Abstract—Software obfuscation or obscuring a software is an approach to defeat the practice of reverse engineering a software for using its functionality illegally in the development of another software. Java applications are more amenable to reverse engineering and re-engineering attacks through methods such as decompilation because Java class files store the program in a semi complied form called 'byte' codes. The existing obfuscation systems obfuscate the Java class files. Obfuscated source code produce obfuscated byte codes and hence two level obfuscation (source code and byte code level) of the program makes it more resilient to reverse engineering attacks. But source code obfuscation is much more difficult due to richer set of programming constructs and the scope of the different variables used in the program and only very little progress has been made on this front. We in this paper are proposing a framework named ‘JConstHide’ for hiding constants, especially integers in the java source codes, to defeat reverse engineering through decompilation. To the best of our knowledge, no data hiding software are available for java source code constant hiding.

Index Terms—Reverse Engineering, Constant Hiding, JConstHide, Source Code Obfuscation

1. Introduction

The java based web applications gained popularity because of its Architecture Neutral Distribution Format (ANDF)[12]. During compilation, the Java source code is translated to java class files that contain Java Virtual Machine (JVM) code called the ‘byte code’, retaining most or all information present in the original source code. This is because the translation to real machine instruction happens in the browser of the user’s machine by JIT (Just-In-Time Compiler). Also, Java programs are small in size because of the vast functionalities provided by the Java standard libraries. Decompilation is the process of generating source codes from machine codes or intermediate byte codes. Though decompilation is in general hard for most programming languages, the semi compiled nature of Java class files makes it more amenable to reverse engineering [2] and re-engineering attacks through decompilation. Reverse engineering can be defined as the process of analyzing a subject system to 1) identify the system’s components and their interrelationships, and, 2) create representations of the system at higher levels of abstraction. Reverse engineering involves the extraction of design elements from an existing system, but it does not involve modifying the target system or generating new systems. Reengineering is the modification of a software system that takes place after it has been reverse engineered, generally to add new functionality, or to correct errors. This makes it easier for the competitors to extract the proprietary algorithms and data structures from Java applications in order to incorporate them into their own programs in order to cut down their development time and cost. Such cases of intellectual property thefts [6, 7, 8] are difficult to detect and pursue legally. Statistics [5] show that four out of every ten software programs is pirated worldwide and over the years, global software piracy has increased by over 40% and has caused a loss of more than 11 billion USD. Software obfuscation [1,6,15,16,17,18] is a popular approach where the program is transformed into an obfuscated program using an ‘obfuscator’[3] in such a way that the functionality and the input/output behavior is preserved in the obfuscated program whereas it is much more difficult to reverse engineer the obfuscated program. Code obfuscation applies transformations to the code to make their analysis very hard and thus safer from being reverse-engineered. They do not change the functionality of the program though. The obfuscation can be preformed on the source code, the intermediate code or the machine executable code [1, 9]. Data transformation [1, 4] and constant hiding [5] are the two well studied obfuscation techniques. In data transformation, Array transformation in particular is popular. Array splitting [13], array folding and array flattening (Figure 1) are the three well known array transformation methods [1, 6]. Though, obfuscation is a more economical method for preventing reverse engineering, there are ‘deobfuscators’[10,14] available to defeat some of the less sophisticated obfuscation strategies. The popular transformation techniques employed for obfuscation are (i) layout transformation which makes the structure of the transformed program difficult to comprehend (ii) data transformation that obscures the crucial data and data structures (iii) control transformation to obscure the flow of execution. The effectiveness of obfuscation is usually measured in terms of a) the potency that is the degree to which the reader is confused, b) the resilience that is the degree to which the obfuscation attacks are resisted and
finally c) the cost which measures the amount of execution time/space penalty suffered by the program due to obfuscation. Over the years, a number of software protection methods have been proposed. The remote service based methods provide the maximum protection against piracy because the application resides in a remote server and only the results of the computation is returned to the client application without exposing the algorithmic details of the server application. But such methods suffer from limited network bandwidth and latency. Even the approach of running only the crucial software components in a remote server suffer from similar drawbacks [1]. The approach of encrypting [19] the executable is effective only if the entire decryption/execution process takes place in the hardware [1]. Furthermore, there is dramatic difference in the cost of encryption and decryption in any public key encryption system [1]. Watermarking and Tamper Proofing are techniques other than obfuscation for software protection by cutting down piracy [6]. The paper [20] presents details of image watermarking based on mean removed vector quantization.

Array Splitting: One dimensional array A is split into A1 and A2.

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

\[
\begin{array}{cccccc}
A1: & A_0 & A_2 & A_4 & A_6 & A_8
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

\[
\begin{array}{cccccc}
A2: & A_1 & A_3 & A_5 & A_7 & A_9
\end{array}
\]

(1) int A[10];
(2) A[1]=...;

Array Folding: One dimensional array D is folded into a two dimensional array D1.

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

\[
\begin{array}{cccccc}
D1: & D_0 & D_2 & D_4 & D_6 & D_8
\end{array}
\]

(1) int D[10];
(2) D[0]=...;
(3) D[i]=...;
(4) D[i]=...;

Array Flattening: Two dimensional array E is flattened into a one dimensional array E1.

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

\[
\begin{array}{cccccc}
E1: & E_0 & E_1 & E_2 & E_3 & E_4
\end{array}
\]

(1) int E[3][3];
(2) E[i][j]=...;

Figure 1. The array restructuring techniques – array splitting, array folding and array flattening.

<table>
<thead>
<tr>
<th>N</th>
<th>Y</th>
<th>Y mod n</th>
<th>Depth k</th>
<th>F(Y,k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x3+5</td>
<td>5x2+7</td>
<td>7 mod5</td>
<td>0</td>
<td>F(7mod5,0)</td>
</tr>
<tr>
<td>5x6+11</td>
<td>11x7+18</td>
<td>18mod11</td>
<td>1</td>
<td>F(18mod11,1)</td>
</tr>
<tr>
<td>11x12+23</td>
<td>23x16+41</td>
<td>41 mod23</td>
<td>2</td>
<td>F(41mod23,2)</td>
</tr>
<tr>
<td>2x3+47</td>
<td>47x4+88</td>
<td>88 mod 47</td>
<td>3</td>
<td>F(88mod47,3)</td>
</tr>
<tr>
<td>47x48+95</td>
<td>95x88+183</td>
<td>183mod95</td>
<td>4</td>
<td>F(183mod95,4)</td>
</tr>
<tr>
<td>95x96+191</td>
<td>191x183+374</td>
<td>374mod191</td>
<td>5</td>
<td>F(374mod191,5)</td>
</tr>
<tr>
<td>191x192 =383</td>
<td>383x374+757</td>
<td>757mod383</td>
<td>6</td>
<td>F(757mod383,6)</td>
</tr>
<tr>
<td>383x384 =767</td>
<td>767x757+1524</td>
<td>1524mod767</td>
<td>7</td>
<td>F(1524mod767,7)</td>
</tr>
<tr>
<td>767x768 =1535</td>
<td>1535x1524+3059</td>
<td>3059mod1535</td>
<td>8</td>
<td>F(3059mod1535,8)</td>
</tr>
<tr>
<td>1535x1536 =3072</td>
<td>3072x3059+6130</td>
<td>6130mod3072</td>
<td>9</td>
<td>F(6130mod3072,9)</td>
</tr>
<tr>
<td>3072x3072 =6144</td>
<td>6144x6130+12273</td>
<td>12273mod3072</td>
<td>10</td>
<td>F(12273mod3072,10)</td>
</tr>
<tr>
<td>6144x6144 =12287</td>
<td>12287x12273+24560</td>
<td>24560mod12287</td>
<td>11</td>
<td>F(24560mod12287,11)</td>
</tr>
<tr>
<td>12287x12288+24575</td>
<td>24575x24560+49135</td>
<td>49135mod24575</td>
<td>12</td>
<td>F(49135mod24575,12)</td>
</tr>
<tr>
<td>24575x24576+49131</td>
<td>49131x49135+98286</td>
<td>98286mod49131</td>
<td>13</td>
<td>F(98286mod49131,13)</td>
</tr>
</tbody>
</table>

Table 1. Possible F(Y, k) Calls up to depth 13 for hiding 2

We have proposed a tool for source code obfuscation based on constant hiding technique which is discussed in the following section.

In [5], Ertaul et. al proposed a novel constant hiding techniques using \(y\)-factors. The \(y\)-factors are essentially a predefined increasing sequence of ‘m’ prime numbers \(y_0, y_1, y_2,..., y_m\). The \(y\)-factors can be used to transform a non negative number ‘x’ which is less than \(y_0\) as follows.

Let the function ‘F(A, k)’ be defined as \(F(A, k) = ((...((A \mod y[k]) \mod y[k-1]) \mod y[k-2]) ... \mod y[0])\). Now replace ‘x’ by the expression F(A, k) such that F(A, k) evaluates to ‘x’. Now to hide any large positive constant say ‘c’ in the program, first ‘c’ is replaced with a simple expression of the form 2*d + r where ‘r’ is 0 if ‘c’ is even and ‘r’ is 1 if ‘c’ is odd. Now, the constants 2 and ‘r’ in the resulting expression can be hidden by replacing it with the corresponding F( ) function.

a) The ConstHide Module

To compute the function F( ) as defined in the introduction, we use an array Y[m] of ‘m’ pairs where Y[i] = (Pi, Qi) denote the pair at the i-th index of Y. These pairs have the following properties: a) for any pair Y[i] = (Pi, Qi), Pi + Qi is a prime number and b) if i < j then Pi + Qi < Pj + Qi. That is, sum of the numbers in any pair is a prime number and the pairs are stored in Y array in the increasing order of their sum value. The following sequence of pairs for example can be the contents of the Y-array given by (2, 3), (5, 6), (11, 12), (23, 24), (47, 48), (95, 96), (191, 192), …. (12287, 12288), (24575, 24576).
Function $F(\cdot)$ is computed using the following algorithm.

```java
int F(A, k)
{
    //k is a number between 1 and m which
    //denotes the depth of the obfuscation.
    Y[m]={(P1,Q1),(P2,Q2),..........(Pm,Qm)}
    r = A;
    for (i :k .....1) {r = r mod (Pi + Qi);} 
    return r; }
```

The Table 1 shows the different Data Hiding function calls $F(\cdot)$ for hiding constant $2$, for depths up to $13$. In the table, for depth $'k'$, $N$ is the sum of pair of elements of index $'k'$ of the Static array $Y[]$ and for up to depth $k$, $Y$ is given by $Y=Y+N$ with initial value for $Y=2$.

The JConstHide would for example hide the constant '2' by replacing it with an expression with randomly chosen $F(Y,k)$ (Table1) from the list, say, $F(4123,2)$, $F(374191,5)$, $F(757383,6)$ and so on. Though most compilers simplify the expressions of the form $374\%191$, we still use these expressions to ensure that the source code itself is difficult to comprehend.

The Figure 2 gives a snapshot of the JConstHide framework, containing main modules for Source Code Formatting (SCF) and Constant Hiding (CH). The input to the framework is the java source code, say in Figure 7 and the outputs are constant hidden obfuscated codes (Figures 9, 10). The first iteration for obfuscation is performed by the SCF module and the next set of subsequent iterations is performed by the CH module. The functionality of the SCF module is to rewrite the source code in a different format for effective source code obfuscation and the CH module implements obfuscation by hiding the constants of the formatted code with expressions. During a session of obfuscation, the tool provides only a single iteration option for 'formatting' and multiple iterations option for obfuscation. The formatted file can be chosen repeatedly to implement further levels of obscurity by data hiding.

The SCF module (Figure 3) is a combination of a source code parser and a source code rewriting module. On tool invocation and selection of the code, initially the parser parses each source code statement for symbols ; and End of Line (EOL), and for each statement, the Source code rewriting module rewrites the statement into a new file terminated either with a ‘;’ or an EOL. We call the output of the SCF module as the ‘formatted code’, say in Figure 8.

The newly written source code file is again parsed by the CH module of the tool for specified tokens [ , , = , ( , + , / , * , - , space < > % and thereby replaces the first constant immediately followed by any of the tokens, by $F(a, b)$ call where the function definition is included in the file say 'obfuscate.java'.

We call the CH module output file as an 'obfuscated code' and on its selection, the CH module (Figure 2) further parses the code for $F(a, b)$ call, and the first constant, say 'h' of argument 'a' of $F(\cdot)$ will be transformed into $2*d + r$ with the constant 2 hidden by the $F(\cdot)$ call and for each successive iteration, this process is again repeated by CH module for the first argument of the recursive $F(\cdot)$ calls.
The operation of CH module is displayed in Figure 4. The Constant Parser & position identifier parses each statement for the first constant and gets its position and thereby contacts Random Depth generator for generating a random index ‘k’ with a value from 0 to 13, which is passed to Data Hiding Function call handler module (Figure 6). The handler accesses the function calls from the repository (Figure 5) for index ‘k’ which a pointer to the static array that is storing different strings of data hiding Function calls. Using the position returned from the Constant parser and the F() call returned by the handler, the constant replacement module replaces the constant in the statement by an expression incorporating the data hiding F() call.

```
String[] repository={"F(7%5,0)" ,"F(18%11,1)" ,"F(41%23,2)" ,
"F(88%47,3)" ,"F(183%95,4)" ,"F(374%191,5)" ,"F(757%38,3,6)" ,
"F(1524%767,7)" ,"F(3059%1355,8)" ,"F(6130%3071,9)" ,
"F(12273%6143,10)" ,"F(24560%12287,11)" ,"F(49135%24575,12)" ,
"F(98286%49151,13)"};
```

**Figure 5.** Repository of strings storing F() calls

The detailed operations of the tool are discussed by considering the following snippet of source code ‘leapyears.java’:

```
class leapyears
{
    public static void main(String[] args)
    {
        int i=2006;int n;for (n=1990; n<=i ; n++)
        {
            l=n%4;if (l==0) {
                System.out.println("leap year: "+n);
            }
        }
    }
}
```

**Figure 7.** Java code to find Leap years between 1990 and 2006

The output after the first iteration by SCF module of the framework is given by

```
public class leapyears_mod123 extends obfuscate
{
    public static void main(String[] args)
    {
        int i=(1003*F(12273%6143,10));
        int n;
        for (n=(995*F(757%383,6)); n<i ;
            n++)
        {
            l=n%2*F(49135%24575,12));
        if (l==0*F(374%191,5))
            System.out.println("leap year: "+n);
    }
}
```

**Figure 8.** Output of SCF module

The output after the first iteration of CH module is given by

```
public class leapyears_mod123123 extends obfuscate
{
    public static void main(String[] args)
    {
        int i=(1003*F(12273%6143,10));
        int n;
        for (n=(995*F(757%383,6)); n<i ;
            n++)
        {
            l=n%2*F(49135%24575,12));
        if (l==0*F(374%191,5))
            System.out.println("leap year: "+n);
    }
}
```

**Figure 9.** Output after the first iteration of CH module

The output of the CH module, after the second iteration is given by

```
public class leapyears_mod123123123  extends obfuscate
{
    public static void main(String[] args)
    {
        int i=(1003*F(12273%6143,10));
        int n;
        for (n=(995*F(757%383,6)); n<i ;
            n++)
        {
            l=n%2*F(49135%24575,12));
        if (l==0*F(374%191,5))
            System.out.println("leap year: "+n);
    }
}
```

**Figure 10.** Output of CH module after second iteration

The CH module was modified, especially removing the Data Hiding Function Call handler module and the Static Array Repository. Instead of the handler the Function Response module (Figures 11, 12) was incorporated, which computes the F() call of string type from a random depth k and returns it to the constant replacement module.
public static String FunctionResponse(int k) {
    int n=2; int y=2; String F= " ";
    for (int depth=0; depth<k; depth++) {
        n=n+(n+1);
        y=y+n;
    }
    F=y+%"%"+n;
    return(F);
}

Figure 11. Function Response Module

Figure 12. Modified CH module with a Data Hiding Function Response module

Hence, we can see that the various obfuscated codes results in recursive function calls leading to a possibility of more execution time. To analyse the execution time of the obfuscated codes, a program ,say, 'search_random.java' and its obfuscated versions are analysed. The algorithm of the code is as follows,

Initialize array 'A' of size 100000, to 0
Read n
for (i = 0 .... n-1)
    Generate a random number say 'num'<n

For 100000 elements, the execution time analysis (Figure 13) of the above code is performed on a system with Intel Core Duo processor, 1.66GHz, with 1GB of RAM and with Windows XP Professional operating system. Let C represent the original code and C1 represent the newly formatted non obfuscated version of the source code. Let C2, C3, C4, C5, C6, C7, C8, C9 represent the different successive obfuscated versions of C, obfuscated by data hiding.

Figure 13. Execution time analysis of a code and its obfuscated versions

The plot clearly reveals that the obfuscated codes are not having much deviation in execution times as with that of the original code for lower levels of iterations. As iteration progresses, it results in variations without significant addition of cost in execution.

The file sizes of the obfuscated codes are analyzed for the memory cost and the plot (Figure 14) shows that higher iterations of obfuscation causes linear growth in file size, not resulting significant increase in file size.

Figure 14. File Size Analysis

Reverse engineering effort depends on the effort for comprehending entire code statements and is proportional to the number of statements. Hence, the repeated operation on the obfuscated code by the tool, adds more reverse engineering effort to the statements by generating recursive F() calls.

Due to the lack of commercial de-obfuscators in the market, this analysis of the obfuscated codes is solely based on its decompiled codes. The decompiled codes for the above obfuscated codes have been analyzed using FrontEnd Plus v1.04 decompiler and the decompiled code for the various iterated versions also contains the same number of recursive
Let 'S' be the minimum number of statements of a F() call contributed by a recursive F() call in the final obfuscated version, then the reverse engineering effort 'RE', and 'N' be the number of successive iterations for its execution. Hence, the tool cuts down reverse engineering effort. Hence, the best obfuscated code suggested is code C_i where E_1-E=E_p and F_1-F=F_p.

Another alternate provision added to the tool is to implement obfuscation by putting into operation a new design for CH module. The new design replaces a constant by F() call in the first iteration and for the second iteration it replaces F() call by an expression. For the next iteration, the first constant of the statement is again replaced by F() and for the subsequent iteration F() is replaced by an expression. This process continues for successive iterations. For this implementation we are considering 2 repositories named 'repository1' and 'repository2'. The repositories are shown below.

String[]repository1={"7","18","41","88","183","374","757","1524","3059","6130","12273","24560","49135","98286"};

String[]repository2={"5","11","23","47","95","191","383","767","1535","3071","6143","12287","24575","49151"};

Figure 15. Reverse Engineering Effort plot analysis

The Table2 also shows that only statements involving constants are obfuscated. Therefore, from plot in Figure 15 we infer that the reverse engineering effort grows linearly and considerably for codes, by adding too many statements for high level of iterations, but not causing too much cost for its execution. Hence, the tool cuts down reverse engineering on the codes involving more constants to an extent by adding more reverse engineering effort and the tool is found to be ineffective on codes without considerable constants.

Hence, to choose the obfuscated version for a code it is better to choose the iterated version of the code with the permitted variation levels for file size and execution time.

Table 2. Analysis of decompiled codes

<table>
<thead>
<tr>
<th>Decompiled Codes</th>
<th>Number of Statements</th>
<th>No. of Obfuscated Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrimeNumbers.java</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>Leapyears.java</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>search_random.java</td>
<td>26</td>
<td>13</td>
</tr>
</tbody>
</table>

Figure 16. Two repositories for alternate design of CH module

This proposal also uses the functionalities of modules SCF and CH with the constant replacement by F() (Figure 12) and Figure 17 gives details of additional actions of CH module performing replacement of F() call by an expression.

The two repositories (Figure 16) are used for computing the expression. The main modules of the tool are the 'F() parser & position identifier', 'expression compute' and the 'F() replacement' modules. The CH module operations are performed on the output of the SCF module.

The 'F() Parser' parses for the F() call in the output of SCF module and marks its position. The position is then passed to the 'F() replacement' module. Then, the replacement module, by passing the F() call of String type, contacts the 'compute module' which computes expressions. The Expression Compute module (Figure 18) parses F() and isolates the depth to compute the expression which is returned to the F() replacement module. The expression is framed with details from repository1 and repository2. With the 'position' returned from the F() parser module and the 'Expression' returned from the Compute module, the F() replacement module replaces F() call with an expression.
Let us discuss the operations of the new design by considering the codes of Figure 7. The first iteration of CH module would result in code shown in Figure 9 and we can see that the codes in Figure 9 contains F() calls and the CH module replaces F() calls by expressions for the second iteration, shown in Figure 19 and the next immediate iteration results in code in Figure 20.

The Expression Compute module for depth denoted by 'count' is given by,

```java
public static String ExpressionCompute(int count)
{
    String s = " ";
    s = "(" + repository1[count] + "%" + repository2[count] + ")";
    for (int i = count - 1; i > 0; i--)
        s = "(" + s + "%" + repository2[i] + ")";
    if (count > 0) s = "(" + s + "%" + repository2[0] + "+")";
    return(s);
}
```

This mode of operation also tends to make the reverse engineering process hard. We have tried to incorporate JConstHide component in our latest framework named JDATATRANS [11] for obfuscating source code array usages, by data hiding. The main modules of the JDATATRANS framework is a class repository named CoBS (Classes for Obfuscating Source codes) implementing the restructured arrays (Figure 1) and the JConstHide component, for obfuscating the Java program by hiding the constants. We also call the JConstHide component as an 'Obfuscator'. Additional capabilities have been added to the Obfuscator of the JDATATRANS Framework for handling non constant index variable.

The methods `setArray()`, `getArray()` and `lengthArray()` are used to store an element at a given index, retrieve the element stored at the given index and to get the length of the array respectively. Implementation for these three methods is mandatory for all the array classes in CoBS repository.
The JDATATRANS-Obfuscator scans the program and identifies those statements, called candidate statements, in the program where either a CoBS based array is declared or an instance of the array is accessed using any of its public methods. To do this, the obfuscator first pre-processes the input program. In the preprocessing phase, the sentence boundaries are detected, the comments are stripped off and the tokens in the sentence are identified. Now in each candidate statement, the obfuscator identifies the constants used, including the array indices. If there are no constants used then the array index variable is multiplied by ’1’ (which clearly does not alter its value) and the constant ’1’ is hidden using ConstHide. If only the lengthArray( ) function is invoked, then it is similarly replaced by lengthArray( ) * 1, where the ’1’ is later hidden by ConstHide. For non candidate statements, the first integer constant is hidden, avoiding alphanumeric strings. We remark that the resulting program can be re-obfuscated again by the obfuscator to obtain further levels of obfuscation.

To illustrate the method, consider the following snippet from the program 'myprog.java' which the programmer wishes to obfuscate. The programmer decides to obfuscate the array 'ar' using SplitArray.

```java
SplitArray<Integer>ar=newSplitArray<Integer>
(100000);
ar.setArray(i,(3*i + 1000) % n);
y = ar.getArray(i);
```

After obfuscation, it is transformed into the following code.

```java
SplitArray<Integer>ar=newSplitArray<Integer>
(50000*F(49135%24575,12));
ar.setArray(i*(4*F((F(49135%24575,12)*1529+F(33%21,2))%1535,8)-
(F(49135%24575,12)*F(35%27,2)+F(33%21,2))),3*i + 1000) % n);
y=ar.getArray(i*(F((F(49135%24575,12)*17+F(33
%21,2))%27,2)-F(12273%6143,10)));
```

But this creates an additional overhead in terms of the execution time as the number of F( ) expressions that needs to be computed in runtime increases with each iteration of the obfuscation. The following plot (Figure 21) shows the tool performance where the analysis is performed on a sample code 'myprog.java' denoted by A and its obfuscated version using SplitArray, FoldedArray and FlattenedArray denoted by B, C and D respectively. The algorithm section of 'myprog.java’ is as follows.

Set ‘n’ elements to an array of size 100000
Print n array elements

Let P2, P3, P4, P5 correspond to successive obfuscated versions of each of the codes A, B, C, D. The graph in Figure 21 shows the execution time analysis between the various obfuscated versions. For 100000 elements, the analysis shows no significant deviation between the original and highly obfuscated codes and the execution time remains almost constant for further levels of obfuscation.

```
Execution Time Analysis

<table>
<thead>
<tr>
<th>Execution Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
</tr>
<tr>
<td>12.5</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>13.5</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>14.5</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>15.5</td>
</tr>
</tbody>
</table>

P P1 P2 P3 P4 P5
```

Source codes and Obfuscated codes

![Execution Time Analysis](image)

**Figure 21.** Execution Time Analysis

We conclude that, further code obfuscation would add more effort in reverse engineering, without too much cost on execution time and storage.
References


Author Biographies

Praveen Sivadasan is an Assistant Professor at Federal Institute of Science and Technology (FISAT), Angamaly, Kerala, India and also a Research Scholar in Information Security at Mahatma Gandhi University, Kerala, India. He earned his Masters Degrees in Mathematics in 1996 and Computer Applications in 1999 from Mahatma Gandhi University. He received his Degree in Master of Philosophy from Manonmaniam Sundaranar University, Tirunelveli, Tamil Nadu, India, in 2003.

Dr. P Sojan Lal is an Adjunct Professor in Computer Science at Rajagiri School of Engineering & Technology, Cochin, Kerala, India and also a Research Supervisor with School of Computer Science, Mahatma Gandhi University, Kottayam, Kerala, India. He is also a Strategic Business Planner and Enterprise Asset Management (EAM) functional consultant, particularly for international oil and gas industries. He received his B.E. degree in Mechanical Engineering from Bangalore University, Karnataka, India in 1985, M. Tech degree in Computer Science from National Institute of Technology (NIT), Warangal, Andra Pradesh, India in 1993 and Ph.D degree in Computer Science from Cochin University of Science and Technology, Cochin, Kerala, India in 2002. His research interests include image processing and security systems. He is a Fellow of The Institution of Engineers (India) since 2004.

Dr. Naveen Sivadasan is a Scientist at the TCS Innovations Lab, Tata Consultancy Services, Hyderabad. He received his B. Tech degree in Computer Science & Engineering from NIT Calicut in 1996, M. Tech degree in Computer Science from Indian Institute of Science, Bangalore in 1999 and Ph.D degree in Computer Science from Max-Planck Institute for Computer Science, Germany in 2004. His research interests include Applied & Theoretical algorithms, Graph theory and Information Extraction & Retrieval.