sqrt{\frac{1}{N}} \tag{3}$$

where,

$$Q_0 = \frac{1}{cos^{-1}\sqrt{\frac{1}{N}}} \tag{4}$$

$$C^{MPR} = H_{norm} \times Q \tag{5}$$

For analysis, the output stream is grouped into 8-bit, 16-bit and 32-bit words for 6400 sequences. The three variants of colors in Fig 3 depict the division of random sequences into three blocks of words, 8, 16 and 32. From the graphs, we can see that as the entropy approaches 1 in Fig. 3 (a), the statistical complexity tends toward 0 in Fig. 3 (b) which indicates good randomness.



(a) (b)

Figure 3. Statistical analysis: (a) Normalized entropy; (b) MPR complexity.

Randomness Tests for GE-CSPRNG

GE-CSPRNG passed the statistical randomness test of output sequences from the NIST[25] Special Publication (SP) 800-22 Statistical Test Suite. This comprises 15 tests, each of which generates a p-value in the range [0, 1] to demonstrate how well the generator holds up under serious attacks. Passing a particular threshold value, α, is indicative of success on a particular test. In all 15 cases, α is defined to be 0.01, which indicates 99% probability of sequences to be random if the test is passed. The random sequences obtained by the *control_flow_incrementor* approach with GE were generated for 4096-bits as modern systems that use RSA[26] for encryption standards usually require keys of 4096 bits. The longest data stream of ***2,048,000*** random bits passed all the 15 tests from the NIST test suite [Supplementary Table S7].

Test	4096-bit strings	
	p-value	Result
Approximate entropy test (block=8)	0.834965	**SUCCESS**
Block frequency test (block=128)	0.445190	**SUCCESS**
Cumulative sums test (forward)	0.372604	**SUCCESS**
Cumulative sums test (reverse)	0.626721	**SUCCESS**
Fast Fourier Transform test (FFT)	0.365919	**SUCCESS**
Frequency test	0.360721	**SUCCESS**
Linear complexity (block=500)	0.694702	**SUCCESS**
Longest runs of ones test	0.313731	**SUCCESS**
Non-overlapping templates test (block=9, 000111101)	0.4857	**SUCCESS**
Overlapping template test of all ones test (block=9)	0.289337	**SUCCESS**
Rank test	0.939590	**SUCCESS**
Runs test	0.133879	**SUCCESS**
Serial test1 (block=16)	0.67609	**SUCCESS**
Serial test 2 (block=16)	0.174800	**SUCCESS**
Universal test (block=6)	0.534872	**SUCCESS**

Table 2. NIST test results: $\alpha = 0.01$.

The results of the tests are given in Table (2). The randomness and entropy of the strings are validated as all the tests are passed by the generated output sequences.

Sampling and Simulations for GE-PRNG

Random numbers are widely used for various simulation and validation of models. We validated the potential of GE-PRNG for sampling and simulations with two different exercises, namely, a Monte Carlo simulation and fitting regression models.

Monte Carlo Simulation for the estimation of pi

Monte Carlo[27] simulations are widely used for ensuring the quality of random numbers generated. They are used to validate whether a given functionality of PRNG successfully achieves its target goal by calculating the value of pi from the random numbers generated by the PRNG. The value of pi is estimated[28] with the help of equation (6) for calculating the radius of a circle.

$$r = \sqrt{(x^2 + y^2)} \tag{6}$$

where x and y are random numbers generated by GE-PRNG. 1,000,000 or more random numbers are generated in the range from 0 to 1 and the value for equation (7) is calculated. If the value of the equation is less than 1, the point is placed inside the circle, i.e., the random numbers pair is valid. If the value of the equation is greater than 1, the point is placed outside the circle and the point is discarded while calculating the value of pi. At the end of the prescribed 1,000,000 runs, the value is obtained by equation (7) and then it is compared with the actual value of *pi*. The closer the estimated value to the actual value of *pi*, the better is the performance of the PRNG.

$$pi = 4 \times \frac{N(pointsinsidethecircle)}{N(totalpoints)} \tag{7}$$

The value of *pi* noted with our observation over 1,000,000 runs is 3.146564000, while the actual value of *pi* is 3.1415926535[29] up to 6 decimal precision which gives a strong validation of randomness.

Fitting the ML regression models with samples generated by GE-PRNG

We used the GE-PRNG to generate samples for the least squares regression approach where we map the relation between a predicted variable and input variable. We generated samples of random numbers from our GE-PRNG for variables such as predicted house prices, annual income for life expectancy, age for cancer, height/weight, etc., covering real, as well as whole numbers for twenty nine benchmark datasets [Supplementary Table S8]. Note that we are not attempting to generate models, rather to use existing models to test the accuracy of the model using our samples.

We fitted the samples with least square regression[30] according to equation (8)

$$y = mx + c \tag{8}$$

Here, y is a predicted variable, x is an input variable, m is the slope of the regression line and c is constant or intercept of the line.

The model was evaluated in terms of accuracy with the error metric root mean square error (RMSE) as shown in equation (9) where y_{pred}, y_{obs} are the predicted and observed values respectively over n observations.

$$RMSE = \sqrt{(y_{pred} - y_{obs})^2} \tag{9}$$

We performed the same experiments with Python's random sampling[31], based on the Mersenne Twister (MT) PRNG, and present a comparative analysis of both the approaches. We calculated the percent error between the predicted values from datasets and predicted values obtained from samples with GE-PRNG and Python's rand() function. Out of 29 datasets, GE surpassed the Python random function in 17 datasets with less error, as observed from RMSE metric, closer to ground truth while in the other 12, there was no discernible difference. This validates the functionality of GE-PRNG as a sampler for simulation models.

Discussion

There have been PRNGs proposed in the literature that incorporate bio-inspired methods, such as GP and Genetic Algorithms (GA)[32]. Koza[33] used GA to transform a seed **J** into a PRNG using the set of operators: {+, -, *, Quot%, Mod%} and a parameter set of {population size: 500, crossover: 0.9 and mutation: 0}. The PRNG proposed by Koza has been tested digitally on software and has passed a Gap square test and Chi-square tests. Poorghanand[34] employed GA using 16 LFSRs with XOR and inverse-XOR to generate high entropy 128-bit random numbers as output and successfully passed all NIST tests. Jhanjharia[35] proposed a Hebbian neural network which used a GA to initialise the network with random numbers. With the evolutionary parameters of mutation 0.05 and keeping the default population size, i.e. 50, it resulted in a 192-bit output and passing Cumulative Frequency, Gap and Chi-square tests. Recent work in the field by Kosemen[36], which is an extension to the work carried out by Koza, uses evolutionary parameters of {popsize: 50, mutation: 0.05} with elitism, resulted in the

design of CSPRNG which passed all NIST tests and produced output within 0.24960s resulting in 49 crossovers over 3 generations.
To design a secure CSPRNG, the potential of GE as an entropy source can be harnessed. GE's rich modularity makes it highly convenient to use alternative search strategies - whether evolutionary, deterministic or any other, to radically change its behaviour by simply changing the production rules from the grammar supplied. Such a flexible approach to Genetic Programming (GP) makes it a robust tool that can be applied to a diverse set of problem domains. It is found that our approach of GE-based PRNG can be used to generate samples for fitting ML regression models. Further GE- based CSPRNG This CSPRNG is applicable across wide domains, such as one-time password (OTP) generation for storage encryption, network encryption, confidential compute, etc. The proposed GE-based CSPRNG is competitive and has the same key functionalities with respect to existing software and hardware CSPRNG [Supplementary Table S9].

Conclusion

We have presented a combined application of GE and our novel approach, *control_flow_incrementor*, for the design of a basic and cryptographically secure pseudo random number generator. We used *control_flow_incrementor* to generate random numbers and GE as the entropy source to return an initial seed. The validations of the CSPRNG with NIST SP800-22 tests indicate the feasibility of potential of GE as a source of initial seeds leading to the efficient construction of CSPRNG. By utilizing the seed repository for reseeding the initial seed, our CSPRNG is able to generate highly uncorrelated random sequences at a faster rate with minimal computational costs, making it highly efficient for securing sensitive data. Monte Carlo simulations were performed to validate the quality of random numbers for sampling with GE-PRNG. Furthermore, extending production rules with additional choices will make it adaptable across a wide range of industrial applications.

Methods

This section discusses the procedure of obtaining initial seeds with GE for the design of PRNG and CSPRNG with GE. The functionality of GE is based upon the generation of structures described by a formal language grammar, typically in the form of a BNF.
At its heart is a GA that searches through the space of syntactically legal structures; the GA individuals, or *genomes*, are variable length binary strings that get mapped onto a *phenotype*, the actual structure being created, which is typically a program or some other, complex structure. The mapping process is guided by the grammar, which, in BNF can be represented as the tuple {S, N, T, P}, where **S** is a start symbol, **T** is the set of *terminals*, that is, items which can appear in the language, **N** is the set of *non-terminals*, temporary items to facilitate the mapping process, and **P**, a set of production rules that expand **S** into a legal structure consisting of only terminals. GE evolves sequences of choices to make, which are subsequently used to perform the expansion and mapping from **S** to **T.** The BNF grammar for generating an initial seed is shown in Fig.4. The mapping process of GE is strongly influenced by the integer value codons as they greatly affect the choices made in the

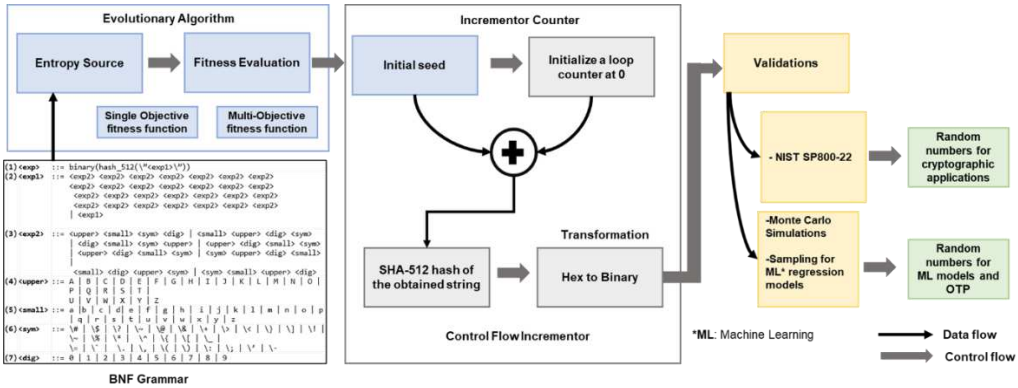


Figure 4. GE-based pseudorandom number generator.

production rules. The GE-based PRNG takes a 512-bit initial seed and combined with the proposed approach, *control_flow_incrementor* as illustrated in Fig. 4 generates random numbers.

The evolutionary process that leads to initial seeds GE can be summarized in the following steps:

1. Initialise evolutionary parameters;
2. Obtain binary string from BNF grammar described in Fig. 4;
3. Evaluate the individuals with a fitness function[37] using Shannon's entropy[38] equation (2) to get the fitness scores;
4. Generate new population for next generation using mutation;
5. Repeat until there is no improvement in fitness[37].

The fitness function as defined in equation (11) is formulated based upon Shannon's entropy function as described in equation (10). Here, ***N*** is the number of characters in a binary sequence and p_i is the frequency of the specific character in the binary sequence.

$$H(X) = -\sum_{i=1}^{N} p_i log_2 p_i \tag{10}$$

$$fitness = \sum_{j=1}^{8} \left(\frac{-\sum_{i=1}^{N} p_i log_2 p_i}{i} \right) \tag{11}$$

The fitness function in equation (11) is the summation of all the *n*-bit entropy values H_1…..H_8, each of which has values in the range [0,1], divided by number of *n*, i.e. number of bits[39]. To eliminate bias while calculating entropy, equation (3) ensures accounting for all valid permutations for bits in a bitwise entropy calculation for the binary string. The example below is the binary notation of the initial seed obtained from GE and calculations on this binary notation yields the entropy value 7.86 for the initial seeds, which is the summation of third column of Table (3) following the steps as discussed below.

0b1010101100001010101100011010100111001110010011000010101001011010101101001010011001110010001010000101000010111010001101010001110110101000010011100111001111011111011100011010111011101110010011001101111111001101111100111001000111110010101100100110101100110100011110001100100010011111011101010100001100101101100111100111101000010000100110000111111001111100000111001010110101001100000100110000111000011010010111001010011000000010111011000001000000100011001100011100010100100101111111011010010000100100010110001010110.

#bits (n)	*H*-value	Entropy (*H*-value/n)
1	H1 = 0.999	0.99
2	H2 = 1.99	0.995
3	H3 =2.99	0.997
4	H4 = 3.98	0.995
5	H5 = 4.86	0.972
6	H6 = 5.92	0.987
7	H7 = 6.83	0.98
8	H8 = 7.62	0.95
∑Entropy		**7.86**

Table 3. Sample fitness score calculation.

Next, a count of each possible value (zero or one) is calculated to get their frequencies, which, in this case, are 0.5 and 0.49 respectively. This gives H_1 as 0.999. Similarly, all the 2-bit possible characters are (00, 01, 10, and 11) and their respective frequencies are 0.24, 0.26, 0.23 and 0.23 respectively. This gives H_2 as 1.99. The final entropy for 2-bit is *H*-value/*n*, i.e 1.99/2, which gives final entropy as 0.995. Dividing these bitwise entropy values by *n* ensures that each permutation of bits has an equal contribution in the final fitness score. We calculate all the *n*-bit entropy values up to 8, and the value for each *n*-bit entropy lies in the range [0-1].

The best individual returned from the evolutionary process is the 512-bit initial seed to the GE-based CSPRNG as illustrated in Table (4). This CSPRNG can be used to generate keys of arbitrary sizes, hence making it suitable for cryptographic applications. The BNF grammar can be varied by modifying the key production rule (4), which controls the number of bits in the resulting string to enable its working as a CSPRNG. This CSPRNG is applicable across wide domains, such as one-

#Generation	Run 1		Run 30	
	Individual (phenotype)	**Fitness (entropy)**	**Individual (phenotype)**	**Fitness (entropy)**
1	Zr;1G7d;Zm^21w=HT0j*2j	7.852622	c9D*dR7?`O3gwY9$T6s{lL3_	7.845765
	?**p**T5**D**1r*}**F7**e-**U5h**&**V**1**l**%**H**0q	7.916708	k5J}Ps~92e_P]cX0eL4@Mf*1	7.865721
	.F1kL5f;Hn&2N9g?^eU5&mH7	7.880614	,bC1dR9^=P3ouO5;b9X#fV4_	7.887011
5	c7F}D5p+}S7g-I5p+H1d%D0i	7.894171	,sG9w3G{#L6koM4@sK3*Jd:2	7.883166
	_pS3**D**5p*+**F7**v-**U5h**^**V5l**{**H**4e	7.904245	U9k=Dj_1~hA0W1p~K1x?Ff}6	7.889589
	@tV1R5r[P6p]Sp<6Q4c;Lr-8	7.890902	<lK7bG9?}F5xuV5%x1W?3n{H	7.882985

Table 4. Sample individuals from the 30 runs.

time password (OTP) generation for storage encryption, network encryption, confidential compute, etc. and regression sampling for ML datasets, etc. For PRNGs, a 16-bit initial seed suffices to make it powerful enough to produce random numbers for samples in various simulations and model fitting applications, following any of the techniques[40]. In our approach, we generate a 512-bit string for CSPRNG, and furthermore, we can extend the length of the seed as desired by random combination of initial seeds returned by BNF grammar.

Data Availability

All data generated or analysed during this study are included in this published article and its Supplementary Information file.

References

1. Park S. K. & Miller, K. W. Random number generators: good ones are hard to find. *Commun ACM.* ***31*****,** 1192-1201 (1988).
2. Stipčević M., & Koç, Ç. K. True random number generators in *Open Problems in Mathematics and Computational Science* (ed. Koç, Ç. K.) 275-315 (Springer, 2014).
3. Vadhan, S. P. Pseudorandomness, *Foundations and Trends in Theoretical Computer Science*, **7,** 1-336 (2012).
4. Preez, V. D., Johnson, M. B., Leist, A. & Hawick, Ken. Performance and Quality of Random Number Generators. *Technical Report CSTN-122* (2011).
5. Divyanjali, A. & Pareek, V. A New Approach to Pseudorandom Number Generation in *2014 Fourth International Conference on Advanced Computing & Communication Technologies ACCT*. 290-295 (2014).
6. Yang YG. & Zhao, QQ. Novel pseudo-random number generator based on quantum random walks. *Sci Rep* ***6,*** 20362 (2016).
7. Murillo-Escobar, M. A., Cruz-Hernández, C., Cardoza-Avendaño, L. & Méndez-Ramírez, R. A novel pseudorandom number generator based on a pseudorandomly enhanced logistic map. *Nonlinear Dyn* ***87***, 407–425 (2017).
8. Bratley, P., Fox, B.L., & Schrage, E.L. A Guide to Simulation. 180-213 (Springer-Verlag, 1983).
9. Eichenauer-Herrmann, J. Pseudorandom Number Generation by Nonlinear Methods, Mathematics of Computation. 60, 375-384 (1993).
10. Tawalbeh, L., Kanakri, W. & Lina, E-B. (2009). Efficient Random Number Generators (RNG) based on Nonlinear Feedback Shift Registers (NLFSR) in *International Conference on Information and Communication Systems (ICICS)* (2009).
11. Blum, L., Blum, M. & Shub, M. A simple unpredictable pseudo-random number generator. *SIAM J. Comput.* ***15,*** 364–383 (1986).
12. Hortensius, P. D., McLeod, R. D., Pries, W., Miller D. M. & Card, H. C. Cellular automata-based pseudorandom number generators for built-in self-test in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, ***8,*** 842-859 (1989).
13. Thottempudi, P., Bhushan, K. N. & Nelakuditi, U. Generation of Cryptographically Secured Pseudo Random Numbers Using FPGA in *AEU- International Journal of Electronics and Communications and technology.* ***5,*** 21-29 (2014).
14. O'Neill, M., & Ryan, C. (2001). Grammatical evolution in *IEEE Transactions on Evolutionary Computation.* ***5*****,** 349-358, (2001).
15. Ryan, C., O'Neill, M. & Collins, J. J. Handbook of Grammatical Evolution,

16. O'Neill, M. & Ryan, C. Grammatical evolution - evolutionary automatic programming in an arbitrary language. *Genetic programming* ***4,*** 1-144 (Kluwer, 2003).
17. O'Neill, M. & Ryan, C. Grammar based function definition in Grammatical Evolution. *GECCO*, 485-490 (2000).
18. Ryan, C. Grammatical evolution tutorial, *GECCO,* 2385-241 (2011).
19. McCracken, D. D., & Reilly, E. D. Backus-naur form (BNF) (2003).
20. Fenton, M., Mcdermott, J., Fagan, D., Forstenlechner, S., Hemberg, E., & Oneill, M. PonyGE2. *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2017).
21. Auger, A., Bader, J., Brockhoff, D., & Zitzler, E. Theory of the hypervolume indicator in *Proceedings of the tenth ACM SIGEVO workshop on Foundations of genetic algorithms - FOGA '09.* 87-102 (2009).
22. Ji, H., & Dai, C. A simplified hypervolume-based evolutionary algorithm for many-objective optimization. *Complexity*, ***2020.*** 1-7 (2020).
23. Lehmer, D. H. Mathematical Methods in Large-Scale Computing Units. *Ann.Comput. Lab.*, *Harvard Univ.*, ***26,*** 141-146 (1951).
24. González, C. M., Larrondo, H. A. & Rosso, O. A. Statistical complexity measure of pseudorandom bit generators. *Physica A: Statistical Mechanics and Its Applications*, ***354,*** 281–300 (2005).
25. Rukhin, A. et. al., A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical Report NIST Special Publication 800-22, U.S. National Institute of Standards and Technology, (April 2010)
26. Hong, J. H., RSA Public Key Crypto-Processor Core Design and Hierarchical System Test Using IEEE 1149 Family, *Ph.D. dissertation*. 322-334 (2000).
27. Strbac-Savic, S., Miletic, A. & Stefanović, H. Z. (2015). The estimation of Pi using Monte Carlo technique with interactive animations in *8th International Scientific Conference "Science and Higher Education in Function of Sustainable Development - SED 2015* (2015).
28. Sen, S. K., Agarwal, R. P. & Shaykhian G. A. Golden ratio versus pi as random sequence sources for Monte Carlo integration. *Mathematical and Computer Modelling,* ***48,*** 161-178 (2008).
29. Nowlan, R. A. (2017) A Short History of π. in *Masters of Mathematics* (2017).
30. Skovgaard, L., Applied regression analysis in *Wiley* ***3,*** (ed. N. R. Draper and H. Smith), 1998.
31. https://docs.Python.org/3/library/random.html
32. Wang, Y., Zhaolong, L., Ma, J. & He, H. A pseudorandom number generator based on piecewise logistic map. *Nonlinear Dyn.* ***83,*** 2373–2391 (2016).
33. Koza, J.R. Evolving a computer program to generate random number using the genetic programming paradigm in *Proc. of the 4th Int. Conference on Genetic Algorithms* (ed. Morgan Kaufmann), 37–44, (1991).
34. Poorghanad, A., Sadr, A. & Kashanipour, A. Generating High Quality Pseudo Random Number Using Evolutionary Methods in *2008 International Conference on Computational Intelligence and Security.* 331-335 (2008).
35. Jhajharia, S., Mishra, S. & Bali, S. Public key cryptography using neural networks and genetic algorithms in *Proceedings of the 2013 6th International Conference on Contemporary Computing IC3. 2013*. 137–142 (2013).
36. Kösemen, C., Dalkılıç, G. & Aydın, Ö. Genetic programming based pseudorandom number generator for wireless identification and sensing platform. *Turkish Journal of Electrical Engineering and Computer Sciences*, ***26,*** (2018).
37. Lima, J. A., Gracias, N., Pereira, H. M. & Rosa, A. (1996). Fitness Function Design for Genetic Algorithms in Cost Evaluation Based Problems in Proceedings of IEEE International Conference on Evolutionary Computation, 207-212 (1996).
38. Shannon, C. E. A Mathematical Theory of Communication. *Bell System Technical Journal,* ***27,*** 379–423, 623–656, (1948).
39. Gholipour, A. & Mirzakuchaki, S. A Pseudorandom Number Generator with KECCAK Hash Function. *International Journal of Computer and Electrical Engineering*. 896-899 (2011).
40. Law, A. M. Statistical Analysis of Simulation Output. *Operations Research*, ***19,*** 983-1029 (1983).

Acknowledgements

This work was supported in part by the Science Foundation Ireland Grant #16/IA/4605.

Author Contributions

C.R. and M.K. conceived the original idea. M.K. and G.V. planned the experiments and G.V. carried out the experiments. S.R. supported G.V. in setting up the experiments. A.C. contributed to the interpretation of results. C.R. and A.C. verified

the findings of this work. C.R., M.K. and G.V. wrote the manuscript. C.R. and A.C. provided critical feedback and shaped the research. C.R. supervised the project. All the authors agreed to the final version of the manuscript.

Competing interests

The author(s) declare no competing interests.

Additional information

Correspondence and requests for materials should be addressed to M.K. and G.V.

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- SupplementaryInformation1.pdf