Proceedings of the Eighth IASTED International Conference
SOFTWARE ENGINEERING AND APPLICATIONS
November 9-11, 2004, Cambridge, MA, USA

JHide – A Tool Kit for Code Obfuscation

Levent Ertaul Suma Venkatesh
Department of Mathematics & Computer Science
25800 Carlos Bee Blvd., Hayward, CA, 94542
lertaul@csuhayward.edu sumav_99@yahoo.com

ABSTRACT
According to Business Software Alliance statistics, four out of every ten software programs is pirated in software business, world wide. Global piracy rate has increased 40% over the past years and nearly $11 billion is lost. This is definitely a clear threat for software producers and thus to global economy. Over the years, several software protection techniques have been developed, code obfuscation is one of them and it is very promising. Code obfuscation is a form of software protection against unauthorized reverse-engineering. In this paper we discuss software protection techniques in general and provide a broad overview of known obfuscation algorithms. We also address the issues related to implementation of obfuscation algorithms. Finally we propose JHide, an obfuscation tool kit for protection of Java code. We conclude our paper identifying the need for reviewing the performance of the algorithms as the future scope of our work.

KEYWORDS
Obfuscation, software protection and software security.

1. Introduction

Fast developments in multimedia and internet technologies have created the need for researching in the areas of securing data. Every company has an intellectual property to protect which often includes algorithms built right into the software that is sold to customers. The secrecy of such software is an edge to beat their competition in the market, so it is not surprising that the approach taken for their protection makes a great deal of difference [1], [2],[3],[4], [5].

Traditionally techniques for securing data resided in the firewalls and gateways of a network or on the operating system of the host. A new idea is to put these defensive mechanisms inside the application software. Vendors of this software distribute them as mobile code in architectural independent formats [1], [2], [4], [5].

Recent statistics [6] show that four out of every ten software programs is pirated worldwide. This is definitely a threat to clean players and thus the global economy. There are two common practices of protecting an intellectual property of a software producer - Legal and Technical methods. Legal methods include getting copyrights on the software and signing legal contracts against creating duplicates. Technical methods include: [1],[4],[7], [8], [9], [10], [11]

- ***Code Authentication***: Developers place license files and identity keys in the software.
- ***Server side execution***: Avoids sending final code to the user.
- ***Program Encoding***: They detect the pirate's attempts to tamper software and protect against those attempts.
- ***Code Obfuscation***: Applying transformations to the code make their analysis very hard and thus safer from being reverse engineered. They do not change the functionality of the program though.

Obfuscation is a new area of research in the field of software protection and gaining more attention in recent years [1], [4],[9]. Although the history of first traits of obfuscation techniques dates back to 1990 [12], their impact got higher as Java technologies dominated the software development world. Java is designed to be compiled into a platform independent byte code format, which means decompilation is easier than with traditional native codes. As a result, the java code can always be reverse-engineered to extract proprietary algorithms from compiled java programs. Obfuscating them make the program more difficult to analyze and uneconomical to reverse-engineer. Software protection tools like DashO, Dot Obfuscator [13], JMangle [14], JObfuscator [15], and Sand Mark [7] are all designed based on the principal theories of code obfuscation techniques.

Based on our research in the software security field and the capabilities of the existing players we strongly believe in the potential of "code obfuscation" techniques as a major software protection tool in the near future and hence our attempt in creating such a tool kit called JHide.

JHide is a tool kit designed for obfuscating software programs written in java. It aids software security researchers to understand how various known obfuscation algorithms work. A user can select a program and apply a desired obfuscation transformation on it so that the code is safe from being reverse-engineered. JHide can be used to obfuscate programs against "malicious host attacks" [16] and make it difficult for an attacker to locate sensitive information. Next we discuss software

protection techniques in general. Followed by implementation issues associated with obfuscation algorithms. Finally we present the features offered by JHide and its overall design.

2. Software Protection Techniques

Generally software code is mobile and distributed across untrusted networks. Their protection must be incorporated into the software and be hardware independent. The main functions of any software protection technique can be listed as detection of pirate attempts to tamper or misuse software, protection against such attempts and alteration of software to ensure that it functionality degrades in an undetectable manner if protection fails. A summary of the most common techniques are discussed here

- ***Protection by Server-Side Execution:*** The application that requires protection is placed on a server and its services are provided to users over a remote connection.
- ***Hardware based solutions:*** The trusted hardware components are placed on the user's machines to identify applications.
- ***Protection by Encryption:*** Program instructions are decrypted by a co-processor (crypto chip) before they are executed by the main processor.
- ***Protection through Signed Native Code:*** The software code to be distributed is first compiled into Java byte codes and stored on a server. Users identify their architecture/operating system combination to the server, which then provides the appropriate native code version of the application.
- ***Tamper Proofing:*** It is a methodology built inside the operating system. This detects any modification to the software code and disables such an action.
- ***Software Aging:*** It is a technique that a periodic update of the software, which is compatible with older versions, is sent out to the customers.
- ***Watermarking:*** It is a technique used to embed ownership marks in software.
- ***Code Obfuscation:*** The final idea is to hide the code. In this technique the application is transformed so that it's functionally identical to the original but it is much more difficult to understand. This technique preserves platform independence.

Advantages and disadvantages of the above techniques can be found in [1] ,[4] ,[7], [8], [9], [10],[11] .

An obfuscated application does not suffer from delays due to network limitations. It also does not require the hardware needed for secure encryption and decryption of code. Java application will not perform illegal actions such as erasing a user's files. So unlike native codes, there is no need to digitally sign a Java application to verify that it is a safe code from a trusted source. There may be situations where the other techniques are better, such as when the source code contains extremely valuable trade secrets, or when run-time performance is critical the transformations employed by an obfuscator may not provide a high level of protection and performance. It has been evaluated that the server-side execution model suffers from network bandwidth and latency limitations. Encryption fails to be effective without specialized hardware and use of signed native code places restrictions on portability which is unacceptable for a language like Java, which is independent of hardware platforms. The use of specialized hardware also incurs higher cost to the end-user. So we must look at hardware-independent means of protecting client executed applications. Java is designed to be compiled into a platform independent byte code format which means that decompilation is easier than with traditional native codes. As a result, the java code can always be reverse-engineered to extract proprietary algorithms from compiled java programs. All the above drawbacks in other protection techniques make code obfuscation a stronger tool for securing programs written in java. Although obfuscation attempts to make decompilation a harder task, given enough time and effort, it is possible to retrieve important algorithms and data structures from such an obfuscated code. The aim here is to increase the time and effort required so that it is economically infeasible for a company to reverse-engineer a rival's application [8], [10], [11].

In the next section we focus on the issues involved in implementing JHide tool kit.

3. JHide Tool Kit

Deciding on whether to analyze the java program at the virtual machine level (byte code) or as a high level language (source code) was a major issue while choosing our obfuscation platform for implementing the transformations. Breaking the anatomy of a java program in its source code level into all the language paradigms requires a great deal of string parsing and program analysis. Determining a variable and its types, identifying the methods and differentiating between a method call and a declaration, identifying loop boundaries, loop conditions and forming logic dynamically suitable to the given code to break the instructions and loops into separate methods and control flow statements, adding extra intelligence to the program to analyze the task of modifying the instructions, splitting classes and merging them are many of the tedious and complex problems associated in analyzing a program to be obfuscated [17]. These tasks need extensive parsing and the use of external parsing and scripting tools like Another Tool for Language Recognition (ANTLR) [18], using this we can write flexible grammars that can support analysis of the java source code. ANTLR also lets us define the rules that the lexer should use to tokenize a stream of characters and the rules the parser should use to interpret a stream of tokens. It can then generate a lexer and parsers which can

be used to interpret any input program written in any language recognized by the ANTLR engine and then translate them to other languages. Since Java classes are compiled into portable binary class files (byte code), it is the most convenient and platform-independent way to implement obfuscation algorithms not by writing a new compiler or parser but by transforming the byte code. These transformations can either be performed after compile-time, or at load-time. Developing such a specialized byte code manipulation tool is tedious and also restricted in the range of their re-usability. Hence to deal with the necessary byte code transformations, we are using the byte code editing library (BCEL) [19], which is a Jakarta common's API [20]. It is a toolkit written in java for the static analysis and dynamic transformation of Java class files. It enables the developers to implement the desired features on a high level of abstraction without handling all the internal details of the Java class file format and thus re-inventing the wheel. The analysis of Java class performed by BCEL is conservative in nature. For example, it is not possible to have a branch to a non-existent instruction number in the byte code. Pointer arithmetic is not permitted in Java, unlike in C and hence there are no instructions to convert between object references to other data types such as integers. This area of obfuscation is not open to Java programs and should be considered as a limitation for implementation [12]. The java beans specification allows a user to create an application out of components downloaded from different vendors. In this scenario, it is not possible to globally change identifier names in the application. This is because some of the identifiers are used to interface to a vendor's component and cannot be changed. Thus only the identifiers local to the user's application are scrambled. Due to limitations in the class file format, the amount of extra code that can be added by an obfuscation transformation is restricted. For example, the method in-liner transformation will replace an inline method call with the actual code of the method being called. This causes a method to exceed the limit on code size. To overcome this problem, excess use of method inliner transformations in a class is avoided.

Keeping all the limitations in mind we have created the obfuscation tool kit JHide which uses BCEL API and implements 30 known obfuscation algorithms. The changes made to the class files are shown using DJ Java [21] decompiler in the view interface of JHide. General system overview of JHide, JHide structure and Obfuscation algorithms in JHide is given in next subsections

3.1. JHide System Overview

The tool is written completely in java and runs on windows operating system. It uses J2SDK 1.4 package and has a graphical user interface (GUI) written in Java Swing. JHide uses BCEL-5.1 API to manipulate java class files and DJ Java decompiler 3.5 to view the changes on the java class after it is subjected to obfuscation transformation. The architecture of JHide is broadly composed of two main parts:

- ***Code obfuscation interface***: Includes 30 known obfuscation algorithms [7], [22] and operations carried out by the user to obfuscate a java source code. The user selects java source code through a GUI and picks one of the obfuscation algorithms from the drop down list
- ***Code View interface***: This interface allows a user to view the code subjected to obfuscation, before and after obfuscation. The user can select the option "source code view "and see the changes in the java source code using DJ decompiler as shown in Fig 3. The user can also choose the option "byte code view" and see the byte code changes displayed as HTML files as shown in Fig 4.

The user invokes JHide on command prompt by using "JHide.bat". A Swing GUI pops up; the user selects a java class file to be transformed and then chooses one among the 30 obfuscation algorithms listed in the drop down menu in the GUI as shown in Fig.1 and Fig.2.

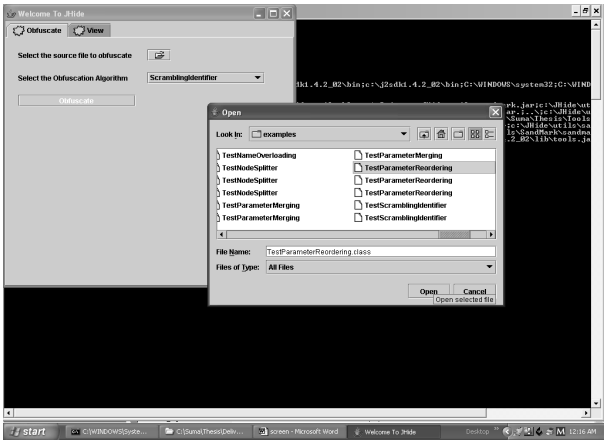

Fig. 1 User selects the source code to obfuscate

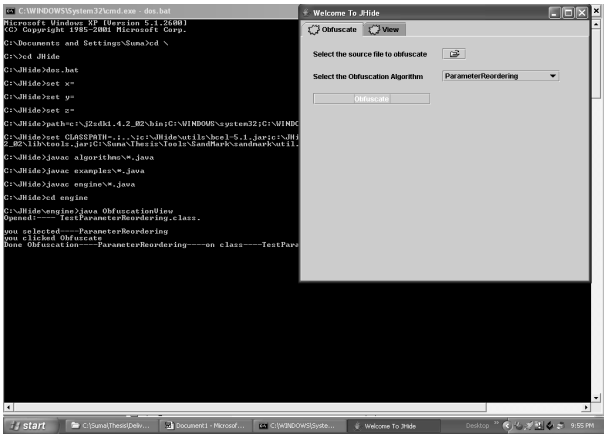

Fig. 2 User selects the obfuscation algorithm

The obfuscation engine applies the selected transformation algorithm on the source and dumps the changed class file into an obfuscated class's directory and retains the original class in the examples directory. Once the obfuscation is done, user can see the "*completed*" message on the console. Now he/she can browse to the view section and select either source code view or byte code view. If the source code view is selected (Fig 3), two instances of DJ decompiler windows are opened to show the changes in the selected java class before and after applying obfuscation transformation.

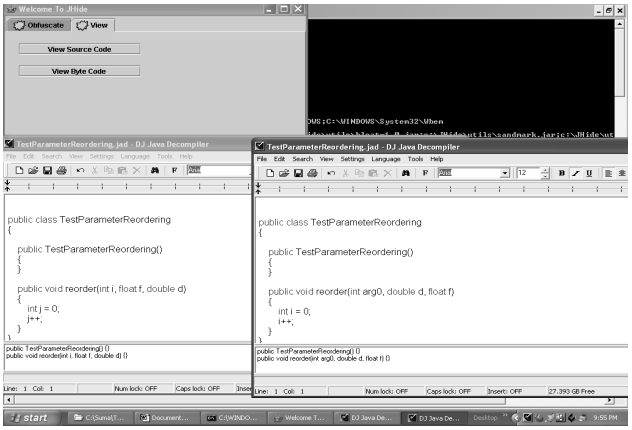

Fig. 3 JHide Source Code View

If the user selects byte code view (Fig 4) two instances of IE (Internet Explorer) is opened to show the changes in the instruction set. BCEL API supports convertToHTML method on a class which will create HTML pages of instruction sets for the class. We have used this feature to show byte code view

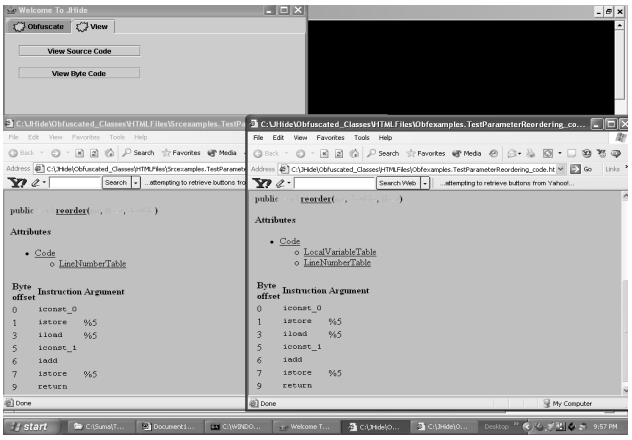

Fig. 4 JHide Byte Code View

3.2. JHide Structure

The selected java class file is parsed by BCEL API into program structures as shown in Table 1.

```
String classname = "TestParameterReordering.class";
JavaClass clazz = Repository.lookupClass ("examples/" +
classname);
ClassGen cgen = new ClassGen (clazz);
ConstantPoolGen cpg = cgen.getConstantPool ();
```

Table 1 Java Class lookup

The program segment in Table 1 looks up for the java class files in the specified directory. If it is not found BCEL Exception is thrown. Once the java class "*TestParameterReordering.class*" is found, a class generator handle (*cgen*) is created on it, using *cgen* the constant pool (*cpg*) is created and the parameters in the methods of the class can be parsed as shown in Table 2.

```
For (int methodNum = 0; methodNum < numMethods;
     methodNum++) {
Method meth = cgen.getMethodAt (methodNum);
MethodGen methGen = new MethodGen (meth,
className, cpg);
String [] oldArgNames = methGen.getArgumentNames ();
Type [] oldArgTypes  = methGen.getArgumentTypes ();
int numArgs = oldArgNames.length;
ArrayList argNums = new ArrayList ();
for (int ii = 0; ii < numArgs; ii++)
    argNums.add (new Integer (ii));
Collections. shuffle (argNums);
String [] newArgNames = new String [numArgs];
Type [] newArgTypes = new Type [numArgs];
int [] indeces = new int [numArgs];
Int [] newIndeces = new int[numArgs];
for (int old Index = 0; old Index < numArgs; old Index++) {
    Integer inewIndex = (Integer) argNums.get (old Index);
    int newIndex = inewIndex.intValue ();
    indices [newIndex] = oldIndex;
    newIndeces [oldIndex] = newIndex;
    newArgNames [newIndex] = oldArgNames [oldIndex];
    newArgTypes [newIndex] = oldArgTypes [oldIndex];}
methGen.setArgumentNames (newArgNames);
methGen.setArgumentTypes (newArgTypes);
Method newMeth = methGen.getMethod ();
cgen.replaceMethod (meth, newMeth);
```

Table 2 Parsing a Java Class

Here total number of methods in a class is computed (*numMethods*) and a method Generator (*methGen*) is created for each method in the class. Using *methodGen* the argument names and types is obtained. New indices for these arguments are created and they are mapped to these new indices using *methGen*. The references to this new order of arguments are updated in *cgen* and *cpg* using the respective generators. The same process is done for each method in the class. This way BCEL reads the entire program constructs in a class file and converts them to byte codes .

3.3 Obfuscation Algorithms in JHide

An obfuscator is a program used to transform program code. The output of an obfuscator is program code that is more difficult to understand but is functionally equivalent to the original. Obfuscation transformations are classified into the following main groups [23]: Layout, Control, Data, Preventive, Splitting, Merging, Reordering, and

Miscellaneous like Method Inliner, Method2RMadness, and Name Overloading transformations. On the basis of these classifications the algorithms supported by JHide can be explained as follows:

Layout Transformations*:* They modify a program's formatting, naming and meta-information they are of two types: ***Format removal*** - removes source code formatting like tabulation and carriage returns. ***Scrambling Identifier Names*** - changes the identifier names in a program to less meaningful ones but an attacker can still infer the meanings based on the context of the program where the variables are used.

Control Transformations: Hide the flow of Control in a program using opaque predicates. Types of Control transformations are ***Control computations***, ***aggregations*** and ***ordering***. ***Control Computation*** - affects the control flow in a program. One of the computations is ***smoke and mirrors***. This technique inserts irrelevant code (dead code) into loops, methods, fields, as method arguments which will never be executed. Each one is implemented as a separate obfuscation technique in JHide. Other computation is called ***High-level language breaking***. This introduces features at the object code level that have no direct source code equivalent and also breaks the code into two halves which perform the same function but obfuscate the second one to such an extent that it is hard to reverse- engineer. The other technique is called ***Alter Control Flow.*** This technique alters the flow of control by finding a sequence of low-level instructions that are equivalent to the construct and then adds redundant termination conditions to the loop. Last technique is known as ***Modify If Else***. This changes the flow conditions in if-else block as shown in Table 3

Before	After

int i = 1;
if (I <10) {
a = a+ b;
i++;
}
else {
a = a- b;
- - i;
}
	int i =1
	if (I > 10) {
	a = a-b;
	- - I;
	}
	else {
	a = a+b;
	i ++;
	}

Table 3 Modify If Else block

Control Aggregation - changes the way in which program statements are grouped together. ***Cloned Methods*** are the only type that is supported in JHide. Here we manipulate the method signature and make it appear as though a different method is being called as shown in Table 4

Control Ordering - alters the order in which statements are executed. We support three types of control ordering. First one is ***Loop Blocking***. Here nesting is applied to loops which would functionally remain the same as the source. Second type is ***Loop Unrolling*** which replicates the body of the loop one or more times. Third type is ***Loop Fission***. This technique breaks the loop into several loops keeping the iteration space same, as shown in Table 5.

Before	After

class C {
method m (int v) {
v = c * r;
p = v * c;
}
}
	class C1 {
	method m (int v) {
	v = c * r;
	}
	}
	class C2 inherits C1 {
	method m (int v) {
	p = v * c
	}
	}

Table 4 Cloned Methods

Before:	After:

for(i =1; i<= n ;i++)
for (j =1;j<=n;j++)
a[i,j] = b[j,i];
	for(I=1;I<=n;I+=64)
	for(J=1;J<=n;J+=64)
	for(i=I;I<=min(I+63,n),i++)
	for(j=J;j<=min(J+63,n),j++)
	a [i,j] =b[j,i];

Table 5 Loop Fission

Data Transformations*:* They affect the data structures used by a program. Different types of data transformations are ***Data storage*** - affects how data is stored in memory. ***Data encoding***- affect how the stored data is interpreted. ***Data aggregation*** - alters how data is grouped together. ***Data ordering*** - changes how data is ordered.

Preventive Transformations: intended to stop decompilers and deobfuscators from functioning correctly. There are two types of preventive transformations. ***Targeted obfuscation***- these are designed to counter specific analysis tools [24]. ***Inherent obfuscation*** - Here loops are reordered.

Splitting Transformations: They include splitting program constructs. Three different types supported are ***Variable Type Splitting*** - the types of the variables are split into smaller ones. ***Node Splitting*** – Every node in the data structure is broken into two parts with an extra field in the first part linking the two together. ***Class splitting*** – Here a class is split into two as shown in Table 6.

Before	After

Class A {
public int a =0;
public int b =1;
method(int c){
c = a+2;
d = b+3;
}
}
	Class A1 {
	public int a =0;
	method(int c) {
	c = a+2;
	}
	}
	Class A2 {
	public int b =1;
	method(int d) {
	d = b+3;
	}
	}

Table 6 Class Splitting

Merging Transformations: merges all possible program

constructs. Different types of merging are: ***Methods merging*** - methods which are logically dependent are merged. ***Parameters merging*** - method parameters which are logically dependent are merged**.** ***Classes merging*** - classes which are logically dependent are merged as shown in Table 7.

Before	After

Class A {
int method1(int a) {
a = a+1;
return (a);
}
int method2 (int a) {
a = a-1;
return(a);
}
}
	Class A {
	int method (int a) {
	a = a+1;
	a = a-1;
	return (a);
	}
	}

Table 7 Method Merging

Reordering Transformations: the sequence of program segments like ***method parameters***, ***local variables*** and ***constant pool*** are rearranged. JHide implements each of these as separate algorithms.

Miscellaneous Transformations*:* They are grouped into three types. ***Method In liner-*** Here an entire method body is substituted for the method call. ***Method2RMadnes-*** This algorithm hides information hidden in methods by disrupting their signatures, argument orders, and moving or combining them. ***Name Overloading*** - This algorithm obfuscates methods so that as many methods as possible have the same name.

4. Conclusion

JHide provides a good starting point for beginners to understand various obfuscation algorithms and the issues involved during their implementation. Users of JHide can see the changes in their code after and before obfuscation through a simple byte code and source code view interface. JHide applies obfuscation to a single selected java source. JHide provides only a show case of obfuscation algorithms and does not support any interface to measure their efficiency in terms of level of obfuscation achieved (Potency) and the maximum execution time/space that the obfuscated code adds to the application (Cost). We want to implement such an interface in our future work where the user would select the % percentage of obfuscation needed and JHide obfuscation interface would automatically apply all the required algorithms to achieve that % level of obfuscation. JHide can be used to build secure mobile agents. One of the main drawbacks of mobile agents is their safety from being corrupted by malicious hosts they interact over the network. Complex encryption algorithms cannot be applied to them with the limitations of memory and bandwidth over the network. In such situations, mobile agents can be obfuscated using JHide to hide the important functions that execute and make reverse engineering uneconomical to the malicious hosts.

References:

[1]M. R Stytz, J. A Whittaker, Software Protection - Security's Last Stand, *IEEE Security and Privacy* January/February 2003

[2] G. McGraw, Software Security, *IEEE Security &Privacy*, March/April 2004.

[3] C. Cowan, Software Security for open source systems, *IEEE Security & Privacy*, Feb. 2003

[4] M.R Stytz, Considering Defense in Depth for Software Applications, *IEEE Security & Privacy*, Feb. 2004.

[5] J.Whittaker, Why Secure Applications are Difficult to Write, *IEEE Security & Privacy*, April, 2003

[6]Business Software Alliance http://global.bsa.org/usa/press/newsreleases/2002-06-10.1129.phtml?CFID=4661&CFTOKEN=73044918

[7] C. Collberg, G. Myles & A.H. Work, Sand mark – A Tool for Software Protection Research, *IEEE Security & Privacy* July/August 2003

[8] C. Collberg , C. Thomborson, Watermarking, Tamper Proofing and Obfuscation – Tools for Software Protection , *Technical Report* February 2000-03.

[9] P. Tyma, Encryption, hashing, & obfuscation, *ZD Net* April 8 2003

[10] G. Naumovich, N. Memon, Preventing Privacy, Reverse Engineering & Tampering , *Innovative Technology for Computer Professionals* , July 2003

[11] D.Low, Java Control Flow Obfuscation. *Thesis Report University of Auckland* June 3 1998

[12] G.Wrobliwski, General Method of Program Code Obfuscation , *PhD Dissertation, Wroclaw University of Technology, Institute of Engineering Cybernetics,* 2002

[13] DashO and Dot Obfuscator http://www.preemptive.com/

[14] JMangle, The Java Class Mangle http://www.elegant-software.com/software/jmangle/

[15]Jobfuscator http://download.com.com/3000-2417-10205637.html

[16] T.Sander, C. F Tschudin, Protecting Mobile Agents against Malicious Hosts, *Lecture Notes in computer science 1419, Mobile Agent Security* Feb 1998

[17] C. Collberg, P.Clark, Breaking abstractions and structuring data structures, *IEEE Computer Language ICCL'*98

[18] ANTLR –Another Tool for language recognition http://www.antlr.org.

[19]Byte code editing library BCEL, http://jakarta.apache.org/bcel/.

[20] Jakarta Commons API www.apache.org

[21] DJ Java decompiler http://dj.navexpress.com/

[22] C. Collberg, C. Thomborson & D. Low, Taxonomy of Obfuscation Transformations, *Technical Report #148* July 1997

[23] D. Low, Protecting Java Code via Code Obfuscation *ACM Crossroads Student Magazine* Spring 1998

[24] H.P. Vliet, Mocha the Java decompiler, http:// wkweb4.cableinet.co.uk/jinja/mocha.html