Techniques of Program Code Obfuscation for Secure Software

Popa Marius
Bucharest Academy of Economic Studies
165 PUBLICATIONS **479** CITATIONS

Journal of Mobile, Embedded and Distributed Systems, vol. III, no. 4, 2011
ISSN 2067 – 4074

Techniques of Program Code Obfuscation for Secure Software

Marius Popa
Department of Economic Informatics and Cybernetics
Academy of Economic Studies
Piaţa Romană 6, Bucharest
ROMANIA
marius.popa@ase.ro

Abstract: The paper investigates the most common obfuscation techniques for software program source code. Engineering elements of the compiling and interpreting processes are presented form the most widely used programming language based on Java Development Kit and .NET Framework. The reverse engineering of software is implemented on taking into account the architectures of the software development platforms used by the most part of software developers. The engineering elements of these architectures facilitate understanding and production of data exchange, disassembling and decompiling. The last ones are essential tools to implement the reverse engineering of software. In order to prevent unauthorized disclosure of software engineering techniques, techniques of the source code obfuscation are used. On the other hand, the reverse engineering of software is used in critical software fields like antivirus program development.

Key-Words: code obfuscation, secure software, compiling process, obfuscator.

1. Introduction

The modern software development platforms use programming languages that compile the source code into intermediate languages. For this kind of programming languages, it can be used a decompiling software to obtain the original source code of the application.
Software applications developed on Java and .NET platforms have the risk to be decompiled because the intermediate code is not a binary code. For instance, the bytecode is the result of compilation for a Java source code, and the Java Virtual Machine is the interpreter of bytecode. The compiling and interpreting process for Java applications is depicted in figure 1, adapted from [15].
In Java compiling and interpreting process, the bytecode resulted from java source code compiling is translated to binary code by an interpreter as a component of Java Virtual Machine. The Java Virtual Machine structure is depicted in the next chapter of this paper.

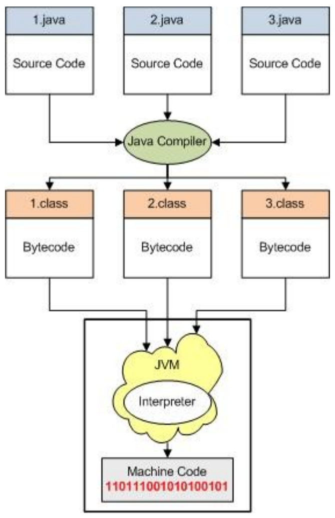

Figure 1. Java compiling and interpreting process

The interpreter translates the intermediate language instructions into machine code and executes it. Translation is made on pieces of bytecode read by interpreter. The interpreter implements specific optimizations and verifications on assembly metadata [5].
For .NET Framework, the intermediate code is called Common Intermediate Language. The Common Language

Infrastructure is presented in figure 2, adapted from [13].

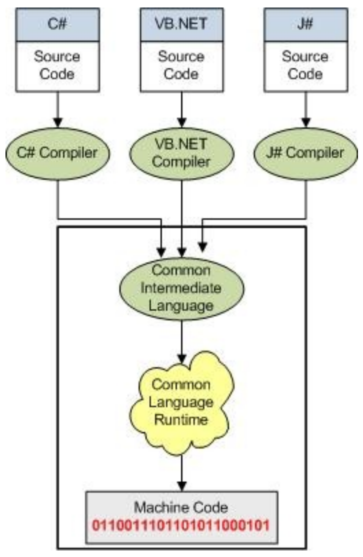

Figure 2 .NET compiling and interpreting process

In .NET compiling and interpreting process, the source codes written in .NET programming languages are compiled into Common Intermediate Language as a neutral language. The Common Language Runtime translates the code from Common Intermediate Language to binary code executed by computer machine.
The Common Intermediate Language, Common Language Runtime and binary code are components of Common Language Infrastructure.
The intermediate languages have CPU- and platform-independent instruction sets that are executed by interpreting infrastructures.
At running time, the interpreter needs to know the names of classes, methods and fields stored by intermediary code. Thus, the intermediate language has the vulnerability to be decompiled into source code.
In this way, an unauthorized person accesses the software functionalities, algorithms, data structures and security issues. The original source code can be modified and recompiled into a new software application. This issue involves the regulations regarding the intellectual property [5].

During the software development life cycle, the code protection can be implemented by:

- Using of ahead-of-time compilers – translation from bytecode to machine code is made before execution; this means that the performance of interpreter is better that a translation made during execution;
- Encryption of intermediate code – the bytecode is encrypted after compiling; at running time, the encrypted bytecode is decrypted to be translated into machine code by interpreter; the method is vulnerable when a method call receives the decrypted class bytes; this call can be intercepted and used to obtain decrypted form of bytecode; also, it is possible that an interpreter cannot load and execute encrypted classes;
- Code obfuscation – the source code and bytecode is transformed to make more difficult decompiling, code reading and analysis; there are some techniques and methods to implement the code obfuscation.

The last code protection way is approached in the following sections.

2. Reverse Engineering of Software

As concept, reverse engineering is the process to discover the technological principles of a product or system based of analysis of its structure, function and operation [14].
Applied in software development field, the reverse engineering represents the process to develop a system at a high level of abstraction, starting from the end software product and deductions of the procedures used in software development process. During reverse engineering, deductions are made with a little or no knowledge of the original procedures [5].
There are three main classes of techniques for reverse engineering of software [14]:

- Observation of data exchange – data exchange is observed through hardware and software products as bus analyzers and packet sniffers;

the observations are analyzed to perform a new implementation that imitates the system;

- Disassembly – a specific software is used to read the binary code of a program; the software is called disassembler and it uses the assembler instruction set mnemonics to translate the binary code to instructions in assembler language; the disassembling process of the binary code is a time-consuming process;
- Decompiling – a specific software is used to read the binary code or bytecode of a program; the software is called decompiler and it is used to recreate the source code of the program in a high-level programming language.

The impact of reverse engineering of software on computer security is very significant and it aims the following issues [3], [4], [6]:

- Malicious software – developers of malicious software locates vulnerabilities in operating software and other software programs; these vulnerabilities are exploited to defeat all security walls built around the software program by developers; also, the reverse engineering is used by antivirus developers to identify the ways in which the malicious software is eliminated from the system and to quantify the damages made within the software and the access to sensitive information;
- Cryptographic algorithms – reverse engineering of software is used to expose cryptographic algorithms; once a cryptographic algorithm is exposed, that algorithm can be broken; also, in key-based algorithms, the encryption and decryption keys can be obtained, extracted or generated from software analysis;
- Digital Rights Management – this technology is used to protect the digital media information from unauthorized duplication; DRM is similar to software copy protection technology and aims to protect the digital media producers from uncontrolled duplication; the crackers attempt to defeat the DRM technology to make copies of digital media information; for this purpose, they use reverse engineering techniques to identify the modification necessary to disable the protection of DRM;
- Auditing program binaries – the open-source software has the source code available for any software engineer; so, the open-source software vulnerabilities are identify faster than the proprietary software ones; however, with the appropriate and strong skills, the source code of proprietary software can be made available and analyzed to establish the vulnerabilities.

As technique for reverse engineering of software, decompiling is applied on binary code or intermediate code.

The Java bytecode is stored in a *.class* file. The bytecode is an array of bytes in which each byte stores a value in accordance to the operation code.

When a Java bytecode file is decompiled, the resulted file contains the operations as offset within the invoked method.

The execution of bytecode is made by Java Virtual Machine. JVM is a stack-based machine. A JVM stack stores frames and it is assigned to each thread. A frame is created for each method and it consists of [7]:

- An operand stack – it is used to push and pop values; some operation code instructions stores values on stack, other instructions extract values from stack, operate them the result is stored on stack; also, it is used to store values from method invocations;
- An array of local variables – it is used to store the method parameters and values of local variables;
- Reference to the runtime constant pool of a class or interface – it stores numeric literals known at compile time and method and field references resolved at runtime.

The conceptual representation of a frame from a JVM stack is depicted in figure 3 [7].

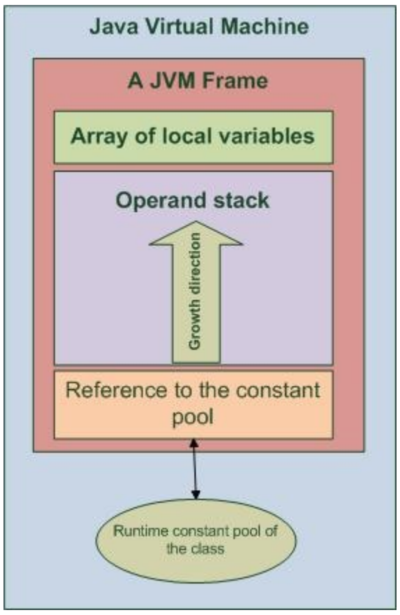

Figure 3. Structure of a JVM Frame

For .NET software programs, the source code written in .NET compatible programming language is compiled to intermediate code CIL. Intermediate code is transformed into machine code by Common Language Runtime component of .NET Framework.
The architecture of CLR is presented in figure 4 [8].

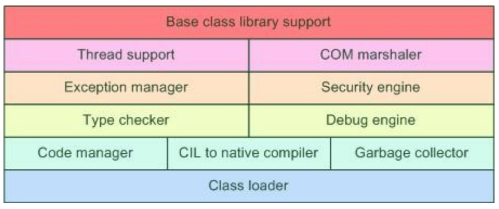

Figure 4. CLR architecture

CLR contains the tools to interpret the CIL code for getting the machine code of the .NET software programs. The roles of the CLR components are:

- Class loader – loading the classes into CLR;
- Code manager – code management during execution;
- CIL to native compiler – conversion of CIL code into native code;
- Garbage collector – automatic management of the memory;
- Type checker – enforce strict type checking;
- Debug engine – allowance to debug applications;
- Exception manager – mechanism for run-time exception handling;
- Security engine – enforce security restrictions: code security, folder and machine security;
- Thread support – multithreading support for applications;
- COM marshaler – allowance to exchange data with COM applications;
- Base class library support – providing the classes necessary for application at run-time.

The environment in which a .NET method is executed by Common Language Interface is depicted in figure 5 and it is called the method state [2].

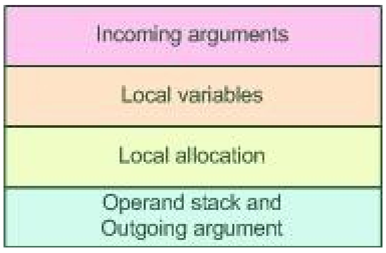

Figure 5. State of .NET method

The following structures are components of CLI method state:

- Instruction pointer – indicates the next CIL instruction to be executed by CLI within method;
- Evaluation stack – is local to the method; it is used to retrieve arguments for CIL instructions and store the instruction results; also, arguments to other methods and their results are pushed on this stack;
- Local variable array – is used to store values of variables define within method; these values are preserved across the method call;
- Argument array – stores the values of incoming arguments of the method;
- Method information handle – contains read-only information as method signature, types of local variables, exception handler data;
- Local memory pool – is used to allocate memory dynamically for

objects; allocated memory is reclaimed on method exit;
- Return state handle – restores the method state for current method caller;
- Security descriptor – is used by CLI security system to record security overrides.

The areas depicted in figure 5 are logically distinct areas. The CLI can map these areas in a contiguous array of memory.

3. Techniques and Methods of Code Obfuscation

Obfuscation is the process which transforms the source code or intermediate code to make it more difficult to be decompiled and analyzed. Thus, the reverse engineering of software is more difficult to be implemented.
The obfuscation process is implemented in automatic tools called obfuscators.
As obfuscation techniques, the following must be considered [10]:
- Name obfuscation;
- Data obfuscation;
- Code flow obfuscation;
- Incremental obfuscation;
- Intermediate code optimization;
- Debug information obfuscation;
- Watermarking;
- Source code obfuscation.

Name obfuscation. It is the process to replace de identifiers with meaningless strings as new identifiers. Usually, the identifiers have meaning for a better recognition of the source code structures like classes, methods, variables and so forth.
Once an identifier is renamed, it is mandatory to provide consistency across the entire application through replacements of the old names by the new identifiers.
For programming languages that allow the method overloading, method name obfuscation is made by the same identifier string for the methods having different signatures.
For instance, the below Java source code

```
import java.*;
import java.lang.*;
class Product extends java.lang.Object
{
public int id;
public String Name;
public Product(String prName,int i){
        Name = prName;
        id = i;
}
public String getName(){
        return Name;
}
public void setName(String prName){
        Name = prName;
}}
```

has the following code after decompiling:

```
Compiled from Product.java
class Product extends java.lang.Object
{
   public int id;
   public java.lang.String Name;
   public
Product(java.lang.String,int);
   public java.lang.String getName();
   public                        void
setName(java.lang.String);
}

Method Product(java.lang.String,int)
  0 aload_0
  1   invokespecial   #3   <Method
java.lang.Object()>
  4 aload_0
  5 aload_1
  6 putfield #4 <Field java.lang.String
Name>
  9 aload_0
 10 iload_2
 11 putfield #5 <Field int id>
 14 return

Method java.lang.String getName()
  0 aload_0
  1 getfield #4 <Field java.lang.String
Name>
  4 areturn

Method                         void
setName(java.lang.String)
  0 aload_0
  1 aload_1
  2 putfield #4 <Field java.lang.String
```

```
Name>
  5 return
```

The Java intermediate code is disassembled by *javap* application included as a tool in Java Development Kit (JDK).

After name obfuscation, the significance of the identifiers is hidden, as it is shown in the following code:

```
Compiled from X.java
class X extends java.lang.Object {
    public int a;
    public java.lang.String b;
    public X(java.lang.String,int);
    public java.lang.String f();
    public void f(java.lang.String);
}

Method X(java.lang.String,int)
  0 aload_0
  1 invokespecial #3 <Method
java.lang.Object()>
  4 aload_0
  5 aload_1
  6 putfield #5 <Field java.lang.String
b>
  9 aload_0
 10 iload_2
 11 putfield #4 <Field int a>
 14 return

Method java.lang.String f()
  0 aload_0
  1 getfield #5 <Field java.lang.String
b>
  4 areturn

Method void f(java.lang.String)
  0 aload_0
  1 aload_1
  2 putfield #5 <Field java.lang.String
b>
  5 return
```

It considers the following C# source code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsTestProj
{
    class Program
    {
        public int id;
        public String Name;
        public Program(String
prName,int i){
            Name = prName;
            id = i;
        }
        public String getName(){
            return Name;
        }
        public void setName(String
prName){
            Name = prName;
        }

        static void Main(string[] args)
        {
        }
    }
}
```

After disassembling of the *exe* file generated by the .NET IDE Visual Studio 2008, the code for *ConsTestProj.exe* has the following content:

```
.module ConsTestProj.exe
// MVID: {8C9F8979-558E-4810-
9148-D8441A5AC1B7}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003          //
WINDOWS_CUI
.corflags 0x00000001    // ILONLY
// Image base: 0x00BC0000


// ================ CLASS
MEMBERS DECLARATION
===================

.class private auto ansi beforefieldinit
ConsTestProj.Program
      extends [mscorlib]System.Object
{
  .field public int32 id
  .field public string Name
  .method public hidebysig
specialname rtspecialname
        instance void .ctor(string
prName,
                         int32 i) cil
managed
```

```
 // SIG: 20 02 01 0E 08
 {
   // Method begins at RVA 0x2050
   // Code size       24 (0x18)
   .maxstack  8
   IL_0000:  /* 02   |                  */
ldarg.0
   IL_0001:  /* 28|(0A)000010 */ call
instance void
[mscorlib]System.Object::.ctor()
   IL_0006:  /* 00   |                  */
nop
   IL_0007:  /* 00   |                  */
nop
   IL_0008:  /* 02   |                  */
ldarg.0
   IL_0009:  /* 03   |                  */
ldarg.1
   IL_000a:  /* 7D   | (04)000002 */
stfld string
ConsTestProj.Program::Name
   IL_000f:  /* 02   |                  */
ldarg.0
   IL_0010:  /* 04   |                  */
ldarg.2
   IL_0011:  /* 7D   | (04)000001 */
stfld int32
ConsTestProj.Program::id
   IL_0016:  /* 00   |                  */
nop
   IL_0017:  /* 2A   |                  */
ret
 } // end of method Program::.ctor

 .method public hidebysig instance
string
         getName() cil managed
 // SIG: 20 00 0E
 {
   // Method begins at RVA 0x206c
   // Code size       12 (0xc)
   .maxstack  1
   .locals init ([0] string CS$1$0000)
   IL_0000:  /* 00   |                  */
nop
   IL_0001:  /* 02   |                  */
ldarg.0
   IL_0002:  /* 7B   | (04)000002
*/   ldfld string
ConsTestProj.Program::Name
   IL_0007:  /* 0A   |                  */
stloc.0
   IL_0008:  /* 2B   | 00               */
br.s       IL_000a
```

```
   IL_000a:  /* 06   |                  */
ldloc.0
   IL_000b:  /* 2A   |                  */
ret
 }   //   end   of   method
Program::getName

 .method public hidebysig instance
void
         setName(string prName) cil
managed
 // SIG: 20 01 01 0E
 {
   // Method begins at RVA 0x2084
   // Code size       9 (0x9)
   .maxstack  8
   IL_0000:  /* 00   |                  */
nop
   IL_0001:  /* 02   |                  */
ldarg.0
   IL_0002:  /* 03   |                  */
ldarg.1
   IL_0003:  /* 7D   | (04)000002
*/   stfld string
ConsTestProj.Program::Name
   IL_0008:  /* 2A   |                  */
ret
 }   //   end   of   method
Program::setName

 .method private hidebysig static void
Main(string[] args) cil managed
 // SIG: 00 01 01 1D 0E
 {
   .entrypoint
   // Method begins at RVA 0x208e
   // Code size       2 (0x2)
   .maxstack  8
   IL_0000:  /* 00   |                  */
nop
   IL_0001:  /* 2A   |                  */
ret
 } // end of method Program::Main

}   //   end   of   class
ConsTestProj.Program
```

The disassembled code is generated by a tool called MSIL Disassembler (ildasm.exe) that is included in .NET Framework SDK. Each component of the C# program can be investigated in MSIL Disassembler.
After the name obfuscation, the disassembled code is:

```
.module ConsTestProj.exe
```

```
//    MVID:    {14B7C248-8DD6-458F-
BBF7-8FF0A750AA41}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem   0x0003                //
WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x01330000


//    ===============    CLASS
MEMBERS                  DECLARATION
===================

.class private auto ansi beforefieldinit
A.X
      extends [mscorlib]System.Object
{
  .field public int32 a
  .field public string b
  .method         public         hidebysig
specialname rtspecialname
         instance void  .ctor(string x,
                         int32      y)     cil
managed
  {
    // Code size       24 (0x18)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:       call    instance    void
[mscorlib]System.Object::.ctor()
    IL_0006:  nop
    IL_0007:  nop
    IL_0008:  ldarg.0
    IL_0009:  ldarg.1
    IL_000a:  stfld      string A.X::b
    IL_000f:  ldarg.0
    IL_0010:  ldarg.2
    IL_0011:  stfld      int32 A.X::a
    IL_0016:  nop
    IL_0017:  ret
  } // end of method X::.ctor

  .method   public   hidebysig   instance
string
         f() cil managed
  {
    // Code size       12 (0xc)
    .maxstack  1
    .locals init ([0] string CS$1$0000)
    IL_0000:  nop
    IL_0001:  ldarg.0
    IL_0002:  ldfld      string A.X::b
    IL_0007:  stloc.0
```

```
    IL_0008:  br.s       IL_000a

    IL_000a:  ldloc.0
    IL_000b:  ret
  } // end of method X::f

  .method   public   hidebysig   instance
void
         f(string z) cil managed
  {
    // Code size       9 (0x9)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldarg.0
    IL_0002:  ldarg.1
    IL_0003:  stfld      string A.X::b
    IL_0008:  ret
  } // end of method X::f

  .method private hidebysig static void
Main(string[] args) cil managed
  {
    .entrypoint
    // Code size       2 (0x2)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ret
  } // end of method X::Main

} // end of class A.X
```

There are some limitations for identifiers renaming as it follows:

- Names of standard classes included in standard APIs;
- Names of serializable classes;
- Names of classes accessed via reflection or using native methods.

Name obfuscation can lead to a shorter bytecode with advantages for mobile and desktop applications.

Data obfuscation. It aims the data structures defined in a software application. Data obfuscation is the process that changes the ways in which data are stored in the memory. Data obfuscation aims [9]:

- Change the role of data in the program – change the place of variable defining to change the variable role (local vs. global);
- Change the data encoding – change the interpretation of stored data through replacement of a value by an expression;

- Data aggregation – change the access to a data structure after its conversion in another one; it is applied for values stored in aggregated structures like arrays;

For instance, it considers the following Java source code:

```
import java.*;
import java.lang.*;

class A extends java.lang.Object {
public static void main(){
        int a[] = new int [10];
        int i;
        for(i=0; i<10; i++)
                a[i] = i*10 + i;
}}
```

After decompiling process, the code is:

```
Compiled from A.java
class A extends java.lang.Object {
   A();
   public static void main();
}

Method A()
  0 aload_0
  1 invokespecial #3 <Method
java.lang.Object()>
  4 return

Method void main()
  0 bipush 10
  2 newarray int
  4 astore_0
  5 iconst_0
  6 istore_1
  7 goto 22
 10 aload_0
 11 iload_1
 12 iload_1
 13 bipush 10
 15 imul
 16 iload_1
 17 iadd
 18 iastore
 19 iinc 1 1
 22 iload_1
 23 bipush 10
 25 if_icmplt 10
 28 return
```

After obfuscation by data aggregation, the Java source code is:

```
import java.*;
import java.lang.*;

class A extends java.lang.Object {
public static void main(){
int a[][] = new int [2][5];
int i, j;
for(i=0; i<2; i++)
for(j=0; j<5; j++)
        a[i][j]=(i*5+j)*10+i*5+j;
}}
```

and the decompiled code is:

```
Compiled from A.java
class A extends java.lang.Object {
   A();
   public static void main();
}

Method A()
  0 aload_0
  1 invokespecial #4 <Method
java.lang.Object()>
  4 return

Method void main()
  0 iconst_2
  1 iconst_5
  2 multianewarray #2 dim #2
<Class [[I>
  6 astore_0
  7 iconst_0
  8 istore_1
  9 goto 47
 12 iconst_0
 13 istore_2
 14 goto 39
 17 aload_0
 18 iload_1
 19 aaload
 20 iload_2
 21 iload_1
 22 iconst_5
 23 imul
 24 iload_2
 25 iadd
 26 bipush 10
 28 imul
 29 iload_1
 30 iconst_5
 31 imul
 32 iadd
 33 iload_2
 34 iadd
 35 iastore
```

```
36 iinc 2 1
39 iload_2
40 iconst_5
41 if_icmplt 17
44 iinc 1 1
47 iload_1
48 iconst_2
49 if_icmplt 12
52 return
```

- Data inheritance – modify inheritance relations between data by redundant structures;
- Data ordering – change the way in which values are ordered in data structures; to establish the correct position of a value in the data structure, a specific method is built in the program.

Code flow obfuscation. It is the process to change the control flow in a software application. The changed control flow must lead to the same results as the initial one, but the software application is more difficult to be analyzed.
Transformations used in code flow obfuscation are [9]:

- Control aggregation – change the group of program statements; for instance, using the inline methods instead of method calls;
- Control ordering – change the order in which the program statements are executed;
- Control computation – change the control flow in the program as:
  - Inserting dead code in the code execution flow to hide the true control flow; the code complexity is increased;
  - Inserting uncontrolled jump instructions to change the control graph of the program;
  - Inserting alternative structures having a controlled evaluation of the condition to broke a statement block in two statement sub-blocks at least;

For instance, it considers the following Java source code:

```
import java.*;
import java.lang.*;

class A extends java.lang.Object
{
public static void main(){
        int x, y, z, S;
        x = 1;
        y = x + 2;
        z = x * y;
        S = x + y + z;
}}
```

The decompiled code is:

```
Compiled from A.java
class A extends java.lang.Object
{
   A();
   public static void main();
}

Method A()
   0 aload_0
   1 invokespecial #3 <Method
java.lang.Object()>
   4 return

Method void main()
   0 iconst_1
   1 istore_0
   2 iload_0
   3 iconst_2
   4 iadd
   5 istore_1
   6 iload_0
   7 iload_1
   8 imul
   9 istore_2
  10 iload_0
  11 iload_1
  12 iadd
  13 iload_2
  14 iadd
  15 istore_3
  16 return
```

Inserting a controlled evaluation together with some dummy code change the Java source code in:

```
import java.*;
import java.lang.*;

class A extends java.lang.Object
{

public static void main(){
        int x, y, z, S;
        x = 1;
        y = x + 2;
        z = x * y;
        if(y > x)
                S = x + y + z;
        else
                S = x - y - z;
}}
```

The decompiled code after obfuscation is:

```
Compiled from A.java
class A extends java.lang.Object
{
   A();
   public static void main();
}

Method A()
   0 aload_0
   1 invokespecial #3 <Method
java.lang.Object()>
   4 return

Method void main()
   0 iconst_1
   1 istore_0
   2 iload_0
   3 iconst_2
   4 iadd
   5 istore_1
   6 iload_0
   7 iload_1
   8 imul
   9 istore_2
  10 iload_1
  11 iload_0
  12 if_icmple 24
  15 iload_0
  16 iload_1
  17 iadd
  18 iload_2
  19 iadd
  20 istore_3
  21 goto 30
  24 iload_0
  25 iload_1
  26 isub
  27 iload_2
  28 isub
  29 istore_3
  30 return
```

- Inserting redundant operations to increase the expression complexity, but with the same result like the initial evaluation;
- Inserting the standard API methods in user defined methods because the standards methods cannot be renamed;
- Inserting the buffer classes to hide extensions of standard API classes.

Incremental obfuscation. It is the process providing code consistency of the updated software programs obfuscated in the previous version.

Intermediate code optimization. It aims the size of intermediate code. As optimization techniques, the following are included [10]:

- Removing the unused methods, fields and strings;
- Evaluation of constant expressions;
- assignment of static and final attributes;
- Inlining of simple methods.

Debug information obfuscation. It is the process for removing the debug information that can help to discover the original source code of the program. The debug information is used to get stack traces. This information aim line numbers or file names.

Watermarking. It is the process to embed information in the software application. This information is used to prevent unauthorized software disclosure.

The watermarking must considered the following requirements [1]:

- Software execution has not to be altered;
- Embedded information is decoded only by the software developer;
- Embedded information is hard to be discovered;
- Embedded information is decoded even only a part of software is stolen;
- Embedded information is resistant to decompiling-recompiling and obfuscation processes.

Insertion of embedded information for watermarking cannot be done for an arbitrary byte string because the intermediate code is verified because its interpreting.

There two methods to insert watermarking:

- Numeric operand overwriting;
- Operation code replacement.

Numeric operand overwriting. Numeric operands of some instructions can be replaced by other values. In Java, such instructions are: *iinc*, *sipush*, *ldc* and so forth. Also, there are instructions having numeric operands that cannot be overwritten because the syntactic rules are not accomplished.

Operation code replacement. It is possible because the replacement follows the syntactic rules. In Java, there are operation codes that can replace each other as: *iadd*, *isub*, *imul*, *idiv*, *irem*, *iand*, *ior* and *ixor*. In these operation codes, three information bits can be encoded. Thus, these operation codes are used to inject the embedded information.

Source code obfuscation. It is the process to hide the source code meaning when this code is disclosed to a third party to test or maintain it. This technique supposes renaming of the program identifiers and removing the comments [10].

The obfuscation of mobile device software programs leads to a smaller code and faster running.

The operations for software obfuscation are implemented by specialized software products called obfuscators. In table 1, a part of many Java obfuscators are presented [10].

Table 1. Java obfuscators

Obfuscator	**License**	**Features**
Allatori	Commercial	▪ Name Obfuscation ▪ Flow Obfuscation ▪ Debug Info Obfuscation ▪ String Encryption ▪ 100% Protection Against Popular Decompilers ▪ Optimizing ▪ Watermarking ▪ Incremental Obfuscation ▪ Stack Trace Utility ▪ Build Tool Interface ▪ J2ME Obfuscation ▪ Android Obfuscation
Dash-O-Pro	Commercial	▪ Cross JAR Renaming ▪ Renaming Prefix ▪ Overload Induction ▪ Incremental Obfuscation ▪ Control Flow ▪ String Encryption
GuardIT	Commercial	▪ Control flow obfuscation ▪ String Encryption and Variable Renaming ▪ Real-time Security Alerts ▪ Developer-Friendly ▪ 100% Verifiable Code
Proguard	Open Source	▪ Bytecode Optimizations ▪ Android Obfuscation ▪ Flow Obfuscation ▪ Incremental Obfuscation ▪ Obfuscation using Reserved Keywords ▪ Stack Trace Translation
yGuard	Free	▪ Name Obfuscation ▪ Code Shrinking
Zelix Klassmaster	Commercial	▪ Name Obfuscation ▪ Flow Obfuscation ▪ String Encryption ▪ Incremental Obfuscation ▪ Stack Trace Translation

For applications based on .NET Framework, the obfuscation is implemented on a specific tool for Visual Studio called *Dotfuscator*. This tool implements the following obfuscation techniques: renaming transforms, control flow obfuscation and string encryption. Also, tamper detection and notification are implemented to alert the users when the proprietary software runs [12].

For the C# source code presented above, using of MSIL Disassembler on obfuscated code obtained with Dotfuscator leads to following code:

```
.module ConsTestProj.exe
//   MVID:   {C4007DA0-F041-47D8-
```

```
AC84-442204E22D88}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem  0x0003           //
WINDOWS_CUI
.corflags 0x00000001   //  ILONLY
// Image base: 0x01710000


// =============== CLASS
MEMBERS DECLARATION
===================

.class public auto ansi sealed
beforefieldinit DotfuscatorAttribute
       extends
[mscorlib]System.Attribute
{
  .custom instance void
[mscorlib]System.AttributeUsageAttri
bute::.ctor(valuetype
[mscorlib]System.AttributeTargets) =
( 01 00 01 00 00 00 00 00 )
  .field private string a
  .method public hidebysig
specialname rtspecialname
          instance void  .ctor(string a)
cil managed
  {
    // Code size       14 (0xe)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  dup
    IL_0002:  call       instance void
[mscorlib]System.Attribute::.ctor()
    IL_0007:  ldarg.1
    IL_0008:  stfld       string
DotfuscatorAttribute::a
    IL_000d:  ret
  } // end of method
DotfuscatorAttribute::.ctor

  .method public hidebysig instance
string
          a() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  1
    IL_0000:  ldarg.0
    IL_0001:  ldfld       string
DotfuscatorAttribute::a
    IL_0006:  ret
  } // end of method
DotfuscatorAttribute::a
```

```
  .property instance string A()
  {
    .get instance string
DotfuscatorAttribute::a()
  } // end of property
DotfuscatorAttribute::A
} // end of class DotfuscatorAttribute

.class private auto ansi beforefieldinit
a
       extends [mscorlib]System.Object
{
  .field public int32 a
  .field public string b
  .method public hidebysig
specialname rtspecialname
          instance void  .ctor(string A_0,
                               int32 A_1) cil
managed
  {
    // Code size       24 (0x18)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void
[mscorlib]System.Object::.ctor()
    IL_0006:  nop
    IL_0007:  nop
    IL_0008:  ldarg.0
    IL_0009:  ldarg.1
    IL_000a:  stfld      string a::b
    IL_000f:  ldarg.0
    IL_0010:  ldarg.2
    IL_0011:  stfld      int32 a::a
    IL_0016:  nop
    IL_0017:  ret
  } // end of method a::.ctor

  .method public hidebysig instance
string
          a() cil managed
  {
    // Code size       12 (0xc)
    .maxstack  1
    .locals init (string V_0)
    IL_0000:  nop
    IL_0001:  ldarg.0
    IL_0002:  ldfld      string a::b
    IL_0007:  stloc.0
    IL_0008:  br.s       IL_000a

    IL_000a:  ldloc.0
    IL_000b:  ret
  } // end of method a::a
```

```
 .method  public  hidebysig  instance
void
        a(string A_0) cil managed
  {
    // Code size       9 (0x9)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldarg.0
    IL_0002:  ldarg.1
    IL_0003:  stfld      string a::b
    IL_0008:  ret
  } // end of method a::a

 .method   private   hidebysig   static
void  a(string[] A_0) cil managed
  {
    .entrypoint
    // Code size       2 (0x2)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ret
  } // end of method a::a

} // end of class a
```

Depending on the goal of the software programs, a software developer decides what it is the appropriate obfuscator. Also, obfuscation techniques are selected for specific situations [11] to ensure the following characteristics for the software program:

- Readability – the code is read and understood;
- Reversibility – the code is reversible using a specific tool;
- Performance impact – effects on program running;
- Maintenance/Support – effects on supporting of transformed assemblies.

Obfuscators are automatic tools for obfuscation implementing. They are defined similar to compilers because a program is taken as input and it is transformed in an obfuscated program.

4. Conclusion

Techniques and methods for code obfuscations are used to prevent reverse engineering of the software. Prevention is necessary to protect intellectual propriety for software developers or digital media information owners.

Obfuscation is a part of solution to not restore the original program code that can be hacked to change functionalities, to extract sensitive information regarding the encryption algorithms or encryption keys or to change the policies of digital media distribution.

Acknowledgment

This work was supported by CNCSIS – UEFISCSU, project number PNII – IDEI 1838/2008, contract no. 923/2009 and the title Implementation of the Quantitative Methods in Distributed Informatics System Audit, financed by The National University Research Council – Ministry of Education, Research, Youth and Sports from Romania.

References

[1] Akito Monden, Hajimu Iida, Ken-ichi Matsumoto, Katsuro Inoue, Koji Torii, A Practical Method for Watermarking Java Programs, Proceedings of *COMPSAC '00 24th International Computer Software and Applications Conference*, IEEE Computer Society Washington, DC, USA, 2000, pp. 191 – 197

[2] ECMA International, Common Language Infrastructure, ECMA – 335, Geneva, December 2010

[3] Eldad Eilam, *Reversing: secrets of reverse engineering*, Wiley Publishing, Indianapolis, USA, 2005

[4] Mihai Doinea, Sorin Pavel, *Security Optimization for Distributed Applications Oriented on Very Large Data Sets*, Informatica Economică, vol. 14, no. 2, 2010, pp. 72 – 85

[5] Marius Popa, Characteristics of Program Code Obfuscation for Reverse Engineering of Software, Proceedings of the *4th International Conference on Security for Information Technology and Communications*, Academy of Economic Studies and Military Technical Academy, Bucharest, 17 – 18 November 2011, ASE Publishing House, Bucharest, pp. 103 – 112

[6] Paul Pocatilu, *Android Applications Security*, Informatica Economică, vol. 15, no. 3, 2011, pp. 163 – 171

[7] Peter Hagar, *Java Bytecode: Understanding bytecode makes you a better programmer*, IBM, 2001

[8] http://www.codeproject.com/KB/dotnet/DotNetWhitePaper.aspx

[9] http://www.cs.arizona.edu/~collberg/

Research/Students/DouglasLow/obfuscation.html

[10] http://www.excelsior-usa.com/articles/

java-obfuscators.html#intro

[11] http://msdn.microsoft.com/en-us/vstudio/ff716624

[12]http://www.preemptive.com/products/dotfuscator/overview

[13] http://en.wikipedia.org/wiki/Common_Language_Infrastructure

[14] http://en.wikipedia.org/wiki/

Reverse_engineering

[15] http://www.javatutorialhub.com/

java-virtual-machine-jvm.html