A Novel Approach for Pseudo-Random Seed Generation

Sacha Pelaiz and Renier Tejera

Complejo de Investigaciones Tecnológicas Integradas CITI, La Habana, Cuba

{spelaiz, rtejera}@udio.cujae.edu.cu

Abstract. Random sequences play an important role in all aspects of Cryptography. All cryptographic systems and protocols are based on secrets and can only be as strong as the random sequence generators they use to generate those secrets. The best cryptographic scheme becomes insecure once its secrets can be predicted or determined. In modern cryptography random sequences are used (1) to generate session keys and initialization vectors for symmetric block ciphers, (2) to generate random values for various digital signature schemes such as DSA and (3) to produce seeds which are used in math routines to get values such as large prime numbers for RSA and also in security protocols. This paper presents the design of RAMG, a pseudo-random seed generator, using a secure symmetric block cipher algorithm. We describe the design principles used for the development of the generator as well as its principal components. We also discuss the idea of using it as a pseudo-random bit generator (DRBG).

Keywords. DRGB, seed, symmetric encryption cipher.

Un nuevo procedimiento de generación seudo aleatoria de semillas

Resumen. Las sucesiones aleatorias juegan un papel importante en todos los aspectos de la criptografía. Todos los sistemas y protocolos criptográficos se basan en el secreto y solo pueden ser tan fuertes como lo sean los generadores aleatorios de sucesiones empleados para generar esos secretos. El mejor esquema criptográfico deviene inseguro una vez se puedan determinar o predecir sus secretos. En la criptografía moderna se emplean las sucesiones aleatorias para: generar llaves de sesión e inicializar vectores para esquemas simétricos de cifrado en bloques; generar valores aleatorios para diversos esquemas de firma digital tales como DSA y ECDSA; generar semillas que se empleen en rutinas matemáticas para obtener valores tales como números primos grandes para esquemas como RSA y ElGamal, entre otras aplicaciones. En este trabajo se describe el diseño de RAMG, un generador seudo-aleatorio de semillas (GSSA) empleando un algoritmo simétrico seguro de cifrado en bloques. Se describen los principios de diseño utilizados para su desarrollo así como sus componentes principales y se analiza la idea de emplearlo como un generador de sucesiones de bits seudo-aleatorios (GBSA).

Palabras clave. Semilla, cifrado simétrico en bloques.

1 Introduction

The generation of truly random sequences, despite being a widely studied topic in cryptography, is a complex issue, particularly because computers in which generation algorithms are implemented are designed to be deterministic. Therefore, a general procedure is to use pseudo-random numbers, which are numbers generated from truly random values and are very difficult to distinguish from the latter.

A pseudo-random bit generator (DRBG) is a deterministic mechanism that processes truly random unpredictable inputs, commonly called seeds, and generates pseudo-random outputs. DRBGs are currently used in many cryptographic applications; some of them have already identified vulnerabilities [5].

A DRBG can be defined as a deterministic function as follows:

$$F:\{0,1\}^k \rightarrow \{0,1\}^n, \quad (1)$$

where F is a one-way function that maps a small input to a much greater length output, and $n >> k$.

If input x is selected uniformly at random, then no efficient algorithm can distinguish

between $F(x)$ and a truly random sequence of the same length. This implies that the output is uniformly distributed and unpredictable without the knowledge of x. The first k bits of the output are used as input for the next iteration, while the remaining $n-k$ bits are the output. Designating the first entry of F as the initial state and each following entry as the next state, one creates a state machine with the transition function F, and then successive application results in a DRBG.

The security of this construction is to keep the internal state secret. The requirement that F is a one-way function ensures that an attacker cannot retrieve the internal state from the outputs.

For the initial state to be secret to a potential attacker, it must be initialized with uniformly distributed secret values (seeds). Also, most DRBGs periodically reset the internal state for two main reasons: to prevent the function F to fall in short cycles and to increase the entropy of the internal state.

DRBGs must fulfill certain requirements [1]. to be considered cryptographically secure. The two fundamental requirements are as follows.

1. Forward security or Progressive resistance. An attacker who can compute the internal state of the generator cannot retrieve previous outputs. This is achieved by ensuring that the transition function is a one-way function.
2. Backward security or Prediction resistance. An attacker who can compute the internal state of the generator cannot predict future outputs. This can be provided if the internal state is periodically reset with data of sufficient entropy.

So, why do we need to design a seed generator? The most common reason for failure of DRBGs in real-world applications is that they overestimate the amount of entropy in the seeds. For example, if a generator have seeds of 128 bits which seem random, but in fact have only 56 bits of entropy, then an attacker can feasibly perform an exhaustive search of the starting point of the DRBG [4]. This is perhaps the most difficult problem to solve in the design of a DRBG.

To generate seeds, one commonly uses some values as entropy inputs; these values are usually obtained from actions taken by a human operator, and also relating to system inputs such as time values and process tables. However, these values usually are poor sources of entropy [3].

An improved variant to obtain suitable inputs is to employ entropy sources associated to unpredictable physical or electronic processes such as thermal noise sources and noise of diodes. However, this implies an additional cost in hardware due to the use of appropriate and specific devices which may not be feasible in different scenarios.

For these reasons we decided to design and implement a software-based pseudo-random seed generator PRSG, named RamG, with good statistical properties to provide the quantity of entropy required for DRBGs.

2 RamG Components

2.1 Entropy Source

According to [9], the source of the entropy input shall be either

– a non-deterministic random bit generator (NRBG),
– a DRBG, thus forming a chain of at least two DRBGs; the highest-level DRBG in the chain shall be seeded by an NRBG or an entropy source, or
– an appropriate entropy source.

In the implementation of RamG we used the function *randomize* as an entropy source. This function is part of the *BigInt* class, developed and implemented by the authors for working with large integers and multi-precision arithmetic. This function was adequate for specific purposes from various functions of the Open Source Library Lidia [6]. It was proved that the *randomize* function generates random sequences with good cryptographic properties for lengths of 256, 384, 512 and 768 bits. Note further that this function can be replaced by any other function to generate pseudo-random sequences of 768 bits.

2.2 Symmetric Algorithm GOST 28147-89

The symmetric block cipher scheme GOST 28147-89 [11] is considered an algorithm with good cryptographic properties. To date, there are no known effective general or specific purpose attacks. This algorithm is originally from the former Union of Soviet Socialist Republics (USSR), and it functions today with certain specifications and employment restrictions according to the symmetric standard of Russia [2].

The algorithm was implemented in the Output Feedback mode (Fig. 1) leading to the function rGOST. Its components are:

- $(K_0,\ldots,K_7)$: 32 bit registers for storing the symmetric key of 256 bits.
- $(S_1,\ldots,S_8)$: 8 S-boxes, each of size 64 bits.
- (R_1,R_2,R_3,R_4): 32 bit registers.
- (R_5,R_6): 32 bit registers which contain two constants C_1 and C_2.
- (CM_3,CM_1): 32 bit adders modulo 2^{32}.
- (CM_2): a Bitwise Exclusive Or adder for 32 bit registers.

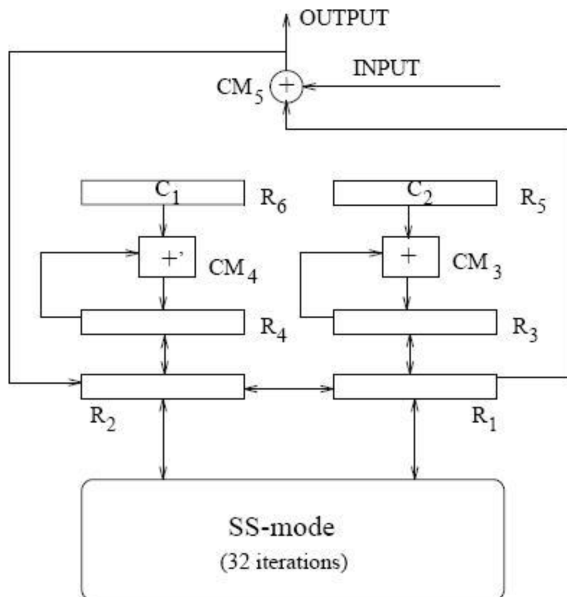

Fig. 1. [7]. GOST algorithm in OFB mode

- (CM_4): a 32 bit adder modulo $2^{32}-1$.
- (CM_5): a Bitwise Exclusive or adder without restrictions of the number of bits.
- (SS)-mode: a Simple Substitution Mode.

2.3 Statistical Tests

To check the quality of the sequences generated by RamG, we used the NIST package of statistical tests proposed in [10] with good results.

Besides, as part of the generation process, we decided to incorporate only 4 of these statistical tests for efficiency reasons, to measure the quality in terms of randomness of the generated sequences. In the following we describe these 4 tests.

2.3.1 Frequency Test (Monobit Test)

Compute the confidence interval:

$$I = np \pm t_\alpha \sqrt{npq}, \qquad (2)$$

Where n is the number of elements of the sequence, $p = 1/2$ is the probability of the output bit to be equal to 0 or 1, and $q = 1 - p = 1/2$.

The sequence is accepted as random if the number of 0's and 1's are both within the computed confidence interval.

2.3.2 Serial Test (Two-Bit Test)

Compute the number of occurrences of (00, 01, 10 y 11) and compute the statistic

$$\chi_b = \frac{(n_{00}-np_{ij})^2}{np_{ij}} + \frac{(n_{01}-np_{ij})^2}{np_{ij}} + \frac{(n_{10}-np_{ij})^2}{np_{ij}} + \frac{(n_{11}-np_{ij})^2}{np_{ij}}, \qquad (3)$$

Where n_{00}, n_{01}, n_{10} and n_{11} are the number of 00's, 01's, 10's and 11's in the generated sequence, respectively; n is the number of elements of the sequence; $p_{ij} = 1/4$ is the probability of occurrence of the two-bit i, j = 0,1.

The computed value χ_b is then compared with the theoretical value $\chi^2_{1-\alpha}$ with $(m-1)^2$ degrees of freedom, ($m=2$). The hypothesis of randomness is rejected if the value of the statistic is greater than the theoretical value.

2.3.3 Run Test

Compute the number of runs (of either zeros or ones) of various length of the sequence using

$$e_i = \frac{n-i+3}{2^{i+2}} \quad (4)$$

Let k be the largest integer i with $e_i \geq 5$, and then compute the statistic

$$\chi_r = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}, \quad (5)$$

Where B_i is the number of runs of 1's of length i, with $1 \leq i \leq k$, and G_i is the number of runs of 0's of length i, with $1 \leq i \leq k$.

The computed statistic is then compared with the theoretical value $\chi^2_{1-\alpha}$ with $(2k-2)$ degrees of freedom. The hypothesis of randomness is rejected if the value of the statistic is greater than the theoretical value.

2.3.4 Autocorrelation Test

Select d with $1 \leq d \leq \left\lfloor \frac{n}{2} \right\rfloor$ and compute

$$A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d}, \quad (6)$$

Where $A(d)$ is the number of bits in the sequence not equal to their d-shifts; s_i is the i element in the sequence with $i = 0,1,\ldots,n-1$; and $\oplus$ denotes the XOR operator. Then compute the statistic:

$$\chi_\alpha = \frac{2A(d) - n + d}{\sqrt{n-d}}. \quad (7)$$

The computed statistic is then compared with the theoretical value $N_{1-\alpha}(0,1)$. The hypothesis of randomness is rejected if the value of the statistic is greater than the theoretical value.

These four tests conform the function *testRandom* used to validate the randomness of the generated sequences. Both the initialization sequences and the outputs sequences must pass these tests to be considered random.

2.4 RamG Functional Scheme

The algorithm to generate 64 bit pseudo-random seeds is as follows.

Algorithm RamG

Input: symmetric key key of 256 bits.

Output: seed sequence Suc_{64} of 64 bits.

do{
  do{
    $Suc_{768} \in_R \left[0, 2^{768} - 1\right] \leftarrow randomize()$
  }**while** (no $testRandom(Suc_{768})$)
  $Suc_{64} \leftarrow rGOST(Suc_{768}, key)$
}**while** (no $testRandom(Suc_{64})$)

In Section 1.1.1, a pseudo-random sequence of 768 bits is generated using the function *randomize*. This step is repeated while the generated sequence Suc_{768} does not pass the four statistical tests. In Section 1.3, a pseudo-random sequence Suc_{64} of 64 bits is generated using rGOST, considering Suc_{768} as the message and key as the symmetric key. This

process is repeated while the sequence Suc_{64} does not pass the four statistical tests.

3 Additional Elements of Randomness

3.1 RamG Reset Process

As shown in Section 2.2, RamG is reset each time required for generating a pseudo-random sequence (seed), i.e., each time it generates a 64-bit seed; it also generates a new 768-bit sequence using the function *randomize*. In other words, the generator is reinitialized with new entropy input in each generation. This mechanism has a disadvantage that if the used entropy source fails from one point without being detected, then a new initialization using this entropy source perhaps cannot provide enough entropy to the generator to operate safely. Therefore, the selection of the entropy source must be correct and the output sequences must be controlled.

Another reset mechanism is to regularly change the key used by the symmetric algorithm GOST 28147-89. This approach is not necessary in practice because the period of the GOST 28147-89 algorithm is approximately 2^{64}, which is undoubtedly sufficiently large.

3.2 S-Box Conformation

A second approach is to construct dynamically and symmetrically key-dependent the 8 S-boxes used by the GOST 28147-89 algorithm. The idea is to conform 8 new and independent S-boxes for each generation process from a set P of 256 permutations of 16 bits. The permutations must be previously generated in a random way and fulfill the necessary security requirements for the S-boxes. The selection process of the 8 permutations or S-boxes among the set of 256 ones implemented in RamG receives as input the sequence of bits of the symmetric key and outputs 8 integers values within 1 and 256 determining the positions of the permutations.

Thus, we have $\binom{256}{8}$ possible manners of conforming S-boxes.

4 RamG as a DRBG

In this section we present the idea of using RamG as a specific DRBG (SDRBG) for the generation of pseudo-random values to be employed as symmetric keys.

As SDRBG, we set a DRBG with the following properties:

- It does not respond to a construction based on a machine of states with a transition function, so there is no relation between the initial values of different iterations;
- It uses a one-way function (symmetric encryption block scheme).

Note that this proposal indeed does not respond to the standard definition of DRBGs.

RamG algorithm as a SDRBG is as follows:

Algorithm RamG
Input: symmetric key key of 256 bits, n number of bits to generate.
Output: seed Suc_n of n bits.

for $i = 0$ to $(n/64)${
  do{
    do{

$$Suc_{768} \in_R \left[0, 2^{768} - 1\right] \leftarrow randomize()$$

    }while (no $testRandom(Suc_{768})$)

$$Suc_n \,\|{=}\, rGOST(Suc_{768}, key)$$

  }while (no $testRandom(Suc_n)$)} .

For example, if we want to generate symmetric keys of 256 bits for GOST 28147-89 algorithm, we have $n = 4$ and the output sequence Suc_n can be interpreted as the result of implementing the function rGOST in the Electronic Code Book mode (ECB) [8].

It is easy to see that this proposal meets the attributes of progressive resistance and

predictions resistance. The first attribute is achieved by the use of a one-way function: the symmetric encryption algorithm GOST 28147-89.

The achievement of the second one is due to the constant reset of the generator. Moreover, the internal states of the generator are not related to each other, i.e., they are independent, so the compromise of a state does not provide any information to determine or predict an earlier or later state.

5 Conclusions

This paper describes the design of RamG, a pseudo-random seeds generator, using the secure symmetric encryption algorithm GOST 28147-89 as a one-way function. We obtained very good results applying the NIST package of statistical tests and proposed two new approaches to provide more entropy to the generation process. Besides, we presented a strategy to employ RamG as a DRBG with specific characteristics to generate symmetric keys, in particular, for the symmetric encryption algorithm GOST 28147-89.

References

1. **Barak, B. and S. Halevi (2005).** A model and architecture for pseudo-random generation with applications to /dev/random*." CCS´05 Proceedings of the 12th ACM conference on Computer and communications security:* 203-212
2. **GOST28147-89 (1989).** National Soviet Bureau of Standards. Information Processing Systems. *Cryptographic Protection. Cryptographic Algorithm*.
3. Gutmann, P. Software Generation of Practically Strong Random Numbers. *7th USENIX Security Symposium, San Antonio, Texas, USA*
4. **Kelsey, J., B. Schneier, et al. (1999).** Yarrow 160 Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. *LNCS 1758: 13-33*
5. **Kelsey, J., B. Schneier, et al. (1998).** "Cryptanalytic Attacks on Pseudorandom number generators." *FSE 1372(Springer): 168-188*
6. LiDIA. A library for computational number theory., Available from http://www.informatik.tu-darmstadt.de/TI/LiDIA/Welcome.html
7. **López, J. C. and R. Monroy (2008).** Formal Support to Security Protocol Development: A Survey. *Computación y Sistemas 12*
8. **Menezes, A., P. v. Oorschot, et al. (2001).** Handbook of Applied Cryptography, CRC Press New York
9. **NIST (2007).** Recommendation for random number generation using deterministic random bit generators. NIST Special Publication 800-90A.
10. **NIST (2010).** A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications (Revised). NIST Special Publication 800-22.
11. **Pieprzyk, J. and L. Tombak (1994).** Soviet Encryption Algorithm.

Sacha Pelaiz received his M.Sc. at Havana University in 2006 and graduated in Mathematics in 2001.

Renier Tejera received his M.Sc. at Instituto Superior Politécnico José Antonio Echeverría, Havana, Cuba in 2011 and graduated in Computer Sciences in 2005 at Havana University.

Article received on 11/10/2012; accepted on 09/01/2013.