Scientific Journal of Riga Technical University
Computer Science. Applied Computer Systems
2010
Volume 43

Implementation of Cryptographic Algorithms in Software: An Analysis of the Effectiveness

Vladislav Nazaruk, *Riga Technical University*, Pavel Rusakov, *Riga Technical University*

***Abstract* - The goal of the paper is to discuss some possibilities of effective implementing cryptographic algorithms in software development. There are concerned two aspects of the term "effectiveness": efficacy (i.e., meeting a goal of using cryptographic algorithms – providing a protection of information) and efficiency (i.e., implementing cryptographic algorithms in an economical way; as well as implementing cryptographic algorithms to the effect that they execute in an economical way). In the paper, there are defined and classified errors which are the causes of possible vulnerabilities in implementation of cryptographic algorithms on a computer. There are given general recommendations, how to escape from these vulnerabilities. There is also considered implementation of cryptographic algorithms in software development by using software libraries and frameworks. Several widespread software frameworks which provide cryptographic functionality are compared with each other by the speed of execution of algorithms. Finally, there are discussed some possibilities of maximising the speed of execution of cryptographic algorithms, emphasising the importance of parallelisation of algorithms. For block ciphers, possible parallelisation is discussed especially deeply, referencing the results obtained in practical experiments that demonstrate and estimate the benefits of parallelising block ciphers.**

***Keywords:* cryptographic algorithms, effectiveness, execution speed, parallelization, software libraries, vulnerabilities**

I. Introduction

People's life is impossible without communication. Sometimes communication must be protected in a certain manner. In this situation, cryptography – a study and practice of protecting information – is often used. The goal of this paper is to discuss cryptography in the context of its effective implementation in software development, emphasising possible errors and vulnerabilities in implementation of cryptographic algorithms, usage of cryptographic frameworks and libraries, as well as the speed of execution of implemented cryptographic algorithms.

At the beginning of the paper, there is given a brief overview of methods for providing information security. Further in the paper, there are defined possible vulnerabilities in implementation of cryptographic algorithms on a computer, and errors which are the causes of these vulnerabilities are classified. There is also considered implementation of cryptographic methods in software development by using software frameworks. Several major software frameworks which provide cryptographic functionality are compared with each other by the speed of execution of algorithms. The goal of this comparison is to construct some guidelines which help select a feasible solution for implementing a cryptographic functionality in software. Finally, there are discussed some general possibilities of maximising the speed of execution of cryptographic algorithms. At the end of the paper, the conclusions about the work are given.

Results concerning the speed of execution of implemented cryptographic algorithms are based on practical experiments – by the authors, there were written and utilised several computer programs (in Java, C# and C++ programming languages) for measuring the time of running specific cryptographic algorithms implemented in some widespread software frameworks/libraries (Java Platform Standard Edition, Microsoft .NET Framework, Botan library). The methodology used for measuring the speed of execution of cryptographic algorithms is discussed further in this paper.

The paper is addressed to specialists in computer science and information security interested in effective (i.e., secure and fast) implementation of cryptographic algorithms in a process of software development.

II. Basic Concepts of Cryptography

Cryptography is a science which investigates methods of protecting information. *Protection of information* is an activity (or a set of activities) for providing *security of the information*; in its turn, *information security* is such a property of the information environment, which implies the presence of at least one of the following attributes (factors) of the information (depending on the requirements) [1]:

- confidentiality – a possibility to access the information only by its sender and addressee(-s);
- integrity – an ability for the receiver of the information to discover the fact of modification of the information (if any) by third parties (after the information has been sent by its sender);
- authenticity – an ability for the receiver of the information to determine the real sender of the information.

Cryptography together with *cryptanalysis* – a study of identifying and exploiting flaws in methods of information protection – forms a joint science called *cryptology* (which sometimes is also called cryptography).

There exist a lot of different algorithms – cryptographic algorithms – the use of which can ensure the fulfilment of specific factors of information security. All cryptographic algorithms provisionally can be divided into the following two main classes: cryptographic primitives and cryptographic protocols. *Cryptographic primitives* can be defined as algorithms which describe atomic mathematical transformations with specific properties (these properties, in their turn, determine a class of cryptographic primitive). *Cryptographic protocols* mainly describe *logic* of using of

cryptographic primitives, in order to implement specific factors of information security. Thus, cryptographic protocols are meta-algorithms which in their work use cryptographic primitives.

Depending on properties of a transformation implemented by a specific cryptographic primitive, these primitives are divided in the following main classes:

- ciphers:
  - symmetric ciphers (*or* private-key ciphers):
    - block ciphers;
    - stream ciphers;
  - asymmetric ciphers (*or* public-key ciphers);
- one-way (*or* cryptographic) hash-functions;
- message authentication codes (MAC);
- digital signature algorithms.

In this paper, the emphasis is on cryptographic primitives, and particularly on mostly-used primitives: symmetric ciphers and hash-functions.

Information security factors which are fulfilled by each class of cryptographic primitives are shown in Table 1. The names of most widespread algorithms from each class of cryptographic primitives are shown in Table 2.

TABLE 1

INFORMATION SECURITY FACTORS
FULFILLED BY A SPECIFIC CLASS OF CRYPTOGRAPHIC PRIMITIVES

Class of cryptographic primitives	Information security factor		
	Confidentiality	Integrity	Authenticity
ciphers	+		
one-way hash-functions		+	
message authentication codes		+	+
digital signature algorithms		+	+

TABLE 2

MOST WIDESPREAD ALGORITHMS
FROM EACH CLASS OF CRYPTOGRAPHIC PRIMITIVES

Class of cryptographic primitives	Examples of algorithms
ciphers:	
— symmetric:	
— block	AES (*or* Rijndael), RC2, Tri-DES, Blowfish, RC6, Twofish, Serpent, IDEA, RC5, DES, GOST
— stream	Salsa20, Panama, RC4 (*or* ARC4), SEAL 3.0, WAKE
— asymmetric	RSA, ElGamal
one-way hash-functions	SHA-1, SHA-2, Tiger, Whirlpool, RIPEMD-160, MD2, MD4, MD5
message authentication codes	VMAC, HMAC, CMAC
digital signature algorithms	DSA, GOST R 34.10-94

III. VULNERABILITIES IN IMPLEMENTATION OF INFORMATION PROTECTION METHODS

When creating or implementing cryptographic methods, it is very hard to prove that the method or its implementation is resistant to cryptanalysis. If people who created a specific cryptographic method, and people who tested it for the resistance, have not found vulnerabilities in it (i.e., in its theoretical model or its implementation), then that does not mean that the method is fully resistant to cryptanalysis – even if manpower used for checking the method was very large. The situations when well-known, widespread cryptographic algorithms, which appeared to be secure for a long period of time, are successfully attacked, are common enough (see [2] and [3] for the illustration of attacking the MD5 hash-function).

Vulnerabilities can arise in the entire life cycle of a cryptographic method. Factors which can raise vulnerabilities in implementation of information protection methods are classified by the authors of this paper in the following way:

- errors in selecting a proper information protection algorithm:
  - errors in selecting a proper class of information protection algorithms;
  - errors in designing cryptographic algorithms;
- errors in implementing cryptographic algorithms;
- usage of cryptographic algorithms in an improper way;
- unaccounted specific features of the external environment.

These factors are considered further in this section.

A. Errors in Selecting a Proper Information Protection Algorithm

The choice of improper class of information protection algorithms can lead to a violation of the security of an information system. For example, in the information system, there can be used methods of information protection which constraint only *access* to the secure information, but does not ensure *hiding* of that information; or in the information system, information protection methods can be not used at all. In these cases, in order to maximise the information security level and to prevent a possibility for unauthorised people to get to the secure information by taking the software in, *cryptographic* algorithms should be used.

By using cryptographic algorithms, many necessary actions (e.g., checking for the correspondence of an entered password to the original) can be implemented either totally without storing secure information (e.g., instead of passwords, storing their hash-values), or without storing secure information in an open way – but in a modified *(protected)* way.

By far not every invented cryptographic algorithm (including cryptographic protocols) is reasonably secure. To become to a certain degree accepted by the public, an algorithm must sustain after many attempts of breaking it during a sufficient period of time (at least several months, and also if there is enough interest to it from cryptanalysis professionals).

It is needless to say that most of home-made cryptographic algorithms are very exposed to breaking. However, even a little modification of a highly secure cryptographic algorithm can reduce almost to zero all advantages of the original algorithm. For example, a modification of one of the efficient group key distribution technique – the subset difference (SD) method – has been showed as insecure in [4].

On the other hand, an algorithm (even which is commonly used) should not be used in areas where a high degree of security is needed if there are facts of the existence of undesirable vulnerabilities in it (and, surely, if the algorithm was broken). For example, some of the widely used hash-functions: MD2, MD4, MD5, HAVAL-128 – are proven to be not acceptable one-way functions (simply stated, they are broken) [5], [6], and therefore, in order to provide a greater security level, they should be replaced by functions not yet broken.

Summarising the information in this section, one should not use a personally created or modified cryptographic algorithm, as well as, where possible and necessary, one should not use algorithms that are proven to be insecure (i.e. broken). The solution is to use trusted non-broken cryptographic algorithms [7].

B. Errors in Implementing Cryptographic Algorithms

Even if a mathematical model of an algorithm (or a protocol) has been proven to be secure, it is not guaranteed that implementation of the algorithm is also secure.

If in the software development, cryptographic algorithms are coded from scratch, then possible errors in the implementation of cryptographic algorithms are programming errors – errors of an improper mapping of operations and logics of algorithms to the source code by software programmers. Programming errors in cryptographic software include the following errors:

- *bugs* – which are common for the entire programming and not specific to cryptography; methods for controlling them are not discussed here separately;
- *usage of cryptographic algorithms in an improper way* (see the corresponding section of this paper);
- *usage of improper algorithms* – for example, if a stream cipher assumes a pseudo-random number generator (PRNG) to be used, then there should be used only *cryptographically secure* PRNGs, but not simple generators which are built in some programming languages (e.g. the function *std::rand()* in C++).

To avoid most of programming errors in implementation of cryptographic algorithms, and to simplify and shorten the implementation process, there are provided a number of cryptographic libraries with implementations of commonly used (and sometimes also many other, not so popular) cryptographic algorithms.

C. Usage of Cryptographic Algorithms in an Improper Way

An example of improper use of already implemented algorithms is the following. If an algorithm uses a PRNG, it should not be forgotten to be seeded; the seed should not be constant, but should be as random as possible. Therefore, if in implementation of a cryptographic algorithm there is used an already implemented routine, its specification and considerations of using it should be studied.

This problem has a considerably effective solution in object-oriented programming languages. Because of the property of encapsulation and existence of constructors and destructors (or finalisers) in these languages, it is much easier to enforce the program to behave more or less desirably.

D. Unaccounted Specific Features of the External Environment

One such vulnerability can occur if an operating system which uses swapping, moves (i.e. swaps out) data from the program memory to the disk. If in the data swapped out, there was confidential information (for example, such as a private key), and the computer at the moment when this information was on a hard drive was abnormally terminated, then the confidential information will be available on the hard drive for an undefined period of time – which is a potential security threat [8]. In order to eliminate such a risk, implementations of cryptographic methods must take actions to prevent swapping confidential information out to the disk (e.g. by using a special system calls). By using self-implemented cryptographic routines or some immature cryptographic libraries, there is a risk of the program being insecure.

IV. Implementation of Cryptography Using Software Libraries and Frameworks

There exist a plenty of cryptographic libraries, and there also exist software frameworks which include implementation of commonly used cryptographic algorithms: Microsoft .NET Framework and Java Platform Standard Edition (Java SE). .NET framework can be used by any CLI language (C#, Visual Basic for .NET, Delphi for .NET, C++/CLI, etc.), and Java SE can be used by the Java programming language and some other programming languages (see below).

Further in this section, there is discussed a usage of cryptographic frameworks/libraries from variety of programming language; also some commonly used cryptographic libraries are compared with each other.

A. Possibilities of Using Cryptographic Frameworks in Different Programming Languages

For some programming languages, there exist slight modifications (ports) of these languages to make them support the Microsoft .NET Framework, the Java Platform or another software framework – so-called *managed* versions of the corresponding programming languages. For such managed programming languages, porting enables them to use the entire cryptographic (as long as other) functionality provided by these software frameworks.

For some natively unmanaged programming languages, there exist cryptographic libraries that are implemented in these languages. An amount and functionality of such libraries vary from one language to another (surely, depending also on the popularity of the language) – for example, for the C++ language, there exist many cryptographic libraries, however,

for the Ada language, very few native (that are written in Ada) cryptographic libraries exist.

Therefore, for some languages, in order to use reliable and secure cryptographic functions, generally the only possible way is, instead of using the native (unmanaged) language, to use a port of the language to a framework which provides cryptographic functions needed. In this way, cryptographic functionality can be implemented in a variety of languages: for example, languages which have their Common Language Infrastructure (CLI) implementations: e. g. C++ (which has a managed version named C++/CLI), Delphi (respectively Delphi.NET), Java (J#), Python (IronPython), Ruby (IronRuby), Lisp (L#, IronLisp), PHP (Phalanger), Prolog (P#) [9]; and languages which have their Java Virtual Machine (JVM) implementations: e. g. Python (Jython), Ruby (JRuby), JavaScript (Rhino), Scheme (Bigloo, Kawa, SISC), Haskell (Jaskell) [10].

It needs no saying that cryptographic functionality of the software frameworks can be used by programming languages which are *natively* managed by the corresponding virtual machines, such as Java, C# and VB.NET.

B. Comparison of Major Cryptographic Frameworks

In this section, three commonly used cryptographic libraries and frameworks which provide cryptographic functionality are compared with each other by the speed of execution of the implemented algorithms. This comparison is an instrument to construct some guidelines for selecting a feasible solution for implementation of cryptographic algorithms in software.

In Fig. 1 and Fig. 2, there are shown the results of a comparison of three cryptographic frameworks: Java SE 6, .NET Framework 3.5 and Botan 1.8.2 (latest versions at the moment of preparing this paper) – by the speed of execution of their implemented algorithms. An absence of some bars in the diagrams, and values 0.00 for speed mean that the corresponding cryptographic framework does not support the corresponding cryptographic algorithm.

For the measurement of the speed, by the authors there were written three similar test programs – a program in Java for Java SE framework, a program in C# for .NET Framework, and a program in C++ for Botan library. (It is valuable to note that the Botan library was compiled with a speed optimisation enabled – a compiler switch "/O2".) To maximally smooth out the differences between results, in every program each cryptographic algorithm was called with data with different size – from 32 B to 32 MB. To measure small amounts of time with a high precision, each cryptographic algorithm with the same input data size was called many times in a loop. The measurement of time intervals needed for algorithms to execute was implemented in the following way: the system time was measured just before and just after the execution of the algorithm, and the difference of these values was considered as the execution time. The system time was measured using system calls, with the precision of several milliseconds. Before calling cryptographic algorithms, all input data have been completely loaded into the RAM. After the amount of time needed for a cryptographic algorithm to complete has been measured, a relative speed of execution (in

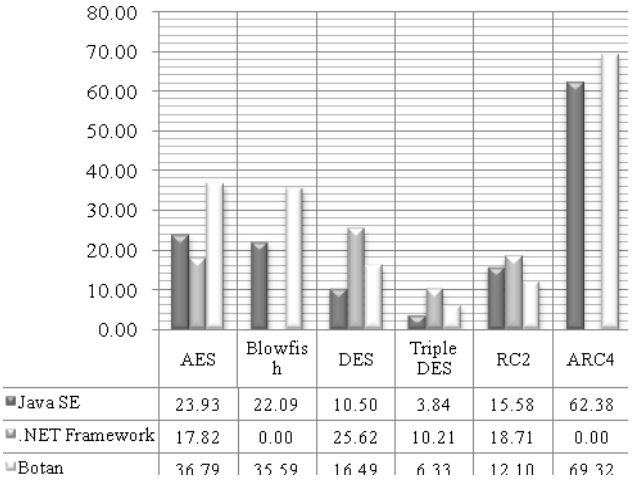

Fig. 1. An average speed rate of symmetrical ciphers in different cryptographic frameworks (MB/s)

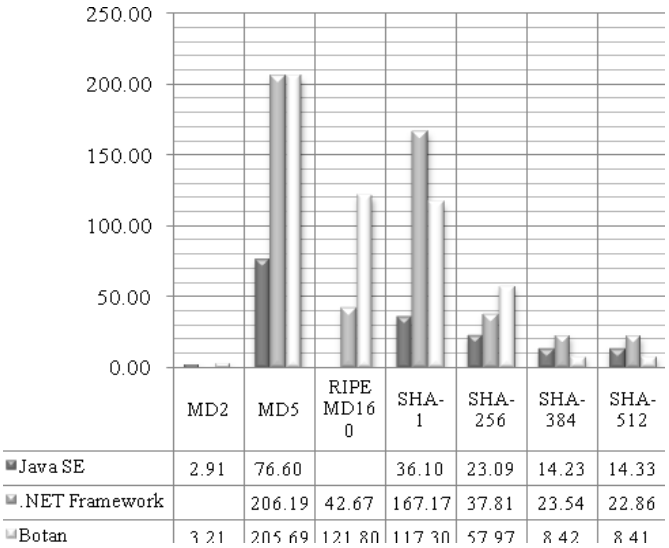

Fig. 2. An average speed rate of hash functions in different cryptographic frameworks (MB/s)

megabytes-of-input-data per second) of cryptographic algorithms was calculated; in diagrams, there are shown average values of the corresponding speeds. The test programs were executed on a computer with an Intel Core2Duo 2.20 GHz CPU, 4 GB RAM, and Microsoft Windows XP SP3 operating system.

Comparing the average speed rate of *hash functions* in frameworks Java SE and Microsoft .NET Framework, one can see that hash algorithms implemented in .NET Framework in all cases are faster than in Java SE. Implementations of hash functions SHA-256, SHA-384 and SHA-512 in .NET Framework are about 40% faster than in Java SE; but implementations of hash functions MD5 un SHA-1 in .NET Framework is about 3 times faster than in Java SE.

By analysing the dependence of time needed for hash functions to run, one can conclude that on data of almost any

size (starting from 1 KB), the speed rate of hash functions (in MB/s) is approximately constant.

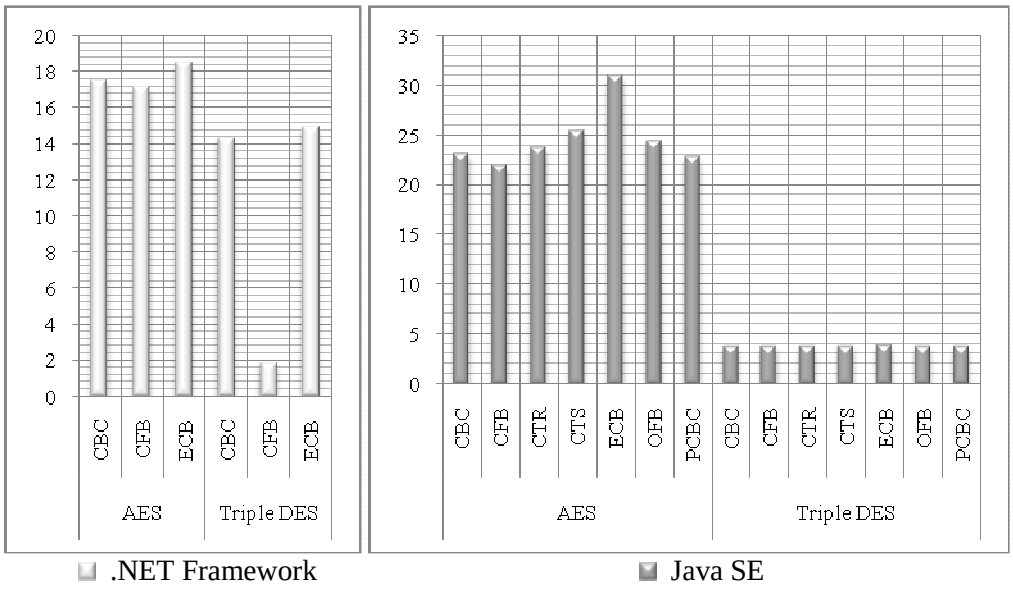

Fig. 3. An average speed rate of some block ciphers depending on their mode of operation, in different cryptographic frameworks (MB/s)

There are some unpredictable results for all block symmetric ciphers implemented in .NET Framework, except for the AES algorithm. By analysing theoretical knowledge about block cipher modes of operation, it is clear that all widespread modes of operation are similar in terms of a number of basic computational operations needed to implement them. Therefore, there should not be a significant difference between speed rates of different modes of operation for the same block cipher. This fact is true for all block ciphers implemented in Java SE; however, this is false for block ciphers DES, Triple DES and RC2 in .NET Framework – here, by unknown cause, a speed rate for the CFB mode of operation is many (approximately 7) times smaller than for other modes (furthermore, for other modes, speed rates are approximately the same). This situation for two block ciphers only (AES and Triple DES) in a graphical way is shown in Fig. 3.

Unlike hash functions, *symmetric ciphers* implemented in Java SE in some cases execute faster than those implemented in .NET Framework. For example, AES implemented in Java SE is about 20% faster than in .NET Framework. Although other ciphers (DES, Triple DES and RC2) in modes CBC and ECB run faster in .NET Framework, the same ciphers in mode CFB run faster in Java SE.

By comparing cryptographic frameworks oriented to languages with *pseudo-compilation* (Java SE, .NET Framework) and to *compiled* languages (Botan), one can see that the speed rate for all of the compared frameworks in average does not differ significantly. For example, block ciphers AES and Blowfish are faster in programs implemented using Botan, however, this fact is not true for other widespread ciphers. The difference between the speed rates for hash functions in Botan and .NET Framework largely depend on the hash function itself.

It is important to notice that during practical experiments with the test programs, there was determined that all corresponding cryptographic frameworks (their latest versions) do not yet have a native support of multi-core processors – all programs executed only on a single core of a CPU. At the end of section “Parallelisation of block ciphers”, this is shown to be a significant disadvantage of these cryptographic frameworks.

V. Methods of Maximising the Execution Speed of Cryptographic Algorithms

In software where cryptographic algorithms are widely used, the speed of execution of cryptographic algorithms is very important. A bottleneck for this speed usually is a processor on which these algorithms are executed. Therefore, in order to efficiently maximise the speed of execution of these algorithms, a processing unit should be used to the extent possible.

An obvious way how to maximise the execution speed of specific implementation of a cryptographic algorithm is to optimise the program. The optimisation can be done either manually or automatically (by a compiler). However, in some cases in order to efficiently optimise a program, some a priori data about the target hardware platform must be known. For example, if it is known that a target CPU is Intel Pentium 4 or compatible, a compiler to build a program can use SSE2 instructions, which in variety of cases provide better efficiency of the resulting program.

Talking about an execution of algorithms, it is important to note that many modern computers have video cards that can also be used for arbitrary (even not related with graphics) calculations, including cryptographic algorithms. Such technique of using graphic processing units (GPUs) for general-purpose calculations is called *general-purpose*

computing on graphics processing units (GPGPU). GPU developers provide free GPU programming libraries (or SDKs), e.g. OpenCL (Open Computing Language), CUDA by Nvidia, Stream SDK by AMD.

Therefore, nowadays (in contrast to a period of several years before) most processing systems belong to one of the following two classes:

- central processing units (CPUs);
- graphic processing units (GPUs).

All GPUs suitable for arbitrary calculations (i.e., GPGPUs) are multi-core; and a large number of modern CPUs are multi-core as well.

Modern GPUs, in contrast to CPUs, are composed of a large number of cores (of the order of several tens). (Moreover, computational power of GPUs in average is comparable to computational power of CPUs.) This means that multi-core CPUs, as well as GPUs provide a big possibility for speeding up execution of cryptographic algorithms.

A vast majority of common algorithms (including cryptographic algorithms) are defined in a sequential way (i.e., the corresponding code of an algorithm is sequential). However, the fact that nowadays most of modern processors are multi-core assumes that for a specific sequential algorithm in order to execute efficiently, it should be parallelised – i.e., divided into several maximally independent (parallel) tasks.

Thus, in order to take advantage of using multi-core processors, cryptographic algorithms should be adapted for parallel execution. However, as there was discovered by the authors (see section "Comparison of major cryptographic frameworks"), most widespread frameworks which provide a cryptographic functionality do not support advantages of multi-core processors (neither they support advantages of multi-core CPUs, nor they have a support of GPUs). This means that (at least for the moment of writing this paper) in order to get a greater performance of cryptographic algorithms, one should implement all needed cryptographic algorithms *manually*, or should use some raw third-party implementations of these algorithms. At least for some time, this way of implementing cryptographic algorithms cannot be guaranteed to be secure to a certain degree.

Automatic optimisation techniques are highly developed for single-tasking environments (where output programs are supposed to execute); but as of yet, there are many difficulties in *automatic parallelisation* of a program code – i.e., in using all the benefits of multi-tasking processing systems [11]. Therefore, for efficient implementation of sequential algorithms on multi-core processing systems, there should be considered manual parallelisation.

Despite of both CPUs and GPUs being multi-core, their architectures differ significantly. According to Flynn's taxonomy [12], multi-core CPUs have in general a *MIMD* (Multiple Instruction stream, Multiple Data stream) architecture, with each core usually having a support for a set of SIMD (Single Instruction, Multiple Data) instruction. Alternatively, all GPGPUs have a *SIMD* architecture.

The difference between the architectures of multi-core CPUs and GPGPUs substantiates differences between optimisation processes for these two types of processing systems. However, all optimisations for a SIMD architecture are also applicable to a MIMD architecture – because a MIMD architecture can be considered as a more enriched SIMD architecture. Therefore, while considering an optimisation of cryptographic algorithms for multi-core processors in this paper, firstly there are considered SIMD-optimisations, and only in case of negative result, there are considered MIMD-optimisations.

Further in this section there are discussed possibilities of parallelising block ciphers – one of the mostly widespread class of cryptographic algorithms.

A. Parallelisation of Block Ciphers

Block ciphers take as an input:

- plaintext (i.e., a message) divided into blocks of specific length and
- a key.

The result is a ciphertext consisting of the same number of blocks as the plaintext. Thus, each block cipher, in order to be able to cipher a message of any length, consists of the following three components:

- block enciphering transformation E, which enciphers a fixed-length block B (usually it is a part of plaintext) with a fixed-length key K, resulting in a block $B^{E,K}$ (whose length equals to the length of block B: $|B| = |B^{E,K}|$;
- mode of operation O, which determines a way how to calculate each block of ciphertext (with a use of block enciphering transformation E);
- padding algorithm P, which supplement plaintext in a special way, in order to make it be divisible in a whole number of blocks.

While ciphering with a block cipher, a padding algorithms is executed only once; furthermore, every padding algorithm consists only of a few instructions. Therefore, when analysing the execution speed of block ciphers, an impact of padding algorithms can be disregarded.

A mode of operation can be thought of as a meta-algorithm for executing a block enciphering transformation. Most common modes of operation are: ECB, CBC, PCBC, CFB, OFB and CTR.

If one defines the i-th block of plaintext as B_i, the i-th block of ciphertext as C_i, a bitwise XOR operation as $\oplus$, enciphering of a block B with a key K as $E^K(B)$, deciphering of a block B with a key K as $D^K(B)$ (where $D = E^{-1}$), an initialisation vector (a constant) as IV, then the common modes of operation can be defined as shown in Table 3 (initial conditions are not shown for the reason of inessentiality in this context).

TABLE 3

DEFINITION OF BLOCK CIPHER MODES OF OPERATION ($i = 1, \ldots, N$)

Mode of operation	Cipher direction	
	Enciphering	Deciphering
ECB	$C_i \leftarrow E^K(B_i)$	$B_i \leftarrow D^K(C)$
CBC	$C_i \leftarrow E^K(B_i \oplus C_{i-1})$	$B_i \leftarrow D^K(C_i) \oplus C_{i-1}$
PCBC	$C_i \leftarrow E^K(B_i \oplus B_{i-1} \oplus C_{i-1})$	$B_i \leftarrow D^K(C_i) \oplus B_{i-1} \oplus C_{i-1}$
CFB	$C_i \leftarrow E^K(C_{i-1}) \oplus B_i$	$B_i \leftarrow D^K(C_{i-1}) \oplus C_i$

OFB	$C_i \leftarrow E^K(E^K(C_{i-1})) \oplus B_i$	$B_i \leftarrow D^K(D^K(C_{i-1})) \oplus C_i$
CTR	$C_i \leftarrow E^K(IV \oplus i) \oplus B_i$	$B_i \leftarrow D^K(IV \oplus i) \oplus C_i$

TABLE 4

POSSIBILITY OF PARALLELISATION OF BLOCK CIPHERS MODES OF OPERATION DEPENDING ON A CIPHERING DIRECTION

Mode of operation	Cipher direction	
	Enciphering	Deciphering
ECB	**high**	**high**
CBC	low	**high**
PCBC	low	**high**
CFB	low	**high**
OFB	low	low
CTR	**high**	**high**

Some modes of operation in a specific ciphering direction, in order to cipher a succeeding block, require the result of ciphering the preceding block to be known. An efficient parallelisation of such *modes of operation* in the specific directions is impossible. This is because a bitwise XOR operation ("⊕"; supported directly by any CPU) is incomparably faster than an enciphering or deciphering operation (*E* or *D*); and none of the mentioned modes of operation uses other operations than ⊕, *E* or *D*.

On the contrary, the rest modes of operation (in combination with a direction of ciphering) are so-called embarrassingly parallel – which means that they can be parallelised in a most efficient way. Furthermore, they san be parallelised both in SIMD and MIMD architectures.

The summary of possible levels of parallelisation for block ciphers modes of operation is shown in Table 4. Here, the word "high" means that the corresponding modes of operation are embarrassingly parallel, and the word "low" means that the corresponding modes of operation cannot be significantly parallelised.

A demonstrable schematic illustration of an enciphering process in the CTR mode of operation is shown in Fig. 4. In this figure, one can see that all the operations for each block of plaintext are made independently on the processing of other blocks.

Surely, it is not true that an execution of block ciphers in modes of operation with a low level of parallelisation cannot be efficiently parallelised at all – only *modes of operation* themselves cannot be efficiently parallelised; one can try to parallelise *a block ciphering transformation* (that is the only thing the definitions of block ciphers – such as DES, AES, Blowfish – usually contain). However, block ciphering transformations usually noticeably differ from each other, and therefore parallelisation processes of different block cipher transformations are significantly different. Thus, for the reason of non-universality, such parallelisation is not considered in this paper.

CTR mode of operation provides both a high level of parallelisation (i.e., separation of the ciphering process into a number of parallel (independent) tasks), and a high level of security (in contrast to ECB).

In order to demonstrate a possibility of the efficient parallelisation of some modes of operation, and in order to measure a performance gain of such parallelisation, by the authors there was written a program which implements the DES cipher in the ECB mode of operation allowing to use the benefits of multi-core CPUs, and which measures the execution speed of the cipher.

As a base for the program, there was used implementation of the DES cipher found in [13], written in the C/C++ language. That implementation was modified in order to use the OpenMP API for parallel programming (see [14]). The resulting program (in C++) was tested in all of the combinations of the following two dimensions: without or with a speed optimisation of the program by a compiler, and without or with a support of OpenMP enabled. As with previous test programs, to improve an accuracy of the results, this program was executed multiple times.

The tests were held on the same computer as the previous tests described in this paper. The results obtained from the execution of the program, in their comparison with the results from Fig. 1 (viz. the execution speed of the DES cipher in Java SE, .NET Framework and Botan library), are shown in Fig. 5.

According to expectations, on a double-core CPU the execution of OpenMP-enabled version of the program was almost 2 times (more precisely – 1.88–1.91 times) faster than the execution of OpenMP-disabled version. This is because of the usage of the second core of the CPU, which was out of action in cases where the use of OpenMP was not enabled.

These results can be generalised: in case of *n*-core processor, execution speed is supposed to be almost *n* times greater than in case of single-core processor (surely, it can be not true in case of very little amount of input data).

At the end of section "Comparison of major cryptographic frameworks" there was stated that none of the implementations of cryptographic algorithms in Java SE, .NET Framework and Botan library has the support of multi-core processors. The authors regard this as a significant disadvantage of these implementations of cryptographic algorithms – so, for example, for the DES cipher executing on a double-core processor, an execution could have been from 1.99 (for .NET Framework) to almost 6.58 (for Java SE) times faster if there was a support of multi-core CPUs.

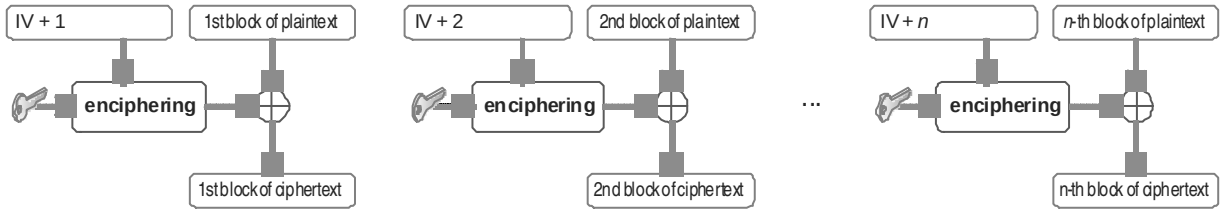

Fig. 4. A schematic view of an enciphering process in the CTR mode of operation

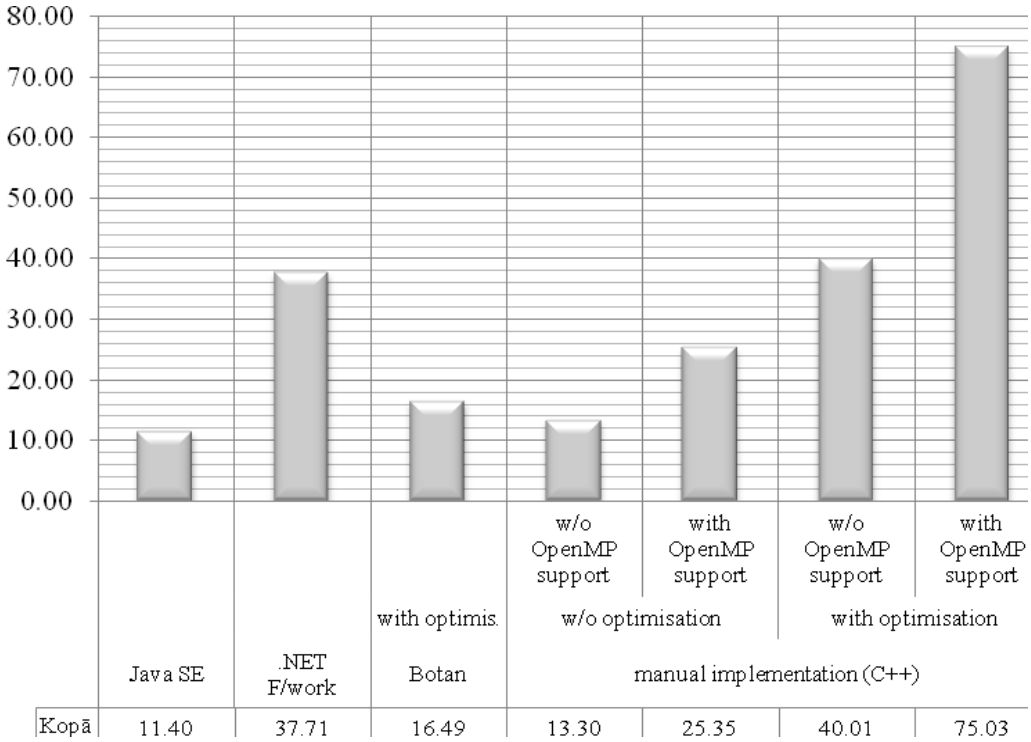

Fig. 5. An average speed rate of different implementations of the DES cipher in the ECB mode (MB/s)

Summarising the content of this section, one can conclude that block ciphers are an example of a class of cryptographic algorithms which in many cases can be executed very effectively on multi-core processors.

VI. Conclusions

The main conclusions of the research can be formulated as follows:

- Conclusions about the secure implementation of cryptographic algorithms:
  - The reason for most errors in implementation of cryptographic algorithms is an inadequate knowledge of base principles and methods of cryptography, a use of unsafe (broken) cryptographic algorithms, as well as a human factor.
  - In order to implement secure software with cryptographic functionality, one shall carefully consider a program execution mechanism and a specificity of the operating system.
- Conclusions about the speed of execution of cryptographic algorithms:
  - None of cryptographic frameworks compared can be considered as the fastest – a rank of each framework can be different depending on a specific cryptographic algorithm.
  - In general case, it is not true that cryptographic algorithms implemented in software libraries written in compiled languages (e.g. C++) are faster than algorithms implemented in software platforms Java SE and Microsoft .NET Framework. However, as it was shown on the example of implementation of the DES block cipher, a manual implementation usually gives a higher execution speed.
  - Multi-core central processing units and also graphics processing units can be used to speed up an execution of cryptographic algorithms several times.
  - Cryptographic frameworks analysed do not have a native support of multi-core processors.
  - If there is needed faster implementation of cryptographic algorithms than cryptographic frameworks compared provide, one shall implement these algorithms manually, using the advantages of processing system. For example, as it was shown in

the paper, manual implementation can run up to 2–6 times faster than the implementation from a software library. However, in this situation, the resulting security level can be very low.

ACKNOWLEDGEMENTS

This work has been supported by the European Social Fund within the project "Support for the implementation of doctoral studies at Riga Technical University" (agreement No. 2009/ 0144/1DP/1.1.2.1.2/09/IPIA/ VIAA/005).

REFERENCES

[1] B. Schneier, *Applied Cryptography*, 2nd ed. John Wiley & Sons, 1996.
[2] J. Black, M. Cochran, and T. Highland, "A study of the MD5 attacks: insights and improvements." [Online]. Available: http://www.cs.colorado.edu/~jrblack/papers/md5e-full.pdf. [Accessed: October 2009].
[3] M. Stevens, A. Lenstra, and B. de Weger, "Vulnerability of software integrity and code signing applications to chosen-prefix collisions for MD5." [Online]. Available: http://www.win.tue.nl/hashclash/ SoftIntCodeSign. [Accessed: October 2009].
[4] T. Asano, "Secure and insecure modifications of the subset difference broadcast encryption scheme," in *Lecture Notes in Computer Science*, 2004, pp. 12-23.
[5] F. Muller, "The MD2 hash function is not one-way," in *Lecture Notes in Computer Science*, Vol. 3329, 2004, pp. 214-229.
[6] X. Wang, D. Feng, X. Lai, and H. Yu, "Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD." [Online]. Available: http://www.insidepro.com/doc/199.pdf. [Accessed: October 2009].
[7] B. Schneier, "Cryptography: the importance of not being different." [Online]. Available: http://www.schneier.com/essay-189.html. [Accessed: October 2009].
[8] N. Ferguson and B. Schneier, *Practical Cryptography*. John Wiley & Sons, 2003.
[9] "dotnetpowered Language List." [Online]. Available: http://www.dotnetpowered.com/languages.aspx. [Accessed: October 2009].
[10] R. Tolksdorf, "Programming languages for the Java virtual machine," is-research GmbH. [Online]. Available: http://www.is-research.de/info/vmlanguages. [Accessed: October 2009].
[11] L. Schwartz, "Method for automatic parallelization of software," US Patent 6,708,331. [Online]. Available: http://www.google.com/patents/ about?id=ZaQSAAAAEBAJ&dq=6708331. [Accessed: October 2009].
[12] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. C-21, 1972, pp. 948.
[13] GnuPG source code, Free Software Foundation, Inc. [Online]. Available: http://www.koders.com/c/fidBFE3AB8285EA2DD9405B37FCD987B38F39444942.aspx. [Accessed: 2009].
[14] OpenMP. [Online]. Available: http://openmp.org/wp/about-openmp. [Accessed: October 2009].

Vladislav Nazaruk was born in 1986. He earned a degree of bachelor of engineering sciences (Bc.sc.ing.) in 2007, a degree of master of engineering sciences (Mg.sc.ing.) in 2009 – both at Riga Technical University, Latvia. Now he is a doctoral student at Riga Technical University.

He works as Information Systems Engineer at the Department of Applied Computer Science, Riga Technical University.

Fields of interests: computer science, mathematics, pedagogy. Special interests: algorithms, information theory, protection of information.

Pavel Rusakov was born in 1972. He earned a degree of master of engineering sciences (Mg.sc.ing.) in 1995, a degree of doctor of engineering sciences (Dr.sc.ing.) in 1998 – both at Riga Technical University, Latvia.

He works as Associated Professor at the Institute of Applied Computer Systems, Riga Technical University and Head of the Laboratory, responsible for the professional bachelor/master studies at Department of Applied Computer Science, Riga Technical University.

Field of interests: computer science. Special interests: programming paradigms, parallel computing, Web technologies, distributed systems, computer graphics, protection of information.

Vladislavs Nazaruks, Pāvels Rusakovs. Kriptogrāfisko algoritmu implementēšana programmatūrā: efektivitātes analīze
Šī raksta mērķis ir aplūkot iespējas efektīvi implementēt kriptogrāfiskos algoritmus programmatūras izstrādē. Tiek skarti divi jēdziena «efektivitāte» aspekti: efektivitāte kā kriptogrāfisko algoritmu izmantošanas mērķa (informācijas aizsardzības nodrošināšanas) sasniegšanas rādītājs, un efektivitāte kā kriptogrāfisko algoritmu implementēšanas procesa ekonomiskuma, kā arī jau implementēto algoritmu izpildes ekonomiskuma (galvenokārt, laika resursu ziņā) rādītājs. Rakstā tiek definētas un klasificētas kļūdas, kas var izraisīt ievainojamības kriptogrāfisko algoritmu implementācijā ar datoru; tiek sniegtas vispārējās rekomendācijas, kā var izvairīties no šādām ievainojamībām. Arī tiek aplūkota kriptogrāfisko algoritmu implementēšana, izmantojot programmatūras bibliotēkas un platformas. Vairākas izplatītas programmatūras platformas, kas piedāvā kriptogrāfisko funkcionalitāti, tiek salīdzinātas savā starpā pēc algoritmu izpildes ātruma kritērija. Visbeidzot, tiek aplūkotas dažas iespējas palielināt kriptogrāfisko algoritmu izpildes ātrumu, izceļot algoritmu paralelizācijas svarīgumu. Bloku šifriem iespējamā paralelizācija tiek aplūkota īpaši dziļi; arī, atsaucoties uz praktisko eksperimentu rezultātiem, tiek demonstrētas un novērtētas bloku šifru paralelizācijas priekšrocības.

Владислав Назарук, Павел Русаков. Реализация криптографических алгоритмов в программном обеспечении: анализ эффективности
Целью данной статьи является рассмотрение эффективной реализации криптографических алгоритмов в разработке программного обеспечения. Затрагиваются два аспекта термина «эффективность»: эффективность как показатель выполнения цели использования криптографических алгоритмов – обеспечения защиты информации, и эффективность как показатель экономичного процесса реализации криптографических алгоритмов, а также экономичного исполнения реализованных алгоритмов. В статье определяются и классифицируются ошибки, которые могут вызвать уязвимости в реализации криптографических алгоритмов на компьютере; даются общие рекомендации, как возможно избежать таких уязвимостей. Также рассматривается реализация криптографических алгоритмов, используя программные библиотеки и платформы. Несколько распространённых программных платформ, предоставляющих криптографическую функциональность, сравниваются между собой по критерию скорости выполнения алгоритмов. В заключение, рассматриваются некоторые возможности увеличения скорости исполнения криптографических алгоритмов, выделяя при этом важность распараллеливания алгоритмов. Для блочных шифров возможное распараллеливание рассматривается особенно глубоко; также ссылаясь на результаты, полученные в ходе практических экспериментов, демонстрирующие и оценивающие преимущества распараллеливания блочных шифров.