Comparative Implementation Analysis of AES Algorithm

Boris Damjanović[1], **Dejan Simić**[2]
[1] *dboris0206@gmail.com*, [2] *dsimic@fon.bg.ac.rs*
Faculty of Organizational Sciences, 11000 Belgrade, Serbia

Case study

DOI: **10.7251/JIT1102119D** UDC **659.2:004.651**

Abstract: Advanced Encryption Standard (AES) is the first cryptographic standard aroused as a result of public competition that was established by U.S. National Institute of Standards and Technology. Standard can theoretically be divided into three cryptographic algorithms: AES-128, AES-192 and AES-256. This paper represents a study which compares performance of well known cryptographic packages - Oracle/Sun and Bouncy Castle implementations in relation to our own small and specialized implementations of AES algorithm. The paper aims to determine advantages between the two well known implementations, if any, as well as to ascertain what benefits we could derive if our own implementation was developed. Having compared the well known implementations, our evaluation results show that Bouncy Castle and Oracle/SUN gave pretty equal performance results - Bouncy Castle has produced slightly better results than Oracle/Sun during encryption, while in decryption, the results prove that Oracle/Sun implementation has been slightly faster. It should be noted that the results presented in this study will show some advantages of our own specialized implementations related not only to algorithm speed, but also to possibilities for further analysis of the algorithm.

Keywords: computer security, cryptography, algorithms, standards, AES, performance.

Introduction

In the literature, there is a certain number of Java cryptographic APIs [10][16] (Application Programming Interfaces). Most of these implementations are constructed with an intention to, as much as possible simplify usage of various cryptographic algorithms and techniques. However, much smaller number of people is engaged in research and implementation of individual algorithms and cryptographic techniques.

Constructing small, specialized implementations of some algorithm which function is devoted to the specific task gives us multiple benefits. Such implementations generally achieve better results accomplishing the mission for which they are made for. In addition, the writing of these programs allows the author to be well acquainted with the ways of functioning of the individual algorithms and to come up with new discoveries related to the different points of view.

This paper represents an empirical study which compares performance of massive and well known cryptographic packages in relation to our own implementations [4] of AES [6][5][3][2][14] algorithm in Java programming language. In the article, we will further compare these results with speed measurements of an experiment with AES algorithm extensions below the key size of 128 bits. As a reference for measuring, we will use two AES implementations, which are parts of the large cryptographic packages - Bouncy Castle [11] and Oracle (former Sun), which both use the Java Cryptography Extension (JCE) [10][16]. Cryptographic implementations in the Sun JDK are distributed through several different providers still using name Sun (“Sun”, “SunJSSE”, “SunJCE”, “SunRsaSign”).

Known cryptographic packages and the length of the keys used in the experiments are:

1. Oracle/Sun JCE [13] version 1.7 with 128, 192 and 256 bit encryption;
2. Bouncy Castle [11] version 1.46 with 128, 192 and 256 bit encryption.

Comprised in our evaluation, we had four of our own implementations as well:

1. Implementation of AES standard algorithm (with 128, 192 and 256 bit key length), each based on Dr. Gladman's [9][12] and Bertoni's [1] ideas;
2. Implementation of the expanded AES algorithm (with key lengths of 32 and 64 bits), based on Dr. Gladman's and Bertoni's ideas each.

Test platform

As a test platform was used an Asus notebook computer with Intel (R) Core (TM) i5 450M processor at 2.40 GHz, (without new AES set of instructions - AES-NI) with 4GB RAM and Seagate@Momentus@ ST9500325AS hard disk and with the MS Windows 7 operating system.

As a development environment we used Eclipse Java EE IDE for Web Developers, Build id: 20110916-0149, Java SE Development Kit 7u1 for Windows and Java Cryptography Extension (JCE) for Java SE Development Kit 7u1 for Windows.

Implementation details

In our own implementation of the AES algorithm we used POJOs (Plain Old Java Objects). In this implementation we experiment with possible extensions of this algorithm according to the simple rules that we will introduce later in the text. Because of these extensions, our own implementation will hereinafter be referred to as EAES (Expanded AES).

To determine how fast our implementation is, we will compare it with implementations of well known manufacturers that use Java Cryptography Extension (JCE) [10][16] – Oracle/SUN and Bouncy Castle [11]. Both implementations are using provider-based architecture. For more details on the implementation of various cryptographic algorithms in Java, readers are referred to [10], [11], [16].

AES algorithm described in FIPS-197 document [6] transforms 128 bit block of data during 10, 12 or 14 rounds using the initial key lengths of 128, 192 and 256 bit. The initial key is then enlarged to (10+1)*16, (12+1)*16 or (14+1)*16 bytes in the key expansion routine. Each round repeats the SubBytes(), MixColumns(), ShiftRows() and AddRoundKey() transformations. AES authors redefine both addition operation within the GF(2^8), which is then conducted by XOR operation at the byte level and multiplication operation which is thus conducted as polynomial multiplication with the conditional modulo polynomial 0x11B. The mentioned multiplication is the most time consuming in the aspect of optimization, because it is intensively used during the MixColumns() transformation.

The most known software implementations of AES algorithm are based on Dr. Gladman's ideas. These implementations use four lookup tables of 4kB each for encryption, commonly referred to us as T tables, and four additional tables of same size for decryption. These tables contain the intermediate results calculated in advance for several transformations at once.

Beside the aforementioned eight large tables, we must point out two smaller tables of 256 bytes in size each, for SBox and inverse SBox, as well as a table with calculated values of RCon operation for which it is usually sufficient to allocate eleven bytes. In those implementations the 128 bit block (State) is represented as a 4x4 byte matrix, and it is processed on column by column basis.

According to Bertoni's idea, State matrix is to be firstly transposed then processed on row-by-row basis. This approach uses only three smaller tables - SBox, inverse SBox and RCon, therefore consumes significantly less memory [1], but uses multiplication more intensively.

Hypotheses

As mentioned fastest software implementations of AES algorithm today are based on ideas of Dr. Brian Gladman [9][12]. These implementations are characterized by the high processing speed, which is based on pre-calculated tables, due to which a great deal of memory is used. On the other side, there is a very interesting idea of Bertoni that achieves very good performance with significant decrease in memory usage [1], because the idea is based on a significantly smaller utilization of pre-calculated tables with interim results.

To conduct the necessary experiments with higher quality, we implemented both ideas in Java programming language, so that the implementation by Dr. Brian Gladman is marked by EaesG, while slightly changed implementation of Bertoni's ideas is marked by EaesB. You may have already assumed that the letter E in the mark refers to our implementations that reduce the standard to 32 and 64 bit encryption/decryption.

Experiments to be carried out will serve to test the following hypotheses:

- Specialized implementation of AES algorithm shows equally good or better results compared to the well known cryptographic packages,
- Large cryptographic suites lose a lot of the time for the first initialization at engine startup,
- Experimental extensions of AES algorithm for 32 and 64 bit encryption and decryption are achieving even greater differences in processing speed compared to the large cryptographic packages.

Testing methodology

To achieve the highest test results precision, we implemented four applications named SunAes, BcAes, EaesB and EaesG. Each individual implementation was given the same conditions in regard to processor, memory and hard disk usage. Each particular implementation was evaluated using the same test platform as described in section 2. All tests were conducted by consecutive repetition of measurements on files in 512KB-32MB size.

The first series of tests was conducted in such manner that we measured the time required for initialization of particular class, loading data from disk, its processing and saving to disk. Then, to avoid any caching by operating system and hardware, we initiated the subsequent application in another folder, and then the following application in the third folder, etc. After that we computed the arithmetic mean of the achieved results. This way of testing showed that large cryptographic packages (such as Oracle/SUN JCE and Bouncy Castle) consume a lot of time (from 200 to even 700 ms) for initialization, while our implementation was significantly faster due to short initialization time. When we put the same code in the loop, we got significantly different results, as you can see from the following example:

```
infile_16_bytes.txt, aes128, pass: 1
 Time : 641 ms
infile_16_bytes.txt, aes128, pass: 2
 Time : 0 ms
infile_16_bytes.txt, aes128, pass: 3
 Time : 0 ms
```

Code 1: Total time results in loop

This way of testing can give us a twisted picture of large cryptographic packages speed – those are ultimately optimized and extraordinary fast implementations. However, in some applications, the extended time needed for initialization can present a problem which must be taken into account.

That is why we applied a slightly different solution in the following testing series. Firstly we slightly altered the source code, to be able to measure only the time needed for data processing. In accordance with [7][8] and [15] we conducted additional two measuring series. In the first series we measured the time by alternate starting of each application individually, to avoid the influence of caching by the operating system and hardware as much as possible. Achieved results in this step represent the arithmetical mean of five conducted measuring sessions, in which we rejected the highest and lowest result to avoid the influence of other processes in the system. In the second testing series, we put the measurement code in the loop and executed it for six times within one VM call, after which we rejected the first result, which, according to [8] is considered to be the time required for compiling. We took into account only the time required for execution. In the end, we combined two described testing methodologies as to compute the arithmetic mean of the achieved results from the last two test series. Finally, the results are presented as the mean number of milliseconds per megabyte.

Measurement Results– Standard-Defined AES Algorithm

Hereby we set out the measurement results, with the aim to rank our implementations – EaesG and EaesB in comparison to large cryptographic packages.

TABLE 1: 128 BIT ENCRYPTION RESULTS

128-bit encryption	Sun (ms)	EaesG (ms)	BC (ms)	EaesB (ms)
512 KB	4	15	11	14
4096 KB	73	84	69	93
8192 KB	144	140	158	175
16384 KB	290	312	293	365
32768 KB	591	577	593	702
ms/MB	18	18	18	22

FIGURE 1: 128 BIT ENCRYPTION RESULTS

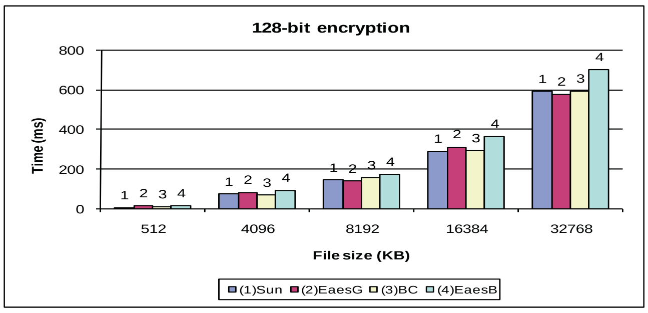
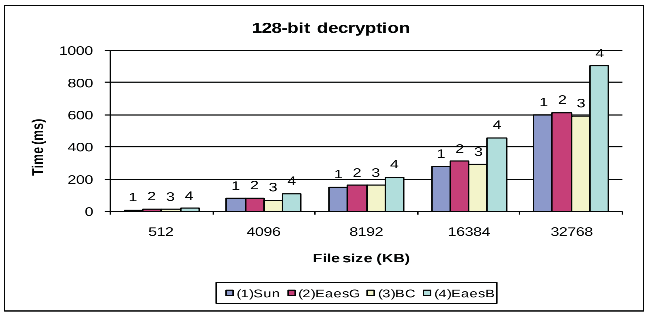


TABLE 2: 128 BIT DECRYPTION RESULTS

128-bit decryption	Sun (ms)	EaesG (ms)	BC (ms)	EaesB (ms)
512 KB	8	15	13	18
4096 KB	79	82	70	111
8192 KB	152	162	161	214
16384 KB	278	311	292	458
32768 KB	600	614	594	902
ms/MB	**18**	**19**	**18**	**27**

FIGURE 2: 128 BIT DECRYPTION RESULTS



Although all tested implementations showed impressive speed, generally speaking, our implementation based on Dr. Gladman's ideas, Bouncy Castle and Oracle/SUN implementations provided slightly better results in the described measuring conditions. Those implementations gave pretty equal results in measuring of 192 bit and 256 bit encryption and decryption:

TABLE 3: 192 BIT ENCRYPTION RESULTS

192-bit encryption	Sun (ms)	EaesG (ms)	BC (ms)	EaesB (ms)
512 KB	11	16	10	15
4096 KB	94	88	89	105
8192 KB	193	182	166	210
16384 KB	375	364	351	403
32768 KB	721	671	688	846
ms/MB	**22**	**21**	**21**	**25**

FIGURE 3: 192 BIT ENCRYPTION RESULTS

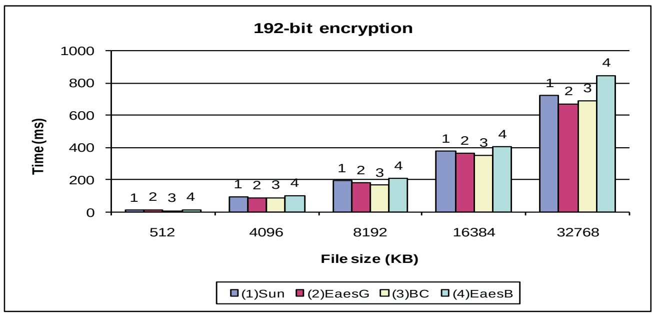
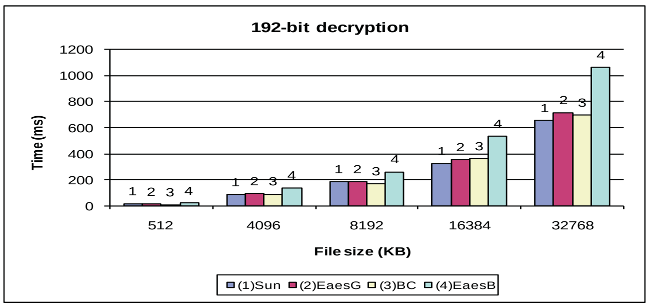


TABLE 4: 192 BIT DECRYPTION RESULTS

192-bit decryption	Sun (ms)	EaesG (ms)	BC (ms)	EaesB (ms)
512 KB	16	14	11	21
4096 KB	88	94	93	139
8192 KB	188	186	171	262
16384 KB	325	354	364	534
32768 KB	653	714	698	1060
ms/MB	**21**	**22**	**22**	**33**

FIGURE 4: 192 BIT DECRYPTION RESULTS



Once again, our EaesG and Bouncy Castle implementations encrypt data slightly faster than implementation based on Bertoni's idea. We had the least available information on Bertoni's idea, according to [1] probably for Bertoni's work had been under patenting process. We therefore gave up making any attempts to optimize implementation based on his idea. Yet, it was included in our test, because we believe that it was an awesome idea with enormous potential for experiments on standard-defined AES algorithm expansion.

Table 5: 256 bit encryption results

256-bit encryption	Sun (ms)	EaesG (ms)	BC (ms)	EaesB (ms)
512 KB	9	15	15	18
4096 KB	110	114	107	136
8192 KB	186	197	209	241
16384 KB	436	394	414	504
32768 KB	835	852	789	998
ms/MB	25	25	25	31

Figure 5: 256 bit encryption results

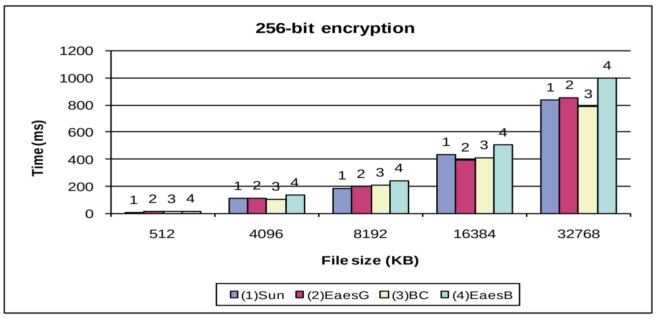
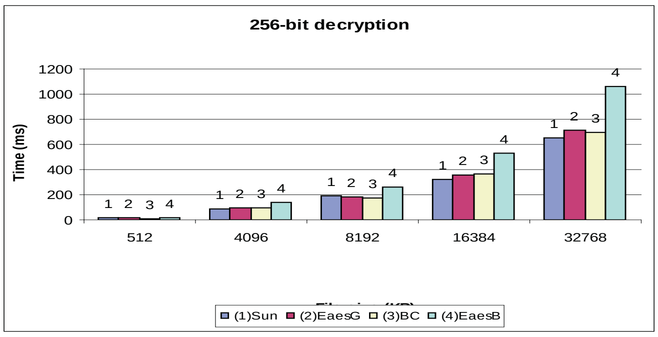


Table 6: 256 bit decryption results

256-bit decryption	Sun (ms)	EaesG (ms)	BC (ms)	EaesB (ms)
512 KB	23	17	21	22
4096 KB	91	103	108	149
8192 KB	187	193	214	306
16384 KB	393	405	409	621
32768 KB	770	805	826	1234
ms/MB	**24**	**25**	**25**	**38**

Figure 6: 192 bit decryption results

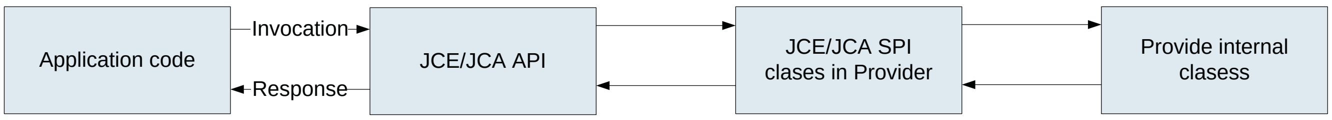


As we come to 256-bit encryption, all three implementations showed equally good results, but for the 256-bit decryption, SUN's implementation has produced slightly better outcomes, to EaesG and BC implementations respectively. Hereby we must stress out that the purpose of the described tests was not to run a dispute over the speeds of large cryptographic packages. If we exclude time needed for initialization, for the shown differences in speed are still insignificant. The complete initial test phase was conducted in order to create a solid ground for determining the real gains in speed expected to attain in our subsequent experimental implementation of 32 and 64 bit expansion of AES algorithm.

Experiment - Details of Expanded Algorithm Implementation

The development of one's own implementation of some cryptographic algorithm makes the essential advantage as a possibility for further specialization in certain applications. It is noticeable that a short and specialized implementation of AES algorithm produces equally satisfactory results and even faster than the ones in large multipurpose implementations. Tested Oracle/SUN JCE and Bouncy Castle implementations use "provider based" architecture, as it is shown in Figure 6a. Objects that provide functionality in the Java Cryptography Architecture (JCA) and its successor Java Cryptography Extension (JCE) are not visible to those who develop an application. Developer in the case of JCA and JCE address to collections of classes that serve as links that provide some cryptographic service. Therefore, the mentioned multifunctional implementations need more time for initialization of proper algorithm, and thus for the execution.

Figure 6a: Provider based architecture



However, performance gain is not the only benefit of writing your own implementation of a particular algorithm. A lot more than mere speed is gained by acquiring the knowledge needed for mastering the certain algorithm – knowledge that can be used for certain improvements of this algorithm. In further text we will present two experiments that explore the possible ways for expansion of AES algorithm, related to 64-bit and 32-bit encryption.

We have already mentioned that the standard-defined AES algorithm transforms the data during 10, 12 or 14 rounds and that the initial key in the key expansion routine is developed at (10+1)*16, (12+1)*16 or (14+1)*16 bytes. Hence, AES uses 10 rounds for the 128-bit encryption, and the initial key is expanded to (10+1)*16=176 bytes. If we continue to follow this logic, for the 64-bit encryption we can use 8 rounds, due to which we will expand the initial key to (8+1)*16=144 bytes, while for the 32-bit encryption we will use 7 rounds, and the initial key will be expanded to (7+1)*16=128 bytes.

This reduction in the number of operations (via the reduction in the number of rounds) should result in certain accelerations, which we must determine by new series of tests.

Measuring Results – Expanded AES Algorithm

Based on the previously conducted measuring sessions we have ranked our implementations in comparison to well known cryptographic packages. The purpose of conducting the following series of tests was to determine the time spared by applying 64-bit and 32 bit encryption in relation to 256, 192 and 128-bit encryption and decryption. For this measuring series we also used the formerly described combination of two testing methodologies to get more precise results, and all the measurements were conducted on both of our implementations (EaesG and EaesB).

The above diagrams show the results of measurements the EaesG algorithm based on Dr. Gladman's ideas, which are marked 1 to 5, while the results of measuring the EaesB algorithm, based on Bertoni's ideas are presented with bars 6 to 10. If we observe each implementation individually, the achieved re-

Table 7: 256, 192. 128 bit vs. 64/32 bit encryption results

Encryption	EaesG 256	EaesG 192	EaesG 128	EaesG 64	EaesG 32	EaesB 256	EaesB 192	EaesB 128	EaesB 64	EaesB 32
512 KB	15	16	15	16	16	18	15	14	12	16
4096 KB	114	88	84	63	63	136	105	93	79	55
8192 KB	197	182	140	103	94	241	210	175	158	139
16384 KB	394	364	312	270	224	504	403	365	308	271
32768 KB	852	671	577	484	442	998	846	702	608	529

Table 8: 256, 192, 128 bit vs. 64/32 bit decryption results

Decryption	EaesG 256	EaesG 192	EaesG 128	EaesG 64	EaesG 32	EaesB 256	EaesB 192	EaesB 128	EaesB 64	EaesB 32
512 KB	17	14	15	12	7	22	21	18	14	11
4096 KB	103	94	82	63	64	149	139	111	94	81
8192 KB	193	186	162	120	107	306	262	214	187	169
16384 KB	405	354	311	246	203	621	534	458	386	325
32768 KB	805	714	614	632	469	1234	1060	902	755	667

Figure 7: 256, 192, 128 bit vs. 64/32 bit encryption results

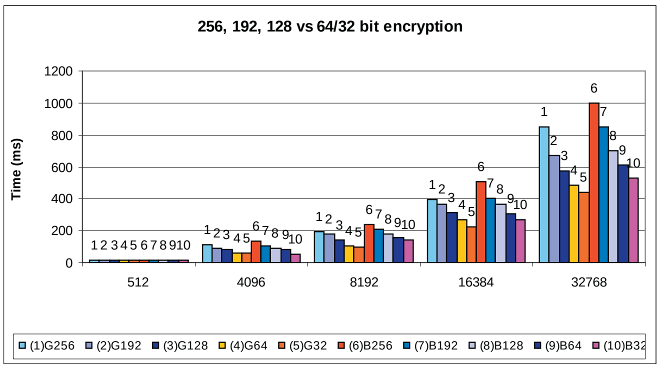
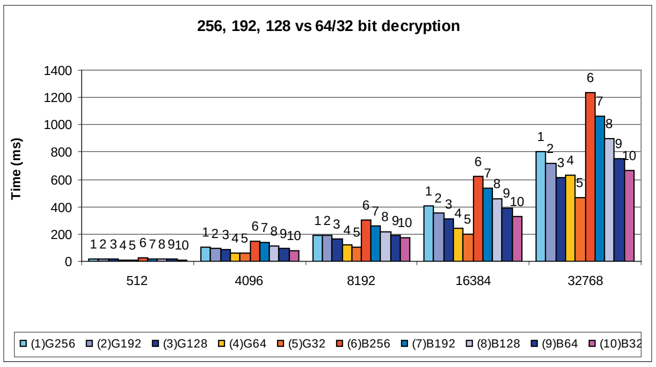


Figure 8: 128 bit vs. 64/32 bit decryption results



sults show that the time necessary for data processing is almost proportionally reduced as the number of algorithm rounds goes down.

Conclusion

Tested Oracle/SUN JCE and Bouncy Castle implementations use "provider based" architecture. According to our experimental results small, specialized implementations of the AES algorithm can be eqauly good or even faster than its large and multi-function counterparts. The multifunctional implementations take more time for initialization of proper algorithm or cryptographic tool, thus the data processing becomes longer.

Our tests have shown that when comparing well-known implementations, Bouncy Castle produces slightly preferable performances related to encryption time, while Oracle/Sun implementation is better when the criteria is decryption time. If we compare all implementations, EaesG brings equally good results as Bouncy Castle and Oracle/Sun when considering 128-bit encryption but slightly worse results when it comes to decryption. Both EaesG and BC appear to have equally preferable outcomes in the 192-bit encryption. However, taking into consideration the process of decryption, it is shown that Oracle/Sun implementation runs a bit faster. Finally, as we come to 256-bit encryption, all three implementations showed equally good results, while Oracle/Sun gets a better score in decryption.

Also, it should be mentioned that EaesG implementation based on Dr. Gladman's ideas shows significant improvements to EaesB implementation founded on Bertoni's idea no matter if it is related to encryption or decryption. On the other hand, it should be noted that EaesB implementation consumes significantly less memory, while still achieving satisfactory results.

We can point out that the conducted experiments have proven that AES algorithm can be expanded to 64 and 32 bit encryption given its high flexibility. This can lead to significant accelerations in its operation. Displayed results show that, depending on the number of both rounds and implementations, we can gain as much as 20-30% higher speed compared to 128 bit encryption and decryption.

From the presented experimental results it is clear that a certain acceleration can be achieved by constructing small and specialized implementation of AES algorithm instead of the use of the large implementations of the well-known software manufacturers. But the greatest advantage of constructing our own implementations is the possibility of further experimentation with a given algorithm for the purpose of research and comprehensive analysis.

Acknowledgments

The work presented here was partially supported by the Serbian Ministry of Science and Technological development (project Multimodal biometry in identity management, contract no TR-32013).

References

[1] Bertoni, G., et al. (2002). Efficient Software Implementation of AES on 32-Bit Platforms. CHES 2002: 159-171

[2] Carlos, C., et al. (2006). Algebraic Aspects of the Advanced Encryption Standard, Springer Science-Business Media, LLC.

[3] Daemen J., Rijmen V., (2002). The Design of Rijndael, Springer-Verlag, Inc.

[4] Damjanović, B. (2008), Implementation and extension of AES algorithm, Master's thesis, Faculty of Organizational Sciences, University of Belgrade,

[5] Dobbertin, H., et al. (2005). Advanced Encryption Standard AES, 4th International Conference, Bonn, Germany, 2004, Springer-Verlag

[6] Federal Information Processing Standards Publication 197, (2001). Specification for the ADVANCED ENCRYPTION STANDARD (AES), Available at: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (Accessed: December 2011)

[7] Francia, G., et al. (2007). An Empirical Study on the Performance of Java/.Net Cryptographic APIs, Information Security Journal: A Global Perspective, 16: 6, 344 - 354

[8] Georges, A., et al., (2007). Statistically Rigorous Java Performance Evaluation, Department of Electronics and Information Systems, Ghent University, Belgium

[9] Gladman, B. (2007). A Specification for Rijndael, the AES Algorithm, Available at: http://gladman.plushost.co.uk/oldsite/cryptography_technology/rijndael/aes.spec.v316.pdf (Accessed: December 2011)

[10] Hook D., (2005). Beginning Cryptography with Java, Wrox Press

[11] http://www.bouncycastle.org/ (Accessed: December 2011)

[12] http://www.gladman.me.uk/ (Accessed: December 2011)

[13] Java SE security, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html (Accessed: December 2011)

[14] Konheim, A. (2007). Computer security and cryptography, John Wiley & Sons

[15] Van Etten, D., (2009). Why Many Java Performance Tests are Wrong, Available at: http://java.dzone.com/articles/why-many-java-performance-test/ (Accessed: December 2011)

[16] Weiss, J., (2004). Java cryptography extensions: practical guide for programmers, Morgan Kaufmann

Submitted: December 13, 2011
Accepted: December 31, 2011