Int. J. of Computers, Communications & Control, ISSN 1841-9836, E-ISSN 1841-9844
Vol. V (2010), No. 4, pp. 483-489

Stream Ciphers Analysis Methods

D. Bucerzan, M. Crăciun, V. Chiş, C. Raţiu

Dominic Bucerzan, Mihaela Crăciun, Violeta Chiş
"Aurel Vlaicu" University of Arad
Faculty of Exact Sciences
Department of Mathematics-Informatics
România, 310330 Arad, 2 Elena Drăgoi
E-mail: dominic@bbcomputer.ro, qbt@rdslink.ro, viochis@yahoo.com

Crina Raţiu
DARAMEC srl, Arad
România, Sofronea FN
E-mail: ratiu__anina@yahoo.com

Abstract: The purpose of this paper is to present and to discuss analysis methods applied in symmetric cryptography, especially on stream ciphers. The tests were made on some algorithms and also on the personal symmetric cryptographic algorithm, HENKOS, based on a pseudorandom number generator. The test confirms that the algorithm appears to be secure and fast. The paper describes first the main parts of the cryptosystem, its implementation and different analysis methods. The code is written in the C/C++ language. The software application and the tests applied were processed on a PC computer. The quality analysis presents the results of many classical statistical tests, comparing some algorithms based especially on pseudo random number generators. The tests use standard sequence of 12.5 MB resulted from some test generators. The main part of the work presents selected results for the most important statistical tests like: FIPS 1401, FIPS 1402 , ENT tests, Diehard battery of tests, NIST Statistical Test Suite. The final question is: are these tests enough to certifie the quality of a tested algorithm?
Keywords: stream cipher, synchronous stream cipher, pseudorandom number generator (PRNG), performance analysis, statistical tests.

1 Introduction

Stream ciphers are an important class of encryption algorithms. They encrypt individual characters (usually binary digits) of a plaintext message one at a time, using an encryption transformation which varies with time. Various design methods where proposed for stream ciphers and the specialists proposed many analysis methods. However, the reality is that in the literature we can find relatively few fully-specified stream cipher algorithms. One possible explanation can be the fact that many stream ciphers used in practice tend to be proprietary and confidential.

A stream cipher generates what is called a keystream (a sequence of bits used as a key). Encryption is accomplished by a simple operation combining the keystream with the plaintext, usually with the bitwise XOR operation. Stream ciphers can be either symmetric-key or public-key. The focus of this chapter is symmetric-key stream ciphers. A stream cipher generates successive elements of the keystream based on an internal state. This state is updated in essentially two ways: if the state changes independently of the plaintext or ciphertext messages, the cipher is classified as a synchronous stream cipher. By contrast, self-synchronizing stream ciphers update their state based on previous ciphertext digits.

For the synchronous stream ciphers some properties are mandatory.

(i) synchronization requirements. The sender and the receiver must be synchronized – using the same key and operating at the same position (state) within that key. If synchronization is lost due to ciphertext digits being inserted or deleted during transmission, then decryption fails and can only be restored through additional techniques for re-synchronization.

(ii) no error propagation. A ciphertext digit that is modified (but not deleted) during transmission does not affect the decryption of other ciphertext digits.

(iii) active attacks problem. As a consequence of the synchronization requirement, the insertion, deletion, or replay of ciphertext digits during an attack causes loss of synchronization, and offer the possibility to be detected by the attacker. An active attack offers the possibility to make changes to selected ciphertext digits, and find out what affect these changes have on the plaintext. This conclusion proves that the data origin authentication and data integrity must be assured by additional mechanisms.

In a synchronous stream cipher a stream of pseudo-random digits is generated independently of the plaintext and ciphertext messages, and then combined with the plaintext (to encrypt) or the ciphertext (to decrypt). In the most common form, binary digits (bits) are used, and the keystream is combined with the plaintext using the exclusive or operation (XOR). This is called a binary additive stream cipher.

Another approach uses several of the previous N ciphertext digits to compute the keystream. Such schemes are known as self-synchronizing stream ciphers or asynchronous stream ciphers. The idea of self-synchronization has the advantage that the receiver will automatically synchronize with the keystream generator after receiving N ciphertext digits, making it easier to recover if digits are dropped or added to the message stream. Most stream cipher designs are for synchronous stream ciphers. For further details see [5].

2 Design of a Stream Cipher

The design of a new stream cipher involves some important goals:

– To deduce the internal state from the result should be impossible,

– There should be no short cycles,

– It should be cryptographically secure,

– It should be easy to implement,

– The code should be optimized for speed,

– To create as much confusion and diffusion as possible.

I tried to achieve the same goals in designing a new stream cipher named HENKOS (see [1], [2]). This cryptosystem is a symmetric synchronous stream cipher encryption system designed for a software implementation.

After many efforts, here are the results:

– An easy to implement algorithm,

– A cryptographically secure algorithm (proven by statistical tests),

– A very fast algorithm: a 50 megabytes file is encrypted / decrypted in less then one second,

– A fast pseudorandom number generator: a 12,500,000 byte stream needs 0.30 sec for C/C++ code.

This cryptosystem uses a binary additive stream cipher and two types of keys:

– a short-term key named data key (*DK*) with a fixed length of 1024 bytes that is an input in the keystream generator. This key can be generated with a PRNG (not necessarily a cryptographic secure PRNG) or can be an ordinary file, if PRNG is not available.

– a long-term key named master key (*MK*) with a fixed length, which contains 1024 numbers, used to mix the data key and the internal state of the keystream generator. This key must be generated with a true RNG (hardware).

– If during the transmission an attacker intercepts the encrypted data, it is not possible to decrypt the ciphertext correctly without having the master key, because there is a very large number of possible combinations of decrypted ciphertext.

Every attempt to find the master key produces a different plaintext, including the one with the same numbers but the changed order of the numbers in the key affects the decryption process.

2.1 Index keys generation

In this section of the algorithm the master key *MK* is transformed into two index keys *MKS* and *MKT* in two steps. Two functions **Sum** and **Inv** are used: the first one is an additive function and the second one produced a sort of symmetrical figures of number transformation.

The function **Sum** is **Sum** : $\{1,2,\ldots,1024\} \rightarrow \{1,2,\ldots,1024\}$,

$$\mathbf{Sum}(i;MK) = \sum_{j=0}^{i} MK(j) \; modulo \; 1024 \; .$$

The function **Inv** is **Inv** : $\{1,2,\ldots,1024\} \rightarrow \{1,2,\ldots,1024\}$, $\mathbf{Inv}(i) = i^* \; modulo \; 1024$, where i^* is the number obtained by writing the digits of the number i in reverse order. The index keys *MKS* and *MKT* are:

$$\begin{aligned} Step\ 1:\ MKS(i) &= \mathbf{Sum}(i;MK),\ i \in \{1,2,\ldots,1024\}\ , & (1)\\ Step\ 2:\ MKT(i) &= \mathbf{Inv}\big(MKS(i)\big),\ i \in \{1,2,\ldots,1024\}\ . & (2) \end{aligned}$$

The transformation has two targets:

- Not to use the original *MK* key directly in the process,
- To create confusion and diffusion for master key.

Keystream generation transform the *DK* key to obtain the real K key for encryption using two functions: the first one is the essential function in this algorithm the "switch function" **Sw** and the second function **Ad** is an additive one:

$$(\mathbf{Sw}) : DK(j) \leftrightarrow DK(k),\ \text{where } j = MKS(i) \text{ and } k = MKT(i) \text{ for } i \in \{1,2,\ldots,1024\}\ , \quad (3)$$

$$(\mathbf{Ad}) : DK(i) = DK(i) + DK(i+1) \; modulo \; 256,\ i \in \{1,2,\ldots,1023\} \text{ and } DK(1024) = DK(1024) + DK(1)\ . \quad (4)$$

These functions create a totally changed image of the data key *DK*. After these two transformations we obtain DK1; the new key is the input for transformations (3) and (4) and the process will be repeated 64 times:

$$DK \rightarrow DK1 \rightarrow DK2 \rightarrow \ldots \rightarrow DK63 \rightarrow DK64\ . \quad (5)$$

To obtain the keystream bytes $K(i)$ of the final K key, the last operation is:

$$K(i) = \big(DK64(i) + DK64(i+1)\big) \oplus DK64(i) \text{ for } i \in \{1,2,\ldots,1023\};$$
$$K(1024) = \big(DK64(1024) + DK64(1)\big) \oplus DK64(1024)\ . \quad (6)$$

The encryption / decryption process will transform a plain text P of 1024 bytes into a cipher text C of 1024 bytes by using the encryption key K and the function $\oplus$:

$$C(i) = P(i) \oplus K(i)\ .$$

For every stream of 1024 bytes of plain text, another K key wil be used. The new encryption K key will be obtained by the algorithm with the last values of DK as input and the operations described in (4), (5) and (6) will be effectuated one time. This sequence will run until the plain text is finished for one session. Remarks:

- The confusion and the diffusion of the bits are given specially from (3) and (4),
- The data key for the next session we have generated with the same algorithm.

3 Quality analysis

The quality of a stream cipher is measured performing statistical tests. These tests FIPS 140–1, FIPS 140–2, ENT tests, Diehard battery, NIST Statistical Test Suite (a statistical test suite for testing the pseudo–random number generators used in cryptographic applications) were performed on large ciphertext samples of 12.5 megabytes.

3.1 FIPS 140–1/FIPS140–2 Test

FIPS statistical tests contain the Monobit Test, the Poker Test, the Runs Test and the Long Run Test. The following tests are based on performing a pass/fail statistical test on 5000 sequences of 2500 bytes each. In my results the well known generators SHA–1 and CCG together with the new HENKOS pass FIPS 140–1 in proportion of 100%. SHA–1 passes FIPS 140–2 in proportion of 99.6%, CCG in proportion of 99.4% and HENKOS in proportion of 99.64%. For these statistical tests, even if the generators present good statistical properties this isn't a guarantee that the algorithm is good for cryptographic purposes.

3.2 DIEHARD Statistical Tests

The next set of tests was designed to identify weaknesses in many common non–cryptographic PRNG algorithms. These tests analyze a single large file from the output of the generator of 11 megabytes or more (see [3]). The battery of tests include: birthday spacing test, overlapping 5–permutation test, binary rank test 31×31, binary rank test 32×32, binary rank test 6×8, bitstream test, opso, oqso and DNA tests, count–the–1's test on a stream of bytes, parking lot test, minimum distance test, 3Dspheres test, etc.

Majority DIEHARD tests return a $p-$value, which should be uniform on [0,1) if the input file contains truly independent random bits. Those $p-$values are obtained by $p = F(X)$, where F is the assumed distribution of the sample random variable X. When a bit stream really fails, it get $p-$values of 0 or 1 (or close to 0 or 1) to six or more places.

For SHA–1 generator we have 2 $p-$values very close to 1, and for Cubic Congruent Generator we have 28 $p-$values near to 1. For HENKOS we don't have $p-$values close to 0 or 1.

3.3 ENT Tests

ENT applies various tests to a sequence of bytes stored in files and reports the results. The program can be used to evaluate pseudorandom number generators for encryption and compression algorithms. It calculates entropy, optimum compression, chi–square distribution, arithmetic mean, Monte Carlo value for *pi* and serial correlation coefficient. HENKOS obtained good results.

3.4 NIST Statistical Test Suite

The package includes statistical tests for: frequency, block frequency, cumulative sums, runs, long runs, Marsaglia's rank, spectral (based on the Discrete Fourier Transform), nonoverlapping template matching, overlapping template matching's, Maurer's universal statistical, approximate entropy, random excursions (due to Baron and Rukhin), Lempel–Ziv complexity, linear complexity, and serial.

The NIST framework, like many tests, is based on hypothesis testing.

– State your null hypothesis. Assume that the binary sequence is random.

– Compute a sequence test statistically. Testing is carried out at the bit level.

– Compute the $p-$value which must be less than 0.01, otherwise, failure is declared.

The Cubic Congruential Generator fails Frequency, Cumulative Sums, Runs, Aperiodic Template at 11 from 284 templates, Approximate Entropy test, serial and Lempel–Ziv test and Micali generator has 3 fails at Aperiodic Template. HENKOS and BBS pass all this tests.

4 Security Analisys

What is a secure stream cipher? That is a question with no definitive answer, but I can make some assumption on that subject. The entire test package presented here can eventually reveal weaknesses but, even if the ciphers pass with good results, that is not a guarantee of its security. That does not make it fail proof.

Cryptographers consider that there are two main conditions for the security of a stream cipher with a $k-$bit key.

- The attacker should not be able to predict future keystream generated by the cipher in any conditions: recovering the secret key, recovering the internal state of the cipher at some point, or otherwise. The attacker can obviously test all possible secret keys, so the complexity of a brute force attack (requiring at most $2k$ executions of the algorithm) gives a performance baseline to which any alleged attack should be compared to.

- The attacker should not be able to recover the cipher's key or internal state from the keystream. Cryptographers also demand that the keystream be free of even subtitle biases that would let attackers distinguish a stream from random noise, and free of detectable relationships between keystreams that correspond to related keys or related nonce. This should be true for all keys (there should be no weak keys), and true even if the attacker can know or choose some plaintext or ciphertext.

Like other attacks in cryptography, stream cipher attacks can be certificational, meaning they aren't necessarily practical ways to break the cipher but they indicate that the cipher might have weaknesses.

Securely using a secure synchronous stream cipher requires that one never uses the same keystream twice; that generally means that a different nonce or key must be supplied to each invocation of the cipher. Application designers must also recognize that most stream ciphers don't provide authenticity, only privacy: encrypted messages may still have been modified in transit.

Stream Cipher	Creation Date	Speed (cycles/ byte)	bits			Attack	
			Key Length	Init. vector	Internal State	Best Known	Comp. Compl.[1]
$A5/1-2$	1989	Voice	54	114	2^6?	Active	2^{40}
FISH	1993	Quite Fast	Huge	?	?	K-p A[2]	2^{11}
Grain	≤ 2004	Fast	80	2^6	160	Key D[3]	2^{43}
HC-256	≤ 2004	2^2?	2^8	2^8	2^{15}	?	?
HENKOS	2005	7.8	2^8-2^{10}	2^8	2^{10}	?	2^{1024}
ISAAC	1996	$2.38-4.69$	$40-2^8$	N/A[4]	8288	2006 WIS[5]	5×10^{1240}
PANAMA	1998	2	2^8	2^7?	1216?	2001 HC[6]	2^{82}
Rabbit	2003	$3.7-9.7$	2^7	2^6	2^9	2006 N/A	2006 N/A
RC4	1987	Impressive	$40-2^8$	2^3	2064	Key D	$2^{13}-2^{33}$
SEAL	1997	Very Fast	?	2^5?	?	?	?
SOBER-128	2003	?	$\geq 2^7$	?	?	Mes. Forge	2^6
Trivium	≤ 2004	2^2-2^3	80	80	288	Brute Force	2^{135}

Table 1: Comparison of some well known stream ciphers

5 Performances Analysis

5.1 Testing Platform

For testing we select only one platform among many other possible platforms, on the criteria of disponibility and reproductibility of measurements. There are also results, reported in papers, that make possible only a relative comparison between the performances of different algorithms.

5.2 Performances Measurement

The performance algorithms are mesuread by a special program, written in Visual C and based on reading the processor clock before and after the calls of the main phases implementing functions, using the routines get_start_time and get_stop_time written in ASM. The result is calculated as the difference between the two clock readings, minus the additional time consumed with the calls of the clock reading routines.

In the HENKOS case the CPU time for K keys generation is 7.8 cycles/byte, and the encryption process performs 181 megabytes/second. Comparing it to well–known algorithms and the algorithms tested in the Estream Project like CryptMT, Dragon or Salsa20, which are among the top algorithms in the final list, my results are good enough (for details see [6] and [4]).

5.3 Implementation details

The performance was measured reading the processor clock cycles using the RDTSC instruction.

5.4 Comparison of performances

In the real life, in very few cases the authors reveals all the details and the performances of the ciphers. In table 1 there are some results for some well known stream ciphers, [8].

6 Conclusions

There are a lot of stream ciphers used in cryptography because of the speed, but in this case nobody tried a standardisation like in block cipher area. The European Union–based NESSIE project [7], which was aimed at evaluating the security of various cryptographic primitives, did not recommend any stream ciphers in their report.

The performances and quality analysis on cryptographic stream ciphers algorithms are an ambitious goal for all the designers of algorithms. In majority of cases there is no proof of the behaviour of the new cipher, but it's possible to verify the quality by performing statistical tests, and also to measure the performances of implementation and the speed by software means.

In the future, new stream ciphers will appear so that new methods for analysis will be a permanent preoccupation for the cryptographic community.

Bibliography

[1] Bucerzan D. and Gheorghiţă M., HENKOS – *A New Stream Cipher: Performance Analysis*, WARTACRYPT '04 The $_4{}^{th}$ Central European Conference on Cryptology, Bedlewo, Poland, July 2004.

[2] Bucerzan D., *A Cryptographic Algorithm Based on a Pseudorandom Number Generator*, SYNASC'08, Timişoara, October 2008.

[3] Marsaglia G., Diehard Statistical Tests, `http://stat.fsu.edu/pub/diehard/`

[4] Matsumoto M., Saito M., Nishimura T. and Hagita M., CRYPTMT Stream Cipher Version 3, eSTREAM project, `http://www.ecrypt.eu.org/stream/`

[5] Schneier B., *Applied Cryptography*, J. Wiley & Sons Inc, (second edition), 1996.

[6] ***, eSTREAM, `http://www.ecrypt.eu.org/stream/`

[7] ***, NESSIE European Proiect, `http://www.cosic.esat.kuleuven.be/nessie/`

[8] ***, `http://www.answers.com/topic/stream-cipher`

Dominic Bucerzan (b. May 17, 1956) received his M. Sc. in Information Technology from "Aurel Vlaicu" University of Arad, Romania and a PhD in Economic Cybernetics from the "Bucharest Academy of Economic Studies" (2005), with a paper in the field of Information Security. Currently he works as a lecturer in informatics at the Department of Mathematics-Informatics, Faculty of Exact Sciences, "Aurel Vlaicu" University of Arad, România. His current research interests include aspects of IT Security and Cryptography. He is author or co-author of 4 books and more than 45 papers and participated in 35 conferences and workshops.

Mihaela Crăciun (b. March 10, 1972) received her Master of Science in Information Technology from "Aurel Vlaicu" University of Arad, România. At present she is a candidate for a PhD in Computer Science at "Politehnica" University of Timişoara, România. Her current research is focused on Decision in Enterprise Analysis. She published articles in her field of interest.

[1] Computational Complexity
[2] Known-plaintext Attack
[3] Key Derivation
[4] Not Available
[5] Weak Internal State
[6] Hash Collisions