VOL. 13, NO. 8, APRIL 2018 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences

www.arpnjournals.com

# EMBEDDING THE HIDDEN INFORMATION INTO JAVA BYTE CODE BASED ON OPERANDS' INTERCHANGING

Andrey Vladimirovich Krasov, Aleksander Sergeevich Arshinov and Igor Aleksandrovich Ushakov
Federal State Budget-Financed Educational Institution of Higher Education The Bonch-Bruevich Saint - Petersburg State University of Telecommunications Russia, Saint-Petersburg, Prospect Bolshevikov, Russia
E-Mail: andrey.v.krasov@mail.ru

**ABSTRACT**

Software piracy becomes more and more a serious issue by the day, software companies lose money because of that. In this article the technique called software watermarking has been considered. A digital watermark is embedded by means of slightly changes of the executable file. Such changes must not be found and not change the program's logic. The system of embedding digital watermarks into executable java files by means of operands' interchanging has been designed. The designed system does not change the file volume and the execution time. This system can be used to protect copyright on software or on a part of it.

**Keywords:** java virtual machine, information embedding, executable files, byte code, digital watermarks.

## 1. INTRODUCTION

Using unlicensed program software is detrimental to companies who develop them. In order to prevent software piracy different means are being used, one of them is digital watermarking. The main idea of digital watermarking could be described as the special message that is embedded in every program copy with special algorithms [1]. The embedded message allows to identify a pirate copy and to find a source file from which it has been copied [2]. The main requirements for algorithm are do not change program's logic and not to be detectable [3]. As of today, programming systems, which compile programs not into executable machine code, but into virtual machine code, became wide spread. One of such virtual machines is Java Virtual Machine.

In this article we will discuss a method of information hiding into files, being executed by Java Virtual Machine. This topic has been selected because there are no papers devoted to the description of watermarking algorithms for java files since the most papers on digital watermarking are devoted to embedding into audio, video or image files, and furthermore, a small part of papers on digital watermarking for executable files are devoted to exe files [4] [5].

Embedding methods, which implement executable files in the function of covering message, allow embedding smaller amount of information in contrast to other covering messages (i.e. video, audio files, graphical files). However, they allow solving tasks of joint usage of embedded information with software. Here are some examples of such usage [6]:

- Digital signature embedding;
- Embedding of copying operation counter;
- Embedding of license #;
- Embedding of information about copyright;
- Embedding of information about programs' integrity and its separate parts;

## 2. JAVA BYTE CODE

Compiler converts initial java text into byte code, which is executed on Java virtual machine (JVM) and does not depend on processor's architecture. Java virtual machine is delivered as virtual stack machine. In the function of data structure, where operands are placed, stack is used. Operations receive data from a stack, process it and put the results into a stack based on the LIFO rule (the last came, the first gone). The example of how a*4+b expression is calculated on a virtual machine is presented in Table-1.

**Table-1.** Expression calculation.

| Bytecode | Command | Operation with stack |
|---|---|---|
| 0x1b | iload_1 | Insert variable a into a stack |
| 0x05 | iconst_2 | Insert number 4 into a stack |
| 0x68 | imul | Remove from stack number 4 and a renamed a, multiply it and then put the result into a stack. |
| 0x1c | iload_2 | Put a variable b into a stack |
| 0x60 | iadd | Remove variable b and the result of multiplying, total it and then put the result into a stack |

The structure of executable class file is described in Table-2. Any method or procedure may have unlimited number of attributes, some of which may have their own attributes. In order to go directly to the method's byte-code, it's necessary to look in detail at one of its attributes, the Code attribute, which contains a sequence of byte-code instructions, which describes the logic of the method.

**Table-2.** Structure of class file.

| Number of bytes | Fields |
|---|---|
| 4 | Identifier of the class file format |
| 2 | Minor Version |
| 2 | Major Version |
| 2 | Number of constant in pool |
| Depends on number of commands | Constants pool |
| 2 | Access flags |
| 2 | Link to constant class name in pool |
| 2 | Link to constant with name of super class |
| 2 | Interfaces count |
| Continuation of the Table 1. | |
| 2 on each interface | Interfaces |
| 2 | Fields count |
| Depends on fields and number of attributes | Fields |
| 2 | Methods count |
| Depends on methods and number of attributes | Methods |
| 2 | Attributes count |
| Depends on attributes and their number | Attributes |

Operation code (or Opcode) field is a mandatory field in the command format. It defines action over data blocks, which are called operands. Operands can be set implicitly in cases when the operation code determines them. When operands are explicitly declared, they are either present in the command itself, or represented by their addresses in the so-called targeted fields. Commands operands with targeted fields are hardly used. The addressing part of the command, which serves to indicate location of the operand address, contains address of OP cells or numbers of general-purpose registers (GPR).

## 3. METHODS

Let's list possible methods of embedding into executable files [6] [7]:

A. Changing the sequence of operations.
B. Replacement of constructs for equivalent.
a) the replacement of existing branch of code
b) the replacement for equivalent mathematical operations.
C. Inserting operations, which do not change the program logic.

Methods comparison is shown in Table 3.

**Table-3.** Methods comparison.

| Embedding method | Changing of sequence | Replacement of constructs for equivalent | Insert of operations |
|---|---|---|---|
| Increases file volume | No | Yes* | Yes |
| Increases file execution time | No | Yes* | Yes |
| Volume of embedding | limited | limited | unlimited |

Source: * mathematical operations

The first method for information embedding into executable file will be discussed since it doesn't change the size of executable file and it doesn't change the speed of execution. This method will be described in detail in the next section.

The second method - replacement of constructs for equivalent. One of the options is the replacement of existing branch of code. Each branch, written in Java, always can be replaced by a similar one, but with opposite logical expression. In this case, the logic of this branch will be changed [8]. To prevent this, along with the replacement of a logical expression, it's necessary to replace the branches. For illustration, consider an example

of pseudo-code in Figure-1, there is an example in a byte-code in Table-4.

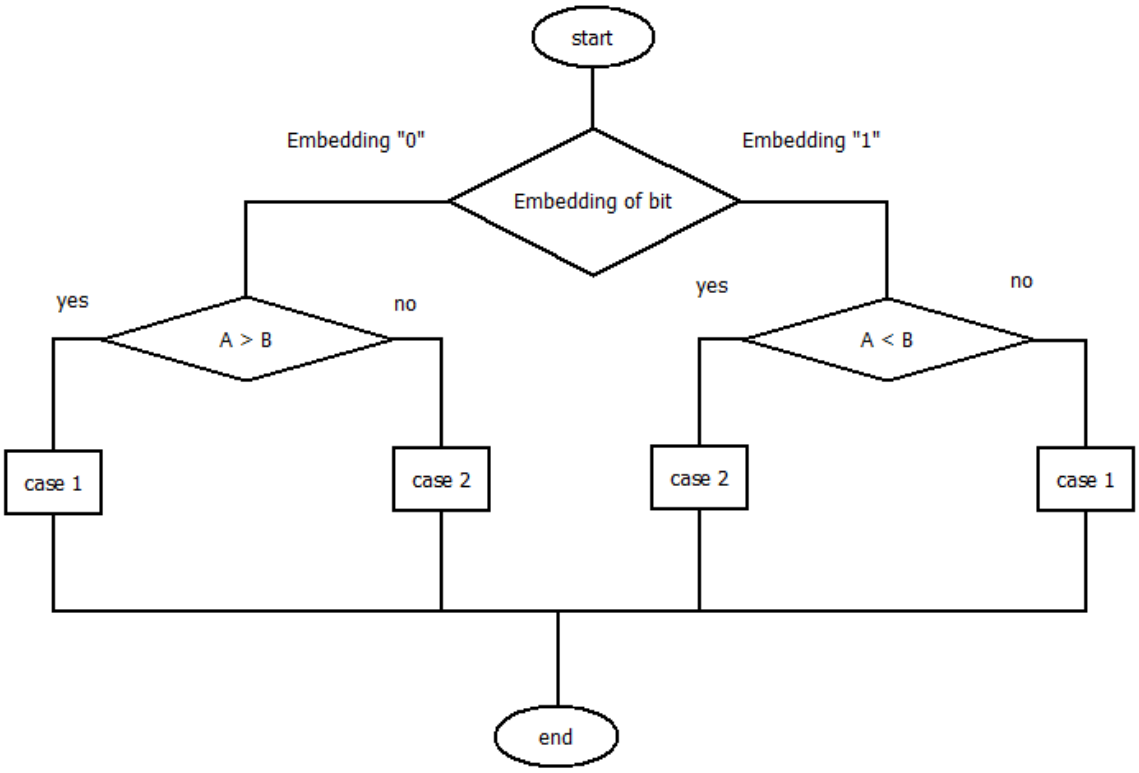


**Figure-1.** Execution of commands in different variants of code.

In the figure above, block "case 1" will be executed in case of the truth, and block "Case 2" in case of falsity of conditions. If you set the contrary condition, for example, "a <= b", and swap "Case 1" and "Case 2" places, the logic implementation of such a code in the programming language will not be changed.

**Table-4.** Embedding of 1 bit.

| Source code | Byte code | Commands | Recoverable bit |
|---|---|---|---|
| if ( a > b )<br>{<br>c = 0;<br>}<br>else<br>{<br>c = 1;<br>} | 0ч1и<br>0ч1c<br>0чф4<br>0ч03<br>0ч3y<br>0чф7<br>0ч04<br>0ч3y | iload_1<br>iload_2<br>if_icmple 14<br>iconst_0<br>istore_3<br>goto 16<br>iconst_1<br>istore_3 | 0 |
| if ( a < b )<br>{<br>c = 1;<br>}<br>else<br>{<br>c = 0;<br>} | 0x1b<br>0x1c<br>0xa2<br>0x04<br>0x3e<br>0xa7<br>0x03<br>0x3e | iload_1<br>iload_2<br>if_icmpge 14<br>iconst_1<br>istore_3<br>goto 16<br>iconst_0<br>istore_3 | 1 |

Thus, having N branches in one or more methods, and taking one version as zero, and other as one, it is possible to embed sequence of N bits. Such embedding may be possible both at the level of source code of programs, and at the level of the class files of virtual Java machine.

The second option of replacement of constructs for equivalent is replacement for equivalent mathematical operations [9]. There is an example in Table-5; it is based

on the replacement of addition and subtraction. The size increases by one command by embedding 1.

**Table-5.** Embedding of 1 bit.

| Source code | Byte code | Commands | Recoverable bit |
|---|---|---|---|
| int c = a + b; | 0x1b<br>0x1c<br>0x60<br>0x3e | iload_1<br>iload_2<br>iadd<br>istore_3 | 0 |
| int c = a -(-b); | 0x1b<br>0x1c<br>0x74<br>0x64<br>0x3e | iload_1<br>iload_2<br>ineg<br>isub<br>istore_3 | 1 |

The third method - inserting operations not changing the program logic. There is an example in Table-6, it is based on that adding zero does not change the result. When we embed 1 the size increases by two commands. That method changes the size of executable file and changes the speed of execution but it allows embedding unlimited size of information [10].

**Table-6.** Embedding of 1 bit.

| Source code | Byte code | Commands | Recoverable bit |
|---|---|---|---|
| int a = 2;<br>int b = a + 3; | 0x15<br>0x06<br>0x60<br>0x3d | iload_1<br>iconst_3<br>iadd<br>istore_2 | 0 |
| int a = 2;<br>int b = a + 3 + 0; | 0x15<br>0x06<br>0x60<br>0x03<br>0x60<br>0x3d | iload_1<br>iconst_3<br>iadd<br>iconst_0<br>iadd<br>istore_2 | 1 |

## 4. METHODOLOGY OF FIRST METHOD

The first method - changing of sequence method is based on the possibility of reordering of assignment operations, which follow each other, and its results are not dependent from each other and their sequence. Such changes cannot affect the correctness of the computing program. In order to make embedding, we should change the order of assignment operations going in lexicographic order for embedding of one bit, or in opposite order, to embed the other.

Methodology of embedding of information into an executable file is based on the arguments before some commands, which take two arguments form stack, can be interchanged and the result of calculation will not be changed. Such commands are as follows:

- Totaling: *iadd, ladd, fadd, dadd*;
- Multiplying: *imul*, *lmul*, *fmul*, *dmul;*
- Logistic OR: *ior*, *lor;*
- Logistic AND:*iand*, *land;*
- Exclusive OR:*ixor*, *lxor;*
- Congruence of numbers:*lcmp, fcmpl, fcmpg, dcmpl, dcmpg;*
- If-statement for whole numbers (equals to, not equals): *if_icmpeq, if_icmpne.*

In the function of arguments these commands can take on a value:

- Variables(*iload,lload,dload*);
- Constants(*sipush,bipush,ldc*);
- Class fields(*getfield,getstatic*);
- Output computation of methods(*invokevirtual, invokestatic, invokeinterface*);
- Output computation of operations (*iadd, imul, iand, ior,ixor*).

Quantity of bites, which can be embedded into an executable file, equals:
$K_b = K_1 - K_2$
where$K_1$ – quantity of commands before which arguments can be interchanged, $K_2$ – quantity of commands before which arguments can be interchanged, providing that arguments are equal.
Example of embedding of 1 bit is described in Table-7

**Table-7.** Embedding of 1 bit.

| Source code | Bytecode | Commands | Recoverable bit |
|---|---|---|---|
| intc = b+a; | 0x1c<br>0x1b<br>0x60<br>0x3e | iload_2<br>iload_1<br>iadd<br>istore_3 | 0 |
| intc = a+b; | 0x1b<br>0x1c<br>0x60<br>0x3e | iload_1<br>iload_2<br>iadd<br>istore_3 | 1 |

One bit is embedded by means of interchanging of arguments before totaling, in the function of arguments - two variables.

The Table-8 shows an example of embedding of two bites.

**Table-8.** Embedding of 2 bits.

| Source code | Bytecode | Commands | Recoverable bit |
|---|---|---|---|
| int c = a*2+b; | 0x1b<br>0x05<br>0x68<br>0x1c<br>0x60<br>0x3e | iload_1<br>iconst_2<br>imul<br>iload_2<br>iadd<br>istore_3 | 00 |
| int c = 2*a+b; | 0x05<br>0x1b<br>0x68<br>0x1c<br>0x60<br>0x3e | iconst_2<br>iload_1<br>imul<br>iload_2<br>iadd<br>istore_3 | 01 |
| int c = b+a*2; | 0x1c<br>0x1b<br>0x05<br>0x68<br>0x60<br>0x3e | iload_2<br>iload_1<br>iconst_2<br>imul<br>iadd<br>istore_3 | 10 |
| int c = b+2*a; | 0x1c<br>0x05<br>0x1b<br>0x68<br>0x60<br>0x3e | iload_2<br>iconst_2<br>iload_1<br>imul<br>iadd<br>istore_3 | 11 |

The first byte is embedded by means of interchanging of arguments before totaling, in the function of arguments - of the multiplying and a variable. The second byte is embedded by means of interchanging of arguments before multiplying, in the function of arguments - a constant and a variable.

## 5. EXPERIMENTAL VERIFICATION

The result of the experimental verification of this method of embedding information is shown in Table-9.

**Table-9.** Experimental verification of embedding.

| Retrivial | Quantaty ofclass files | File size | Embedded size | Embedding speed |
|---|---|---|---|---|
| tools.jar | 4145 | 14,5 МБ | 4145bit | 1 bit on 4.6 Kbytes |
| rt.jar | 373 | 0,98 МБ | 21 bit | 1 bit on 47 Kbytes |
| jfxrt.jar | 5770 | 14,4 МБ | 93 bit | 1 bit on 159 Kbytes |
| total | 155282 | 630 МБ | 85860 bit | 1 bit on 7.5 Kbytes |

For the research of this method of embedding of information approximately 155 thousands of class-files were selected. The medium size of class-file is 4.15 Kbytes. On the average 1 bite of information can be embedded into 7.5 Kbytes.

This method allows making hidden embedding of information into executable code, which allows implementing protection of copyright rights on computer software or some parts of it. The advantages of this method are that with embedding the size and the execution time of the file remains unchanged.

## 6. APPLICATIONS

For example, somebody has written a library as a class file. That person wants to track who is using his library in own program. To solve this problem there is a digital signature. The library will be signed with a private key and it can be checked with a public key.

The algorithm of embedding the digital signature into the class file is shown in Figure-2:

a) Count the number of available bites (bits_available).
b) Calculate the sum of all bytes (sum_bytes).
c) Calculate the hash function end encrypt with a private key.
d) Calculate modulo of $2^{\boldsymbol{bits\_available}}$.
e) Embed the bits sequence into the file.

$$signature = encryption\ (\ hash(\text{sum\_bytes}), \text{K}_{\text{private}})\%2^{bits\_available},$$

We calculate the sum of all bytes because embedding digital signature does not change.

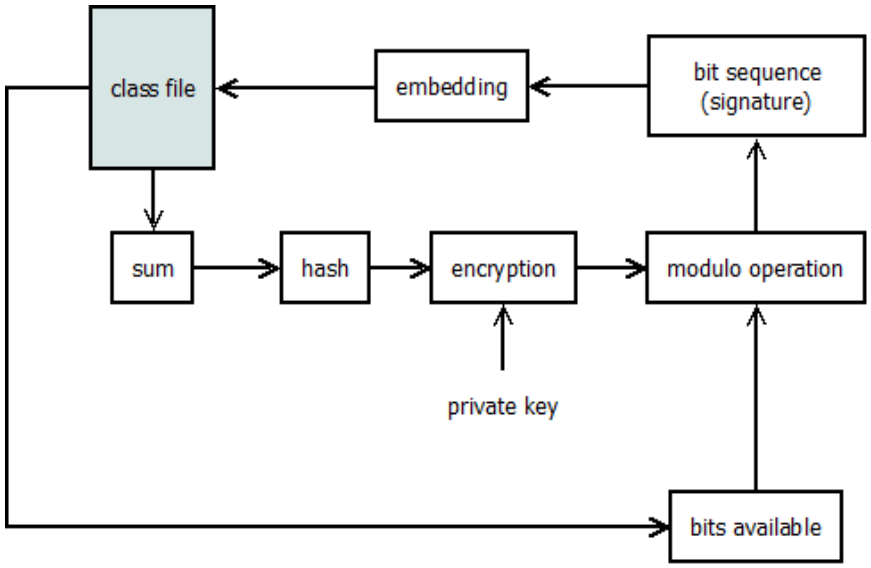


**Figure-2.** Embedding the signature.

The algorithm of verification class file shown in Figure-3:

a) Extract the sequence from the class file (seq_extr) and sum of all bytes (sum_bytes).
b) Calculating $\boldsymbol{s1 = decryption(seq\_extr, K_{public})\%2^{bits\_available}}$
c) Calculation $\boldsymbol{s2 = hash(sum\_bytes)\%2^{bits\_available}}$
d) If $\boldsymbol{s1 = s2}$ the signature is right

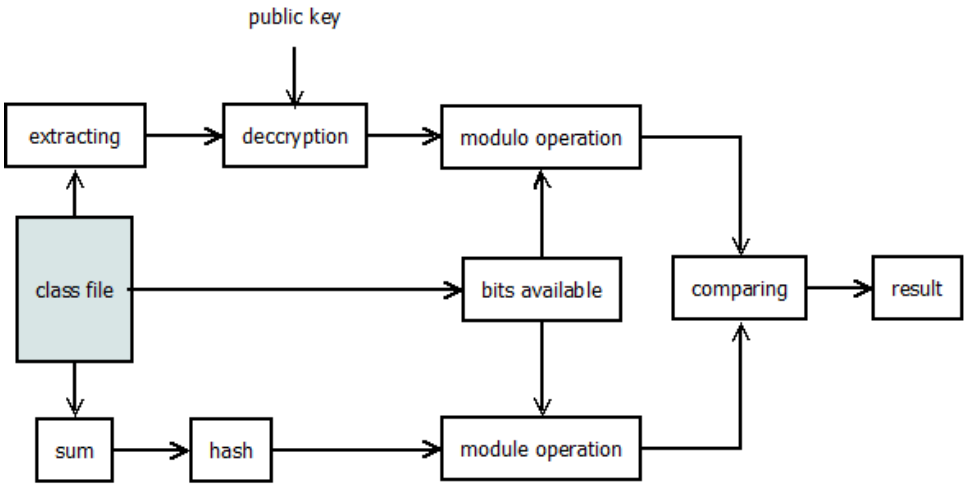


**Figure-3.** Verification the signature.

## 7. CONCLUSIONS

The application of these techniques of byte-code embedding is belong to undocumented features of Java machines, which may vary, depending on the version and the experimental verification needs.

Embedding into Java byte-code does not affect the structure and the integrity of the program. Hidden embedding, which was introduced in the file, will be difficult to detect due to the fact that the changes within the file do not affect size or functionality of the executable code. The proposed method is simple to implement and is cost-effective. Concealed embedding, implemented by methods of equivalent substitution of operators, allows realization of copyright protection at the technical level by using digital signature. This method allows embedding a small amount of information; however the practicability of this method is caused by the fact that the embedded information is used jointly with the executable file.

## REFERENCES

[1] V. Korzhik, K. Vebaeva, E. Gerling, I. Dogil and I. Fedyanin. 2016. Digital steganography and digital watermarking. SPb. SUT. p. 226.

[2] A. Krasov and S. Shterenberg. 2014. Methods for embedding hidden data into executable scripts. Key Issues in Modern Science - 2014, Sofia, Bulgaria. p. 9.

[3] A. Krasov, A. Vereshchagin and A. Cvetkov. 2013. Authentication Software by using embedding digital watermark into executable code. Telecommunications. (S7): 27-29.

[4] A. Krasov, Y. Tregubov and S. Shterenberg. 2015. Research of copy protection methods software based on embed of digital watermarks into executable and library files. Cambridge Journal of Education and Science. 2(14): 565-573.

[5] S. I. Shterenberg, A. V. Krasov and I. A. Ushakov. 2015. Analysis of Using Equivalent Instructions At The Hidden Embedding of Information Into The Executable Files. Journal of Theoretical and Applied Information Technology. 80(1): 28-34.

[6] A. Krasov, A. Vereshagin and V. Abaturov. 2012. Methods of the Hidden Embedding of Information in Executed Files. Saint-Petersburg state electro technical university LETI. (8): 51-55.

[7] I.N. Homyakov and A.V. Krasov. 2014. Possibility of hiding information into java byte. Information technology modeling and management. 2(86): 185-191.

[8] A. Krasov, Y. Tregubov and S. Shterenberg. 2015. Analysis of use of equivalent instructions in the embedding hidden information into the byte-code of JAVA. London Review of Education and Science. 2(18): 811-819.

[9] A. Krasov and Y. Tregubov. 2015. Code Noising with embedding information into an executable file by using Semantic Equivalents Replacement Instructions method. Telecommunication and computer systems. pp. 108-111.

[10] I.N. Homyakov and A.V. Krasov. 2014. Hiding Information into java byte code structure. Control Systems and Information Technology. 56(2): 89-93.