© 2005 - 2015 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ANALYSIS OF USING EQUIVALENT INSTRUCTIONS AT THE HIDDEN EMBEDDING OF INFORMATION INTO THE EXECUTABLE FILES

STANISLAV IGOREVICH SHTERENBERG, ANDREY VLADIMIROVICH KRASOV, IGOR ALEKSANDROVICH USHAKOV

Federal State Educational Budget-Financed Institution of Higher Vocational Education
The Bonch-Bruevich Saint-Petersburg State University of Telecommunications
Prospekt Bolshevikov, 22, St. Petersburg, 193232, Russian Federation

ABSTRACT

The article examines the problems and their possible solutions on the specifics of using equivalent instructions for the hidden embedding of information into the executable and library files. This article considers an example of hidden embedding into an executable file of exe format or into a similar elf-format for the Unix/Linux systems. In order to embed information into an executable code by the synonyms substitution method, it is suggested to use instructions that execute one and the same operation and have similar length, but that are encrypted by different opcodes. Such instructions are to be called equivalent instructions. The idea of semantic substitution must be implemented in the executable and library files. An executable code, and not an executable file itself, is used as a container for the information embedding.

Keywords: *Machine Code, Authentication Of The Executable Code, Executable File, Equivalent Instructions, Steganography.*

1. INTRODUCTION

This article examines an element of interaction with the static libraries. The Linker combines object files of libraries and object files of your program into a single executable file (for example, exe format for Windows or elf for Linux).

The reassembled executable files enable using alphabetical mnemonic operation codes, assigning symbolic names to the computer and memory registers in one's sole discretion as well as specifying personally convenient addressing schemes (or example, an index or indirect scheme). Unlike a natural language, a low-level programming language has a limited vocabulary (language statements) and a strict set of rules for using that vocabulary, while the semantic rules the same as for any formal language are clearly and unambiguously defined. In this article we present an example and method for semantic substitution of language statements, in order to implement methods for the equivalent replacement of statements, which will allow for provision of copyright protection at the technical level when using digital signatures [4].

An executable code, and not an executable file itself, is used as a container for the information embedding. Such method of embedding is based on the features of an executable code of particular processor architecture and does not depend on the format of an executable file. It is much more difficult to discover the fact of information embedding into an executable code, since there is no one-to-one association between the source code of the program and the compiled executable code. One and the same source code can be associated with various executable codes depending on the compiler and its settings [2].

The x86 family processors have an excessive set of instructions. One and the same action can be executed with the help of different instructions. Such redundancy can be used for hidden embedding of information into an executable code without disrupting its integrity. This article discusses and analyses the methods for hidden embedding of information that do not change the code size and use a general purpose instruction set [15], [16].

© 2005 - 2015 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

2. METHODOLOGY

Many instructions with 2 operands contain in their opcode a direction bit indicating which operand is a source, and which – a receiver. Such instructions include adc, add, and, cmp, mov, or, sbb, sub, and xor. Thus, for example, the forms of the add instruction (add reg, r/m, and add r/m, reg) vary by the direction bit's value and, accordingly, have different opcodes.

The first instruction adds values from the register or memory (depending on the ModRM byte's content) to the register's value. The second instruction adds register's value to the value that is in the register or memory. Thus, if two operands are served by the registers, the add instruction can be encoded by any of the methods presented. Table 1 shows an example of such encoding. The add eax, ebx instruction can be assembled as 03 BA or 05 E1 (in hexadecimal notation) (Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A, 2008; Intel 64 and IA-32 Architectures Software Developer's Manual, Vo lume 2B, 2008).

Table 1. Equivalent Variants Of The Add Edx, Ecx Instruction Encoding

add eax, ebx			
add r/m, reg		add reg, r/m	
Opcode	ModRM byte	Opcode	ModRM byte
0000 0011	10111010	0000 0101	11100001
03	BA	05	E1

Some instructions working with an immediate value can be replaced by the inverse ones. In this case, an immediate value must be recalculated.

For example, the inverse instructions include add and sub or ror and rol.

The immediate values are to be calculated as follows:

- For the add and sub instructions:

$$imm2 = (not\ imm1 + 1) \bmod 2^{size} \quad (1)$$

- For the ror and rol instructions:

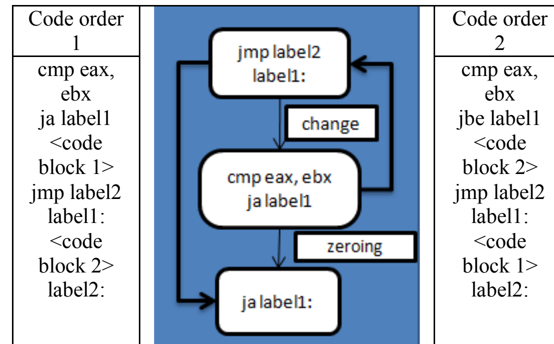
$$imm2 = (size - imm1) \bmod size \quad (2)$$

In both expressions, imm1 and imm2 are the immediate values for the inverse instructions; size is the size of a register, over which an operation is executed [9].

The xor reg, reg instruction is often used for resetting a register. The same operation can be carried out with the help of the sub reg, reg instruction. In some cases the interchangeable instructions differently change the eflags register's flags. Therefore, one instruction can be replaced by another one only in the event if within the limits to the next instruction, which changes the flags, the flags values do not affect the program flow, i.e. there are no instructions depending on the eflags register, such as conditional jumps. If to swap the operands in the instruction of comparison, the instruction obtained will set the opposite flags. Thus, it will be enough to replace the commands of a conditional jump by the opposite ones on the interval from the instruction of comparison to the next instruction that changes the flags (for example, "cmp eax, ebx ja label1" by "cmp ebx, eax jbe label1") [9].

Besides, it is also possible to change the sequence order of the code's functionally independent blocks (Table 2).

Table 2. The Equivalent Sequences Of Instructions With The Changed Order Of The Code's Functionally Independent Blocks

<table>
<tr><td>Code order 1</td><td rowspan="2">
</td><td>Code order 2</td></tr>
<tr><td>cmp eax, ebx
ja label1
<code block 1>
jmp label2
label1:
<code block 2>
label2:</td><td>cmp eax, ebx
jbe label1
<code block 2>
jmp label2
label1:
<code block 1>
label2:</td></tr>
</table>

In order to make the process of identifying the embedded information in an executable code as complicated as possible, it is required that the statistics of certain instructions in the code with embedded information would minimally differ from the similar statistics in the code without embedded information. This article examines the statistics of instructions occurrence in the executable code without embedded information, and analyses the most suitable for the embedding instructions [10].

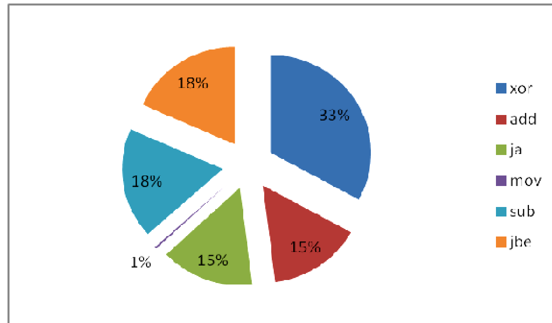
For this study, the authors have chosen the instructions for adding and subtracting a consonant

© 2005 - 2015 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

with the help of add/sub, resetting the register with the help of xor/sub, and the opposite conditional jumps. In addition, a semantic substitution ja/jbe is shown. Figures 1a and 1b demonstrate the distribution diagrams of each of the equivalent instructions variants and ratio of the variants of the semantic substitutions of statements.

	no optimize	size	time
xor	99	99	99
add	45	60	60
ja	45	45	50
mov	1	1	1
sub	55	40	40
jbe	55	55	50

a)



b)

Figure 1. A) Distribution Of The Equivalent Instructions Variants In The Executable Code Without Optimization, With Optimization By Size And Time; B) Ratio Of Variants Of The Semantic Substitutions Of Statements.

It is evident that in order to reset a register, the GCC compiler always uses the xor instruction. This means that the embedding through the substitution of the add/sub, ja/jbe instructions can be easily detected. The rest of the instructions synonyms are distributed relatively uniformly and can be used for embedding information. [19].

Also, the authors have analyzed the frequency of the instructions, using registers as both operands, occurrence in the executable code compiled at various variants of optimization (Figure 2).

The most frequent mov and xor instructions are the best suitable for embedding information.

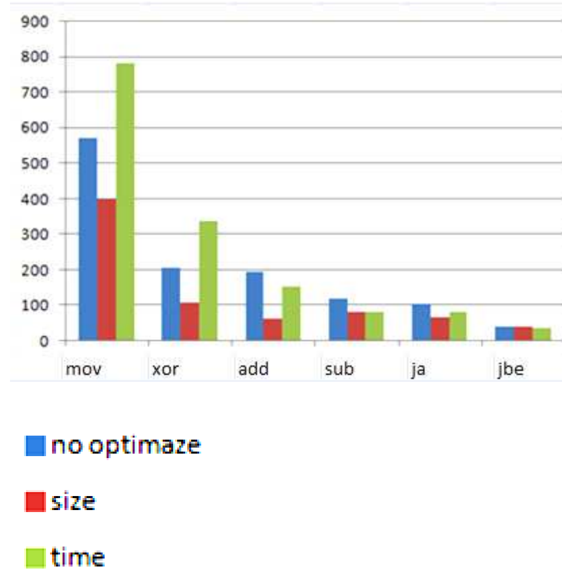


Figure 2. Number Of Instructions In An Executable Code That Use Registers As Both Operands.

Besides, the authors have calculated the amount of information, which can be embedded into an executable code (Table 3). The size of the executable code considered amounted to 10,122 bytes [12].

Table 3. Evaluation Of The Information Volume Being Embedded

Type of embedding	Number of instructions	Ratio of the embedded information volume to the code volume
Substitution of *xor-mov*	1,502	0.4
Substitution of *add-sub*	455	0.3
Substitution of *ja-jbe*	321	0.1
Exchange of the *cmp* operands	121	0.1
Addressing of conditional jumps	1,584	0.2

Therefore, when using the xor-mov substitution, it is possible to embed 1 KB of hidden information into each 100 KB of a container [9].

3. RESULTS AND DISCUSSION

Journal of Theoretical and Applied Information Technology
10th October 2015. Vol.80. No.1
© 2005 - 2015 JATIT & LLS. All rights reserved·

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The following listing demonstrates an example of embedding into the executable code when using semantic substitution of the equivalent xor-mov statements. The experiments on implementation of the hidden embedding require an executable file. A code sample is shown in Listing 1.

```
…….! main:                           ;xref o80482d7
…….! push       ebp
8048385 ! mov          ebp, esp
8048387 ! sub          esp, 8
804838a ! and          esp, 0fffffff0h
804838d ! mov          eax, 0
8048392 ! sub          esp, eax
8048394 ! mov          dword     ptr      [esp],
strz_STEGO_coder
804839b ! call         wrapper_123
80483a0 ! leave
80483a1 ! ret
80483a2 ! nop
…………….
80483an ! nop
```

Listing 1. Place for embedding information

It is required to replace pop esi/xor ebp, ebp by jmp by our code. With that it becomes possible to perform all the conceived actions, to execute these commands, and return back. But first it is necessary to prepare a code for integration [1], [3].

To make it simple, we will display a short greeting. In assembly language it will sound approximately like this:

```
mov eax, 4                ; system call write
mov ebx,1                 ; standard output ID
mov ecx, offset begin_msg    ; a pointer to the first character of the message being displayed
mov edx, offset end_msg      ; a pointer to the last character of the message being displayed
int 80h                   ; displaying
pop esi                   ; saved commands
xor ebx,ebp
jmp 80482C3h              ; back to the program
```

Listing 2. A short greeting with an on-screen

This is not the most optimum variant, and it can be optimized if to rewrite as follows:

```
xor eax,eax
add al, 4
xor ebx,ebx
inc ebx
mov ecx, offset begin_msg
mov edx,ecx
add edx, sizeof(msg)
int 80h
pop esi
xor ebp, ebp
jmp 80482C3h
```

Listing 3. Optimization of the output variant

When executing file in a hex-editor, let us find and write out the start addresses of all chains of NOPs suitable for implementation. If two adjacent chains are located within a short jump (roughly – within a hundred bytes), then three NOPs will be quite enough (2 bytes for a jump command, one – for any single-byte command of the useful code, for example, inc ebx or pop esi). Otherwise, a chain of at least 6 NOs is required: five – for a short jump command and one – for a useful command [17].

In this case, it turns out as follows:

8048306h 10 bytes

80483a2 14 bytes

8048464 12 bytes

Total – 36 bytes.

This place is taken to demonstrate a semantic substitution. Then the filling of the chain of NOPs with the useful code starts. After the first attempt, it looks as follows:

```
8048306    31 c0            xor    eax, eax
8048308    04 04            add    al, 4
804830a    e9 93 00 00 00   jmp    80483a2h
804830f    90               nop
```

At the same time, the last one NOP remains lost, but there is no other way. The XOR EBX,EBX command needs two bytes and cannot be executed here. Now it is time to implement the methods of semantic (equivalent) substitution of statements. Then, it is required to transfer add al,4 to the next NOP chain, and to insert XOR EBX,EBX/INC EBX instead of it [15], [16], [13].

Journal of Theoretical and Applied Information Technology
10th October 2015. Vol.80. No.1
© 2005 - 2015 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
8048306    31c0            xor   eax, eax
8048308    31db            xor   ebx, ebx
804830a    43              inc   ebx
804830b    e9 92 00 00 00  jmp   80483a2h
```

Then, the next chain will be filled as follows:

```
80483a2    0404            add   al, 4
80483a4    b9 ?? ?? ?? ??  mov   ecx, offset begin_msg
80483a9    89ca            mov   edx, ecx
80483ab    e9 b4 00 00 00  jmp   8048464h
```

Listing 4. An implemented example of the semantic substitution XOR EBX,EBX/INC EBX.

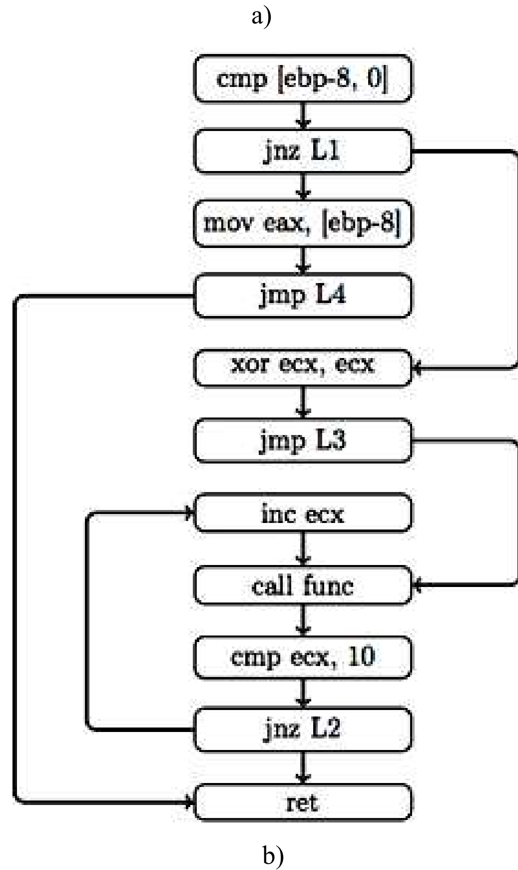
1. The rest of the code is too big in size to be included into the third last chain of NOPs, as it lacks one single byte. It is required to try to shrink the code a little bit more. For example, the pairs of *mov edx,ecx/add edx,sizeof(msg)* instructions with the size of 5 bytes can be used lea edx,[ecx+sizeof(msg)] [11].

The message can also be placed in the data segment. Since there is not so much free space there, we can restrict ourselves to the string "hello". There is no need for a null-terminated string, as the system call "write" displays exactly as many characters as it is set to display. At this stage the example of the semantic substitution is considered to be completed [14].

An executable code is formed on the basis of a graph of the function execution flow. This method can be used for verifying the integrity of separate segments of an executable file of the .exe format for implementation of the hidden embedding of information. A machine code of the executable file's function (Figure 3a) can be represented in the form of a scheme (Figure 3b), in which vertexes the instructions are placed, and the edges correspond to the possible control jumps between them. Such scheme has only one initial command and at least one final command. The source of this graph is the instruction located at the function's entry point address (for example, cmp or mov), and stocks are the instructions of return from the function (for example, ret).

```
    cmp [ebp-8], 0
    jnz L1
    mov eax, [edp-8]
    jmp L4
L1: xor ecx, ecx
    jmp L3
L2: inc ecx
L3: call func
    cmp ecx, 10
    jnz L2
L4: ret
```

a)



b)

Figure 3. A) Assembly Code Of The Executable File's Function;
B) Assembly Code Of The Function Represented In The Form Of A Scheme.

The scheme representing the function in such a form can contain certain cycles, so it must be transformed into a simply connected directed acyclic graph by running the depth first search algorithm from the graph's source. At the algorithm's output a depth first search tree will be

© 2005 - 2015 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

obtained $T=(V,E')$, the root of which is the function entry point.

This tree must be binary, since in the original graph, no more than two edges come from each its vertex. The obtained binary tree T of the height h can be compared to the binary heap of the height h (Figure 4), and unambiguously assign the sequence $m=\{0,1\}^n$ to the tree T, where $n=\sum_{i=0}^{h}2^i$, according to the following rule:

$$m_i = \begin{cases} 1, \text{by index i of the bnary heap, the} \\ 0, \text{by index i of the binary heap, the} \end{cases} \quad (3)$$

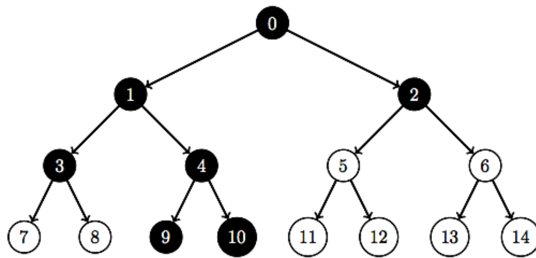


Figure 4. Concordance Between The Tree T And The Binary Heap.
Painted Are The Vertexes Entering Into T (M = 1 11 1100 00110000)

The numerical representation of the function control flow graph is used to generate a hidden embedding into an executable file. The above entered number m is exponentially dependent on the height of the tree T. Therefore, it makes sense to apply to it a hash function, so that the numerical representation would have a fixed size (Figure 5). Changing the instructions to the equivalent ones does not disturb the function control flow graph, and hence it is possible to embed into the executable code of the same function by the method of synonyms substitution [7].

When verifying the code validity, the same way is used to generate the numerical representation of the function control flow graph m. Afterwards, it is compared to the function extracted from the executable code [18].

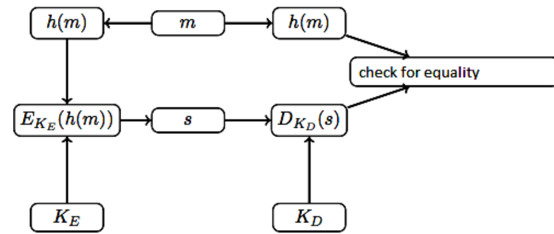


Figure 5. Scheme Of The Executable Code Authentication

4. CONCLUSION

Using the example of the executable code authentication scheme, it is possible to explain why exactly these methods of embedding into the structure of an executable file and embedding into an executable code are implemented. With the help of these methods, an "attacker" can be distracted by the instructions considered that do not change the program control flow and by the vertexes presented with one exiting edge. Replacement of such instructions by the equivalent ones does not change the function control flow graph. Therefore, after embedding information into an executable code, its numerical representation, which was used for generating a digital signature, remains unchanged. [15].

Therefore, it is proposed to use instructions that execute one and the same operation and possess similar lengths for embedding digital signature into an executable code by the method of synonyms substitution. The idea of the semantic substitution of statements will allow distracting an "attacker" when verifying the integrity of the separate segments of an executable code. It will be difficult to notice the hidden embedding integrated into the file because the changes inside the file will affect neither the size nor the functionality of the executable code. The method suggested is simple for implementation and cost-effective. On the shown example with the use of a NOP chain, it is evident that the equivalent substitution of the mov and xor statements, proposed in this work, is highly effective and is being successfully implemented.

The methodology of using semantic and equivalent substitutions will allow implementing the possibility of signing not only the executable files but also the library files, since all the necessary functions are included in a single executable file. The hidden embedding implemented through the method of equivalent substitution of statements

© 2005 - 2015 JATIT & LLS. All rights reserved·

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

allows for copyright protection at the technical level by using a digital signature.

REFERENCES:

[1] *ELF. Files Structure*. (n.d.). Retrieved May 30, 2015, from http://www.gfs-team.ru/articles/read/149.

[2] Gribunin, V., Okov, I., & Turintsev, I. (2002). *Digital Steganography* (pp. 227). Moscow: Solon-Press.

[3] *Implementation of Additional .ELF Segments in QNX.* (2014). Retrieved May 30, 2015, from http://web.archive.org/web/20081120013357/http://qnx.org.ru/article9.html.

[4] Cox, I. (2008). *Digital Watermarking and Steganography* (pp. 593). Second Edition: Morgan Kaufmann.

[5] *Intel 64 and IA-32 Architectures Software Developer's Manual.* (2008). Volume 2A: Instruction Set Reference. A-M: Intel Corp.

[6] *Intel 64 and IA-32 Architectures Software Developer's Manual.* (2008). Volume 2B: Instruction Set Reference. N-Z: Intel Corp.

[7] Krasov, A., & Shterenberg, S. (2013). Development of Methods for the Software Copyright Protection Based on the Digital Watermarks Integrated into the Executable and Library Files. *Actual Problems of Infocommunication in Science and Education, Saint Petersburg,* pp.847-852.

[8] Krasov, A., & Vereshchagin, A. (2012). Certificate of State Registration of a Computer Program No. 2013612237. *Program for Embedding Digital Watermarks into the Executable and Library Files.* Copyright holder: Federal State Educational Budget-Financed Institution of Higher Vocational Education the Bonch-Bruevich St. Petersburg State University of Telecommunications. Incoming date: December 25, 2012. Registered in the Register of Computer Programs on February 18, 2013.

[9] Krasov, A., Vereshchagin, A., Abaturov, V., & Reznik, M. (2012). Methods for Hidden Embedding of Information into the Executable Files. *Proceedings of the Saint Petersburg Electrotechnical University "LETI", Saint Petersburg*, 8, pp.51-55.

[10] Krasov, A., Vereshchagin, A., & Tsvetkov, A. (2013). *Software Authentication Using the Embedding of Digital Watermarks into an Executable Code. Telecommunications*, (pp. 27-30). Moscow.

[11] Norton, P., & Souhe, D. (1992). *Assembly language for IBM PC*. Moscow: Computer.

[12] Venkatesan, R., Vazirani, V., & Sinha, S. (2001). A Graph Theoretic Approach to Software Watermarking. *4th International Information Hiding Workshop*, pp.157-168.

[13] Secrets of the Elves Conquest. (20056 November). *Hacker Journal*, 83. Retrieved May 30, 2015, from http://inf.tltsu.ru/res/strogov/j/xa083/106/6.htm.

[14] Shterenberg, S., & Andrianov, V. (2013). Variants of Modifying the Structure of the Executable Files of PE Format. *Prospects for Development of Information Technologies, Novosibirsk,* pp.134-143.

[15] Shterenberg, S., & Andrianov, V. (2014). Methods of Selecting the Optimal Methods for Copyright Protection on Unix Systems Using the Executable and Library Files. *Efektivní Nástroje Moderních Věd, Praha.*

[16] Shterenberg, S., & Andrianov, V. (2014). Study of the Adaptive Attacks Methodology Based on the Hidden Embedding into the Executable Files. *Science, Technology, Innovation, Bryansk*, pp.287-294.

[17] Shterenberg, S., Andrianov, V., Lipatnikov, V., & Kostarev, S. (2015). Certificate of State Registration of a Computer Program No. 2015611539. *RPA (Rationable Progressimo Aggredi) (lat.)*. Copyright holder: Federal State Educational Budget-Financed Institution of Higher Vocational Education the Bonch-Bruevich St. Petersburg State University of Telecommunications. Incoming date: December 2, 2014. Registered in the Register of Computer Programs on January 30, 2015.

[18] Shterenberg, S., & Prosikhin, V. (2014). Methodology for Applying a Mathematical Model of Stegosystem for the Hidden Embedding of Information. *Eurasian Union of Scientists, Moscow*, 9.

[19] Stern, J., Hachez, G., Koeune, F., & Quisquater, J. (1999). Robust Object Watermarking: Application to Code. *Information Hiding*, pp.368-378.