Dynamic Multi Levels Java Code Obfuscation Technique (DMLJCOT)

Adwan Yasin *adwan.yasin@aauj.edu*
Faculty of Engineering &Information Technology
Computer Science Department
Arab American University
Jenin, 240 ,Palestine

Ihab Nasra *ehab.nasra@aauj.edu*
Faculty of Engineering &Information Technology
Computer Science Department
Arab American University
Jenin, 240 ,Palestine

Abstract

Several obfuscation tools and software are available for Java programs but larger part of these software and tools just scramble the names of the classes or the identifiers that stored in a bytecode by replacing the identifiers and classes names with meaningless names. Unfortunately, these tools are week, since the java, compiler and java virtual machine (JVM) will never load and execute scrambled classes. However, these classes must be decrypted in order to enable JVM loaded them, which make it easy to intercept the original bytecode of programs at that point, as if it is not been obfuscated. In this paper, we presented a dynamic obfuscation technique for java programs. In order to deter reverse engineers from de-compilation of software, this technique integrates three levels of obfuscation, source code, lexical transformation and the data transformation level in which we obfuscate the data structures of the source code and byte-code transformation level. By combining these levels, we achieved a high level of code confusion, which makes the understanding or decompiling the java programs very complex or infeasible. The proposed technique implemented and tested successfully by many Java de-compilers, like JV, CAVJ, DJ, JBVD and AndroChef. The results show that all decompiles are deceived by the proposed obfuscation technique.

Keywords: Software Obfuscation, Reverse Engineering, Byte Code, Java Reflection, De-compiler.

1. INTRODUCTION

Java program is compiled down to platform independent and intermediate languages which is the bytecode and Java could link entities from different libraries using symbolic references in order to achieve platform independency [1], therefore, the names of classes , methods ,fields, , , and data types stored in the constant pool within the byte code file. The semi-compiled form of bytecode file makes the java applications more susceptible to analysis by reverse engineering and decompiled by the attackers.

On the other hand, Java Virtual Machine (JVM) works as an interpreter of bytecode, which executes and translates them into binary code. In addition, the dependencies resolved at runtime when classes loaded by java virtual machine [18].

Source code intellectual property needs protection against piracy and tampering which is a highly required at this time. A lot of software and applications that required hard work, money, intelligence, and a lot of time are pirate over the last years and worldwide which cause a significant financial loss estimated by millions of dollars every year [19].

Code obfuscation can be used as one of basic technique needed to protect the source code from manipulating and prevent malicious reverse engineering of software because the attacker should understand the source code of the software before they can make any specified modifications.

Code obfuscation can be defined as transformation, and altering the source code in such a way, which can make it too difficult to analysis by reverse engineering and difficult to understand, by human [1]. Obfuscation should not change the functionality of the program. It can be performed on source code, intermediate code or the machine code.

It is important to note, that we can use code obfuscation to protect the software against attackers; it can be also used to hide a malicious content. As an example, viruses can be resorted using obfuscation techniques that prevent them from detection by antivirus software.

Reverse engineering can be defined as the process of analyzing and extraction the design elements from the software system. It involve modification of the system such as adding a new functionality, modify the source code and return the source code which include the structure of program, control flow, methods and variables [19].

Many available tools and software help and makes it easier for the attacker and reverse engineer to decompile the software and extract the proprietary source code from Java programs in order to combined it into their own programs and steal the intellectual property. Such stealing are difficult to detect or tracking easily, such tools increase the challenge of code obfuscation process.

De-compilation can be defined as the process of retrieving source codes from intermediate bytecode or machine code; therefore, there is no obfuscation that can completely counter against extremely dedicated hackers [18].

Over the last years, number of software obfuscation tools have been proposed. Commercial and freely available obfuscation tools are often not good as they rely on “security through obscurity” which include renaming variables, string literals and adding nonsense instructions. These techniques may deter some impatient adversaries, but against a dedicated adversary they offer little to no security [18].

A lot of traditional obfuscation software use string processing techniques to carry out the obfuscation process, or to operate on binary files. despite that string processing techniques can sometimes work , however it often fails because the programming languages have a complex name resolution rules and different formatting and, therefore processing such techniques usually requires a complete language parsing, not only a string hacking.

When nested commands ,multiple statements per line, , comments placed around incomplete blocks of codes, unusual and peculiar conventions of identifiers and function names are encountered, as they often exist in complex and large systems it will case failure in processing the program correctly, that produce an easily broken obfuscated code via reverse engineering. Nevertheless, binary obfuscators work quite well, but in general it is limited to the standard simple binary formats [3].

In addition, Semantic Designs Obfuscation tools can work quite well because it generally remove whitespaces and nice indentation , strip comments, rename identifiers from their original name to nonsense names which convey no information , and encode constants in an inconveniently readable ways.

Most popular obfuscation techniques, can be classified into the following general classifications: [19].

1. Layout transformation, which transform the program structure and make it difficult to understand by human.
2. Data transformation, which transform the critical data and data structures into obfuscated form.
3. Control transformation, which transform the flow of execution make it more confuse for reverse engineering.

Most reliable way that used to build obfuscation tools simply parsing the source code according to language syntax and lexical rules in the compiler such as the data structures, carry out the obfuscation, and then unparsing them back to the source code [3]. We use this way in the first level on our obfuscation approach.

This paper proposed a dynamic multi-level obfuscation approach, which depends on combination of many techniques, as using only one technique is not sufficient to deter reverse engineers from de-compilation the code. We combine source and lexical code obfuscation with byte code and data obfuscation in order to make our approach robust against many forms of reverse engineering attacks. However, we focus our efforts to work at run time environment, which will make the byte code not easily possible to recompile again or make the code too difficult to understand by human.

We integrate our proposed encryption algorithms based on java metaprogramming concepts, which can be defined as the process of writing programs that manipulate other programs or itself depend on metadata with the ability to treat programs as their data. [12] It means that a program could developed and design to access, modify analyses, read and transform other programs, and even modify itself at running time. [13] Working on Meta level allows us to customize the java object model during the runtime environment. Therefore, our approach works at a level much closer to the java virtual machine rather dealing with the higher-level language where most compile time obfuscation tools and approaches worked.

We can determined the scope of the customization by which methods and byte code instructions are wrapped at load time, however the real nature of the customization is adjustment at runtime environment.

Rewriting byte code also can be applied at load time rather than run time; using an application level class loader or prior to load time by directly rewriting class files.

In some cases, it gives the programs more flexibility and efficient handling new situations without the need of recompilation. These techniques give us the ability to use java objects & classes later at runtime, allowing us manipulating and modifying startup of the application and its behavior.

The rest of this paper organized as follows. We presented the related works in section 2, Section 3, gives a brief description and analysis of available code obfuscations techniques. In section 4, we give a detailed description of our proposed approach. Section 5 contains the experiment result of our proposed approach and finally the conclusion.

2. RELATED WORK

Many researches had been placed in order to achieve the most level of obfuscation and deter reverse engineering form de-compilation the code. In spite of there are many free products and commercial tools that available now a days the need for a robust obfuscation techniques is highly required.

Chan, et al [1] proposed an approach that scramble the identifiers in the java byte code by adding additional information to the identifiers, and stored them into the byte code file. This additional information increase the size of the byte code file and require additional computation time, which reduce the efficiency of program. Furthermore, they proposed several techniques that introduce a syntax and semantic errors into the decompiled program while preserving the original behaviors of the byte code.

Memon, et al [2] proposed to remove the name of the variables and methods, which did not allow the completion of code statements, although this technique can fool some de-compilers, but unfortunately, most of recent smart de-compilers can substitute these names with sequentially names and exceed this trick easily.

In addition, the techniques cannot be applicable to all methods such as the instance method that implements an abstract method of a superclass or the instance method that overrides an inherited method of a superclass.

On another hand, Balachandran et al [5] proposed a software obfuscation algorithm that move and hide some of the vital source code information such as jump instruction from the original code into data segment. After that reconstructed it dynamically at run time. It also used the concept of junk bytes addition in order to increase the complexity of disassembly process. However, the size of the program will be duplicated and increase about 2.2 of the original program due to addition of the reconstructed instructions in the program.

The signals based approach proposed by Debray, et al [7] obfuscate the control flow with the help of signals. The idea is to replace the control flow instructions, such as jump, return and call instruction with a trap instruction. When the trap instruction is raised a signal at run time it will trigger the program signal handler, which will transfer the system control to the original target address again.
Madou, et al [8] proposed an obfuscation technique that based on dynamic code mutation. The main of their work is to mutate the program by running edit scripts. Therefore, some parts of the procedures in the orginal program removed and placed a stub at the entry point of the procedure. During the run time of the program, the parts will be restored. In addition, the routine will go into the stub to execute the editing engine and then the stub will be removed.

P. Sivadasan, et al [19] proposed a framework for hiding integer in java code using Y-factors; they improved the constant hiding techniques proposed by Ertaul et al [21] where the Y_factors could be used to transform the non-negative numbers into simple expression which followed the form of “2*d + r”. They used an array of prime numbers where the sum of the numbers in any pair should be a prime number and the pairs then stored in that array in an increasing order of their sum value.
H.Chen et al [23] propose a scheme to obfuscate the whole control flow of a program and insert bogus code. They use the tags as opaque predicates with modest performance degradation in order to prevent malicious code injection, defeat software piracy and hinder reverse engineering analysis. Their work focus on using two features the architectural for automatic propagation of tags and violation handling of tag misuses. They implement a prototype based on Itanium processors that exploit exception propagation and user-level exception handling.

S.Schrittwieser et al [24] proposed a novel that used the concept of software divarication and apply it to program control flow in order to prevent dynamic attack, they split the code into small parts before diversification where the control flow graph of the software reconstructed before executing the code. They utilize the concept of a branching function by inserting indirect jumps that cannot detect their real jump target until runtime; therefore, the attacker needs to collect all information in order to obtain a complete view of the program.

P. Sivadasan, et al [22] proposed a tool for restructuring arrays of java code, they first spit the array into two arrays then merge and folding the arrays, finally the flatting the array. Their tool generate a class that encapsulate the array object where the instantiated objects of those classes used for source-code writing.

3. CODE OBFUSCATION TECHNIQUES ANALYSIS AND REVIEW

Code obfuscation techniques can be classified into the following: Source code obfuscation, bytecode obfuscation, name obfuscation, String obfuscation, control flow obfuscation, Data obfuscation, and Debug info obfuscation.

Source code obfuscation involve renaming the identifiers and variables with meaningless names, remove comments, changing the formatting of the source code and removing the debugging information. [4] This process will leads to reduce size of software and make running of that software more fast especially in smart phones devices.

Bytecode obfuscation encrypt the identifiers and class names in bytecode files, however the JVM cannot not load and execute encrypted classes. Therefor the encrypted classes should be decrypted in order to enable the JVM load it.

Name obfuscation is the process of replacing classes and methods names with meaningless names or sequences of characters, which will reduce the understanding of the code. [4].

One of the nice advantage of name obfuscation is the considerable class file size reduction, which will reduce the size of the application. [5] However, name obfuscation has its shortage and limitations: (1) names of the standard java API classes, which are a part of the JRE, cannot be obfuscated. (2) You may cannot rename the entities that accessed via reflection at run time. This happened due to reason that the particular method or class might be dynamically accessed, especially if it belongs to a third party or framework or to a part of another application. (3) Serializable classes names may cannot be obfuscated [5]. Many obfuscators tools might automatically excluded the classes that implement the Serializable interface.

String encryption involved replacing string literals values with calling to a method that decrypts its parameter in order to makes the attacker life more difficult. Nevertheless, this method face a major problem that is the strings should be decrypted at run time, so the particular code must be included in the application. Moreover, the attacker can easily decompiled that code. [5].

Control flow obfuscation is the process of modifying the program by replacing the instructions produced by a Java compiler with jump instructions to change the control flow of the program that may not be decompiled into valid well-structured Java source or lead to unrelated code.

Therefore, the de-compiler may fail to return the original code. However, not all de-compilers are that dumb. In addition, these techniques have their drawbacks and disadvantages, which are (1), code flow obfuscation reduce the performance of program. (2) Field engineering may be difficult. (3) Standard java API classes may cannot be obfuscated.

Debug info obfuscation work by hiding debug information generated by java compiler, this information will be need to get meaningful stack traces such as line number information and source file names to the resulting class files. In case of obfuscation all these information may remove, or change file names to meaningless strings.[4] However a reverse mapping utility need in order to retrieve the original stack trace again.

Due to our comprehensive study, we find that most of the available tools and software suffer from the same problem, which is the disclosure of their decryption process after a hard work of encryption. All of them have certain shortage and drawbacks, and do not solve the fundamental problem of undesirable exposure of obfuscation and encryption algorithms including the ideas, data formats, licensing and security mechanisms that enable reverse engineers from hacking proprietary applications Therefore, the need for a new robust approach is highly recommended.

4.PROPOSED DMLJCOT

The Proposed DMLJCOT resolves the main problems that other approaches could not handle by applying the following powerful features:

1. Obfuscating identifiers names and encrypting variables value should not relying on the application that delivered in byte code form.
2. Integrating of multi-levels of obfuscation because depending on one level will not be sufficient to prevent reverse engineering.
3. Compiled the obfuscated application down in order to optimize the native source code.
4. Reduce the size of programs by replacing the names of variables and method with short names, which will reduce the program compilation time.
5. Hide the decryption algorithms and the mechanisms used in obfuscated program.
6. We do not need to decrypt the obfuscated source code at the first level of obfuscation, because we encrypt the source code without violating the java language specifications.
7. We use advanced programming techniques such as Compile time Reflection and Metaprogramming. Which give us the ability to inspect classes, interferes fields and methods at runtime, which enable us to develop and design encryption/decryption algorithms that can access, modify other programs and the program itself at run time.

The proposed DMLJCOT integrates the following levels of obfuscation:

- Source code obfuscation level: At this level, proposed technique, obfuscate the source code of the program by replaced identifiers such as variables, functions and classes names with nonsense names that convey no information.
- Data obfuscation level: at this level, the proposed technique encrypts the values of constants, local and global program variables in order to make the de-compilation process more complex.
- Bytecode obfuscation level: at this level, the proposed encryption algorithm will substitute the identifiers names that stored in byte code with Illegal obfuscated identifiers, which will generate a syntax and compilation errors by de-compilers.

4.1 First Level: Source Code Obfuscation

At this level, the proposed technique obfuscate the source code of the program by removing whitespaces and indentation , strip comments, encode the constants in inconveniently readable ways, and replaced identifiers such as variables, functions and classes names with nonsense names that convey no information. It aims to prevent the human understanding of the code, which complicates the statistics analysis of the source code. In addition, this mechanism will reduce the size of program as a result of replacing long names of identifiers with short nonsense names. The generated nonsense names should not violate the java language naming specifications or causing any compilation or syntax errors. There are several methods in java that could not be obfuscated like instance methods that implements an abstract method of a superclass, instance methods that overrides an inherited method of superclass and instance methods that used as a callback function and they should be excluded from obfuscation by including them in an exception list. However, the others methods could be obfuscated without any problems.

Proposed source code obfuscation algorithm has the flowing general steps:

1. Traverse the java package and class structure from top to down.
2. If the method is in exception list, keep it without obfuscation; otherwise apply the “stringShuffle” algorithm that will replace the original identifiers with randomly generated nonsense names. The generated nonsense names should contain a letter, number, and specific allowed special characters which are dollar sign & underscore characters ($, _).
3. Apply the cleaning and optimizing process that will be removing nice indentation and whitespace, strip comments, remove annotation, and hide debug information.
4. Save the updated file and continue to another file on the application.

Source code obfuscation algorithm Flow chart is illustrated in in figure 1.

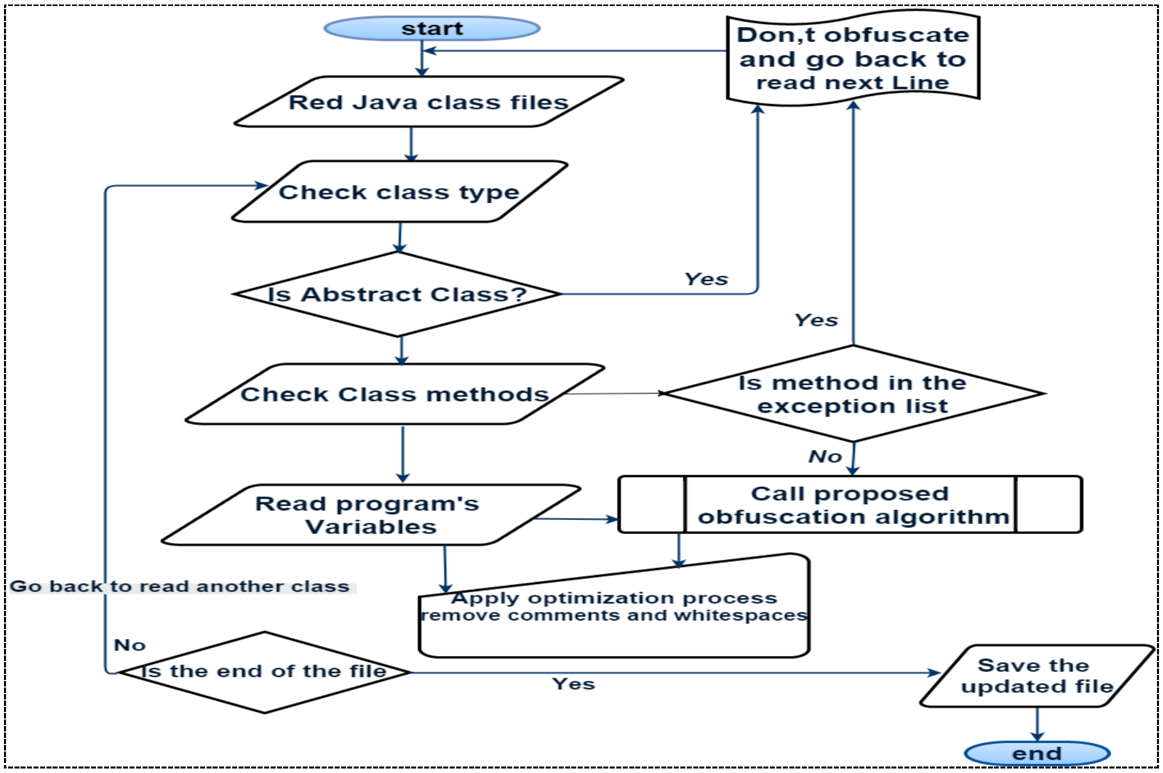

start
Red Java class files
Check class type
Is Abstract Class?
Yes
Don,t obfuscate and go back to read next Line
Check Class methods
Is method in the exception list
Yes
No
Read program's Variables
Call proposed obfuscation algorithm
Apply optimization process remove comments and whitespaces
Go back to read another class
No
Is the end of the file
Yes
Save the updated file
end

FIGURE 1: Source code obfuscation algorithm.

Proposed generator algorithm generate a random nonsense names by applying a combination of letters, numbers, and specific allowed special characters which are the dollar sign and underscore ($, _) characters. The generated nonsense names should follow the java naming specifications and acceptable as an identifiers names.

The way of generating the nonsense involved the following steps:

(1) Generate random characters in the ranges of (a-z, A-Z).
(2) Generate random numbers from (0 -10).
(3) Combine the generated value randomly with allowable special characters ($, _) symbols.
(4) Shuffled the generated nonsense by applying the proposed string shuffle algorithm.
(5) Check if the shuffled nonsense follow the java naming specifications, if so accept it as a nonsense; otherwise reject it and generate another one.

Random seed generator algorithm shown in algorithm 1.

Algorithm 1: Random Seed Generator Algorithm

```
PUBLIC STATIC INT RANDNUMINT() {
RANDOM RANDOM = NEW RANDOM();
INT  RANDOMNUM = GENERATE_RANDOM_INTEGER (SEAD) ;  // GENERATE RANDOM INTEGER FROM (0-9)
   RETURN RANDOMNUM;  }
  PUBLIC STATIC CHAR RANDOMCHARACTERGEN() {
  CHAR BASEVALUE GENERATE_RANDOM_CHARACTERS() ;
    RETURN (CHAR) RANDOM;  }
```

```
PUBLIC STATIC STRING RANDOMSEEDGENERATOR()
{     STRING SEED="";        STRING SPECIAL_CHARACTOR="$_";
    CHAR CHARACTER;     INT RANDOM;
       FOR (INT J = 1; J <= 3; J++)
       {       CHARACTER =  RANDOMCHARACTERGEN();
         RANDOM = RANDNUMINT ();      SEED = SEED +  CHARACTER + RANDOM;          }
SEED = SEED + SPECIAL_CHARACTOR;   // GENERATE RANDOM SEAD
RETURN SEED; }
```

In order to complicate the way of generating the nonsense names and make it more confused we randomly permutated and shuffled the generated nonsense names. However, we should ensure that the generated shuffled nonsense should not violate the java language specification; consequently, we replaced all identifiers names with the shuffled nonsense names. The way of generating and shuffle, the nonsense names shown in algorithm 2 and figure 2 bellow.

Algorithm 2: String Shuffle Algorithm

```
// SEND SEED TO STRING SHUFFLE METHOD AS A PARAMETERS USING STRING BUILDER CLASS.
STRINGBUILDER SB = NEW STRINGBUILDER(SEAD);
STRINGSHUFFLE(SB);    //   STRINGSHUFFLE METHOD
PUBLIC STATIC VOID STRINGSHUFFLE(STRINGBUILDER SHUFFLESTR) {
   RANDOM NEWRAND = NEW RANDOM();
   FOR (INT K = SHUFFLESTR.LENGTH() - 1; K > 1; K--) {
      INT SWAPWITH = NEWRAND.NEXTINT(K);
      CHAR TMPCHAR = SHUFFLESTR.CHARAT(SWAPWITH);
      SHUFFLESTR.SETCHARAT(SWAPWITH, SHUFFLESTR.CHARAT(K));
      SHUFFLESTR.SETCHARAT(K, TMPCHAR);    } }
```

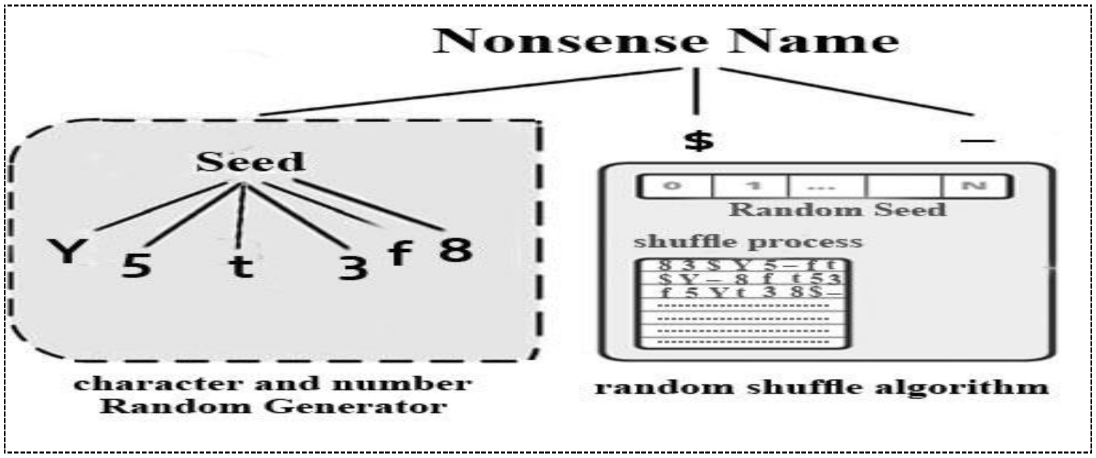


FIGURE 2: Random generation & Shuffle nonsense names.

As a next step, we optimized the source code by removing the whitespaces; comments remove annotation, hide debug information.

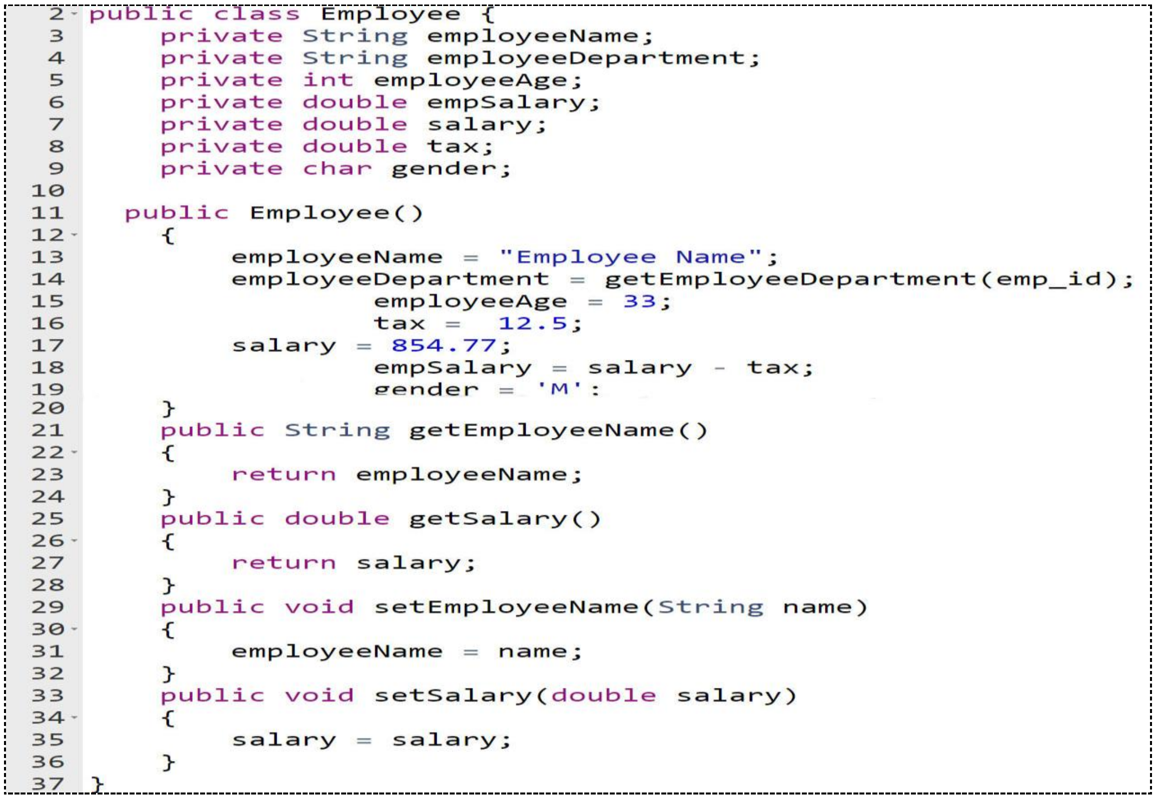
The results after applying the first level of obfuscation shown in figure 3 and 4 below. As can see that all identifiers are transformed to nonsense names, which will make the code less clear and difficult to comprehend by human, as an example variable "employeeName" transformed to "_5X1g$8J" nonsense name.

4.2. Second level: Data Obfuscation

At this level, the proposed technique encrypts the values of constants, local and global program variables in order to make the de-compilation process more complex.

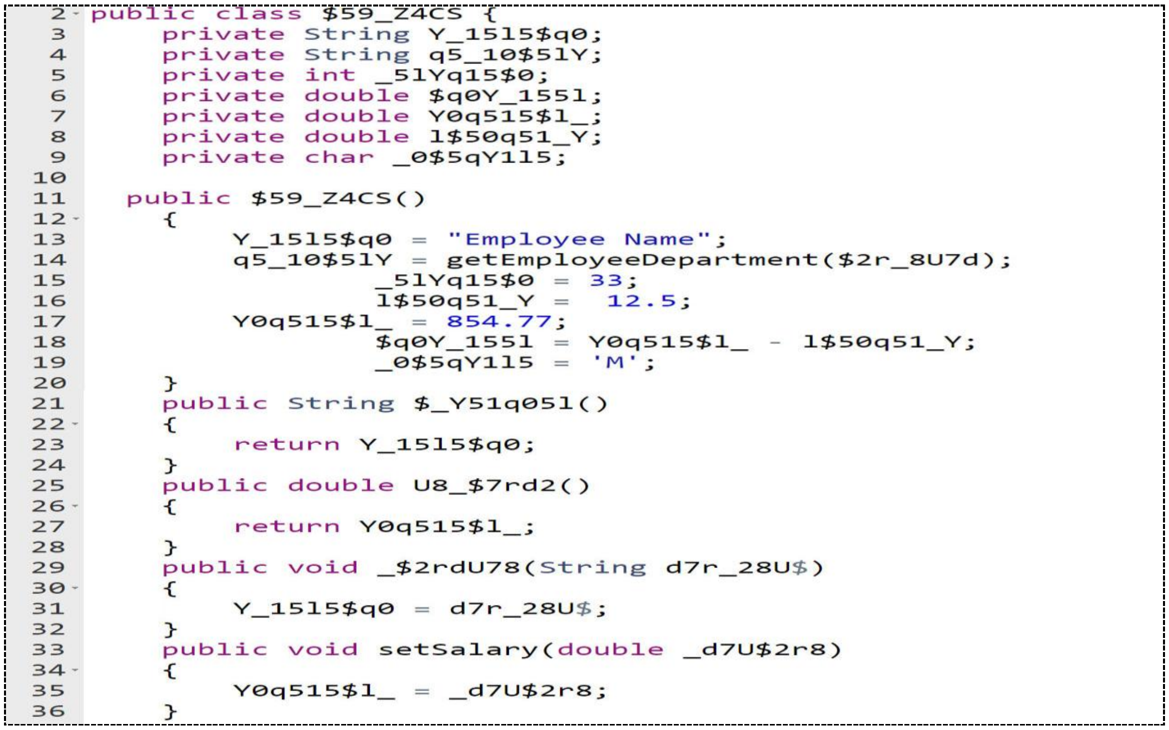
In order to complicate the process of data manipulation and understanding we apply several encryption algorithms and each of them dedicated for one variable data type. The proposed technique allow the selection of encryption algorithm for string, characters, integers and decimal number such as float and double in a random manner. The same encryption key used in all encryption algorithms.

The static data will be encrypted during the program initialization and the dynamic data are encrypted and decrypted by using dedicated methods that manipulate variables values. The encryption algorithm shown in algorithm 3 bellow.



```
public class Employee {
    private String employeeName;
    private String employeeDepartment;
    private int employeeAge;
    private double empSalary;
    private double salary;
    private double tax;
    private char gender;

  public Employee()
    {
        employeeName = "Employee Name";
        employeeDepartment = getEmployeeDepartment(emp_id);
                employeeAge = 33;
                tax =  12.5;
        salary = 854.77;
                empSalary = salary - tax;
                gender = 'M';
    }
    public String getEmployeeName()
    {
        return employeeName;
    }
    public double getSalary()
    {
        return salary;
    }
    public void setEmployeeName(String name)
    {
        employeeName = name;
    }
    public void setSalary(double salary)
    {
        salary = salary;
    }
}
```

FIGURE 3: Employee class original source code before obfuscation.



```
public class $59_Z4CS {
    private String Y_1515$q0;
    private String q5_10$51Y;
    private int _51Yq15$0;
    private double $q0Y_1551;
    private double Y0q515$1_;
    private double 1$50q51_Y;
    private char _0$5qY115;

  public $59_Z4CS()
    {
         Y_1515$q0  =  "Employee Name";
         q5_10$51Y  =  getEmployeeDepartment($2r_8U7d);
                 _51Yq15$0  =  33;
                 1$50q51_Y  =   12.5;
         Y0q515$1_  =  854.77;
                 $q0Y_1551  =  Y0q515$1_  -  1$50q51_Y;
                 _0$5qY115  =  'M';
    }
    public String $_Y51q051()
    {
         return Y_1515$q0;
    }
    public double U8_$7rd2()
    {
         return Y0q515$1_;
    }
    public void _$2rdU78(String d7r_28U$)
    {
         Y_1515$q0  =  d7r_28U$;
    }
    public void setSalary(double _d7U$2r8)
    {
         Y0q515$1_  =  _d7U$2r8;
    }
```

FIGURE 4: Employee class source code after first level of obfuscation.

Algorithm 3: Data encryption algorithm

```
PUBLIC CLASS DATAENCRYPTION {
  STRING ENCRPTIONKEY= "";
  //GENERATE RANDOM ENCRYPTION KEY USING RANDOM KEY GENERATOR
  PUBLIC DATAENCRYPTION()    {
 ENCRPTIONKEY = KEY_RANDOM_GENERATOR();
 //GENERATE THE PERMUTATION VECTOR
 FOR(KI IN ENCRPTIONKEY)
    PERMUTATIONVECTOR[I] = KI MOD 8 // SEQUECE FORM 0-7
    END FOR LOOP
   //TRAVERSE_PROGRAM_VAIABLES
    BEGININDEX = 0;
   FOR(VARIBLE I IN PRGRMA)
   {     //DETERMINED VARIABLE DATA TYPE
   VARDATATYPES =DETERMINED_VARAIBLES_DATA_TYPES(VARIBLE I);
   NOOFBITS = VARIBLE.LENGTH() ;
   ENDINDEX = NOOFBITS;
   IF(VARDATATYPES EQUAL "INTEGER")
         CALL INTEGER_ENCRYPTIONALGORITHM(VARIBLE I);
   ELSE  IF(VARDATATYPES EQUAL "FLOAT" OR VARDATATYPES EQUAL "DOUBLE")
      CALL DECIMAL_ENCRYPTIONALGORITHM(VARIBLE I);
   ELSE IF (VARDATATYPES EQUAL "STRING")
      CALL STRIN_ENCRYPTIONALGORITHM(VARIBLE I);
   ELSE IF (VARDATATYPES EQUAL "CHAR")
      CALL CHAR_ENCRYPTIONALGORITHM(VARIBLE I);
   }  }  }
```

4.2.1. Description of used encryption Algorithms

In order to avoid producing unprintable or not allowed symbols in the output we construct a Code table that contains all possible characters and their codes that may be used in the programming languages and it will be used by all encryption algorithms see table 1 below.

a	b	c	d	…	z	A	B	…	Z	0	1	…	9	!	@	#	%	?	Space	…
0	1	2	3	…	25	26	27	…	51	52	53	…	61	62	63	64	65	66	67	…

TABLE 1: Transformation Table.

The used encryption algorithms are based on the Permutation and Substitution encryption principles in order to enhance performance, confusion and diffusion characteristics of the Ciphertext.

4.2.2. Permutation Algorithm (PA)

The Permutation algorithm is a block cipher that handle each plaintext block independently and the length of each block equal the length of the keyword in terms of positions number. The permutation process starts by initialization the vector of permutation (VP) which is key dependent and computed by using the following formula:

$$Vpi = Ki*(i+1) \text{ Mod } N \qquad i= 0, 1….N-1; \qquad (1)$$

Where, N - the length of block cipher;
Ki = code of ith Key Character;

For implementation issues related to data presentation we suggest that N=8 and consequently the key size =8; the resulting VP will be as shown in table 2. The first element of VP dictates that the first element of plaintext should be permuted with fourth element (0←→4). From table 2, it is clear that the proposed algorithm allows the duplication of swapped position in order to enhance its confusion characteristics. The proposed algorithm uses sliding window technique to deal with the Blocks that their size less than 8 by borrowing from the previous block the needed number of character to fill the last block.

Position	0	1	2	3	4	5	6	7
Transposition	4	5	2	1	0	7	3	4

TABLE 2: Permutation Vector.

4.2.3. Substitution Algorithm (SA)

SA algorithm encrypts the integer, by XORing the last element of the key with the integer, the result value of XOR operation XORed with the previous element of the key. This process repeated until we reach the first element of the key. The encryption function takes two parameters A and K, where A is the integer value and K is the key, which is treated as an array of characters. This process is described in algorithm 4.

Algorithm 4: Integer Encryption Algorithm

```
PUBLIC CLASS DATAENCRYPTION {
  PUBLIC INT ENC(INT A,STRING K){
 // INITIAL VALUE OF ENCRYPTED INTEGER IS A
   INT ENCRYPTED_INTEGER = A;
   FOR(INT I =LENGTH[K] ; I<=0 ; I--)    {
   //XORING VALUE OF INTGER WITH THE KEY
   ENCRYPTED_INTEGER = ENCRYPTED_INTEGER ⊕ K[I];
   } //END LOOP
   RETURN ENCRYPTED_INTEGER ;
  }  }
```

We encrypt the float and double using the same technique described above but with small modification that we ignore the decimal point and take the entire number, as an integer. I.e. a float number such as 10.5 treated as 105 ignoring the decimal point.

In order to retrieve the original value of number at run time we send the count the digits before the decimal point as another parameter to the decryption algorithm. On other hand, the string and characters encrypted using PA described previously.

The next step that follows the encryption process of data is the replacement of data value by calling the decryption algorithm in order to obfuscate the real value which will complicate the disassembly of the code , and this lead to preventing tracing memory tools from detecting the variable value.

In addition, we obfuscate the name of our decryption algorithm to be “Ob_f57_oR” in order to hide its real name from attackers.

Proposed technique call fake methods instead of original ones in case if the variable assignment is a calling to function. As an example if we have a variable called x which assigned to calling to function named mem1 suchlike.

“x = mem1 ()” it will be replaced with “x= fakeMethode1 ()”, fake method names will start by “Fa_” sequence of characters and the rest characters will be generated randomly. During run time, the decryption algorithm will reverse the process and return the original method names.

The structure of Data encryption process and the results of second level obfuscation shown in figure 5 and figure 6.

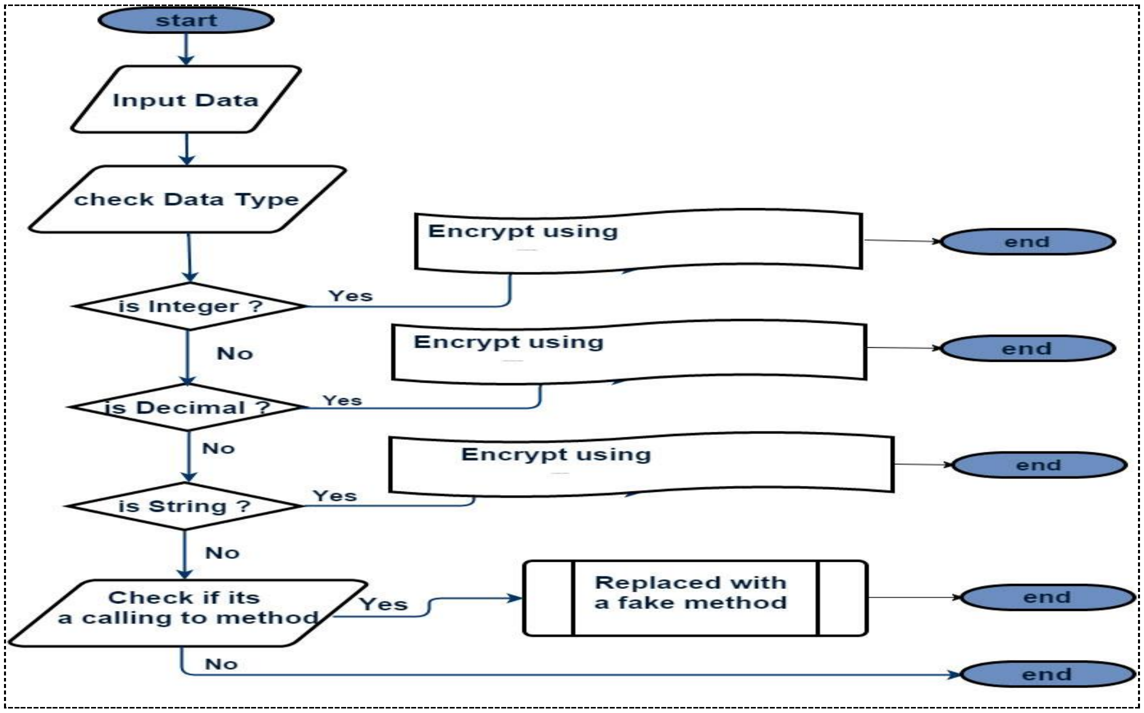
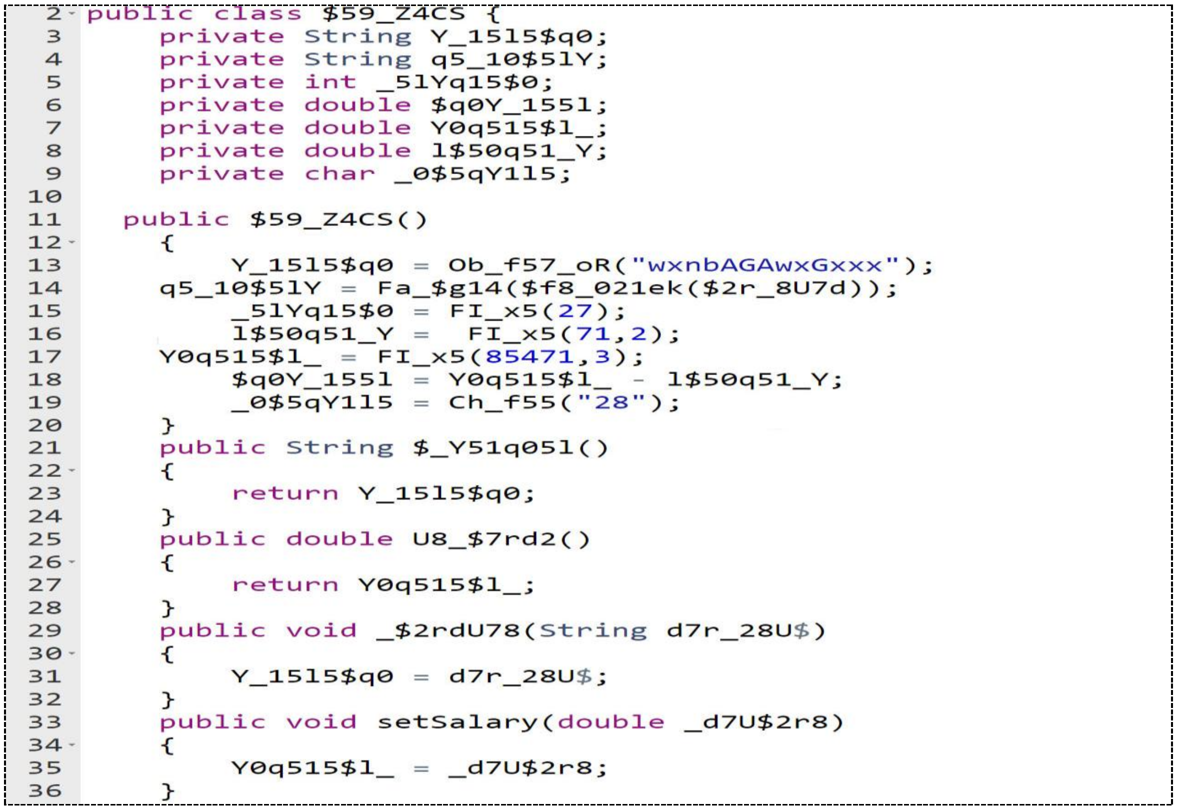


FIGURE 5: Structure of Data encryption process.



```
public class $59_Z4CS {
    private String Y_1515$q0;
    private String q5_10$51Y;
    private int _51Yq15$0;
    private double $q0Y_1551;
    private double Y0q515$1_;
    private double 1$50q51_Y;
    private char _0$5qY115;

  public $59_Z4CS()
    {
        Y_1515$q0 = Ob_f57_oR("wxnbAGAwxGxxx");
    q5_10$51Y = Fa_$g14($f8_021ek($2r_8U7d));
        _51Yq15$0 = FI_x5(27);
        1$50q51_Y =  FI_x5(71,2);
    Y0q515$1_ = FI_x5(85471,3);
        $q0Y_1551 = Y0q515$1_ - 1$50q51_Y;
        _0$5qY115 = Ch_f55("28");
    }
    public String $_Y51q051()
    {
        return Y_1515$q0;
    }
    public double U8_$7rd2()
    {
        return Y0q515$1_;
    }
    public void _$2rdU78(String d7r_28U$)
    {
        Y_1515$q0 = d7r_28U$;
    }
    public void setSalary(double _d7U$2r8)
    {
        Y0q515$1_ = _d7U$2r8;
    }
```

FIGURE 6: Employee class source code after the second level of obfuscation.

4.3. Third level: Bytecode Obfuscation (BO)

At this level, proposed encryption algorithm will substitute the identifiers in byte code with Illegal obfuscated identifiers in order to generate a syntax and compilation errors when it decompiled and recompiled again by attackers.

Java language specification states that an identifier cannot be begin with number or contain some special character such as (;), (:), (/), (%), (!), (#) or space, etc. It should be start with a letter followed by a mixture of letters and digits.

In addition, it cannot be similar to a reserved keyword such as null or Boolean literal.

Lexical analyzer can exploit these rules in order to parsing and analyzing a program. However, in the byte code these rules need not be complied because JVM loads the bytecode without verifying whether the names in the constant pool obey with the identifiers definition or follow the Java language naming specification. Consequently the constant pool of the bytecode can contains illegal characters, keywords, null or Boolean literals and this is will be exploited by our proposed technique by changing the identifiers to be illegal in order to cause in compilation error When the obfuscated bytecode decompiled and recompiled again.

Proposed BO algorithm exploit these rules in order to obfuscate the identifiers names stored in byte code with illegal names that does not follow the lexical of Java language specification. Therefore, the attacker will spend a lot of time and effort debugging it, which is useless. In addition, the de-compiler tools will face a big problem treating such illegal symbols and names, and this will make it too difficult or impossible for any, decompiling tools to obtain the original names or handle these illegal characters and by this way we can prevent the dynamic reverse engineering. There are some characters and symbols that cannot be used as an identifier as they have a specific meanings for JVM such as ‘‘<init>’’ which used by JVM to call the constructors, ‘‘<clinit>’’ which used by JVM for static members initialization. In addition to the characters, ‘‘/’’, ‘‘:’’ and ‘‘n’’, as the JVM used these characters as a path separator in the file systems host. Furthermore the character ‘‘$’’ is used by JVM as a separator between type and its nested types [1]. Proposed BO algorithm uses a combination of illegal special characters, which are chosen randomly, by using the following steps:

1. Generate random obfuscated name
2. Select randomly number or illegal special characters from the list |! | # | % | @ | * | _ | . | ; |.
3. Append the selected characters generated from step 2, at the beginning of the obfuscated names.
4. Replace all identifiers names in constant pool with the generated obfuscated names.

Proposed BO algorithm used semi colon (;) which means end of statement therefore decompiles will treat this name as two variables. As an example if we have the flowing statement (x; y). Decompiles divided into two variables x and y instead of treated it as one variable.

We used the dot character in our encryption process which treated as separators of tokens in a source program, which will make the task of de-compiler more difficult because the Java compiler will consider ‘‘.’’ as a separator between a reference and its members object or a type. As we will see in the experimental results that all de-complies fooled by this illegal symbol.

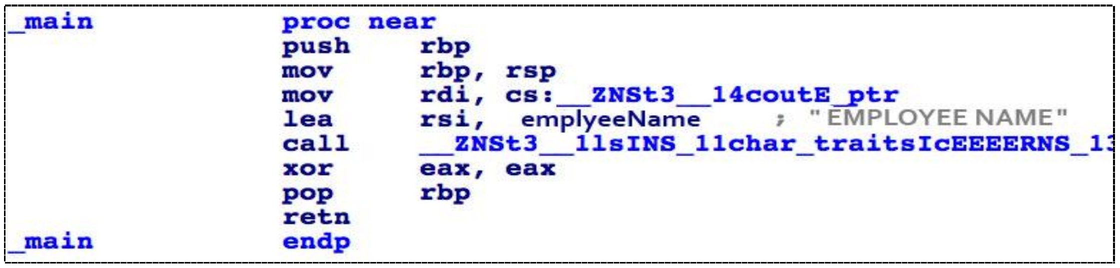
5. EXPERIMENTAL RESULTS

Our experiment proves that any attacker cannot be aware of what happened at run time even when he is trying to disassembly the code or tracing the memory. In order to evaluate our approach against dissemblers and memory tracing tools we select a portion of code from employee class shown in figure 3 and 4. The code we selected shown in figure7.

```
PUBLIC STATIC VOID MAIN(STRING[] ARGS) {
PRIVATE STRING emplyeeName;
 emplyeeName = "EMPLOYEE NAME";
SYSTEM.OUT.PRINTLN(emplyeeName);  // PRINT THE VALUE ON CONSOLE
}
```

FIGURE 7: Portion of code from employee class (without obfuscation)

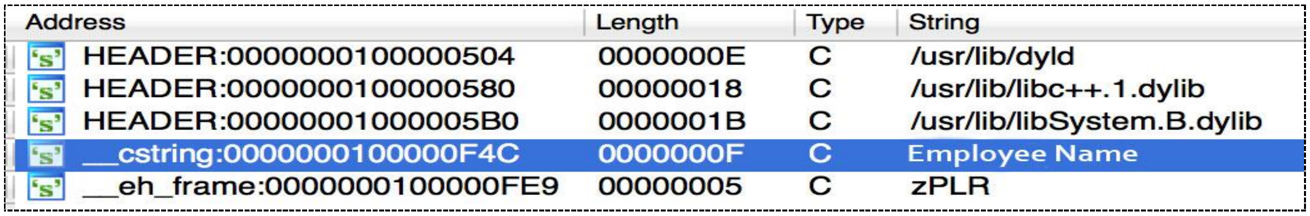
We use Jasmin Java Assembler software that convert Java classes into binary “Java. Class” files that are convenient for loading into a Java Virtual Machine by taking the ASCII descriptions of Java classes. In order to tracing the memory, we used VisualVM, which is a visual tool that give us the capability of performance analysis and monitoring for the Java SE platform. VisualVM integrated several command line JDK tools and lightweight profiling capabilities so it can be used for both production and development. The disassembly results of the code in figure 7, shown in figure 8 below:



```
_main               proc near
                    push    rbp
                    mov     rbp, rsp
                    mov     rdi, cs:__ZNSt3__14coutE_ptr
                    lea     rsi,  emplyeeName      ; "EMPLOYEE NAME"
                    call    __ZNSt3__11sINS_11char_traitsIcEEEERNS_1
                    xor     eax, eax
                    pop     rbp
                    retn
_main               endp
```

FIGURE 8: Disassembly results of code in figure 7.

As obvious from figure 8 above that the variable value detected easily after disassembly the code, also it can be easily traced if we use any memory-tracing tool as obvious in figure 9 bellow



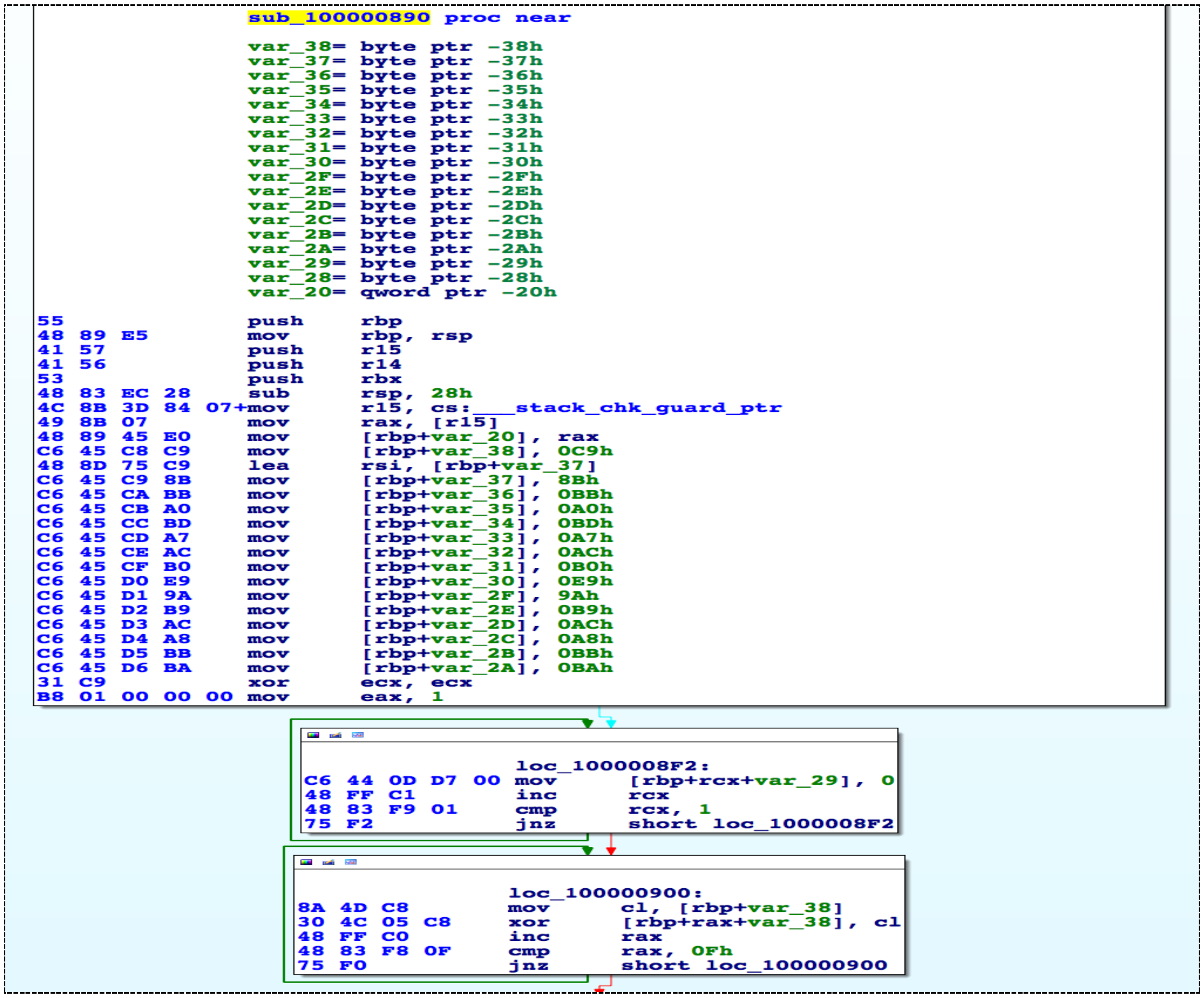
Address	Length	Type	String
HEADER:0000000100000504	0000000E	C	/usr/lib/dyld
HEADER:0000000100000580	00000018	C	/usr/lib/libc++.1.dylib
HEADER:00000001000005B0	0000001B	C	/usr/lib/libSystem.B.dylib
__cstring:0000000100000F4C	0000000F	C	Employee Name
__eh_frame:0000000100000FE9	00000005	C	zPLR

FIGURE 9: Tracing memory of the code in figure 7.

The equivalent obfuscated code after applying the first and second levels of obfuscation using our proposed algorithms shown in figure 10 below:

```
    PUBLIC STATIC VOID MAIN(STRING[] ARGS) {
    PRIVATE STRING Y_15L5$Q0;                //AFTER OBFUSCATE THE IDENTIFIER NAME
    Y_15L5$Q0 = OB_F57_OR("WXNBAGAWXGXXX"); // WHERE"OB_F57_OR"IS OUR OBFUSCATOR
METHOD ,"WXNBAGAWXGXXX" IS THE OBFUSCATED VALUE;
    SYSTEM.OUT.PRINTLN(Y_15L5$Q0);           // PRINT THE VALUE ON CONSOLE
    }
```

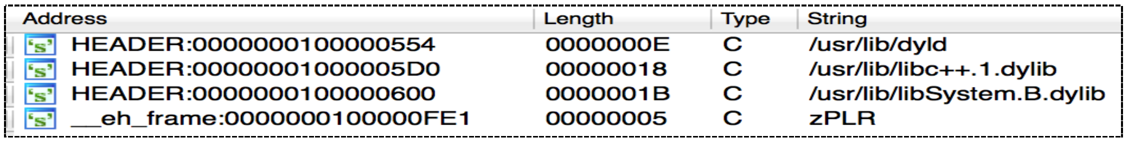
FIGURE 10: Equivalent obfuscated code of the code in figure 7.



```
                     sub_100000890 proc near

                     var_38= byte ptr -38h
                     var_37= byte ptr -37h
                     var_36= byte ptr -36h
                     var_35= byte ptr -35h
                     var_34= byte ptr -34h
                     var_33= byte ptr -33h
                     var_32= byte ptr -32h
                     var_31= byte ptr -31h
                     var_30= byte ptr -30h
                     var_2F= byte ptr -2Fh
                     var_2E= byte ptr -2Eh
                     var_2D= byte ptr -2Dh
                     var_2C= byte ptr -2Ch
                     var_2B= byte ptr -2Bh
                     var_2A= byte ptr -2Ah
                     var_29= byte ptr -29h
                     var_28= byte ptr -28h
                     var_20= qword ptr -20h

55                   push    rbp
48 89 E5             mov     rbp, rsp
41 57                push    r15
41 56                push    r14
53                   push    rbx
48 83 EC 28          sub     rsp, 28h
4C 8B 3D 84 07+mov   r15, cs:___stack_chk_guard_ptr
49 8B 07             mov     rax, [r15]
48 89 45 E0          mov     [rbp+var_20], rax
C6 45 C8 C9          mov     [rbp+var_38], 0C9h
48 8D 75 C9          lea     rsi, [rbp+var_37]
C6 45 C9 8B          mov     [rbp+var_37], 8Bh
C6 45 CA BB          mov     [rbp+var_36], 0BBh
C6 45 CB A0          mov     [rbp+var_35], 0A0h
C6 45 CC BD          mov     [rbp+var_34], 0BDh
C6 45 CD A7          mov     [rbp+var_33], 0A7h
C6 45 CE AC          mov     [rbp+var_32], 0ACh
C6 45 CF B0          mov     [rbp+var_31], 0B0h
C6 45 D0 E9          mov     [rbp+var_30], 0E9h
C6 45 D1 9A          mov     [rbp+var_2F], 9Ah
C6 45 D2 B9          mov     [rbp+var_2E], 0B9h
C6 45 D3 AC          mov     [rbp+var_2D], 0ACh
C6 45 D4 A8          mov     [rbp+var_2C], 0A8h
C6 45 D5 BB          mov     [rbp+var_2B], 0BBh
C6 45 D6 BA          mov     [rbp+var_2A], 0BAh
31 C9                xor     ecx, ecx
B8 01 00 00 00 mov   eax, 1

                     loc_1000008F2:
C6 44 0D D7 00       mov     [rbp+rcx+var_29], 0
48 FF C1             inc     rcx
48 83 F9 01          cmp     rcx, 1
75 F2                jnz     short loc_1000008F2

                     loc_100000900:
8A 4D C8             mov     cl, [rbp+var_38]
30 4C 05 C8          xor     [rbp+rax+var_38], cl
48 FF C0             inc     rax
48 83 F8 0F          cmp     rax, 0Fh
75 F0                jnz     short loc_100000900
```

FIGURE 11: Disassembly results of the code in figure 10.

As a result, of implementing our approach we noticed that the disassembly of code became very difficult. In addition to that, the value of obfuscated variable can't be detected by memory tracing tools as obvious in figure 12 bellow, because we replaced its value by calling a method that obfuscate the value by using encryption algorithm. The tracing does not contain the original string and the encrypted one does not appear. We apply same techniques described previously in order to obfuscate the other identifiers of an application.



```
Address                              Length     Type   String
HEADER:0000000100000554              0000000E   C      /usr/lib/dyld
HEADER:00000001000005D0              00000018   C      /usr/lib/libc++.1.dylib
HEADER:0000000100000600              0000001B   C      /usr/lib/libSystem.B.dylib
__eh_frame:0000000100000FE1          00000005   C      zPLR
```

FIGURE 12: Tracing memory of the code in figure 10.

The proposed technique has been applied on Employee class and then test against several available de-compilers. The results show that all decompiles are fooled by the proposed technique as the decompiled program contains a superfine bugs which it is difficult to discover. The results in Table 3 show that Java Decompiler Project (JD) are smarter than other de-compliers when handling keywords identifier as it can retrieve the ordinary keyword identifier automatically. Also it can replace some illegal symbols with underscore character, while the other decompiles use in the decompiled program the identifiers names similar to the keyword without any changes. The results of testing obfuscated code produced by our technique by using many Java De-compilers such as Cavaj, DJ, JBVD, and AndroChef show that they all fooled by the illegal symbols and failed to decompile the obfuscated byte code.

	Decompiled results After using illegal symbols	**Decompiled results After using semi colon ";"**	**Decompiled results After using dot "."**
Original Identifier	String employeeName="Employee Name"	String employeeName="Employee Name"	String employeeName="Employee Name"
Obfuscated Identifiers	String #_1#l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx")	String #_1;l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx")	String #_1.l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx")
JD	String Y_1_l_q_0 = Ob_f57_oR("wxnbAGAwxGxxx")	String Y_1; l_q_0=getEmployeeDepartment($2r_8U7d) Can't find symbol : l_q_0	String Y_1.l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Exception error
Cavaj	String Y_1#l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Illegal character :'#' <identifier> expected Illegal character:'!' <identifier> expected Can't find symbol : class $q0	String Y_1;l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Illegal character :'#' <identifier> expected Illegal character:'!' Illegal character:'%' Can't find symbol l <identifier> expected	String Y_1.l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Exception error
DJ	String Y_1#l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Illegal character :'#' <identifier> expected Illegal character:'!' <identifier> expected Can't find symbol : class $q0	String Y_1;l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Illegal character :'#' <identifier> expected Illegal character:'!' Illegal character:'%' Can't find symbol l <identifier> expected	String Y_1.l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Exception error

JBVD	String Y_1#l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Illegal character :'#' <identifier> expected Illegal character:'!' <identifier> expected Can't find symbol : class $q0	String Y_1;l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Illegal character :'#' <identifier> expected Illegal character:'!' Illegal character:'%' Can't find symbol l <identifier> expected	String Y_1.l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Exception error
AndroChef	String Y_1#l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Illegal character :'#' <identifier> expected Illegal character:'!' <identifier> expected Can't find symbol : class $q0	String Y_1;l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Illegal character :'#' <identifier> expected Illegal character:'!' Illegal character:'%' Can't find symbol l <identifier> expected	String Y_1.l!q%0 = Ob_f57_oR("wxnbAGAwxGxxx") Exception error

TABLE 3: De-compilation testing results.

It is clear from the above table that JD are smarter than others when it handled illegal symbols such as "!,#", it replaced them by underscore character "_", where the others return the same obfuscated statement as it is without any modification which cause compilation error and results an exceptions.

On the other hand, all de-compliers fooled when they treated with semi colon and dot symbols, although the JD tried to handle the semi colon by dividing the variable into two different statements and replace the illegal characters with underscore characters but it causes an exception and failed to decompile because the Java compiler cannot find declaration of variable " l_q_0".

The proposed approach confused all de-compliers and prevent them from return the original code, and therefore the professional attacker has to spend a lot of time trying to debug the code.

6. COMPARATIVE EVALUATION AND IMPROVEMENTS

Most obfuscators provide only one protection level, some of them worked with high program level; they worked only with layout obfuscation. Therefore, they could not prevent the de-compilers from decompiling the source codes obfuscated by them. On the other hand, other obfuscators only scramble identifier names, which we consider a low-level protection as it does not work well with recent smart de-compilers. Although such obfuscators could successfully obfuscate the simple source codes; however, they could not properly obfuscate the complex programs or logical Programs.

Layout obfuscation simply parsing the data structures of source code according to the language lexical rules and syntax, carry out the obfuscation, and then un-parsing them back to the original source code while in our approach we improved such transformation with a way that break the relationship between code statements through replacing the meaningful identifiers with nonsense names that convey no information where the generated nonsense names should not violate the java language naming specifications or causing any compilation or syntax errors. Therefore, we did not need to un-parsing the code back to the original source code such as the tools that depend on layout transformation.

Array restructuring working only in transforming integer arrays where no string encryption added to the program. Other data obfuscators transform only one datatype such as integer or string where other types will not be obfuscated, which leads to low level of obscurity while in our approach we apply several encryption algorithms each of them dedicated for one datatype, we obfuscate string, characters, integers and decimal number such as float and double in a random manner. By this way, we complicated the process of data manipulation and understanding; we take in our concern the case if the variable assignment is a calling to function where in this case, the proposed technique call fake methods instead of original ones that will be return back at run time. Such improvement convey most data types, which give our approach more robustness.

Control flow obfuscation simply restructuring branching statements and loops in program to change the control flow of the program. However, altering the control flow may increase the runtime to such a drastic level that could affect the efficiency of obfuscation. The criteria used in evaluating the quality of obfuscation depend on how much obscurity added to the program. However, the combination of control flow obfuscation with data obfuscations techniques might be a good way for an obfuscator to defy against de-compilers.

From discussed above, we see that the best way to improve the level of obfuscation is make a combination of different obfuscation techniques; therefore, in our approach we make a combination of the source code obfuscation with byte code and data obfuscation, which make the proposed approach robust against many forms of reverse engineering attacks. In addition, we improve the way of producing illegal names stored in bytecode in order to complicate the life of attacker.

When we evaluated our work with other works, we see that it can defeat de-compilers in more efficient manner than others because it work on different obscure levels and obfuscate many datatypes while others approaches worked only on one level or one datatype. The result of evaluations and comparisons related approaches with the proposed approaches appear in table 4 bellow.

Approach	**Obscure levels worked on**	**Methodology**	**Drawbacks & Limitations**
Chan, et al. approach [1]	Bytecode level.	Scramble the identifiers in the java bytecode.	- Increase size of bytecode. - require additional computation time. - reduce the efficiency of program.
Memon, et al. approach [2]	Variables and methods level.	Hide the name of the variables and methods	- Most of recent smart de-compilers can substitute these names with sequentially names and exceed this trick easily. - Cannot be applicable to all methods such as the instance method that implements an abstract method.
Balachandran et al approach [5]	Layout level.	Move and hide some of the vital source code information such as jump instruction from the original code into data segment.	Size of the program will be duplicated and increase about 2.2 of the original one.
P.Sivadasan, et al approach [19]	Data level.	Hiding integer in java code using Y-factor.	Obfuscate only non-negative integer without obfuscating other types such as string.
S.Schrittwieser et al approach [24]	Control Flow level.	Split the code into small parts before diversification where the control flow graph of the	- Require additional computation time due to vast amount of inserted jumps, which will reduce the

		software reconstructed before executing the code.	efficiency of program. - Not cover inter gadget diversification.
P.Sivadasan, et al approach [22]	Array Restructuring level.	Restructuring arrays of java code.	Working only on transformation the integer arrays where no other datatypes encryptions added to the program.
Our Approach (DMLJCOT)	- Variables and methods level. - Layout level. - Data Level. - Bytecode level.	- Obfuscate the source code of the program by replaced identifiers such as variables, functions and classes names with nonsense names that convey no information. - Encrypts the values of constants, local and global program variables - Substitute the identifiers names that stored in byte code with Illegal obfuscated identifiers.	Not cover control flow level.

TABLE 4: Comparative evaluation of related approaches and the proposed approach.

7. CONCLUSION

The power of our approach come from the combination of many obfuscation techniques used to build it; we combine the source code obfuscation with byte code and data obfuscation, which make our approach robust against many forms of reverse engineering attacks. In addition, we used advanced programming techniques such as Compile time Reflection and Metaprogramming for Java. Which give us the ability to inspect classes, interfaces, fields and methods at runtime. The proposed approach in this paper satisfy all levels of obfuscation including the source code, byte code and data obfuscation and preserving the semantics of the byte code which is an important criterion of the obfuscation process. The most cracking tools cannot easily undo the obfuscation effects of our approach, as the attacker will consume a lot of time removing the bugs of the decompiled buggy program. Our experimental results prove that the proposed technique provides stronger bytecode protection than other existing techniques. The proposed obfuscation technique can be implemented in other languages.

8. FUTURE WORK

In order to improve the proposed approach it is possible to add the flow control level as a forth level to make this approach more robustness against many forms of reverse engineering attacks. In addition, future researches may be conducted on binary level of obfuscation; which required working on hardware level.

9. REFERENCES

[1] J.T. Chan and W. Yang, “Advanced obfuscation techniques for java bytecode,” Journal of systems and software, Vol. 71, 2004.

[2] J.M. Memon, S. Arfeen, A. Mughal, F. Memon, “Preventing Reverse Engineering Threat in Java Using Byte Code Obfuscation Techniques”. 2nd International Conference on Emerging Technologies, IEE, Peshawar, Pakistan 13-14 November 2006.

[3] M. Popa, “Techniques of Program Code Obfuscation for Secure Software. Journal of Mobile, Embedded and Distributed Systems, vol. III, no. 4, 2011, ISSN 2067 – 4074.

[4] F. BUZATU, "Methods for Obfuscating Java Programs. Journal of Mobile, Embedded and Distributed Systems, vol. IV, no. 1, 2012,ISSN 2067 – 4074

[5] V. Balachandran and S. Emmanuel, "Software Code Obfuscation by Hiding Control Flow". IEEE, 978-1-4577-1019-3/11/$26.00, 2011.

[6] X. Yao, J. Pang, Y. Zhang, Y. Yu, J. Lu, "A Method and implementation of control flow obfuscation using SEH". Fourth International Conference on Multimedia Information Networking and Security, IEEE, 978-0-7695-4852-4/12 $26.00, 2012

[7] I. Popov, S. Debray, and G. Andrews, "Binary obfuscation using signals," in Proc. of 16th USENIX Security Symposium on USENIX Security Symposium table of contents, USENIX Association Berkely, CA, USA, 2007.

[8] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, K. De Bosschere, "Software Protection Through Dynamic Code Mutation, "LECTURE NOTES IN COMPUTER SCIENCE 3786, 2006, 194.

[9] W. Dingab, Z. Gua, F. Gaoa, "Reconstruction of Data Type in Obfuscated Binary Programs". ICACT, ISBN 978-89-968650-3-2, 2014.

[10] S. Andrivet, "C++11 metaprogramming applied to software obfuscation". Black Hat Europe conference, Amsterdam ,2014

[11] W. Miao, J. Broberg and J. Siek, "Compile-time Reflection and Metaprogramming for Java". ACM, 978-1-4503-2619-3/14/01. . . $15.00,2014

[12] É. Tanter a, R. Toledo a, G. Pothier a, J. Noyéb, "Flexible metaprogramming and AOP in Java". Science of Computer Programming 72 (2008) 22–30.

[13] M. Garci, "Runtime metaprogramming via java.lang.invoke.MethodHandle". LAMP, EPFL, 2012, http://lamp.epfl.ch/~magarcia.

[14] J. MacBride, C. Mascioli, S. Marks, Y. Tang, L.M. Head, and R., P. Ramachandran. "A COMPARATIVE STUDY OF JAVA OBFUSCATORS". Proceeding of the Nith IASTED International Conference SOFTWARE ENGENEERING AND APPLICATION, November 14-16, 2005, Phoenix, AZ, USA.

[15] N. Jeenjun S. Khuntaweetep and S. Somkuarnpanit. "Byte code Interpreter for 8051 Microcontroller". Proceeding of the International Multi Conference of Engineers and Computer Scientists, 2010 Vol II, IMECS 2010, March 17-19, 2010, Hong King.

[16] E. E. Ogheneovo and C.K. Oputeh, "Source Code Obfuscation: A Technique for Checkmating Software Reverse Engineering". International Journal of Engineering Science Invention ISSN (Online): 2319 – 6734, ISSN (Print): 2319 – 6726 www.ijesi.org Volume 3 Issue 5‖ May 2014 ‖ PP.01-10.

[17] S. Treadwell, M. Zhou, "A Heuristic Approach for Detection of Obfuscated Malware". 978-1-4244-4173-0/09/$25.00 ©2009 IEEE.

[18] F. BUZATU, "Methods for Obfuscating Java Programs". Journal of Mobile, Embedded and Distributed Systems, vol. IV, no. 1, 2012, ISSN 2067 – 4074

[19] P. Sivadasan, P .S. Lal, N. Sivadasan, "JConstHide: A Framework for Java Source Code Constant Hiding". Journal of Information Assurance and Security 4 (2009) 21-29.

[20] I. Welch and R. J. Stroud, "Kava - A Reflective Java based on Bytecode Rewriting",ACM, Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering:

Reflection and Software Engineering, Papers from OORaSE 1999, Pages 155-167 Springer-Verlag London, ISBN:3-540-67761-5,200.

[21] L. Ertaul and S.Venkatesh. “Novel Obfuscation Algorithms for Software Security”. Proceedings of the 2005 International Conference on Software Engineering Research and Practice, SERP’05, June, Las Vegas, June, (2005).

[22] P.Sivadasan and P.Sojan Lal. “ARRAY BASED JAVA SOURCE CODE OBFUSCATION USING CLASSES WITH RESTRUCTURED ARRAYS”, CoRR (2008).

[23] H.Chen, L.Yuan, B.Huang, P.Yew. “Control Flow Obfuscation with Information Flow Tracking”, MICRO 42 Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture,Pages 391-400 , ACM New York, NY, USA (2009).

[24] S.Schrittwieser and S.Katzenbeisser. “Code Obfuscation against Static and Dynamic Reverse Engineering”, IH'11 Proceedings of the 13th international conference on Information hiding,Pages 270-284 ,Springer-Verlag Berlin, Heidelberg (2011).