Annales UMCS Informatica AI 8(1) (2008) 5-13
10.2478/v10065-008-0001-9

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Stream security particularities in Java

Michał Chromiak*, Zdzisław Łojewski

Institute of Computer Science, Maria Curie-Skłodowska University,
pl. M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland

Abstract

Regarding numerous threats connected with sending and storing confidential data, there is a problem of assuring the efficiency. As an answer to those needs, we discuss the SUN's Java Virtual Machine mechanism provided to assure security to a single object. Thanks to the mechanism of serialization in Java, it is possible to provide secure solution. In this paper, we compare the efficiency of algorithms such as DES, Blowfish, AES, RSA and ECC as means of securing serialization of an object.

1. Introduction

Concerning increasing threats including storing and transferring confidential data, there is a growing demand for secure solutions. The Java Virtual machine provided by Sun uses most of the modern ways of securing the transmission of data. As every object can be sent as a serialized stream, there are mechanisms that allow its encryption. Since the amount of data being encrypted rises, choice of an efficient algorithm becomes crucial. In this paper we examine some particularities of the encryption of objects graph[1] byte stream. We have covered some areas that can also be found in [1].

1.1. Main goals

The main objective of the paper is to present the results of tests made as a part of a larger project. The project goal is to provide platform independent security architecture for sensitive data transfer. The conclusions from this paper are being used for the project development.

The following questions are to be answered:

1. What are the cryptographic aspects of Java streams?

*Corresponding author: *e-mail address*: mchromiak@hektor.umcs.lublin.pl

[1]All objects referenced directly or indirectly from within the serialized object and implementing the tagging `Serializable` interface.

2. What is the time efficiency regarding stream cryptography using symmetric algorithms?
3. What are the differences comparing symmetric and asymmetric cryptography algorithm implementations in Java?

This paper does not intend to discuss the accuracy of the time measurement. Even though it is possible that some discrepancy will occur, it is believed that this will be eliminated by more precise hardware timing sources that will hopefully be soon designed and utilized by operating system calls.

2. Testing environment

The choice was the linux because of the Windows JVMcalls to System.nanoTime() is implemented using the *QueryPerformanceCounterQuery PerformanceFrequency API* (if available, else it returns *currentTimeMillis*$*10^6$). As its default 10 ms timer interrupt period can be modified by application programs using the *timeBeginPeriod/timeEndPeriod API*'s there is no guarantee that a wanted period will be supported, moreover its accuracy has been questioned in some reports [2]. The testing procedure for all tests has been performed on 100 different objects of approx. 5KB each. The testing architecture was as follows:

- Hardware: Intel®Core™ 2 Duo E6600 @ 2.40Ghz, 2GB RAM
- Software: Linux (2.6.23.9-85), Fedora 8, Sun JDK 1.6.0_03

3. Byte distribution in streams

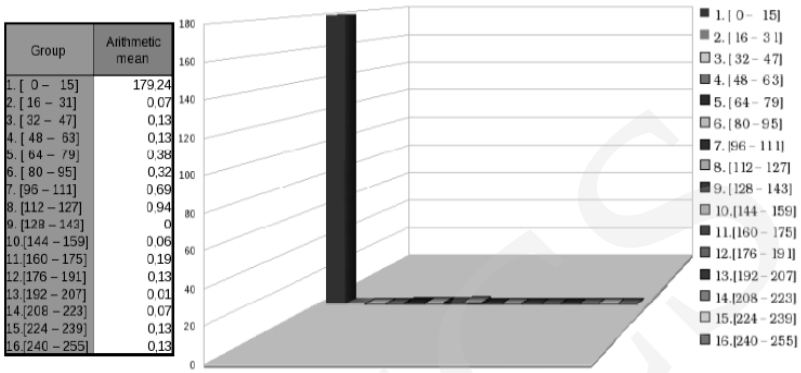
As a security must, the information sent throughout the potentially hostile environment e.g. the Internet, should be as secure as it is possible regarding the efficiency aspect. One of the security bases is the distribution of data structure along the transfer domain (i.e. all possible values of the information byte). At the basic level, the byte distribution is the issue. That is why in this paper we present the Java basic stream mechanism analysis.

3.1. The regular stream distribution

To find the distribution of byte values in the transmitted stream an external class was made. Its purpose was to check each byte of data from the stream and add it to an appropriate analyzing structure. As an obvious way of representing the results, is to treat the byte values of a stream as 0 to 255 codes[2].

As a number of serialized objects were sent as a stream the statistics displayed in Figure 1 were obtained.

[2]Just as it takes place in the case of ASCII codes.



Group	Arithmetic mean
1. [0 – 15]	179,24
2. [16 – 31]	0,07
3. [32 – 47]	0,13
4. [48 – 63]	0,13
5. [64 – 79]	0,38
6. [80 – 95]	0,32
7. [96 – 111]	0,69
8. [112 – 127]	0,94
9. [128 – 143]	0
10.[144 – 159]	0,06
11.[160 – 175]	0,19
12.[176 – 191]	0,13
13.[192 – 207]	0,01
14.[208 – 223]	0,07
15.[224 – 239]	0,13
16.[240 – 255]	0,13

Fig. 1. The pure secured stream byte distribution

It can be clearly seen that the byte distribution provided is unacceptable regarding the frequency analysis attacks that can take advantage of this stream structure. This behaviour is not permanent but the tendency for such a disproportion holds across multiple tests. From the test results, it can also be claimed that the majority of the values (i.e. over 80%) from the first group obtain zero value.

3.2. Entropy provided by cryptographic mechanisms

As the above results for pure stream proves that it appears to be insecure, some tests have been performed on securing it by using the CipherOutput/ InputStream classes provided by the JCA/JCE architecture. For this purpose the Bouncy Castle [3] provider has been used. The same tests have been performed for the streams secured with the DES, TripleDES, Blowfish and AES algorithms as the most widespread. The acquired results displayed in Figure 2 provided much more reliable byte distribution regarding the security aspect.

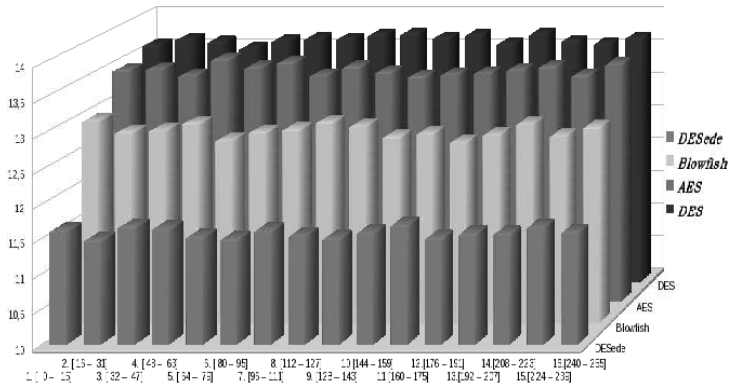


Fig. 2. The secured stream byte distribution

As the figures are not as clearly visible as for the pure stream some statistical analysis needs to be applied. For the perfect byte distribution, the standard deviation should be null and each byte value in this case would have the average value of one out of 256. This of course is just a theory, because the results (Figure 3) for the encryption of a stream provide much better byte distribution than the pure stream did, but not to such an extent. Some tests have also been made on the byte distribution provided by the RSA. Of course, it needs to be mentioned that the RSA is not a kind of cipher used for a stream cryptography. Because of its particularities, the length of a message cannot be longer than the key length. Therefore for the key of 1024 bits the message can be, as in the examined case, maximum 117 Bytes[3]. To avoid the byte distribution provided by the message itself the stream bytes were all zeros. RSA is a cryptosystem used just for symmetric algorithms key distribution, therefore it is combined with one of the symmetric ciphers.

	None	DES	DESede	Blowfish	AES
Average	11,25	11,25	11,25	11,25	11,25
Std. Dev.	175,39095	0,33479	0,33791	0,32474	0,30951

Fig. 3. Statistical analysis for byte distribution in an encrypted stream

Apparently, following Figure 4, the byte distribution provided by the RSA outperforms the symmetric ciphers. This of course, is not conclusive in terms of choosing the RSA as the stream encryption algorithm[4] but reveals an interesting particularity of the RSA cipher.

Group RSA	Arithmetic mean
1. [0 – 15]	0,01
2. [16 – 31]	8,91
3. [32 – 47]	8,22
4. [48 – 63]	7,93
5. [64 – 79]	8,11
6. [80 – 95]	7,96
7. [96 – 111]	8,08
8. [112 – 127]	7,44
9. [128 – 143]	7,98
10.[144 – 159]	8,18
11.[160 – 175]	7,74
12.[176 – 191]	8,24
13.[192 – 207]	7,95
14.[208 – 223]	7,76
15.[224 – 239]	7,86
16.[240 – 255]	7,63

Group RSA + AES	Arithmetic mean
1. [0 – 15]	11,15
2. [16 – 31]	11,36
3. [32 – 47]	11,32
4. [48 – 63]	11,33
5. [64 – 79]	11,28
6. [80 – 95]	11,17
7. [96 – 111]	11,19
8. [112 – 127]	11,17
9. [128 – 143]	11,31
10.[144 – 159]	11,39
11.[160 – 175]	11,25
12.[176 – 191]	11,14
13.[192 – 207]	11,2
14.[208 – 223]	11,19
15.[224 – 239]	11,36
16.[240 – 255]	11,26

	RSA	RSA+AES
Average	8	11,25
Std. Dev.	0,07	0,3291

Fig. 4. Statistical analysis for byte distribution in an encrypted stream

[3]PKCS#1 uses the high order 11 Bytes of the RSA input to implement its padding scheme (128 B -11 B = 117 B).

[4]Further tests show that the time efficiency of RSA is outperformed by the symmetric ciphers in scale of order of magnitude.

4. The System.nanoTime()

To measure the time consumption of an algorithm, to some extent, it would be the best to measure it natively. For that purpose the use of Java Native Interface would be a must. Although this would be the best for watching the efficiency of bytecode itself JNI could provide extra overhead to JVM. That is why the solution provided by System.nanoTime() is considered in this document.

The `System.nanoTime()` method according to [4] provides the most precise available system timer, in nanoseconds. The value returned represents nanoseconds from some fixed but arbitrary time e.g from boot up or JVM start. It is also claimed that this method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. Its general issue is that it only provides nanosecond precision, but not necessarily nanosecond accuracy.

For this kind of JVM functions, it would be preferable to use the finest grained clock possible. For that purpose, the `POSIX clock_gettime()` with the `CLOCK_MONOTONIC`, as described in [5] and [2], seems to be the right choice. As it is known that the `clock_gettime()` is supported in current linux libraries and also according to [6] sufficiently recent versions of GNU libc and the Linux kernel supports the `CLOCK_MONOTONIC` clock as well. As this feature is not supported in most distributions, including the testing environment, the `System.nanoTime()` call relies on alternative, after [7], function `gettimeofday()`[5] as described in [8], that works with microseconds accuracy depending on a system.

Regarding this and as a conclusion from discussion [2], we have chosen the System.nanoTime() to manage with most of the tests workload, on measuring the elapsed times. What also needs to be mentioned is that the disk utilization[6] was not participating in the testing procedures.

5. The time efficiency of symmetric algorithms

The tests performed on time consumption of each symmetric algorithm were all based on the same scheme. Every encryption-decryption phaze was divided into some partial procedures (i.e. their times) evolved from the cryptographic and JCA/JCE architecture. Each test consists of 100[7] repetitions of each partial procedure of the entire process.

[5] It has been found that the native operation executes the following: nanoseconds = (seconds * 1000000) + microseconds) * 1000 As the strace analyze and the function code [9] has been analyzed.

[6] As a potential source of overhead.

[7] The scheme modified to 1000 repetition model did not change the general tendency.

5.1. The time efficiency – results

As the tests were performed the following results have been obtained.

Descryption	Time in ns
Key Generation	657248,44
IV Generation	20361,6
Cipher object gen.	19713,14
Cipher object initialization	15499,47
Stream Prepare	9240,64
ENCRYPTION	8390,38
DEC Cipher object gen.	27193,94
DEC Cipher object initialization	17003,85
DEC Stream Prepare	42297,62
DECRYPTION	8106,41

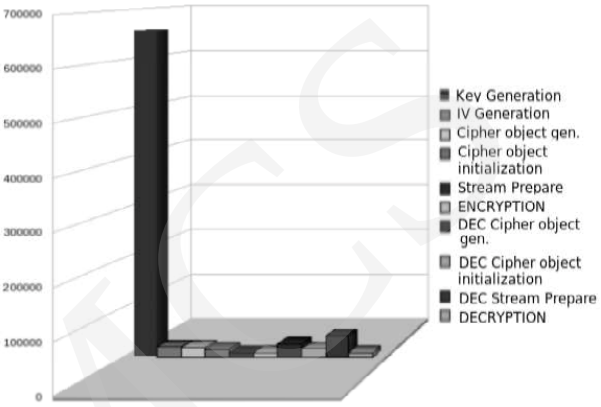


Fig. 5. Time consumption of the DES algorithm

There is only one particularity in the time efficiency of the first two algorithms that can also be found in all of the tested algorithms, meaning the time spent on key generation. It can be clearly seen that it is the main factor that decides for the entire process time. It might not be unexpected regarding the fact that it includes the generation of secure random, but it is important to note how significant it is regarding the entire process[8].

Descryption	Time in ns
Key Generation	647703,72
IV Generation	16631,85
Cipher object gen.	18378,59
Cipher object initialization	38478,15
Stream Prepare	8031,22
ENCRYPTION	11359,46
DEC Cipher object gen.	16994,32
DEC Cipher object initialization	34272,39
DEC Stream Prepare	80215,8
DECRYPTION	15041,29

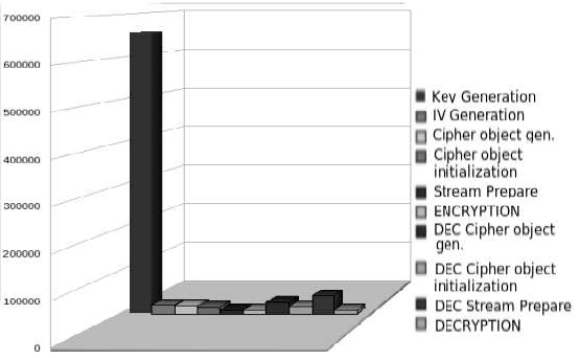


Fig. 6. Time consumption of the Triple DES algorithm

All of the tests on symmetric algorithms lead to the similar pattern of time consumption. One more regularity can be noticed regarding the Blowfish algorithm apart from the time consumed for key generation. The time spent on

[8]It will be essential for the comparison to the ECC.

cipher object's initialization took relatively more time than other activities during the entire process. But this does not disturb the general tendency.

Descryption	Time in ns
Key Generation	609203,58
IV Generation	17251,17
Cipher object gen.	21852,66
Cipher object initialization	79731,72
Stream Prepare	8373,17
ENCRYPTION	5993,47
DEC Cipher object gen.	24718,72
DEC Cipher object initialization	77329,02
DEC Stream Prepare	34294,3
DECRYPTION	6666,43

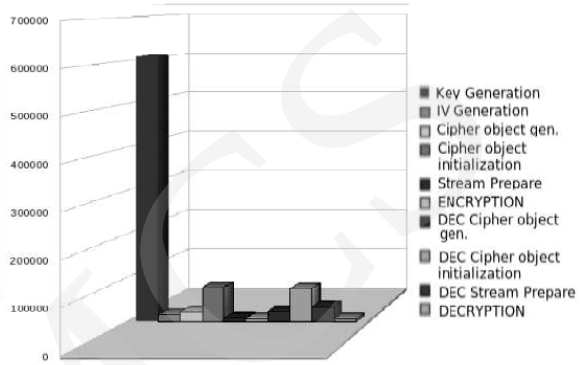


Fig. 7. Time consumption of the Blowfish algorithm

Descryption	Time in ns
Key Generation	622078,07
IV Generation	21644,47
Cipher object gen.	17830,67
Cipher object initialization	6051,44
Stream Prepare	16042,69
ENCRYPTION	5727,17
DEC Cipher object gen.	19161,06
DEC Cipher object initialization	5659,98
DEC Stream Prepare	30885,7
DECRYPTION	6971,5

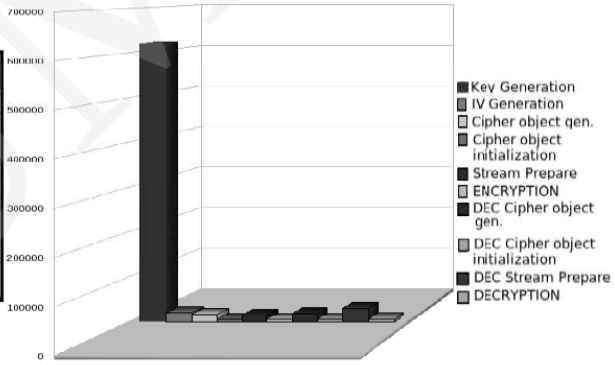


Fig. 8. Time consumption of the AES algorithm

6. Symmetric and Asymmetric algorithms comparison

The same testing procedure was applied to the RSA asymmetric cryptosystem. The results presented in Figure 9 display however, some interesting particularity that comes from the RSA algorithmic characteristic. From the theory of the RSA we know that the private exponent of the private key is much larger than the public exponent. Therefore using the RSA private key causes slower computations than the public key.

Descryption	Time in ns
Key Generation	24225142,47
IV Generation	969234,22
Cipher object gen.	50744,76
Cipher object initialization	7638,86
Stream Prepare	10330,44
ENCRYPTION	7653,78
DEC Cipher object gen.	6055755,91
DEC Cipher object initialization	7530,98
DEC Stream Prepare	35380,4
DECRYPTION	7771,74

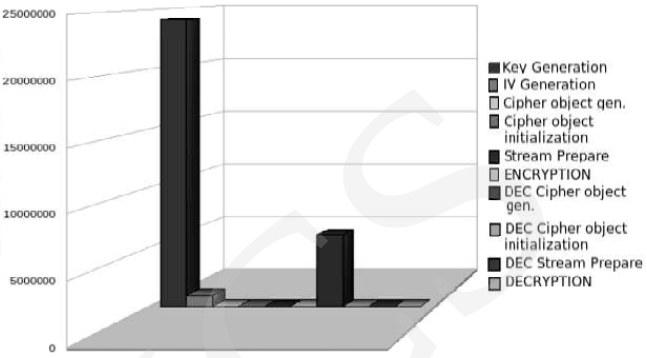


Fig. 9. Time consumption of the RSA algorithm

7. Elliptic Curve Cryptography

Following [10] and key length (Figure 10) it must be claimed that combining it with the results of performed tests ECC becomes the most suitable scheme for securing data.

Symmetric	ECC	DH/DSA/RSA
80	163	1024
112	233	2048
128	283	3072
192	409	7680
256	571	15360

Fig. 10. Key length comparison for different cryptosystems with the same strength

As the time of key generation in the tested algorithms became the main factor; reducing it while retaining the same cryptographic strength would be the optimal solution.

Conclusions

Along the performed tests it has been shown that regarding the symmetric algorithms comparison, the AES algorithm with this Java implementation provider, has outperformed other algorithms in terms of better byte distribution and also presented that time efficiency has given better results.

Thanks to preserving RSA keys' size disproportion particularity in Java implementation, it can be used for further project work to allow ciphering on clients with limited computational resources. Thanks to numbers achieved from those tests, it can be confirmed that the RSA is also much slower comparing to the symmetric ciphers.

Further work needs to be done on promising comparison for asymmetric cryptosystems with the NIST results for ECC; ECC advantage needs to be confirmed regarding the Java ECC architecture and implementation.

References

[1] Lamprecht C., et al., *Investigating the efficiency of cryptographic algorithms in online transactions*. School of Computing Science, University of Newcastle upon Tyne, UK I. J. of Simulation, 7(2).
[2] http://blogs.sun.com/dholmes/entry/inside the hotspot vm clocks, Evaluation
[3] http://www.bouncycastle.org/
[4] Sun Microsystems, Inc. (Santa Clara, CA, USA), *Java*TM*Platform, Standard Edition 6 API Specification*. Copyright 2006 Sun Microsystems, Inc. All rights reserved. available on-line at: http://java.sun.com/javase/6/docs/api/java/lang/System.html
[5] http://mia.ece.uic.edu/~papers/WWW/books/posix4/DOCU 007.HTM; The Department of Electrical and Computer Engineering,University of Illinois
[6] http://ibm5.ma.utexas.edu/cgi-bin/man-cgi?clock gettime+3
[7] http://bugs.sun.com/bugdatabase/view bug.do?bug id=6298653
[8] http://linux.die.net/man/2/gettimeofday
[9] http://lxr.linux.no/linux-bk+v2.6.10/arch/i386/kernel/time.c#L94
[10] http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2 Mar08-2007.pdf; National Institute of Standards and Technology, Computer Security Division, Computer Security Resource Center Recommendation for Key Management Part 1: General (Revised), March 2007
[11] Sun Microsystems, Inc. (Santa Clara, CA, USA), *Java*TM*Cryptography Architecture API Specification & Reference*. available on-line at: http://java.sun.com, Last Modified: 25 July 2004.