JShield: A Java Anti-Reversing Tool

A Thesis

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Deepti Kundu

May 2011

© 2011

Deepti Kundu

ALL RIGHTS RESERVED

The Designated Thesis Committee Approves the Thesis Titled

JShield: A Java Anti-Reversing Tool

by

Deepti Kundu

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2011

Dr. Mark Stamp	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Mr. Ronald Mak	Department of Computer Science

ABSTRACT

JShield : A Java Anti-Reversing Tool

by Deepti Kundu

Java is a platform independent language. Java programs can be executed on any machine, irrespective of its hardware or the operating system, as long as a Java virtual machine for that platform is available. A Java compiler converts the source code into 'bytecode' instead of native binary machine code. This bytecode contains a lot of information from and about the source code, which makes it easy to decompile, and hence, vulnerable to 'reverse engineering attacks'. In addition to the obvious security implications, businesses and the wider software engineering community also risk widespread IP theft - proprietary algorithms, for example, that might be implemented in Java could be easily reverse-engineered and copied.

This thesis addresses the problem of reverse engineering attacks on software written in Java. It analyzes the present defense techniques used to protect software from such attacks, examines their limitations and provides a new tool that implements several anti-reversing techniques.

ACKNOWLEDGEMENT

With great satisfaction, I can call this thesis complete today. The accompanying research and hard work that went with it, however, are just a beginning, as much remains to be explored further in this field.

It gives me immense pleasure to thank the many people who helped and guided me during my research. Special thanks are due to my advisor, Dr. Mark Stamp, who made this work possible. His timely guidance, encouragement, patience, enthusiasm and immense knowledge helped me push further and produce some quality work. I could not have asked for a more knowledgeable person as my advisor and mentor.

I would like to thank members of the committee, Dr. Robert Chun and Dr. Ronald Mak, for their valuable feedback, encouragement and advice. I am grateful to Dr. Sigurd Meldal for his support during my research. Mr. Naveen Roperia deserves a special mention for his guidance all through my work.

I am indebted to my family, my sister Anupam Singh and my brother Rahul Kundu for their emotional support and care. I wish to thank my parents, Dr. Ram Niwas Kundu and Ms. Santosh Kundu, for their unconditional love and endless support. They believed in me when I was in self-doubt. I dedicate this thesis to my beloved parents.

TABLE OF CONTENTS

1.0 INTRODUCTION 1
1.1 Reverse Engineering 1
1.2 Process of Reverse Engineering 2
1.3 Anti-Reverse Engineering 3
1.4 Anti-Reversing Tools 4
1.5 Threat to Java Software 5
1.6 Organization of this Thesis 7
2.0 LITERATURE REVIEW 8
2.1 History 8
2.2 Relation to Reengineering 9
2.3 Decade of Achievements 12
2.3.1 Program Analysis 12
2.3.2 Architecture and Design Recovery 13
2.3.3 Visualization 14
2.4 Future Trend 15
3.0 ANTI-REVERSING TECHNIQUES 16
3.1 Protecting Java Code 16
3.1.1 Compilation Flags 17
3.1.2 Implementing Two Versions of the Application 17
3.1.3 Applying Obfuscation 17
3.1.4 Using Web Services and Server-Side Execution 18
3.1.5 Encryption 18
3.1.6 Digital Rights Management 18
3.1.7 Fingerprinting the Code 19
3.1.8 Selling Source Code 19
3.1.9 Employing Native Methods 19
3.2 Obfuscation Techniques 19
3.2.1 Layout Obfuscation 21

3.2.2 Control Obfuscation 22
3.2.2.1 Computation 22
3.2.2.2 Aggregation 27
3.2.2.3 Ordering 30
3.2.3 Data Obfuscation 30
3.2.3.1 Storage and Encoding 31
3.2.3.2 Aggregation 32
3.2.3.3 Ordering 33
3.3 Some Terminology 33
4.0 EXISTING OBFUSCATORS 36
4.1 Tool Support 36
4.2 Brief Analysis of Existing Tools 37
4.2.1 ProGuard 37
4.2.3 Zelix KlassMaster 42
4.2.4 Semantic Designs Java Obfuscator 44
4.3 Summary 46
5.0 PROPOSED TOOL - JShield 47
5.1 Introduction 47
5.2 JShield Functionality 47
5.4 Design and Implementation 49
5.4.1 Implementation Platform 49
5.4.2 Control Flow 50
5.4.3 Algorithm and Result 53
5.4.3.1 Scramble Identifiers 53
5.4.3.2 Insert Dead or Irrelevant Code 57
5.4.3.3 Extend Loop Condition 59
5.4.3.4 Insert Bogus Class 61
5.4.3.5 Reorder Methods 64
5.4.3.6 Convert Static to Procedural Data 66

5.4.3.7 Add Redundant Operands 67
5.4.4 Result Validation 70
5.4.4.1 Observations 70
5.4.4.2 User Test Statistics 73
6.0 CONCLUSION AND FUTURE WORK 79
6.1 Conclusion 79
6.2 Future Work 80
References 83
APPENDIX A: ANTLR Parser 90
APPENDIX B: Terminology 93
APPENDIX C: JShield Example 95

LIST OF FIGURES

Figure 1 Process of Reverse Engineering 3

Figure 2 Machine Codes 6

Figure 3 Bytecodes 6

Figure 4 Objectives of Reverse Engineering 9

Figure 5 Reengineering 10

Figure 6 Architecture Reengineering 11

Figure 7 Relationship of terms 12

Figure 8 Ways of Protecting Java Code 16

Figure 9 Obfuscation – A Classification 21

Figure 10 ProGuard 38

Figure 11 Jshrink 41

Figure 12 Zelix KlassMaster 43

Figure 13 Semantic Designs Java Obfuscators – Output 45

Figure 14 JShield 48

Figure 15 JShield: Control Flow 52

Figure 16 Scramble Variable Names: Before 55

Figure 17 Scramble Variable Names: After 55

Figure 18 Scramble Method Names: Before 56

Figure 19 Scramble Method Names: After 57

Figure 20 Inserting Dead Code: Before 58

Figure 21 Inserting Dead Code: After 59

Figure 22 Extend Loop Condition: Before 60

Figure 23 Extend Loop Condition: After 61

Figure 24 Bogus Class Example 63

Figure 25 Eclipse Workspace: After Bogus Class Insertion 64

Figure 26 Reorder Methods: Before 65

Figure 27 Reorder Methods: After 65

Figure 28 Before String Obfuscation 67

Figure 29 After String Obfuscation 67

Figure 30 Redundant Operand: Example 1 69

Figure 31 Redundant Operand: Example 2 69

Figure 32 User Statistics for Complex1 75

Figure 33 User Statistics for Complex2 76

Figure 34 User Statistics for Complex3 76

Figure 35 User Statistics for Complex4 77

Figure 36 User Groups Statistics 78

Figure 37 ANTLRWorks Interface 91

Figure 38 Code Snippet 92

Figure 39 Simple Calculator Output 99

Figure 40 Obfuscated Calculator Program Output 103

LIST OF TABLES

Table 1 Java Obfuscators 36

Table 2 Comparison of Java Obfuscators 37

Table 3 ProGuard Obfuscation Observations 40

Table 4 Jshrink Obfuscation Observations 42

Table 5 Zelix KlassMaster Obfuscation Observations 44

Table 6 Data Structures used in JShield 50

Table 7 Comparison of Tools 72

Table 8 Test Programs 73

Table 9 User Statistics: Before Obfuscation 74

Table 10 User Statistics: After Obfuscation 75

Table 11 Average time taken by users 77

Table 12 Simple Calculator: Before Obfuscation 95

Table 13 Simple Calculator: After Obfuscation 99

1.0 INTRODUCTION

"Any sufficiently advanced technology is indistinguishable from magic."

(Arthur C. Clarke)

1.1 Reverse Engineering

The process of extracting knowledge or design blueprints from anything man-made is known as reverse engineering (Eilam, 2005). In other words, reverse engineering may be understood as a systematic methodology for analyzing the design of an existing device or system, either as an approach to study the design or as a prerequisite for re-design. "Reverse engineering is the process of analyzing a subject system to (i) identify the system's components and inter-relationships and (ii) create representations of the system in another form or at a higher level of abstraction" (Chikofsky & Cross, 1990).

In the field of software, developers sometimes do need to understand how existing software works. The concept of reverse engineering, when applied to software leads to many interesting consequences. Various problem areas where reverse engineering has been successfully applied are recovery of design patterns (Antoniol et al., 2001), code smell detection (Emden & Moonen, 2002), redocumentation of programs (Benedusi et al., 1992), renewal of user interfaces (Merlo et al., 1995, Moore, 1998), migration of legacy code (Canfora et al., 2000), translation of program from one language to another (Byrne, 1991), and architecture recovery (Koschke, 2000).

Reverse engineering has proved very helpful in many ways. But on the contrary, it has lead to many serious problems. "Each year software piracy results in billions of

dollars in lost revenue" (Chen et al., 2006). Hacking is one of the challenges that reverse engineering has brought into picture (The terms 'hacking' and 'reverse engineering attacks' are used interchangeably in this paper. It refers to the hacking attacks that are based on reverse engineering). "Stealing or replicating someone else's ideas has always been the easiest way of creating competitive products" (Kalinovsky, 2004). The process of reverse engineering helps in understanding the logic of software which makes it easy to alter its behavior or copy the algorithms. The removal of usage restrictions from software, exploitation of software flaws, cheating in the games and breaking the digital rights of a system are some such reasons for which the hackers resort to reverse engineering (Stamp, 2006).

1.2 Process of Reverse Engineering

"To reverse engineer a software application it is first necessary to gain physical access to it" (Low, 1998). The process of reverse engineering consists of three steps: (i) Parsing and semantic analysis of code, (ii) Extracting information from the code, and (iii) Dividing the product into components, as indicated by Figure 1 (Chikofsky & Cross, 1990). The software code is parsed and semantic analysis is performed on the parsed code. The information thus obtained is stored in an information base and then this information is used to understand the basic functionality and algorithms of the software. This knowledge can be used for legitimate reasons like creating a new system with better design and functionality or it can be misused.

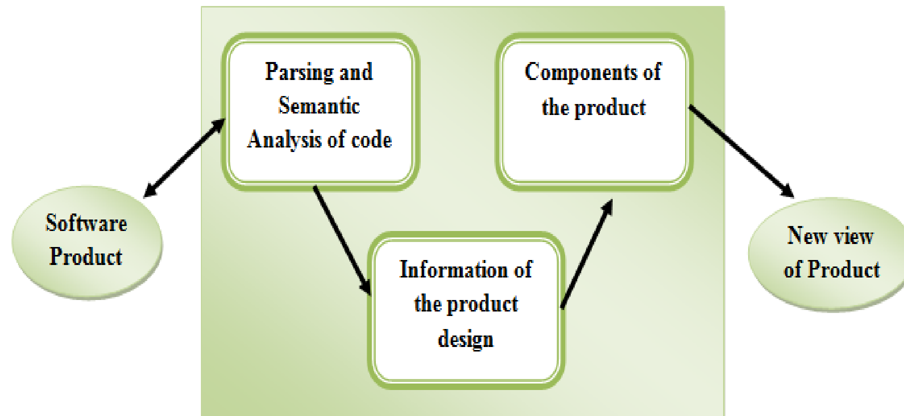

Figure 1 Process of Reverse Engineering

1.3 Anti-Reverse Engineering

The defense techniques implemented in software in order to protect it from malicious attacks are referred to anti-reversing techniques. It has become a challenge for the software community to protect software from attackers and to prevent its misuse. The patent system is not quite as effective with software as it is with traditionally engineered tangible artifacts. While a patent mandates IP protection – it is next to impossible to prove or even suspect any IP theft in a software product that might have been the result of a malicious reverse engineering attack on a patented competitor. After all, such a product, implemented slightly differently from the original, yet using the same core ides and algorithms could simply be deemed as an inventive step over 'prior art' (Kalinovsky, 2004).

(Eilam, 2005) states in his book "It is never possible to entirely prevent reversing" and (Chen et al., 2006) states "The goal of any "anti" reverse engineering technique is to substantially increase the amount of work that a reverse engineering attempt entails,

hopefully beyond the useful lifetime of a software application (or a particular version of the application)”. This indicates that it is possible to evaluate the effectiveness of an anti-reversing technique using empirical metrics.

It is not easy to define criteria for evaluating the different reversing techniques. The criteria that can be used for examining the effectiveness of a reversing technique are (Nolan, 2004):

- Potency – How confused the decompiler is?
- Resilience – Can it rebuff the decompilation attempts?
- Cost – How much overhead does it cause?

1.4 Anti-Reversing Tools

“Reversing is impossible without the right tools” (Eilam, 2005). There are various software tools available on market, free as well as those costing hundreds of dollars. The tools available for reverse engineering include disassemblers available for extracting assembly code from the executables, debuggers for dynamic analysis of code during execution, and decompilers for generating high-level source code from the executables (Chen et al., 2006).

The most popular disassembling and debugging tools available include OllyDbg (Yuschuk, 2000), IDA Pro (Guilfanov, n.d.), SoftICE (SoftICE, n.d.), WinDbg, etc. These tools not only extract the assembly code but also help in viewing many other details of the software. They help in analyzing and patching the code as well.

Java programs are more prone to reversing attacks as “It is more feasible to recover Java source code from Java bytecode than it is to recover C/C++ code from

machine code" (Cipresso, 2009). Just a few of the various decompilers available include Jad (Kouznetsov, 1997), JODE (Hoenicke, 2002), and Jdec (Belur & Bettadapura, 2006).

A lot of research is being done in the software field in order to find out successful ways of protecting software from reverse engineering attacks. The techniques proposed to make reverse engineering difficult include obfuscating the code (Collberg et al., 1997), protecting the computing platform physically (Doorn et al., 2003), encryption of executables (Chen et al., 2006), and watermarking (Collberg & Thomborson, 2002).

1.5 Threat to Java Software

The threat of reverse engineering attacks has been taken more seriously since the advent of Java, because the applications written in Java are easier to reverse engineer (Cipresso, 2009). To understand why, we have to know the difference between Java bytecode and machine code.

- "Machine code or processor instructions are a system of instructions and data executed directly by a computer's central processing unit" (Machine code, 2010). These instructions are specific to the processor on which they are generated. Figure 2 illustrates this scenario.
- "Bytecode is a set of instructions that looks a lot like some machine code, but is not specific to any one processor" (Lemay & Perkins, 1996). "It is the intermediate representation of Java programs just as assembler is the intermediate representation of C/C++ programs" (Haggar, 2001). Figure 3 illustrates the generation of bytecode.

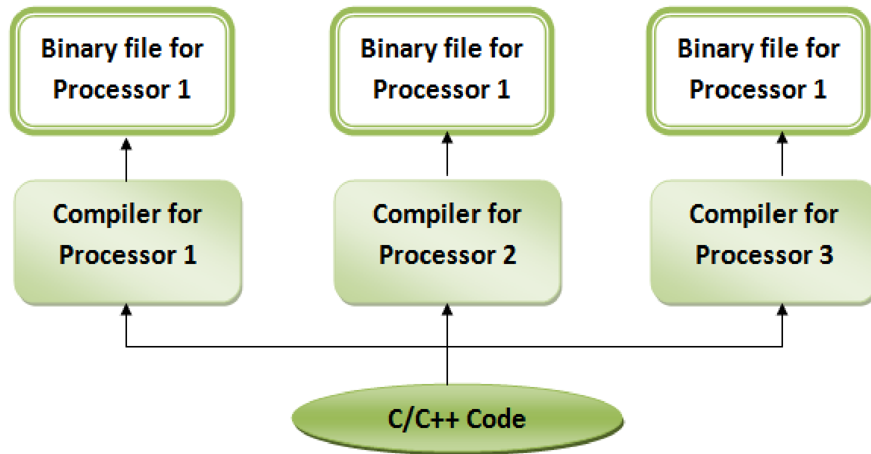

Figure 2 Machine Codes

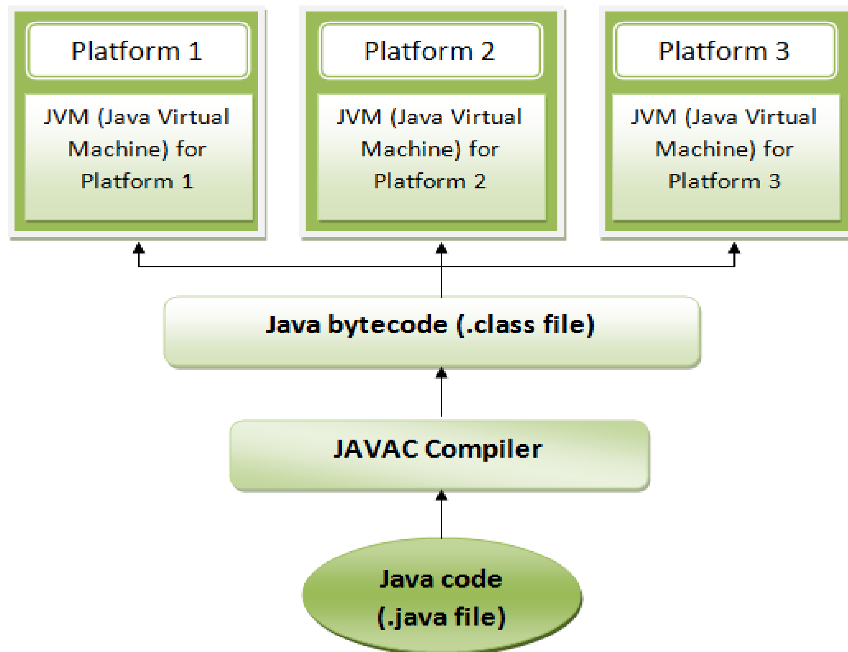

Figure 3 Bytecodes

Java was designed for supporting platform-independent development. This was done by converting the source code into platform-independent bytecode for compilation. "Java bytecode is standardized and well documented" (Kalinovsky, 2004). It contains a lot of information about the code and thus it can be easily decompiled to the source code.

Another characteristic of Java that proves beneficial to the reverse engineering attackers is the use of standard library routines which keeps the size of the application small.

The design of Java language itself, thus, makes it highly prone to reverse engineering attacks. This has become a big problem, as a number of mission critical applications in industries like banking, or simply closed-sourced proprietary applications and games are being developed in the Java language. The purpose of this thesis is to analyze the existing anti-reversing techniques that can be implemented to make Java code immune to reversing attacks and suggest a tool that automates the process of implementing anti-reversing techniques for Java software.

1.6 Organization of this Thesis

The work done by various researchers is discussed in Section 2.0. Section 3.0 introduces the various anti-reversing techniques and Section 4.0 discusses the tools available for obfuscation. Section 5.0 explains the functionality provided by JShield, along with the approaches applied in the prototype tool. It also presents validation of the tool and verification of the results. Section 6.0 concludes the thesis and proposes the related future work to be done in this field. In the next section, we will discuss research done in past years.

2.0 LITERATURE REVIEW

“Men are only as good as their technical development allows them to be.”

(George Orwell)

There is a significant body of literature documenting the work done so far in the field of reverse engineering and anti-reverse engineering. This section cites some prior research in the field of reverse engineering and then describes the importance of anti-reverse engineering and how and why it came into picture.

2.1 History

A lot has been done in the field of reverse engineering over the past 20 years (Canfora & Penta, 2007). Research in the field of reverse engineering had started in the early nineties. Initially, the research was mainly focused on the analysis of procedural software for understanding it and to deal with the Y2K problem (Low, 1998). Architecture recovery was another focus area that was facilitated by reverse engineering. A number of techniques were proposed for component recovery.

In short, most research during the nineties was focused on three main problems (Canfora & Penta, 2007):

- Program Analysis
- Design Recovery
- Software Visualization

The origin of reverse engineering can be traced to software maintenance processes and techniques. The definition of reverse engineering is quite broad today as it encompasses a number of fields like aiding software test by creating representations of

code (Memon et al., 2003), evaluating software design or examining software security (DaCosta et al., 2003). (Chikofsky & Cross, 1990) states that the objective of reverse engineering in software is "most often to gain a sufficient design-level understanding to aid maintenance, strengthen enhancements, or support replacement".

2.2 Relation to Reengineering

Reverse engineering is sometimes understood to be a restructuring technique used for redevelopment of software, which is not precisely what reverse engineering is all about. The objective of the reverse engineering techniques can be broadly classified into two categories: redocumentation and design recovery (Canfora & Penta, 2007), as shown in Figure 4. "Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level" (Chikofsky & Cross, 1990) and "Design Recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains" (Biggerstaff, 1989).

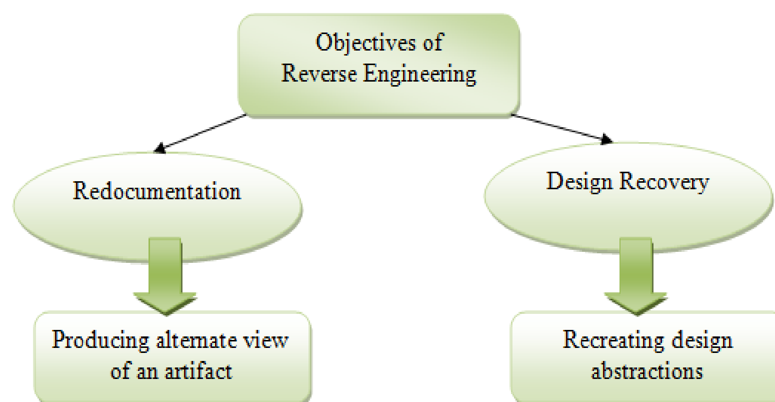

Figure 4 Objectives of Reverse Engineering

The argument given in support of this theory is that by definition reverse engineering does not include restructuring or reengineering. Instead, the process of reverse engineering is just a phase of reengineering. Reengineering can be understood as a process with three phases - reverse engineering, architecture transformation and forward engineering. As Figure 5 shows, the reverse engineering phase aims at obtaining an abstraction of the target software and the forward engineering phase aims at the restructuring part.

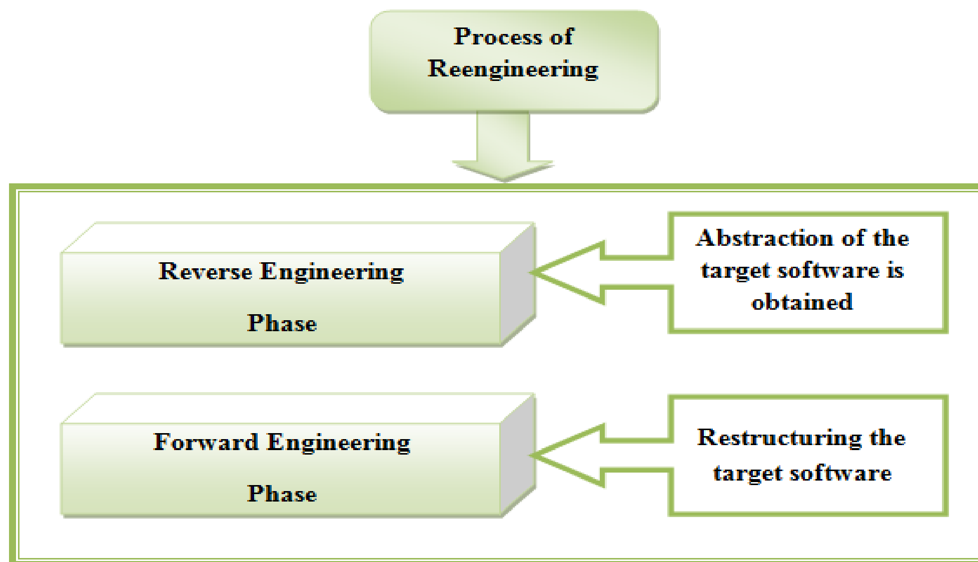

Figure 5 Reengineering

Figure 6 presents the Architecture Reengineering process (Kazman et al., 1998). It indicates that architecture recovery is the reverse process of Architecture Development. For the transformation of software architecture from one form to another, we have to recover the coding approach followed and the architectural plan of the given software. This in turn helps us in figuring out the design patterns implemented in the software.

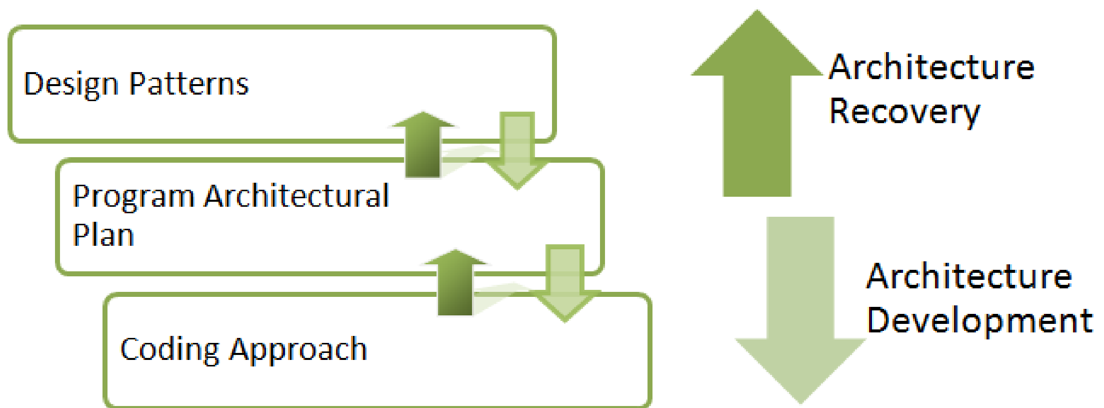

Figure 6 Architecture Reengineering

(Chikofsky & Cross, 1990) give a clear definition and distinction between the terms reverse engineering, forward engineering, restructuring and reengineering using three software life-cycle stages. The three life-cycle stages that they use are – requirement analysis, design, and implementation. Figure 7 pictures the relationship between the aforesaid terms.

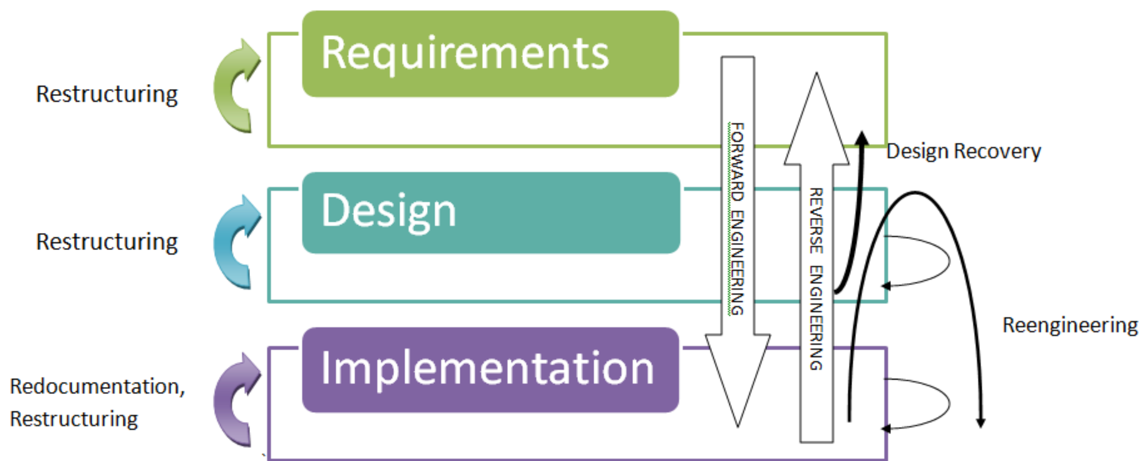

Figure 7 Relationship of terms

2.3 Decade of Achievements

A lot has been done in the field of reverse engineering over the past 20 years (Canfora & Penta, 2007). We see significant advancements made over the past decade. The work done in the field of software reverse engineering has been disseminated in multiple software engineering conferences and journals. As discussed earlier, the research work was focused on the problems of program analysis and its applications, architecture and design recovery, and visualization. The following sections discuss the advancements made in these fields over the past 10 years.

2.3.1 Program Analysis

A number of tools have been developed to help in the analysis of computer programs. Initially these tools used static analysis, but eventually this approach was found lacking for many programs where dynamic analysis was required (Systä, 2000). Dynamic analysis is necessary in many situations and is widely used despite being expensive and incomplete (Ernst, 2003). A number of new analysis techniques have been developed to address the different challenges faced by the software community. For

example, the complexity of program analysis increases with program size. So, techniques like island parsing and lake parsing are employed to analyze only small fragments of code at a time instead of entire programs in one go (Moonen, 2001).

Another event that inspired the research effort in the field of program analysis is the presence of clones in software systems (Canfora & Penta, 2007). The different techniques developed as an outcome include token-based (Baker, 1995), AST-based (Baxter et al., 1998), and metrics-based (Leblanc et al., 1996) techniques.

2.3.2 Architecture and Design Recovery

Initially, the role of reverse engineering in the field of architecture and design recovery was focused on recovering high level architectures from procedural code. With the diffusion of object oriented languages and Unified Model Language (UML), it became important to recover UML models as well from source code.

(Potrich & Tonella, 2005) proposed the static approach for recovering class diagrams and also demonstrated that static analysis was insufficient as it did not contain any information about flow propagation. They successfully extracted sequence diagrams using static analysis on data flow. (Systä, 2000) recovered the UML diagrams by using a combination of static and dynamic analysis techniques.

Another concept that had become very popular along with object-oriented development was design patterns. Recovering the design pattern from the code was helpful in code reuse and assessing code quality. Both static (Antoniol et al., 2001) and dynamic analysis techniques (Heuzeroth et al., 2003) were used to recover design patterns.

2.3.3 Visualization

Software visualization is a blessing to the reverse engineers. A pictorial representation of information greatly benefits both the analyzer and the developer. The proper visualization of the program and the information extracted from its analysis is very important for gaining clearer understanding the code. The code flow becomes much easier to understand with a tool that is capable of presenting relevant information at the right level of detail (Canfora & Penta, 2007). A number of such tools are available, like Rigi (Muller et al., 1995), CodeCrawler (Marziali, n.d.), Seesoft (Easterbrook et al., 2003), and sv3D (Feng et al., 2003). All these tools provide useful visualization of the software using various techniques. One of these tools, Rigi, can show architectural views, while sv3D can render software architecture metrics in a 3D visual representation. "Code Crawler combines the capability of showing software entities and their relationships, with the capability of visualizing software metrics using polymetric views, which show different metrics using the width, the length, and the color of the boxes" (Canfora & Penta, 2007).

These advancements in the field of reverse engineering not only indicate the progress made, but also portray the potentials of reverse engineering. With the tools developed for the purpose of helping the software community, another set of people have been benefitted – the hacker community. With so many tools at hand, they can misuse or reuse a lot of licensed software and the algorithms, without paying a dime to the owners.

2.4 Future Trend

While researchers are working on development of more advanced tools to facilitate the process of reverse engineering, in doing so, they are also making the job of hackers much easier. With the advancement in the field of dynamic analysis of programs, hackers can not only analyze their target software statically but can also uncover the exact implementations of its underlying algorithms. The availability of a wide range of efficient decompilers for high level languages like Java makes it all the more difficult to protect software as it is now possible to recover an almost exact copy of the source code from a class file. We have already discussed (Section 1.3) that copyrights and patents are not very effective. So it is a big challenge for IP owners to protect their code by incorporating anti-reversing techniques into their code.

In the next section, we will discuss about various anti-reversing techniques used to protect java programs from malicious attacks.

3.0 ANTI-REVERSING TECHNIQUES

"A lock only ever stopped an honest man."

(Ancient Egyptian Proverb)

3.1 Protecting Java Code

The software development community has been working on this problem of protecting Java software for many years. The techniques that can currently be used to protect Java source code are given in Figure 8 (Nolan, 2004). These techniques are briefly discussed here:

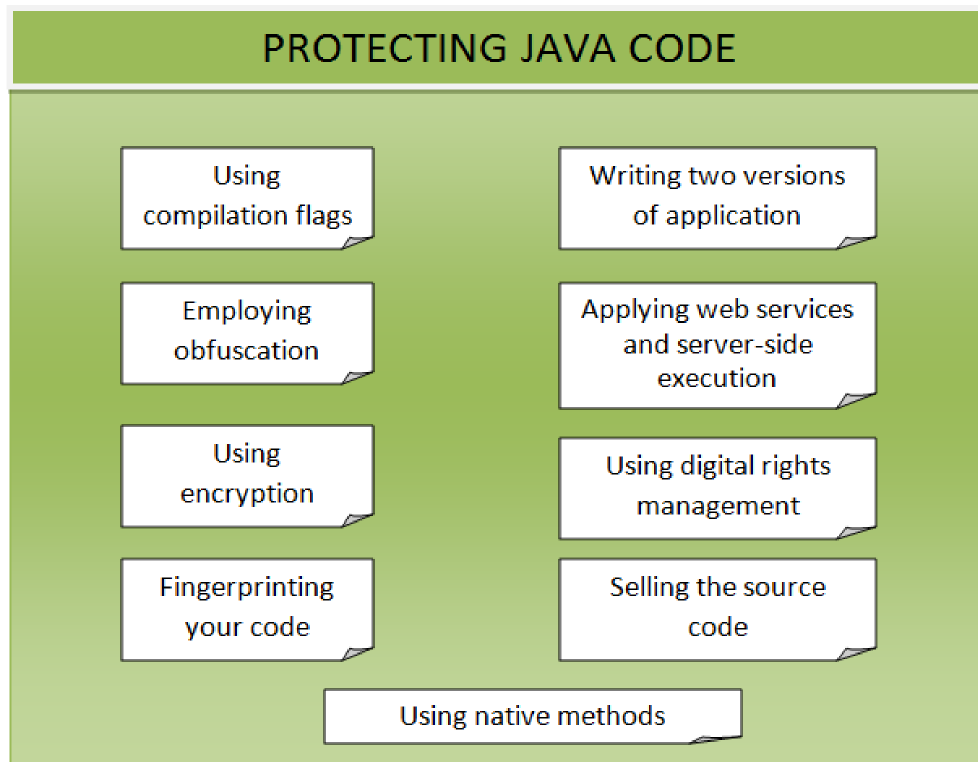

Figure 8 Ways of Protecting Java Code

3.1.1 Compilation Flags

The bytecode generated by the compiler is affected by different types of compilation flags (Nolan, 2004). Use of the `-g` flag during compilation generates debugging tables that contain information about line numbers and local variables (javac, n.d.). This information is very useful for the decompiler to retrieve the source code. So, compilation with `-g:none` keeps information like lines, vars, and the source file attributes out of the classfile (Nolan, 2004).

3.1.2 Implementing Two Versions of the Application

It is a popular trend in the software industry to let users download a fully functional evaluation copy of the software that can be used up to a predefined period of time or a certain number of usages. This introduces the potential threat of malicious users removing these limitations to activate a functional copy of the software without having paid for it after their trial period expired. A possible solution is to implement two versions of the software; with a cut-down trial version that does not reveal all its functionality. Thus the user is forced to buy the original software if they like the trial version. (Nolan, 2004)

3.1.3 Applying Obfuscation

"Obfuscated code is source or machine code that has been made difficult to understand for humans" (Obfuscated code, 2010). There are a number of techniques used to obfuscate code and it is the method used in this thesis. The different techniques for obfuscation have been discussed in detail in the next section (3.2 Obfuscation Techniques).

3.1.4 Using Web Services and Server-Side Execution

Most modern software applications have multi-tier architectures as a best practice – mainly to keep them maintainable, to keep different layers decoupled, accessible through browsers, and to facilitate ease of enhancements etc. But, as a positive side effect, splitting applications into presentation and business tiers also protects the code from inquisitive eyes. By keeping the business layer (which contains the actual business logic/code) on a remote server, the client side applet or application presents just the user interface which does not contain much information, apart from, maybe remote URI's and top level API's (Nolan, 2004).

3.1.5 Encryption

"Throughout the ages, mankind has turned to encryption when trying to protect secret transmissions" (Nolan, 2004). A common solution suggested for preventing the code from decompilation is to encrypt the class files. These class files are not decrypted until before they are executed.

3.1.6 Digital Rights Management

It is clear from our discussion so far that the bytecode needs to be kept out of reach of the end user in order to prevent them from decompiling the code. Ultimately, it would be wiser to protect the code by simply securing the browser and class loader using a trusted browser. The browser should not let the end user access the cache which contains code. (Nolan, 2004)

3.1.7 Fingerprinting the Code

Digital fingerprinting is a string of binary digits that uniquely identifies a file (Digital Fingerprint, n.d.). It is usually in the form of a copyright notice that helps you to identify your code. Inserting a fingerprint does not provide any protection but it helps in protecting the copyright by providing a way for the developer to prove that the code was originally written by him. (Nolan, 2004)

3.1.8 Selling Source Code

"If source code is so readily available, then why not just sell it at a higher price?" (Nolan, 2004). The decompiler can be discouraged to decompile if you sell the source code directly to him. It will bring in some more revenue for the programmer and the programmer will not have to worry about protecting the code.

3.1.9 Employing Native Methods

We discussed in Section 1.5 that code written in Java is more difficult to protect than that written in C/C++. (Nolan, 2004) suggests that we can protect our Java code by compiling it in C or C++. It is possible to do this in Java by using the Java Native Interface (JNI). It might cause portability issues but is useful if portability is not an issue.

3.2 Obfuscation Techniques

There are a number of techniques that can be used to make software immune to reversing attacks. Many of these techniques are used by the obfuscators available in the market. This section discusses various obfuscation techniques that can prove beneficial in protecting Java software from reversing attacks. Before we discuss the different techniques of obfuscation, it is important to know that obfuscation techniques can also be

classified based upon the stage at which the obfuscation is applied. Obfuscation can be classified into three classes (Sogiros, n.d.):

- Source code obfuscation: The obfuscation is performed on the source code.
- Bytecode obfuscation: The transformations are performed on the bytecode of the compiled software.
- Binary code obfuscation: The obfuscation is achieved by rewriting the instructions at machine code level.

Figure 9 gives another classification of the obfuscation techniques (Nolan, 2004, Collberg et al., 1997) based upon how the code is obfuscated.

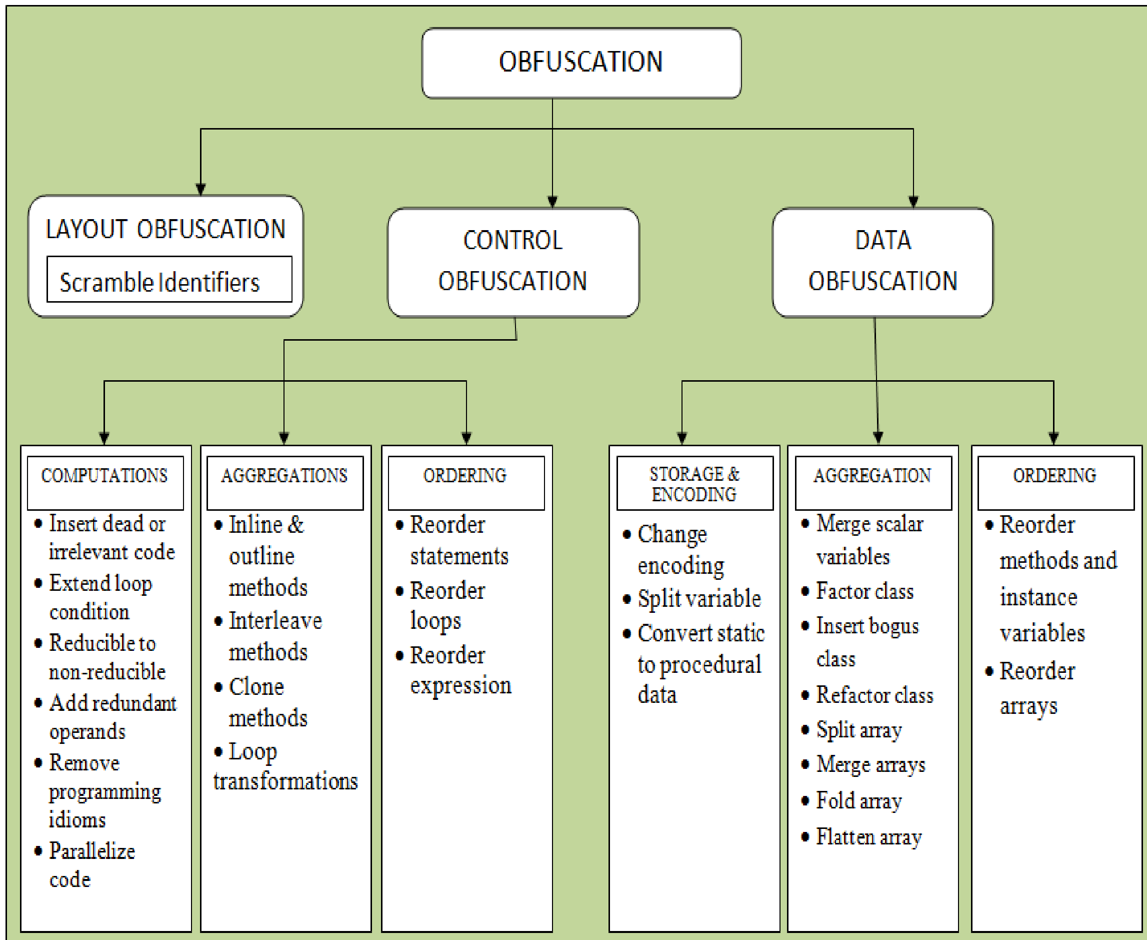

Figure 9 Obfuscation – A Classification

3.2.1 Layout Obfuscation

Obfuscators available on the market work by scrambling the identifiers in the classfile to make the decompiled source useless. The variables are renamed with automatically generated garbage variables which do not affect the code functionality as the classfile uses pointers to methods and variables instead of actual names. It becomes difficult to understand the code but it is not impossible. A disassembler can be used to rename the variables in order to generate more meaningful names. (Nolan, 2004)

3.2.2 Control Obfuscation

The idea behind control obfuscation is to disguise the real control flow (Low, 1998). The control flow of the source code is altered to confuse anyone looking at the decompiled code (Nolan, 2004). (Kalinovsky, 2004) states, “The best obfuscators are capable of transforming the execution flow of bytecode by inserting bogus conditional and goto statements”. (Collberg et al., 1997) classifies control obfuscation into three different categories – computation, aggregation, and ordering.

3.2.2.1 Computation

Computation techniques alter the control flow in a program. It is a type of control obfuscation which can be broken down into following techniques (Nolan, 2004):

- Insert Dead or Irrelevant Code

  The insertion of dead code or junk code confuses the attacker. You insert code that will never be executed and/or will never contribute to the functionality of the program. “This code can include extra methods or simply a few lines of irrelevant code” (Nolan, 2004). It is important to note here that this dead code is to confuse the decompiler and the attacker. Unless the size of program is too small, it will take some effort and time for the attacker to figure out that those chunks of code are actually doing nothing.

  (Nolan, 2004) states, “Don’t just limit yourself to thinking about inserting Java code, there’s no reason why you can’t insert irrelevant bytecode”. The reason why incorrect bytecode can be inserted into the class file without affecting the functionality of the program is that the rules of bytecode format verification

are not strictly enforced by the JRE. This corrupted code does not affect the functionality of the original code but crashes on a decompilation effort. (Kalinovsky, 2004)

- Extend Loop Condition

Complicating the loop conditions introduces obfuscation in the code. This can be done by extending the loop condition with a second or third condition that doesn't do anything (Nolan, 2004). For example, in the following example we have a simple if condition.

Before:	After:

`int x = 1;`
`if (x > 200)`
`{`
`   …`
`   x ++;`
`   // call function`
`   abc(x)`
`        }`
	`int x = 1;`
	`while (x> 200 ||`
	`   x%200==0)`
	`{`
	`   …`
	`   x ++;`
	`   // call function`
	`   abc(x)`
	`}`

This condition is easy to understand as it has no calculation involved. But if we replace this code with condition that does the same job but looks complex, it might make it a little more time consuming for an attacker to understand the logic.

- Reducible to Nonreducible

"The Holy Grail of obfuscation is to create obfuscated code that cannot be converted back into its original format" (Nolan, 2004). We can devise some transformations that make the code nonreducible to its original form. For example, the Java bytecode has goto instruction while no equivalent statement

exists in the Java language. So, the flow graphs produced from Java programs are always reducible, while those from Java bytecode may express non-reducible flow graphs. Expressing non-reducible flow graphs is inconvenient in Java due to unavailability of goto statements, so we need to do some transformation for converting the reducible flow graph into a non-reducible one. We can achieve this by converting a structured loop into a loop with multiple headers (Collberg et al., 1997). For example, see the code below:

Before:

```
Statement 1;
while (condition1)
{
    Statement2;
}
```

After:

```
Statement 1;
if(condition2)
{
Statement2';
while (condition1){
        Statement2;
}
else {
while (condition1){
         Statement2;
 }}
```

In this example, we had a simple while condition. We split the statement to make it appear more complicated than it actually is.

- Add Redundant Operands

Adding some insignificant terms to the code, in the basic calculations confuses the reverse engineer. For example, let's assume that there is an integer variable, 'p' that stores the product of two integer variables – 'a' and 'b'. The code below shows we can make the calculations look complex to the attacker. (Nolan, 2004)

Before:

```
public int sum{
     int a = 5;
     int b = 7;
     int p;

     p = a * b;

System.out.println("
Product =" + p);
}
```

After:

```
public int sum{
     int a = 5, b = 7;
     double i = 0.0005;
     double j = 0.0007;
     double p;

     p = (a * b) + (i*j);
     System.out.println("
Product =" + (int) p);
}
```

Both of these code snippets will generate exactly the same output, just that the second one looks more complex than the original one. (Nolan, 2004) warns that using this technique all through the application has the potential to degrade its performance.

- Remove Programming Idioms

Most programmers reuse the code that they have written for some previous application. In other words, they reuse the components, methods, and classes they had written for a previous application in a slightly different manner. The book Refactoring: Improving the Design of Existing Code (Addison-Wesley, 1999) written by Martin Fowler has created a standard for programming in Java by presenting how to refactor some existing code into shape. Such standardization created a series of programming idioms which prove helpful to the hacker in reversing the code. (Collberg et al., 1997) states, “An experienced reverse engineer will search for such patterns to jump-start his understanding of an unfamiliar program”. So, in order to prevent such hints the programmer

should write sloppy code. It is not good for the performance and long-term maintenance of code but ensures that the hacker does not gather much knowledge about your code without even reversing it. (Nolan, 2004)

According to (Collberg et al., 1997), “Most programs written in Java rely heavily on calls to standard libraries”. This also provides a lot of information to the reverse engineers. These calls are made to the library artifacts by name and hence these names cannot be obfuscated. The solution to this problem is to make your own version of standard libraries and then calling them instead. (Collberg et al., 1997)

- Parallelize Code

One thing that can staggeringly increase the complexity of your program is the introduction of threads (Nolan, 2004). The parallelization process is usually done to increase the performance, but the motive of introduction of threads is to hide the actual flow of code from the hacker (Collberg et al., 1997). The two suggested methods of doing this by (Collberg et al., 1997) are:

1. Create dummy processes which do not actually perform anything useful.
2. Split a sequential section of the program into multiple sections executing in parallel.

(Nolan, 2004) points out that there is a programming overhead to ensure that the threads are in proper order and is not interfering with the proper functioning of the program.

3.2.2.2 Aggregation

Aggregation obfuscation alters how statements are grouped together (Gupta, 2005). (Collberg et al., 1997) has included following techniques in this category:

- Inline and Outline Methods

  In Java, inlining (replacing a method call with the actual body of method) results in ballooning of code which makes it difficult to understand the code. It makes inlining an excellent technique to obfuscate the code. It should be noted that this is a one way transformation. Once the method call has been replaced by the actual code, the function is removed and all traces of abstraction are removed from the code. (Collberg et al., 1997)

  "You can also balloon the code by taking some of the inlined methods and outlining them into a dummy method that looks like it's being called but doesn't actually do anything" (Nolan, 2004).

- Interleave Methods

  It is an important and difficult task in reverse engineering to detect interleaved code (Collberg et al., 1997). (Rugaber, 2000) writes:

  "Subcomponents interact with each other. If the interactions are limited and occur through explicit interfaces, the component is said to be encapsulated. If, usually for reasons of efficiency, two or more design ideas are realized in the same section of code or by the same data structure, then the components corresponding to those ideas are said to be interleaved."

Two methods of a class can be interleaved by merging their parameter lists and adding an extra parameter that will differentiate between the calls to individual methods (Collberg et al., 1997). It is a significantly more difficult task to separate out the interleaved methods as compared to interleaving them (Nolan, 2004). For illustration of this technique we consider the following example where we have combined two methods, *calTax* and *emailSalDetails* into one method *calTaxEmailSalDetails* just to confuse the hacker.

Before:

```
void calTax (int employeeGrade, double salary){
if(employeeId < 4){
     printSalaryStub (salary*0.3);
} else{
     printSalaryStub (salary*0.4);
}

void emailSalDetails (int employeeId){
    printHeader();
    printSalDetails(employeeId);
    printFooter();
}
```

After:

```
void calTaxEmailSalDetails(int choice, int
employeeId, int employeeGrade, double salary){
     printHeader();
if (choice == 1){
     if(employeeId < 4){
         printSalaryStub (salary*0.3);
     } else{
         printSalaryStub (salary*0.4);
     }
else{
    printHeader();
    printSalDetails(employeeId);
```

```
    printFooter();
}}}
```

- Clone Methods

  It is important for a reverse engineer to understand the purpose of a function and it is equally important to understand the different conditions under which the function is called (Collberg et al., 1997). We can create clones of a function and make calls to these functions under identical circumstances. We can call the function depending on any external factor, which appears to be a deciding factor but is actually not. One good example would be to call a different function based on the day of the week. (Nolan, 2004)

- Loop Transformations

  In order to improve the performance of numerical applications, a number of loop transformations have been designed. Some of these transformations tend to increase the complexity of the code and hence are of interest to us (Collberg et al., 1997). Some of these transformations are – loop blocking ("breaks up the iteration space so that the inner loop fits in cache thus improving the cache behavior" (Collberg et al., 1997)), loop unrolling ("replicates the body of the loop one or more times. If the loop bounds are known at compile time, the loop can be enrolled in its entirety" (Collberg et al., 1997)), and loop fission ("turns a loop with compound body into several loops with the same iteration space" (Collberg et al., 1997)).

3.2.2.3 Ordering

Ordering transformations relate to altering the order in which the statements will be executed in the application (Gupta, 2005).

- Reorder Statements and Expressions

  The reordering of statements and expressions does not complicate the code much for the reverse engineer. But reordering the expressions obfuscates the code significantly if applied at bytecode level as it disrupts the link between the Java source code and bytecode. (Nolan, 2004)

- Reorder Loops

  A simple obfuscation technique is to reorder the loops. For example, transforming a loop so that it moves backwards. It is shown below:

Before:

```
val = 0;
while (val < maxVal)
{
   arr[val] += res[val];
   val++;
}
```

After:

```
val = maxVal;
while (val >0)
{
    val--;
    arr[val] += res[val];
}
```

3.2.3 Data Obfuscation

Data obfuscation techniques refer to the transformations that obnubilate the data structures in the source code. These techniques are classified into four categories based on how they affect data – storage and encoding, aggregation, and order. (Collberg et al., 1997)

3.2.3.1 Storage and Encoding

These techniques target the data structures. They change the way data is stored and how the stored data in interpreted. For example, changing the type of a variable or replacing an existing value of a variable with a more complex looking equivalent. We will now discuss all these techniques in detail here.

- Change Encoding

  (Collberg et al., 1997) show a simple example of encoding in the paper. An integer variable `i = 1` is transformed into `i' = x*i+y`. If we choose `x = 6` and `y = 5`, we get transformations shown below:

Before:	After:

`int i = 1;`
`while (i <= 100) {`
`    result = arr[i-1];`
`    i++;}`
	`int i = 11;`
	`while (i <= 605) {`
	`    result = arr[(i-`
	`5)/6];`
	`    i+=6;}`

- Split Variables

  The variables with restricted range like Boolean can be split into two or more parts in order to make them less obvious to the reverse engineer. (Collberg et al., 1997) says, "We will write a variable V split into k variables $p_1, \ldots, p_k$ as $V = [p_1, \ldots, p_k]$." For example, if we have to define the Boolean value of a variable `bool = true`, then we can split it into `bool1 = 0` and `bool2 =1`, and use the following lookup table to change it back to the Boolean value. (Nolan, 2004)

bool1	bool2	bool
1	0	false
0	1	true

- Convert Static to Procedural Data

  The strings in the source are used to store very critical information like copyright information. If this static data is converted to procedural data, the job of the hacker will become significantly tough. As an example, the copyright information could be generated programmatically within the code rather than being stored directly in a string. This kind of transformation is not very practical as it is not trivial to implement and it cannot be automated. (Nolan, 2004)

3.2.3.2 Aggregation

Aggregation transformations change the grouping of the data. An example is splitting an array into several sub-arrays. (Gupta, 2005)

- Merge Scalar Variables

  "The variables can be merged together, or converted to a different base and then merged together. The variables' values can be stored in a series of bits and pulled out using a variety of bitmask operators (Nolan, 2004)."

- Class Transformations

  A series of class transformations can prove helpful in making the program difficult to understand. As the depth of an application's class hierarchy increases, so does its complexity. One good way of achieving this is to use inheritance and

interfaces to the extreme to create deep class hierarchies that will make it more difficult for the hacker to understand the application. (Nolan, 2004)

Inserting a bogus class can confuse the reverse engineer. But it is important that the bogus class should be called by the actual program else shrinker (Section 3.3) will very easily get rid of the class.

- Array Transformations

  Just like variables, we can split, merge, or interleave arrays in order to obfuscate the code. We can split the array depending on the index position – placing all the values on even indices stored in one array and those on odd indices in another. (Nolan, 2004)

3.2.3.3 Ordering

As noted in the previous section, randomizing computations obfuscates the code. Similarly, randomizing the order of declarations also confuses the reverse engineer.. (Collberg et al., 1997)

- Reorder Methods, Arrays, and Instance Variables

  It is a good idea to just move methods, arrays and data declarations across the code, so that the reverse engineer spends some time figuring out details of each. We should ensure that the data elements remain in the appropriate scope while doing this. (Nolan, 2004)

3.3 Some Terminology

Before we move on to the next section, it will be helpful to understand a few terms clearly.

- Shrinker:

  Shrinking removes all the unused code from the application. It is not exactly an obfuscation technique but many tools do provide shrinking as an additional benefit (Kalinovsky, 2004). Eliminating the unused code reduces the size of the jar file.

- Optimizer:

  An optimizer helps to optimize and verify the compiled Java applications. It analyses the application for problems like memory leaks, code bottlenecks, presence of unwanted attributes in classfile, etc. It improves the performance of the application. (Foley, 2009)

- Obfuscator:

  A tool that applies obfuscation techniques to software in order to protect it from reverse engineering attacks is called an obfuscator.

- Preverifier:

  "Preverification performs certain checks on the Java bytecode ahead of runtime. If this first verification pass is OK, the preverifier annotates the classfiles and then saves the annotated class files" (jGuru, 2000). When Kernel-based Virtual Machine (KVM) attempts to execute the application, it checks the Java class files for these preverification annotations. If the proper annotations are present in the class files, it guarantees that certain compile-time checks were made. So the verification and the security checks of KVM are passed faster, thus ensuring faster execution times. (jGuru, 2000)

In the next section, we will examine the functionality and capability of some existing tools.

4.0 EXISTING OBFUSCATORS

"It has become appallingly obvious that our technology has exceeded our humanity."

(Albert Einstein)

4.1 Tool Support

There are a number of tools on the market, both open source and commercial that claim to obfuscate software, making it immune to reverse engineering attacks. These include ProGuard (Lafortune, n.d.), Jshrink (Eastridge Technology, n.d.), Zelix KlassMaster (Zelix KlassMaster, n.d.), and Semantic Designs Java Obfuscator (Semantic Designs, n.d.). Table below highlights the features of each one of them (Google directory, n.d.).

Table 1 Java Obfuscators

Tool	Works on	Feature	Type of Tool
ProGuard	Bytecode	Shrinker, Optimizer, and Obfuscator	Open Source
Jshrink	Bytecode	Obfuscator	Commercial
Zelix KlassMaster	Bytecode	Obfuscator	Commercial
Semantic Designs Java Obfuscator	Source code	Obfuscator	Commercial

An obfuscator can definitely protect software to a certain extent. But it does not imply that the software cannot be reverse engineered. With enough time and effort, an attacker can still retrieve enough information about the software and misuse it. We

discussed in Section 1.3 that the level of security that is provided by any obfuscator depends on three factors – potency, resilience, and cost. (Nolan, 2004)

4.2 Brief Analysis of Existing Tools

Table 2 compares the popular Java Obfuscators available in market. Different obfuscators are listed along with the anti-reversing techniques that they use in order to obfuscate Java programs. It also compares their prices. This comparison is not for all the techniques that we discussed in Section 3.0; it just verifies if the tool implements at least one technique from that category.

Table 2 Comparison of Java Obfuscators

PRODUCT	**ProGuard**	**Jshrink**	**Zelix KlassMaster**	**Semantic Designs Java Obfuscator**
Price	Free	$95	$199-$399	$200-$260
Layout Obfuscation	Yes	Yes	Yes	Yes
Data Obfuscation	Yes	Yes	Yes	Yes
Control Obfuscation	No	No	Yes	No
Shrinking	Yes	Yes	Yes	No
Optimization	Yes	Yes	Yes	No

4.2.1 ProGuard

"ProGuard is a free Java class file shrinker, optimizer, obfuscator, and preverifier. It detects and removes unused classes, fields, methods, and attributes. It optimizes bytecode and removes unused instructions. It renames the remaining classes, fields, and

methods using short meaningless names. Finally, it preverifies the processed code for Java 6 or for Java Micro Edition.” (Lafortune, n.d.)

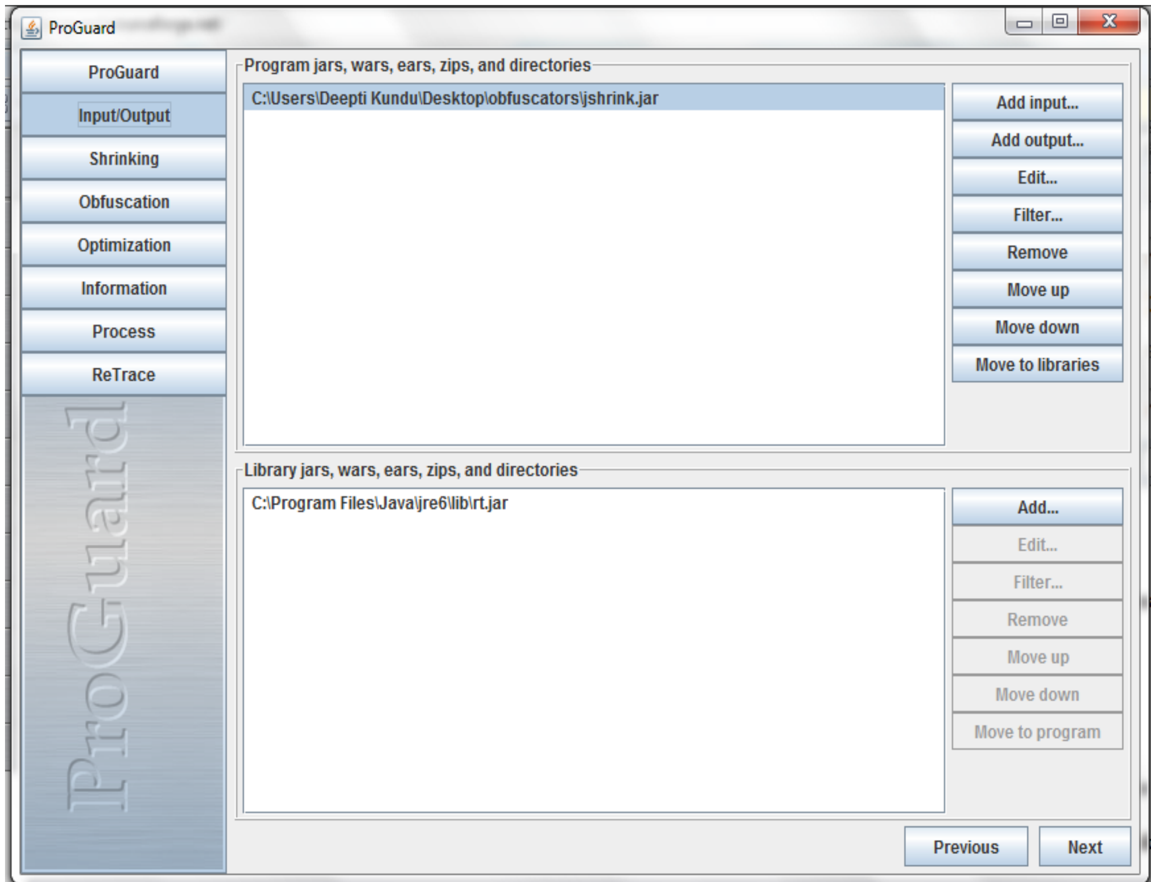

Figure 10 ProGuard

We used a simple calculator program to test all these tools. Our program was packaged in a jar file called calc.jar and this is how we will refer to it hereafter.

<u>Observations:</u>

ProGuard is available for free use under the GPL (General Public License). We used ProGuard to obfuscate calc.jar. Here is my list of observations:

- The resultant jar file (referred to as calc_proguard.jar hereafter) did not execute successfully.

- We were able to decompile cal_proguard.jar using Jad (Kouznetsov, 1997).
- After decompiling the cal_proguard.jar, we compared it with the program's original source code. The techniques used by ProGuard to obfuscate code are – layout obfuscation and data obfuscation.
- ProGuard uses name mangling to implement layout obfuscation. All the methods and variable names were converted to single alphabets like a, or b, so on. It also provides an option to generate a mapping file to print the mapping between the old names and new names for classes and class members.
- Data obfuscation was implemented by reordering the methods. The control flow was left undisturbed.
- Additionally, debugging information was removed by the tool. It is not exactly an obfuscation technique but helps improve security by removing any hints for the reverse engineer in the form of line numbers, vars, etc.
- Table 3 summarizes the obfuscation techniques implemented by ProGuard.

Table 3 ProGuard Obfuscation Observations

PROGUARD	Yes	No
Did the .jar file run after the obfuscation?		√
Name Mangling	√	
String Encryption		√
Control Flow Obfuscation		√
Reorder Methods	√	
Remove Debugging Information	√	

4.2.2 Jshrink

Jshrink is a Java obfuscator that shrinks the program size by removing unused code and data. It obfuscates symbolic names and performs optimization on the code. Jshrink does produce its results in form of a Java jar file. It comes with an inbuilt decompiler that is used to validate its results. (Eastridge Technology, n.d.)

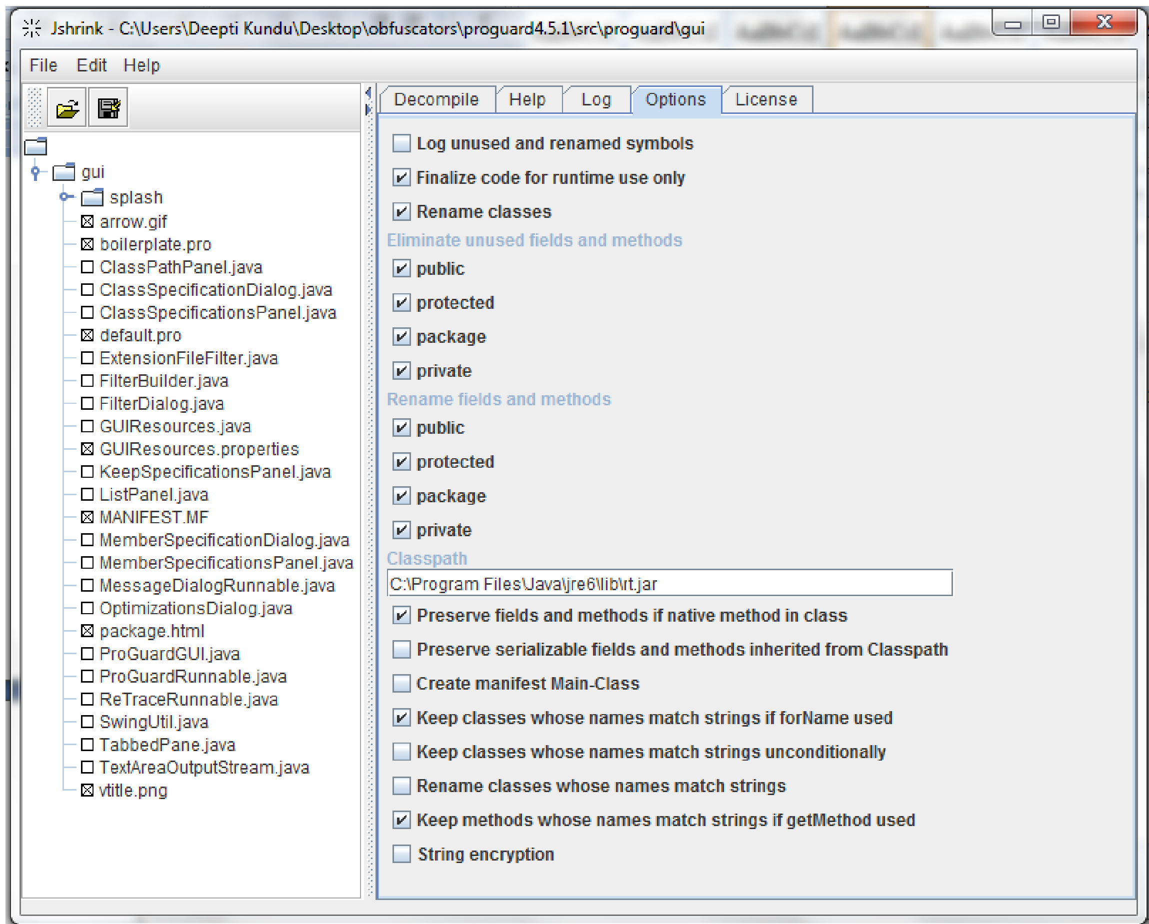

Figure 11 Jshrink

<u>Observations:</u>

An evaluation version of Jshrink is available for free, and the cost of getting a licensed copy of Jshrink is $95. We used Jshrink to obfuscate calc.jar and our observations are listed below:

- The resultant jar file (referred to as calc_jshrink.jar hereafter) did execute successfully.
- None of the methods or variables was renamed in my example. The strings were left without any encryption.

- No control flow obfuscation was implemented. The tool successfully removed all debugging information.
- Table 4 summarizes the obfuscation techniques implemented by Jshrink.

Table 4 Jshrink Obfuscation Observations

JSHRINK	Yes	No
Did the .jar file run after the obfuscation?	√	
Name Mangling		√
String Encryption		√
Control Flow Obfuscation		√
Reorder Methods		√
Remove Debugging Information	√	

4.2.3 Zelix KlassMaster

The various techniques that Zelix KlassMaster uses in order to obfuscate applications are – name obfuscation, flow obfuscation, line number scrambling, and string encryption. It also applies some shrinking, which reduces the size of the input file. Figure 12 shows the user interface provided by the tool.

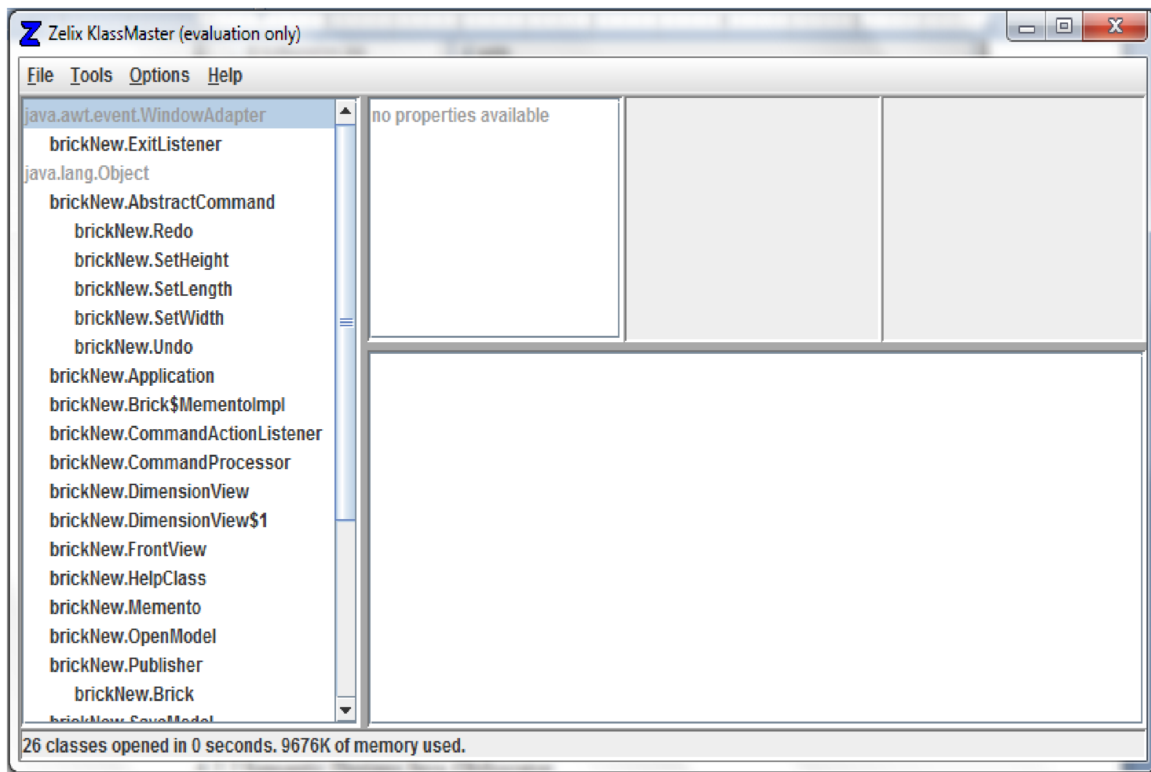

Figure 12 Zelix KlassMaster

Observations:

An evaluation version of Zelix KlassMaster is available for free, and the cost for getting a licensed version is $199-$399. We obfuscated calc.jar using Zelix KlassMaster. Here is a list of observations:

- The resultant jar file (referred to as calc_zelix.jar hereafter) did execute successfully.
- Zelix KlassMaster successfully mangled the names of the methods and the variables of the class.
- Strings which were in plain text prior to obfuscation were successfully encrypted.
- It changed the loops thus altering the control flow for functions. It did these changes to only one function in my example (It is a limitation of the trial version.

Zelix KlassMaster claims that the same code obfuscation is implemented to all the functions in the commercial version of the tool)

- The data was not restructured. It was left unaltered.
- Table 5 summarizes the obfuscation techniques implemented by Zelix KlassMaster.

Table 5 Zelix KlassMaster Obfuscation Observations

ZELIX KLASSMASTER	Yes	No
Did the .jar file run after the obfuscation?	√	
Name Mangling	√	
String Encryption	√	
Control Flow Obfuscation	√	
Reorder Methods		√
Remove Debugging Information	√	

4.2.4 Semantic Designs Java Obfuscator

The Java obfuscator from Semantic Designs is not available for trial. Semantic Designs claims that the tool scrambles the source code making it difficult to reverse engineer. The features that the tool provides are as following:

- Name mangling to replace the identifiers with meaningless names.
- Changes the structure of the code and removes all the comments – to preserve copyright information.

Figure 13 shows a section of code obfuscated by the tool (Semantic Designs, n.d.) (adopted from the company website, as the tool is not available for trial):

Before:

```
while (choice != 6) {

//if(choice!=1||choice!=2||choice!=3||choice!=4||choice!=5)
      // JOptionPane.showMessageDialog(null,"Wrong option
entered",  " error",
      // JOptionPane.ERROR_MESSAGE);
      if (choice == 1) { //calculate the area of circle
        first = JOptionPane.showInputDialog("Enter the value of
radius");
        radius = Double.parseDouble(first);
        area = Math.PI*radius*radius;
        //print out the result
        JOptionPane.showMessageDialog(null,"The area of the
Circle:    "+area,"result",JOptionPane.INFORMATION_MESSAGE);
      } else{…}
```

After:

```
while (l100 != 6) {
if (l100 == 1) {
l10 =
JOptionPane.showInputDialog("\105n\164\145\162\040t\150e\040v\14
1\154\165\145\040o\146\040r\141\144iu\163");
O101 = Double.parseDouble(l10);
l111 = Math.PI * O101 * O101;
JOptionPane.showMessageDialog(null,
"\124\150\145\040\141r\145a\040of\040\164\150e
Ci\162\143l\145\072\040\040 \040" + l111, "result",
JOptionPane.INFORMATION_MESSAGE);
} else{…}
```

Figure 13 Semantic Designs Java Obfuscators – Output

Another fact worthy of being noted here about the commercially available tools is that each tool itself implements only a subset of the available anti-reversing techniques

but none of them implements all the techniques. (Nolan, 2004) states, “Most of the Java obfuscators you’ll meet only perform layout obfuscation with some limited data and control obfuscation.”

The reason is that the automation of many of these techniques is very complicated and it has a tendency to alter the logic, and in some cases, affect the portability of the program. (Nolan, 2004) also verifies this fact, “…the main reason Java obfuscators don’t feature more high-level obfuscation techniques is because the obfuscated code has to work on a variety of Java Virtual Machines (JVMs).”

4.3 Summary

The availability of so many tools in the market clearly indicates the importance of the anti-reversing. The software community needs a strong defense against hackers as current anti-reversing techniques do not make hacking impossible; they just make it difficult and time-consuming.

5.0 PROPOSED TOOL - JShield

"All perceiving is also thinking, all reasoning is also intuition, all observation is also invention." (Rudolf Arnheim)

5.1 Introduction

Applying anti-reversing techniques is a complex procedure. It involves detailed scrutiny of the code, extracting information about its design, and making changes to the data and control flow without altering the program logic. Our tool – JShield, automates a number of obfuscation techniques discussed earlier in this paper. The automation of all the techniques is very difficult because of their complexity and limitations of the implementation language. Manual application of all the techniques is not feasible as it is time consuming and becomes unmanageable with increase in the program size and complexity.

5.2 JShield Functionality

This section outlines the functionality and features provided by JShield. The tool analyzes Java code and applies various obfuscation techniques to the code to make it harder to reverse engineer. JShield is a relatively small tool that uses an ANTLR (ANTLR, n.d.) generated parser to parse the input Java source code. "ANTLR (ANother Tool for Language Recognition) is a language tool that provides a framework for generating parser from grammatical descriptions" (ANTLR, n.d.) (See Appendix A for details). As a proof of concept for our findings, JShield currently works on a single Java file at a time and generates an obfuscated output that is remarkably difficult to reverse

engineer. It can be easily modified and extended to obfuscate an entire project containing several Java source files. Figure 14 shows a screenshot of the JShield user interface.

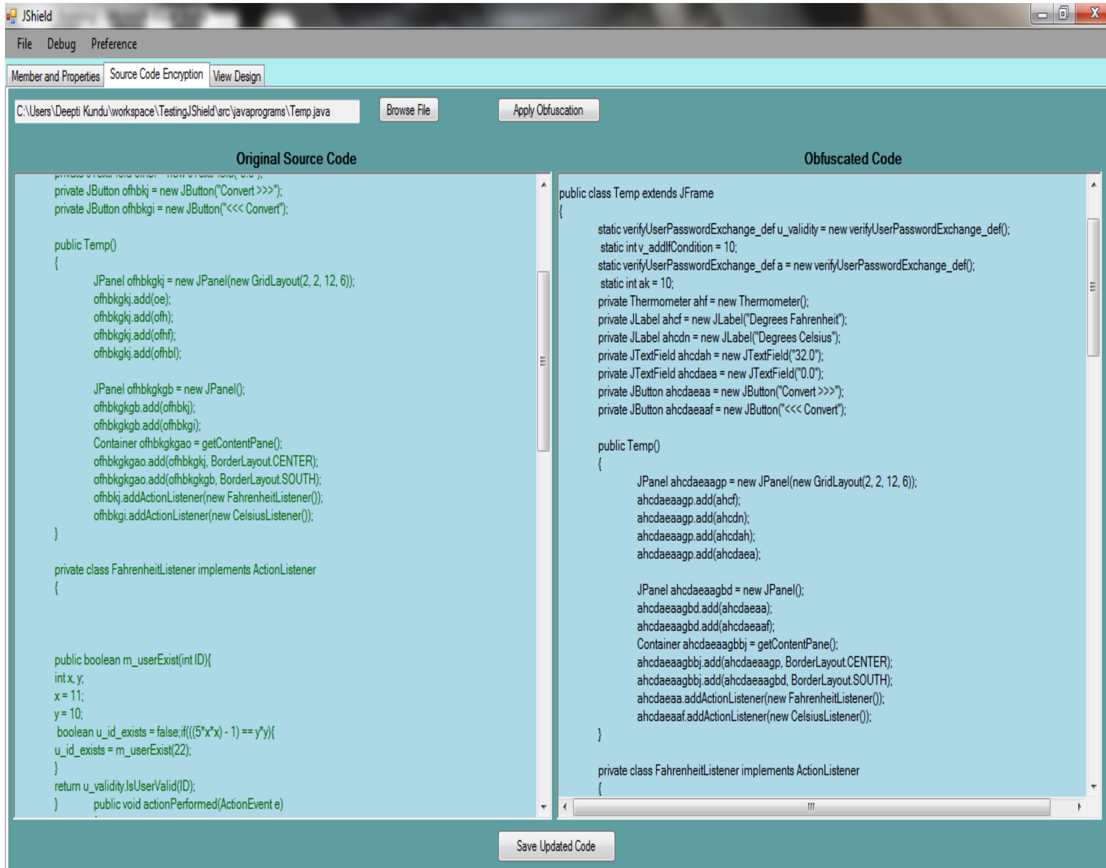

Figure 14 JShield

5.3 Techniques Implemented by JShield

The JShield code itself uses the data structures listed in Section 5.4.1. It then works based on the information generated by the parser. JShield applies the following obfuscation techniques to a Java program: (All the obfuscation techniques implemented by JShield are adopted from Section 3.2 suggested by (Collberg et al., 1997) and (Nolan, 2004)).

- Layout Obfuscation

    - Scramble identifiers
- Control Obfuscation
    - Insert dead or irrelevant code
    - Extend loop condition
    - Add redundant operands
- Data Obfuscation
    - Insert bogus class
    - Reorder methods
    - Convert static to procedural data

The algorithms for implementing each one of these obfuscation techniques are discussed in detail in Section 5.4.3.

5.4 Design and Implementation

5.4.1 Implementation Platform

JShield is implemented in C# and uses an ANTLR generated parser (ANTLR, n.d.) for parsing the program. The IDE used for development is Microsoft Visual Studio 2005. The tool applies all the obfuscation techniques in one step and gives the option of reviewing the code before it is saved. The input and output are both Java source code. As mentioned above, the tool uses various data structures for implementing different obfuscation techniques. The responsibility of these data structures is given in Table 6 below:

Table 6 Data Structures used in JShield

Data Structure Name	Type	Responsibility
importArrayList	Array List	Stores all the imports of the program
packageArrayList	Array List	Stores the package information of the java class
methodArrayList	Array List	Stores the list of all the methods implemented in the source code
variableArrayList	Array List	Stores the list of all the variables used in the program
calledMethodArrayList	Array List	Stores the list of reference objects created in the class
globalVarArrayList	Array List	This array list stores the information of all the globally declared variables
staticStringList	Array List	The static strings appearing in the code are stored in this array list and are obfuscated
mapAlteredCode	Hashtable	The mapping of the method names and variable names to their respective obfuscated values is stored in this hashtable.

5.4.2 Control Flow

The control flow of the tool is illustrated in Figure 15. The ANTLR parser is used to parse the source code. The parser class generated by ANTLR was first altered to capture information about the source code. We added functions and member variables in the parser class to capture the target program information dynamically at run time -

including method names, variable name, static strings declared in the target code, etc. The parser class holds the lists of parsed methods, variables and static strings in different data structures as described in Section 5.4.1. The MD5 hash of the static strings is generated using an inbuilt function of C#. The method and variable names are obfuscated using a random function. The mapping of the original values and the obfuscated values is stored in a hash map. The original values are then replaced with these obfuscated values in the parsed source code. After the name mangling completes, other obfuscation techniques are then applied. The examples of all the obfuscation techniques implemented are given in Section 5.4.3.

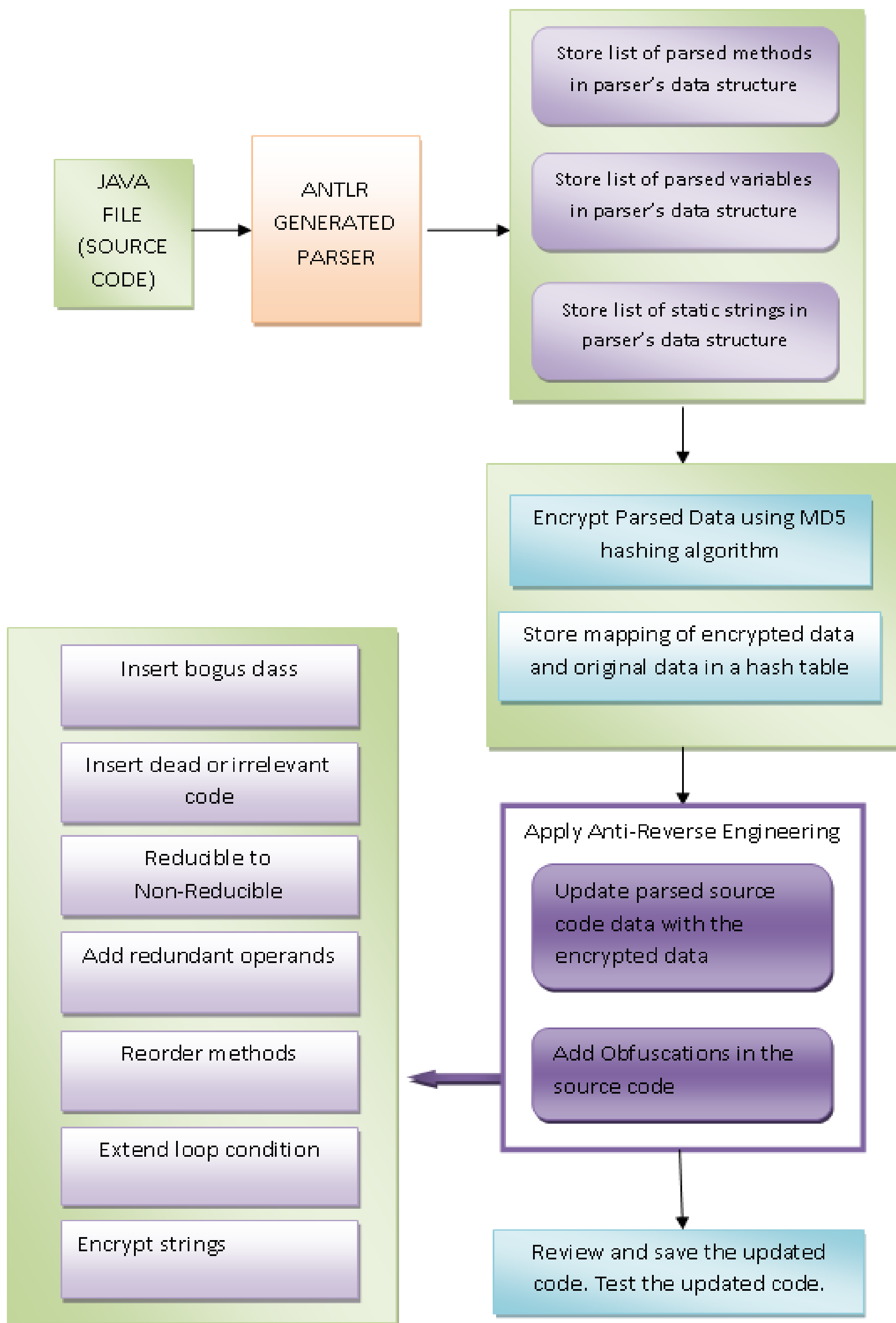

Figure 15 JShield: Control Flow

5.4.3 Algorithm and Result

5.4.3.1 Scramble Identifiers

As introduced in Section 3.2.1, scramble identifiers are used to change the names of variables and methods to meaningless strings. This makes it difficult for the reverse engineer to derive meaningful hints from the actual names given by the programmer. It targets the layout of the code and hence falls under the category of layout obfuscation (Gupta, 2005).

Intent: To replace the method and variable names with meaningless strings.

Algorithm:

1. Using he ANTLR parser (`JShieldGramParser`):
   a. Read the names of all variables and methods in two separate array lists (`variableArrayList and methodArrayList respectively`) (Roperia, 2009)
2. Create an array list (`keyList`) to hold Java keywords and library function names that are not to be renamed
3. Remove the keywords appearing in `keyList` from `methodArrayList` and `variableArrayList`
4. An inbuilt function is used to generate a random string for each name in the `methodArrayList`. The index of the method name in the list is passed to it as parameter.

5. Store the mapping of original and obfuscated names in a hash table (`mapAlteredCode`) where the original name is stored as the key and the obfuscated name is stored as the value
6. All the variable and method names are replaced in the parsed code using the mapping now present in `mapAlteredCode`

Input Validation:

For our proof of concept, the input given to JShield was the simple calculator program written in Java described earlier in chapter 4, where we documented its use in evaluating existing tools. Figure 16 shows the constructor of the input program's main java class before obfuscation. The original variable names are very helpful in understanding the program. For example, the button that clears the contents of the display is named "`clearButton`". After the program has been obfuscated by our tool, the variable names get converted to meaningless strings, as shown in Figure 17.

```
public Calculator() {
        _displayField = new JTextField("0", 12);
        _displayField.setHorizontalAlignment(JTextField.RIGHT);
        _displayField.setFont(BIGGER_FONT);
        JButton clearButton = new JButton("Clear");
        clearButton.setFont(BIGGER_FONT);
        clearButton.addActionListener(new ClearListener());
        ActionListener numListener = new NumListener();
        String buttonOrder = "789456123 0 ";
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(5, 3, 2, 2));
        for (int i = 0; i < buttonOrder.length(); i++) {
                String keyTop = buttonOrder.substring(i, i+1);
                JButton b = new JButton(keyTop);
                if (keyTop.equals(" ")) {
                        b.setEnabled(false);
                } else {
                        b.addActionListener(numListener);
                        b.setFont(BIGGER_FONT);
                }
                buttonPanel.add(b);
        }
```

Figure 16 Scramble Variable Names: Before

In the screenshot above, we can see the altered variable names. For example, "`_displayfield`" got replaced by "`k`", "`clearButton`" to "`abcfqc`", and so on.

```
public Calculator() {
        k = new JTextField("0", 12);
        k.setHorizontalAlignment(JTextField.RIGHT);
        k.setFont(BIGGER_FONT);
        JButton acbfqc = new JButton("xx_75a2aa1d339cb4be6f82324eeabdea6d");
        acbfqc.setFont(BIGGER_FONT);
        acbfqc.addActionListener(new ClearListener());
        ActionListener acbfqci = new NumListener();
        String acbfqcak = "789456123 0 ";
        JPanel acbfqcahc = new JPanel();
        acbfqcahc.setLayout(new GridLayout(5, 3, 2, 2));
        for (int i = 0; i < acbfqcak.length(); i++) {
                String acbfqcahcqg = acbfqcak.substring(i, i+1);
                JButton b = new JButton(acbfqcahcqg);
                if (v_addIfCondition%2 == 1 || acbfqcahcqg.equals(" ")) {
                        b.setEnabled(false);
                } else {
                        b.addActionListener(acbfqci);
                        b.setFont(BIGGER_FONT);
                }
                acbfqcahc.add(b);
        }
```

Figure 17 Scramble Variable Names: After

Similarly, the method names are also converted to meaningless strings. The tool takes care of not converting the constructor name and other library classes like `ActionListener`. The names of accessors and mutators are also obfuscated in order to make the code look complicated to the reverse engineer. In our simple calculator program, the method names shown in Figure 18 are obfuscated and result into names given in Figure 19. A few examples of such conversion from our test runs are presented here:

Before obfuscation	After obfuscation
CalcLogic	ljcfhfccdn
getTotalString	ljcfhfccden
setTotal	ljcfhfccdejn
Subtract	ljcfhfccdejln

```
public class CalcLogic {
        private int _currentTotal;

        /** Constructor */
        public CalcLogic() {
                _currentTotal = 0;
        }

        public String getTotalString() {
                return "" + _currentTotal;
        }

        public void setTotal(String n) {
                _currentTotal = convertToNumber(n);
        }

        public void add(String n) {
                _currentTotal += convertToNumber(n);
        }

        public void subtract(String n) {
                _currentTotal -= convertToNumber(n);
        }
```

Figure 18 Scramble Method Names: Before

```
public class ljcfhfccdn {
	private int rbfabpheeijtkbxleaqaer;

	/** Constructor */
	public ljcfhfccdn() {
		rbfabpheeijtkbxleaqaer = 0;
	}

	public String ljcfhfccden() {
		return "" + rbfabpheeijtkbxleaqaer;
	}

	public void ljcfhfccdejn(String n) {
		rbfabpheeijtkbxleaqaer = ljcfhfccdejlfjln(n);
	}

	public void add(String n) {
		rbfabpheeijtkbxleaqaer += ljcfhfccdejlfjln(n);
	}

	public void ljcfhfccdejln(String n) {
		rbfabpheeijtkbxleaqaer -= ljcfhfccdejlfjln(n);
	}
```

Figure 19 Scramble Method Names: After

5.4.3.2 Insert Dead or Irrelevant Code

As introduced in Section 3.2.2.1, insertion of dead or irrelevant code falls under the category of control obfuscation. Adding irrelevant code alters the control flow of the program thus making the program more obscure than the original (Gupta, 2005).

Intent: To insert dead or irrelevant code.

Algorithm:

1. Parse the code using the ANTLR parser (ANTLR, n.d.)
2. Create code templates to be inserted in the source code. The code templates should be syntactically correct.
3. Search the parsed code for the method signature of `main()` – the starting and ending braces. Return index of the method starting point.

4. Randomly insert one of the code templates into the target code using the index position. The junk code is inserted in first position if the class has only one method and at random position if the class has more than one method.

Input Validation:

To validate the insertion of junk code, the input given to the tool is a simple program that displays today's date. The code input is given in Figure 20. The output of the code after obfuscation is given in Figure 21.

The tool has inserted a new method "`getPassword`" and instantiated an arbitrary class "`verifyPasswordUserExchange_def()`", which was inserted by the tool. The simple program now looks complicated and it will be more time consuming to figure out the control flow of program.

```
import java.util.Date;

public class HelloDate {
   public static void main (String[] args) {
            System.out.println ("Hello, it's: ");
            System.out.println(new Date());
   }
}
```

Figure 20 Inserting Dead Code: Before

```
import java.util.Date;

public class HelloDate {
        static verifyUserPasswordExchange_def u_validity = new verifyUserPasswordExchange_def();
        int v_addIfCondition = 10;


        public String getPassword(int ID){
        String t_password = "";
        t_password = u_validity.returnPassword(ID);
        if(!t_password.equals("User not found"))
                return t_password;
        else
                return "";
        } public static void main (String[] args) {
        int v_userID = 3;
        if(v_userID == 3)

        System.out.println ("Hello, it's: ");
        System.out.println(new Date());
  }
}
```

Figure 21 Inserting Dead Code: After

5.4.3.3 Extend Loop Condition

As described in Section 3.2.2.1, the basic idea behind extending the loop condition is to make the termination condition more complex (Collberg et al., 1997). The loop condition is extended adding a predicate to the condition. The predicate should not affect the number of times the loop will execute.

Intent: To add a predicate to the loop condition

Algorithm:

1. Parse the code using ANTLR parser (ANTLR, n.d.)
2. Declare a variable in the code with a constant value:

   ```
   v_addIfCondition = 10;
   ```

3. Search the parsed code for “if” conditional loop and return the index of “(” that points to the start of the condition

4. Insert predicate "`v_addIfCondition%2 == 1`" to the loop condition with an OR operator "`||`".

   The loop condition will evaluate to the same value as before.

<u>Input Validation</u>:

For extending the loop condition we parsed the program of simple calculator using JShield. The code has a code segment with "if" condition that checks if the value of `keyTop` variable is empty as shown in Figure 22.

```
for (int i = 0; i < buttonOrder.length(); i++) {
        String keyTop = buttonOrder.substring(i, i+1);
        JButton b = new JButton(keyTop);
        if (keyTop.equals(" ")) {
                b.setEnabled(false);
        } else {
                b.addActionListener(numListener);
                b.setFont(BIGGER_FONT);
        }
        buttonPanel.add(b);
}
```

Figure 22 Extend Loop Condition: Before

After the code was obfuscated by JShield, the output is given in Figure 23. The `if` loop conditional statement now has an additional condition that is `v_addIfCondition%2 ==1`, which will always evaluate to true. The variable `v_addIfCondition` is declared by the tool and inserted with its value set to 10.

```
for (int i = 0; i < ecbbgcbk.length(); i++) {
        String ecbbgcbcili = ecbbgcbk.substring(i, i+1);
        JButton b = new JButton(ecbbgcbcili);
        if (v_addIfCondition%2 == 1 || ecbbgcbcili.equals(" ")) {
                b.setEnabled(false);
        } else {
                b.addActionListener(ecbbgce);
                b.setFont(BIGGER_FONT);
        }
        ecbbgcbcg.add(b);
}
```

Figure 23 Extend Loop Condition: After

5.4.3.4 Insert Bogus Class

We discussed in Section 3.2.3.2, inserting a bogus class increases the amount of effort reverse engineer has to put in order to understand the program. The class should appear as part of the logic of the program. To ensure this, the bogus class initialized in the main class and the function calls are made to the member functions of the bogus class. If it does not appear to be related to the logic of program, the reverse engineer will ignore it.

Intent: Insert bogus class to the program

Algorithm:

1. Parse the code using ANTLR parser (ANTLR, n.d.)
2. Create a static variable
3. Initialize the variable with a randomly generated string (using inbuilt random function).
4. Create a template bogus class and use the randomly generated static variable as its name. For example, "verifyUserPasswordExchange_def" and check if another class exists with the same name in the target source code:

    a. If no, save the template class at the location where the target program is located.
    b. If yes, repeat step 3.
5. Search for the starting index of the main function in the target program.
6. Define a static instance of the class and insert it before the given index position. For example,
7. static verifyUserPasswordExchange_def u_validity = new verifyUserPasswordExchange_def();
8. Use the instance created ("u_validity" in the example) in the inserted junk code templates to ensure that the bogus class is not deleted by a shrinker.

Input Validation:

The code for the inserted bogus class is given in Figure 24. The file is saved at the location where target program is stored. Figure 25 shows the package structure of the workspace after insertion of bogus class.

```java
class verifyUserPasswordExchange_def{
    int user_id = 22;
    String user_password = "crackMeIfYouCan";
    String _fName = "Alicia";
    String _lName = "Morgan";

    public boolean IsUserValid(int ID){
        boolean isValid = false;
        if(ID == 22)
            isValid = true;
        return isValid;
    }

    public boolean IsPasswordValid(int ID){
        boolean isPassValid = false;
        if(ID == 22)
            isPassValid = true;
        return isPassValid;
    }

    public String returnUserName(int ID){
        if(ID == 22)
            return _fName + _lName;
        else
            return "";
    }

    public String returnPassword(int ID){
        if(ID == 22)
            return user_password;
        else
            return "User Not Found";
    }
    public int generatePwdKey(int Min, int Max){
        return 2*Min*(Min+Max);
    }

    public void getUserData(int ID){
        if(ID == 22)
            System.out.println("First Name :"+ _fName+"Last Name :"+_lName);
        else
            System.out.println("No Name Found");
    }

}
```

Figure 24 Bogus Class Example

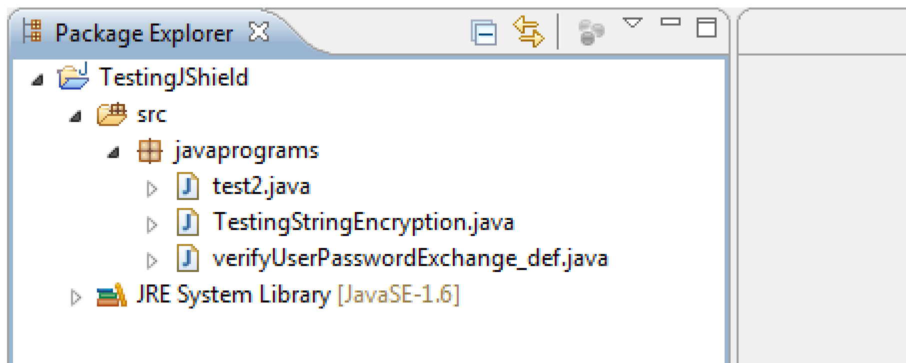

Figure 25 Eclipse Workspace: After Bogus Class Insertion

5.4.3.5 Reorder Methods

As described in Section 3.2.3.3, reordering of methods does not alter the control flow but obfuscates the program by hiding the control flow. By convention, most of the programmers do write functions in an order which makes it easy for the reverse engineer to understand. By altering the order, we can increase the time to be taken by reverse engineer to understand the logic.

Intent: Change the order of the methods in class

Algorithm:

1. Parse the code using ANTLR parser (ANTLR, n.d.)
2. Create templates of bogus methods to be inserted in the source code
3. Find out the number of methods present in the class and return starting index of each function
4. If there is only one function (main()), insert the code template before the main method, else insert the code template at random position before one of the functions

Input Validation:

The code segment from the program obfuscated using JShield is shown in Figure 26. The program has just one method i.e. `main()`, so the code template is added at the index right before the main method. The output of the code is shown in Figure 27.

```
public class TestingStringEncryption {
            public static void main(String[] args) {
                        String name1 = "Bob";
                        String name2 = new String("Bob");
                        String name3 = "Bob";

                        String username = "admin";
                        String password = "sailfish";

                        String teststring = "Having trouble in logging in?";

                        System.out.println("Welcome:"+username+"\nYour password is:"+ password);
                        System.out.println(""+teststring);
```

Figure 26 Reorder Methods: Before

```
public class TestingStringEncryption {
            static verifyUserPasswordExchange_def u_validity = new verifyUserPasswordExchange_def();
            int v_addIfCondition = 10;


            public String getPassword(int ID){
            String t_password = "";
            t_password = u_validity.returnPassword(ID);
            if(!t_password.equals("User not found"))
                        return t_password;
            else
                        return "";
            }           public static void main(String[] args) {
            int v_hash = u_validity.generatePwdKey(3,4);
            if(v_hash < 65){
            for(int v_y = 0; v_y < v_hash/2; v_y++){
                        v_y++;
                        u_validity.getUserData(v_hash);
            }
            }
                        String d = "Bob";
                        String dr = new String("Bob");
                        String ddt = "Bob";

                        String ddlo = "xx_6a4deaae9e6ac5e15cfc1a07374d311b";
                        String ddlet = "xx_47fc74fcabcf9ae2094e96bd3760895b";

                        String ddleur = "xx_fba224233487721ec6afa67740240220?";
```

Figure 27 Reorder Methods: After

5.4.3.6 Convert Static to Procedural Data

As discussed in Section 3.2.3.1, strings with important information about the program can give out a lot of hints about what a section of code is trying to achieve. It may also contain some copyright information. Such strings should be encrypted in order to protect integrity of data.

Intent: Encrypt static strings appearing in the program

4. A corresponding decrypt function is created in the bogus class.

5. All static strings are replaced by decrypt function in the input file, passing encrypted string as parameter.

Algorithm:

1. Extract all static strings present in the code during parsing and store in an array list `staticStringList`
2. Apply encryption to all the strings (We used simple Caesar cipher with a shift of 3) using `encString (string plainText)`
3. Create decryption function to decrypt the strings in the bogus class, named `str_toUpper()`
4. For each string in `staticStringList`, replace all static strings by decrypt function in the input file, passing encrypted string as parameter

   For example:

   `System.out.println("Enter year");` gets converted to:

   `System.out.println(OBJ_ANM.str_toUpper("Hqwhu#|hdu"));`

Input Validation:

```
	System.out.println("Enter year");
		BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));

		try{
			yr = Integer.parseInt(bf.readLine());
		}
		catch ( IOException io)
		{
			System.out.println("Input error");
		}
  boolean isLeapYear;
  isLeapYear = (yr % 4 == 0);
  isLeapYear = isLeapYear && (yr % 100 != 0);
  isLeapYear = isLeapYear || (yr % 400 == 0);
  if(isLeapYear==true)
	System.out.println("Its leap");
  else{
	System.out.println("Its not leap");
```

Figure 28 Before String Obfuscation

```
	System.out.println(OBJ_EAO.str_toUpper("Hqwhu#|hdu"));
		BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));

		try{
			i = Integer.parseInt(bf.readLine());
		}
		catch ( IOException io)
		{
			System.out.println(OBJ_EAO.str_toUpper("Lqsxw#huuru"));
		}
  boolean fo;
  fo = (i % 4 == 0);
  fo = fo && (i % 100 != 0);
  fo = fo || (i % 400 == 0);
  if(v_addIfCondition%2 == 1 || fo==true)
	System.out.println(OBJ_EAO.str_toUpper("Lwv#ohds"));
  else{
	System.out.println(OBJ_EAO.str_toUpper("Lwv#qrw#ohds"));
```

Figure 29 After String Obfuscation

5.4.3.7 Add Redundant Operands

As described in Section 3.2.2.1, adding some insignificant terms to the code, in the basic calculations confuses the reverse engineer. This type of obfuscation can be

done by adding some code with a conditional loop which always returns true and does not affect the functionality of the code.

Intent: Add redundant operand

Algorithm:

1. Parse the code using ANTLR parser (ANTLR,n.d.)
2. Declare two integer variables x and y in the code and initialize them to any arbitrary integer values
3. Find the index position of a function call statement and insert "`if(7x`^{2} `- 1 == y`^{2}"`)` before the call
4. The statement will be executed as per the control flow of the program

Input Validation:

The tool adds redundant operands in the form of opaque predicates. For example, in the code shown in Figure 30 the "`if`" loop has conditional statement `(7x`^{2} `- 1 == y`^{2}`)` which will always be true and hence will not alter the actual control flow of the program. If there are a lot of redundant operands in the program, the actual control flow appears more complex than without obfuscation.

```
public String getPassword(int ID){
int x, y;
x = 10;
y = 12;
String t_password = "";
if(((7*x*x) - 1) == y*y){
t_password = getPassword(22);
} else {

t_password = OBJ_COPELELATFBGLJBJOAHAFFBHHAPBBS.returnPassword(ID);
if(!t_password.equals("User not found"))
		return t_password;
else
		return "";
}
} public static void main(String[] args){
int v_hash = OBJ_COPELELATFBGLJBJOAHAFFBHHAPBBS.generatePwdKey(3,4);
if(v_hash < 65){
for(int v_y = 0; v_y < v_hash/2; v_y++){
		v_y++;
		OBJ_COPELELATFBGLJBJOAHAFFBHHAPBBS.getUserData(v_hash);
}
}
```

Figure 30 Redundant Operand: Example 1

Another example of redundant operand is given in Figure 31. The variable `v_addIfCondition` is declared as a global variable with value 10. The condition inserts an additional statement "`v_addIfCondition%2 == 1`" which will always return true and hence the loop remains unaltered.

```
for (int i = 0; i < ecbbgcbk.length(); i++) {
		String ecbbgcbcili = ecbbgcbk.substring(i, i+1);
		JButton b = new JButton(ecbbgcbcili);
		if (v_addIfCondition%2 == 1 || ecbbgcbcili.equals(" ")) {
				b.setEnabled(false);
		} else {
				b.addActionListener(ecbbgce);
				b.setFont(BIGGER_FONT);
		}
		ecbbgcbcg.add(b);
}
```

Figure 31 Redundant Operand: Example 2

5.4.4 Result Validation

Table 6 compares different tools with JShield on the basis of the different obfuscation techniques implemented by each of them. Most of the commercial tools on the market do not reveal the exact algorithms or techniques that they use in order to make the program stronger against the reverse engineering attacks. The question marks "?" in the table indicate that the tool implements at least one of the techniques listed in the category but does not reveal the details. Based on the information given by the developers of these tools, it is evident that most of the tools do not apply much of control flow obfuscation except Zelix KlassMaster.

Also, it is worth noticing that many obfuscators remove the dead code which contradicts with the basic principle of obfuscation. The data obfuscation techniques emphasize the importance of inserting bogus class and control obfuscation technique indicates the usefulness of having dead or irrelevant code. By removing the unused code from the program, we might make the job of reverse engineer easier.

5.4.4.1 Observations

- JShield implements maximum number of obfuscation techniques as compared to any other tool on the market.
- All the tools on market implement different set of techniques while JShield provides a prototype for a tool that implements most of these techniques in one place.
- JShield makes the Java code difficult to reverse engineer by applying various

obfuscation techniques. The techniques that can be implemented to enhance the tool are mentioned in Section 6.2. It is left as future work to enhance the capabilities of the tool to make it a commercially useful tool.

Table 7 Comparison of Tools

Obfuscation Techniques	JShield	ProGuard	Jshrink	Zelix KlassMaster
Layout Obfuscation	✓	✓	✓	✓
Scramble Identifiers	√	√	?	√
Control Obfuscation	✓	-	-	✓
Insert dead or irrelevant code	√	-	-	?
Extend loop condition	√	-	-	?
Reducible to non-reducible	-	-	-	?
Add redundant operands	√	-	-	?
Removing programming idioms	-	-	-	?
Parallelize code	-	-	-	?
Inline and outline methods	-	-	-	?
Interleave methods	-	-	-	?
Clone methods	-	-	-	?
Loop transformations	-	-	-	?
Reorder statements, loops, expressions	-	-	-	?
Data Obfuscation	✓	✓	-	✓
Change encoding	-	-	-	-
Split variable	-	-	-	-
Convert static to procedural data	√	-	-	√
Merge scalar variables	-	-	-	-
Factor/ Refactor class	-	-	-	-
Insert bogus class	√	-	-	-
Split/ Merge/ Fold/ Flatten arrays	-	-	-	-
Reorder methods and instance variables	√	√	-	-
Reorder arrays	-	-	-	-

5.4.4.2 User Test Statistics

JShield implements obfuscation to a given source code and produces obfuscated source code which is more difficult to reverse engineer than its original version. To test the usefulness of the tool, we performed a few usability tests. The tests were performed with seven Java developers with experience ranging from 3 years to 6 years (this ensured that they have sufficient knowledge of the language to understand the logic of the programs). Four programs of different complexity (named Complex1, Complex2, Complex3, Complex4, with the last one being the most complex) were given to each one of the users and they were timed for understanding the logic of the program. The details of the programs are given in Table 8 below and Table 9 shows the recorded times for understanding the logic of programs prior to obfuscation.

Table 8 Test Programs

Program	Level of Complexity (1-4) 4 being highest	Purpose of the Program
Complex1	1	Simple Hello World application
Complex2	2	A console game application named 21 Sticks
Complex3	3	Temperature Conversion Program with GUI
Complex4	4	Simple Calculator

After this the programs were obfuscated using JShield and the programs were given to the same users again. The time taken by each to understand the logic was recorded again, shown in Table 10.

The programs were given to the users in random order. For example, if program Complex1's obfuscated version was given first; next program might be any other obfuscated program or simply one of the four non-obfuscated versions. This was done to ensure that the users do not get a hint from the program given to them for understanding the next code given to them. The understanding of program was timed using a stopwatch. The time measured for a user for correctly understanding was the time taken by the user to correctly interpret the business logic of the program.

The user tests could not be statically used to validate the significance of results because of the small number of users. The statistics do support that the logic of the program is difficult to understand in terms of time taken to interpret the logic after obfuscation. Due to limited resources and other constraints, we could not establish any vital statistics about measurement of difficulty level to interpret the logic after obfuscation.

Table 9 User Statistics: Before Obfuscation

User	**Time taken in seconds**			
	Complex1	Complex2	Complex3	Complex4
User1	5	180	152	254
User2	8	129	129	350
User3	4	152	139	308
User4	6	120	180	406
User5	5	141	195	496
User6	5	202	184	581
User7	5	190	202	630
Average time	5.43	159.14	168.71	432.14

Table 10 User Statistics: After Obfuscation

User	Time taken in seconds			
	Complex1	Complex2	Complex3	Complex4
User1	60	579	591	1530
User2	78	450	480	1800
User3	40	802	702	1447
User4	98	590	780	1671
User5	46	705	705	1762
User6	67	650	608	1280
User7	44	880	830	1321
Average time	61.86	665.14	670.86	1544.43

The graphs below establish the time difference in understanding the original program and the obfuscated program. Each graph presents the time difference for one program.

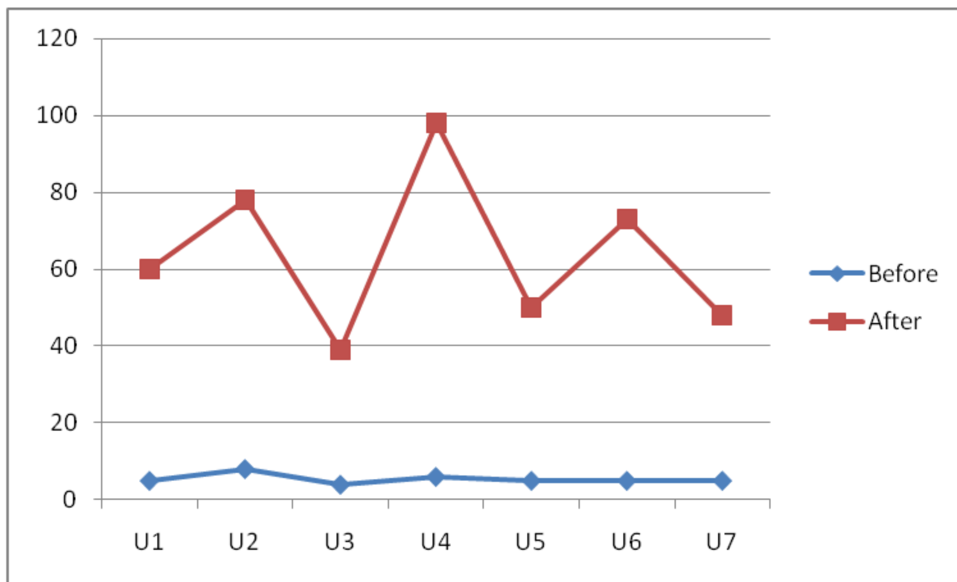

Figure 32 User Statistics for Complex1

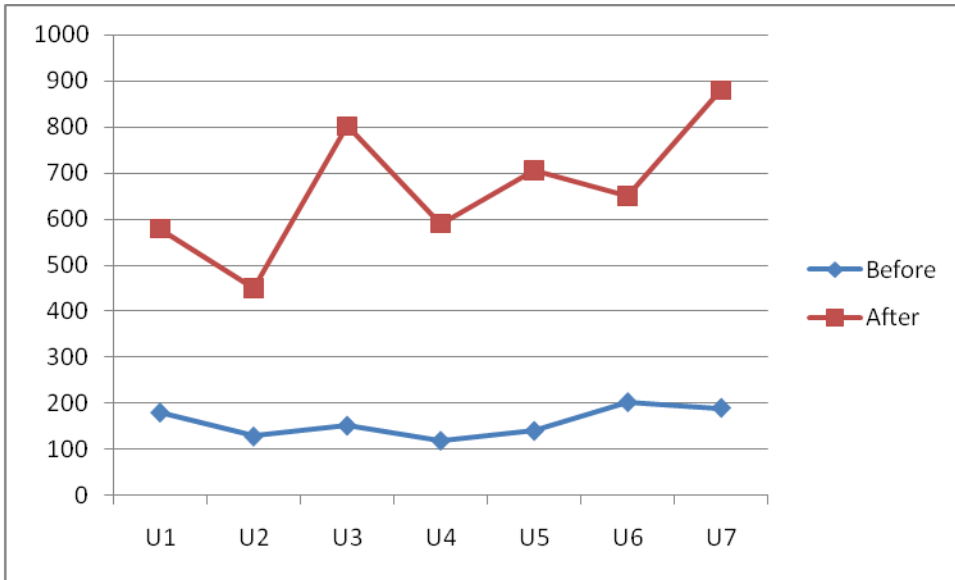

Figure 33 User Statistics for Complex2

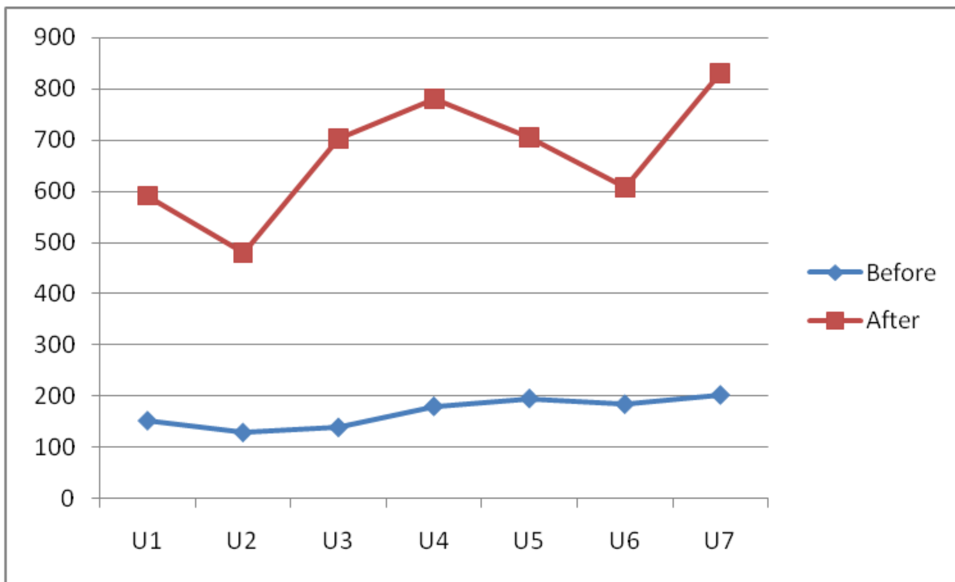

Figure 34 User Statistics for Complex3

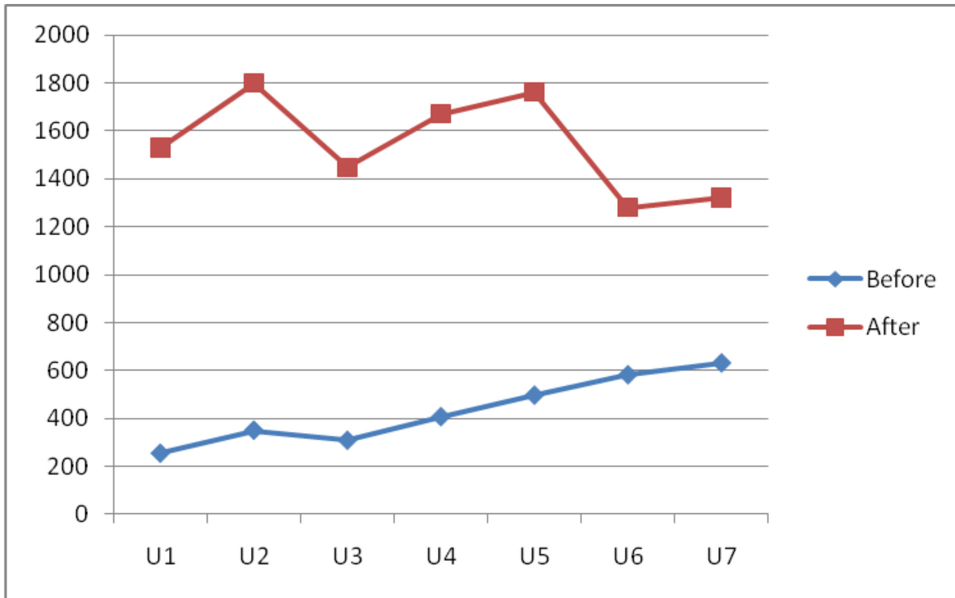

Figure 35 User Statistics for Complex4

To further analyze the statistics derived from these user tests, we divide the users into three groups depending on the years of experience that they do have. The categories thus created are 3-4 years of experience, 4-5 years, and 5-6 years. The users are listed in their respective categories here:

3- 4 years – Group1	4-5 years – Group2	5-6 years – Group3
User1, User3, User7	User2, User4	User5, User6

We do calculate the average time taken by each of the user groups for each one of the four target programs. The table below shows these calculated timings:

Table 11 Average time taken by users

	Complex1	Complex2	Complex3	Complex4
Group1	48	753.67	707.66	1432.66
Group2	88	520	630	1735.5
Group3	56.5	677.5	656.5	1521

The graph shown in Figure 36 indicates experience of the user does not introduce too much variation in the time taken by users in understanding the logic.

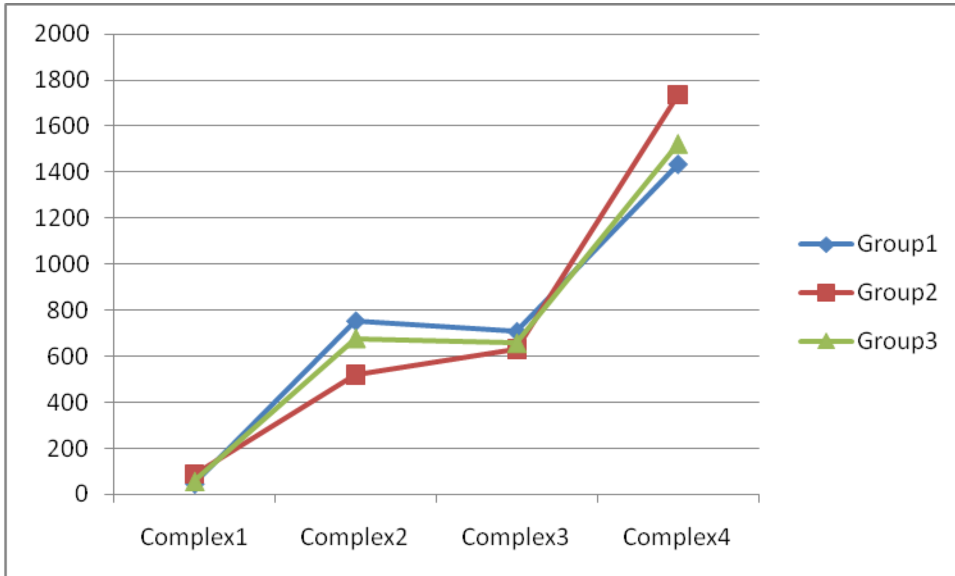

Figure 36 User Groups Statistics

6.0 CONCLUSION AND FUTURE WORK

"The future you see is the future you get."

(Robert G Allen)

6.1 Conclusion

With the availability of so many advanced tools and techniques, Java programs are vulnerable to reverse engineering attacks. The research described in this thesis has lead to the creation of a new tool to automate the application of strong anti-reversing techniques to Java programs. This effort can go a long way in addressing the problems of unauthorized access to source code and IP theft using reverse engineering attacks that the industry currently faces. In 2008, the reported loss to the software industry due to software piracy in general was $47.809 billion (Business Software Alliance, May 2008). This loss increased to $51.41 billion in May 2010 (Business Software Alliance, May 2010). As such, it might very well be impossible to eradicate it but our tool can surely make the reverse engineering effort hard and practically worthless.

In this paper, we presented the different techniques that are helpful in protecting Java software from reverse engineering attacks. We discussed the different obfuscation techniques previously developed. We identified the techniques that could be automated and then developed a prototype to demonstrate the automated application of these techniques. The obfuscation is applied to the java source code files and our tool generates an obfuscated version of the code as its output. During multiple trials and tests (Section 5.4.4.2), we verified that the functionality and performance of the program remained unaffected when compared to the version before the changes were implemented (Appendix C).

Additionally, we analyzed the existing tools on the market that address this problem - both commercial and open source. After detailed analysis, we found each of the existing tools lacking in the set of obfuscation techniques they could automate. We established a genuine gap in the market for a tool that could provide stronger protection and scope for in depth research in this field.

6.2 Future Work

The current prototype of JShield works on one Java source file at a time. A full version could be easily created by enhancing the prototype and that would work on an entire project containing several Java files.

Our proof of concept for this thesis implements seven obfuscation techniques in total. Further research based on this ground work would lead to automation of even more techniques and in fact, development of more advanced techniques based on future needs. Needless to say, if all the known obfuscation techniques could be automated, it would make this tool even more powerful. At the conclusion of this thesis and the accompanying research, we found that certain techniques could only be applied by human intervention while others could not be fit into our proof of concept.

The techniques that have not been implemented are listed here in the order of increasing difficulty level.

- Clone methods
- Reorder statements, loops, expressions
- Reorder/ Split/ Merge/ Fold/ Flatten arrays
- Loop transformations

- Merge scalar variables
- Factor/ Refactor class
- Inline and outline methods
- Parallelize code
- Change encoding
- Split variable
- Interleave methods
- Reducible to non-reducible
- Removing programming idioms

Out of the above listed techniques, we believe it would be most beneficial to implement clone methods, reorder expressions and loops, change the arrays and loop transformations. These techniques will make it difficult for the hacker to understand the logic behind decompiled snippets of code.

The techniques that are most difficult to automate in our opinion are removing programming idioms, reducible to non-reducible, and interleave methods. The technique of removing programming idioms (Section 3.2.2.1) is actually impossible to automate as it deals with changing the way programmers write their code in the first place. The technique of converting reducible to non-reducible (Section 3.2.2.1) is mostly applicable in case of bytecode obfuscation. JShield works with obfuscating the source code and hence it is not possible to add code which is syntactically unacceptable to the compiler. The difficulty in automating the interleave methods (Section 3.2.2.2) is that it needs extensive understanding of the business logic to manipulate the code to interleave two

methods. If the business logic is altered incorrectly, it might affect the functionality of the software.

References

ANTLR (n.d.) Retrieved from the ANTLR website: http://www.antlr.org/

Antoniol, G., Casazza, G., Penta, M.D., & Fiutem, R. (2001, November 15). Object-oriented design patterns recovery, Journal of Systems and Software, Volume 59, Issue 2, Pages 181-196, ISSN 0164-1212, DOI: 10.1016/S0164-1212(01)00061-9. [Electronic version] http://www.sciencedirect.com/science/article/B6V0N-449TJ06-J/2/1194eca49fa9a9d8dbafde4af2041130

Baker, B.S. (1995, July). On finding duplication and near-duplication in large software systems. In proceedings of the Working Conference on Reverse Engineering, pages 86-95

Baxter, I.D., Bier, L., Moura, L., Sant'Anna, M., and Yahin, A. (1998). Clone detection using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368-377

Belur, S. & Bettadapura, K. (2006). *Jdec: Java Decompiler*. Retrieved November 20, 2010 from: http://jdec.sourceforge.net/

Benedusi, P., Cimitile, A., & Carlini U.D. (1992, November). Reverse engineering processes, design document production, and structure charts. Journal of Systems and Software, Volume 19, Issue 3, Pages 225-245, ISSN 0164-1212, DOI: 10.1016/0164-1212(92)90053-M

Biggerstaff, T.J. (1989, July). Design Recovery for Maintenance and Reuse. IEEE Computer

Business Software Alliance (May 2008). Fifth Annual BSA and IDC Global Software Piracy Study. Retrieved on February 2, 2011 from BSA website: http://portal.bsa.org/idcglobalstudy2007/studies/summaryfindings_globalstudy07.pdf

Business Software Alliance (May 2010). Seventh Annual BSA and IDC Global Software Piracy Study. Retrieved on February 2, 2011 from BSA website: http://portal.bsa.org/globalpiracy2009/studies/globalpiracystudy2009.pdf

Byrne, E. (1991). Software reverse engineering. Software – Practice and Experience, 21(12):1349-1364.

Canfora, G., Cimitile, A., Lucia, A. De, & Lucca, G. A. Di (2000). Decomposing legacy programs: a first step towards migrating to client-server platforms. Journal of Systems and Software, 54(2):99-110.

Canfora, G. & Di Penta, M. (2007, May). New Frontiers of Reverse Engineering. In *2007 Future of Software Engineering* (May 23 - 25, 2007). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 326-341. DOI= http://dx.doi.org/10.1109/FOSE.2007.15

Chen, Y., Fu, B., & Richard III, G. (2006, March). Some New Approaches For Preventing Software Tampering. *ACM SE'06* March 10-12, Melbourne, Florida, USA

Chikofsky, E.J.; Cross II, J.H. (1990). Reverse Engineering and Design Recovery: A Taxonomy in IEEE Software [Electronic version]. *IEEE Computer Society*: 13–17. http://seal.ifi.uzh.ch/fileadmin/User_Filemount/Vorlesungs_Folien/Evolution/SS05/chikofsky90.pdf

Cipresso, T. (2009). Software Reverse Engineering Education. *Master's thesis*, San Jose State University, CA. [Electronic version] Retrieved December 3, 2010, from Software Reverse Engineering (SRE) – Web supplement to Master's thesis: http://reversingproject.info/wp-content/uploads/2008/10/cipresso_teodoro_cs299_report.pdf

Collberg, C., Low, D., & Thomborson C. (1997). A Taxonomy of Obfuscating Transformations. *Technical Report.* Department of Computer Science, University of Auckland, New Zealand. Retrieved October 21, 2010 from http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a/A4.pdf

Collberg, C. & Thomborson, C. (2002). Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. IEEE traction on software engineering 8(28), pages 735- 746

Cordy, J.R. & Roy, C.K. (2007, September). A Survey on Software Clone Detection Research. Retrieved 6 March 2011 from http://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf

Cordy, J.R. & Roy, C.K. (n.d.). Scenario-Based Comparison of Clone Detection Techniques. School of Computing, Queen's University, Kingston, ON

DaCosta, D., Dahn, C., Mancoridis, S., & Prevelakis, V. (2003). Characterizing the 'security vulnerability likelihood' of software functions. In ICSM, pages 266-275. IEEE Computer Society

Digital Fingerprint (n.d.). Business Dictionary. Retrieved July 24, 2010 from: http://www.businessdictionary.com/definition/digital-fingerprint.html

Doorn, L.V., Kravitz, J., & Safford, D., (2003). Take control of TCPA, Linux Journal. [Electronic version] Volume 2003 Issue 112. Retrieved October 31, 2010 from http://www.linuxjournal.com/article/6633

Easterbrook, S.M., Holt, R.C., and Elliot Sim, S. (2003). Using benchmarking to advance research: A challenge to software engineering. In *proceedings of the 25th International Conference on Software Engineering* (ICSE 2003), May 3-10, Portland, Oregon, pages 74-83

Eastridge Technology (n.d.). Jshrink: Java Shrinker and Obfuscator. Retrieved March 20, 2010 from Eastridge Technology website: http://www.e-t.com/jshrink.html

Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering*. Indianapolis, Indiana: Wiley Publishing, Inc.

Emden, E.V. & Moonen, L. (2002, November). Java quality assurance by detecting code smells. In Ninth Working Conference on Reverse Engineering (WCRE 2002), Richmond, VA, USA, pages 97-107. DOI:10.1109/WCRE.2002.1173058

Ernst, M.D. (2003, May). Static and dynamic analysis: synergy and duality. In *Proceedings of WODA 2003*, pages 6-9, Portland, Oregon

Feng, L., Maletic, J.I., and Marcus, A. (2003, May). Source Viewer 3D (sv3D) – a framework for software visualization. In *proceedings of the 25th International Conference on Software Engineering* (ICSE 2003), May 3-10, Portland, Oregon, pages 812-813

Foley, S. (2009). IBM Real Time Application Execution Optimizer for Java. *A tool that operates on compiled Java applications to optimize and verify application deployment in specialized environments.* Retrieved January 4, 2011 from IBM website: http://www.alphaworks.ibm.com/tech/javaoptimizer

George, N. & Glafkos, C. (2008) Reverse Engineering: Anti-Cracking Techniques [Electronic version] Retrieved November 5, 2009, from Net Security website: http://www.net-security.org/dl/articles/Reverse.Engineering.AntiCracking.Techniques.pdf

Google directory (n.d.). List of Java Obfuscators. Retrieved February 14, 2010: http://www.google.com/Top/Computers/Programming/Languages/Java/Development_Tools/Obfuscators/

Guilfanov, I. (n.d.). The IDA Pro Disassembler and Debugger. Retrieved November 14, 2010 from http://www.hex-rays.com/idapro/

Gupta, S. (2000). Code Obfuscation. *Article published in Palisade - Application Security Intelligence magazine.* August 2005. Retrieved October 20, 2010: [Electronic version] http://palisade.plynt.com/issues/2005Aug/code-obfuscation/

Haggar, P. (2001). Java bytecode: Understanding bytecode makes you a better programmer. [Electronic version] Retrieved October 21, 2010 from http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode

Heuzeroth, D., Holl, T., Högstorm, G., and Löwe, W. (2003). Automatic design pattern detection. In *11th International Workshop on Program Comprehension* (IWPC 2003), Portland, Oregon, USA, pages 94-103

Hoenicke, J. (2002). *JODE – Decompiler and Optimizer for Java*. Retrieved November 20, 2010, from http://jode.sourceforge.net/

Inoue, K., Kamiya, T., and Kusumoto, S. (2002, July). CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. Retrieved 6 March 2011 from http://www.cs.drexel.edu/~spiros/teaching/CS675/papers/clone-kamiya.pdf

javac – The Java Compiler (n.d.). The Java(tm) Tools Reference Pages. Telemedia, Networks, and Systems Group, MIT Laboratory for Computer Science, Cambridge, MA. Retrieved December 29, 2010 from Telemedia, Networks, and Systems Group's website: http://www.tns.lcs.mit.edu/manuals/java-tools-old/javac.html

jGuru (2000). What is preverification? Forum discussion. Retrieved October 14, 2010: http://www.jguru.com/faq/view.jsp?EID=201507

Kalinovsky, A. (2004). Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering. Bedford, UK: Sam Publications

Kazman, R., Woods, S. S., & Carri`ere, S. J. (1998). Requirements for integrating software architecture and reengineering models: Corum II. *In Proceedings of the Working Conference on Reverse Engineering*, pages 154–163

Koschke, R. (2000). Atomic Architectural Component Recovery for Program Understanding and Evolution. *Ph.D. thesis*, University of Stuttgart, Germany.

Kouznetsov, P. (1997). Jad – *the fast Java decompiler*. Retrieved December 3, 2010, from http://web.archive.org/web/20080214075546/http://www.kpdus.com/jad.html#email

Lafortune, E. (n.d.). ProGuard. *A Java class file Shrinker, optimizer, obfuscator, and preverifier.* Retrieved March 28, 2010: http://proguard.sourceforge.net/

Leblanc, C., Mayrand, J., and Merlo, E. (1996, November). Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance,* pages 244-253, Monterey, CA

Lemay, L. & Perkins C.L. (1996). Teach Yourself JAVA in 21 Days. Indianapolis, Indiana: Sams.net Publishing, Inc.

Low, D. (1998). Java Control Flow Obfuscation. Master's thesis. University of Auckland, Auckland, New Zealand.

Low, D. (1998). Protecting Java Code Via Code Obfuscation. ACM Crossroads, Spring 1998 issue. Retrieved from The University of Arizona website on June 30, 2010: http://www.cs.arizona.edu/~collberg/Research/Students/DouglasLow/obfuscation.html

Machine code (2010). Wikipedia. Retrieved December 4, 2010, from http://en.wikipedia.org/wiki/Machine_code

Marziali, A. (n.d.) Code Crawler. A tool for assisting code review practitioners. Retrieved December 24, 2010, from OWASP Code Crawler website: http://codecrawler.codeplex.com/

Memon, A.M., Banerjee, I., & Nagarajan, A. (2003, November). GUI ripping: Reverse engineering of graphical user interfaces for testing. In Tenth Working Conference on Reverse Engineering (WCRE 2003), 13-16 November, Victoria, Canada, pages 260-269

Merlo, E., Gagne, P.-Y., Girard, J.-F., Kontogiannis, K., Hendren, L.J., Panangaden, P., & Mori, R. de (1995). Reengineering user interfaces. IEEE Software, 12(1):64-73

Moonen, L. (2001). Generating robust parsers using island grammars. *In Proceedings of the Working Conference on Reverse Engineering*, pages 13–22

Moore, M. (1998). User Interface Reengineering, *Ph.D. thesis*, Georgia Institute of Technology, USA

Muller, H.A., Storey, M.D., Tilley, S., and Wong, K. (1995, January). Structural redocumentation: A case study. IEEE Software, pages 46-54

Nolan, G. (2004). *Decompiling Java*. Chapter 4 – Protecting Your Source: Strategies for Defeating Decompilers, pages 79 – 210. New York, USA: Springer-Verlag New York, Inc.

Obfuscated code (2010). Wikipedia. Retrieved November 4, 2010, from http://en.wikipedia.org/wiki/Code_obfuscation

Parr, T. (2007). The Definitive ANTLR Reference. Building Domain-Specific Languages. The Pragmatic Programmers, LLC

Potrich, A. and Tonella, P. (2005). Reverse Engineering of Object Oriented Code. Springer-Verlag, Berlin, New York

Roperia, N. (2009, May). JSMELL: A BAD SMELL DETECTION TOOL FOR JAVA SYSTEMS. Master's thesis – California State University, Long Beach. Retrieved on August 10, 2010 from: http://gradworks.umi.com/1466306.pdf

Rugaber, S. (2000). The use of domain knowledge in program understanding. Reverse Engineering Group at Georgia Institute of Technology, Atlanta. Retrieved on August 30, 2010 from: http://www.cc.gatech.edu/reverse/papers.html

Semantic Designs (n.d.). Java Source Code Obfuscator. Retrieved June 19, 2010 from Semantic Designs company website: http://www.semanticdesigns.com/Products/Obfuscators/JavaObfuscator.html

SoftICE (n.d.). *Wikipedia*. Retrieved December 3, 2010, from http://wiki.laptop. org/go/OLPC_Peru/Arahuay

Sogiros, J. (n.d.). Code Obfuscation Techniques against Strong Software Protection Techniques. Security article published on sooperarticles.com. Retrieved on December 30, 2010 from: http://www.sooperarticles.com/technology-articles/security-articles/code-obfuscation-techniques-against-strong-software-protection-techniques-67981.html

Stamp, M. (2006). *Information Security: Principles and Practices*. New Jersey: John Wiley & Sons, Inc.

Swartz, F. (2007). Java: Example – Simple Calculator. Retrieved on June 19, 2009 from: http://leepoint.net/notes-java/examples/components/calculator/calc.html

Systä, T. (2000). Static and Dynamic Reverse Engineering Techniques for Java Software Systems. PhD thesis, University of Tampere, Finland

The Code Project (n.d.) An Anti-Reverse Engineering Guide. Retrieved October 10, 2009, from The Code Project website: http://www.codeproject.com/KB/security/AntiReverseEngineering.aspx

Yuschuk, O. (2000). *OllyDbg*. Retrieved December 3, 2010, from http://www.ollydbg.de/

Zelix KlassMaster: HEAVY DUTY PROTECTION (n.d.). *Java Bytecode Obfuscator*. Retrieved April 20, 2010: http://www.zelix.com/klassmaster/features.html

APPENDIX A: ANTLR Parser

What is ANTLR?

"ANTLR, ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. ANTLR provides excellent support for tree construction, tree walking, translation, error recovery, and error reporting." (ANTLR, n.d.)

How ANTLR works?

ANTLR provides a grammar development environment, developed by Jean Bovet, called ANTLRWorks (The ANTLR GUI Development Environment). ANTLRWorks combines an editor and an interpreter which helps in rapid prototyping (Parr, 2007). ANTLRWorks needs Java 1.5 or later to execute. We used version 1.1.3 of ANTLRWorks.

Figure 37 shows the high level interface of ANTLRWorks. ANTLRWorks helps in understanding the rules of grammar by providing syntax diagrams of the grammar rules. ANTLRWorks can generate parsers for multiple target languages like Java, C#, C++, and Python (ANTLR, n.d.).

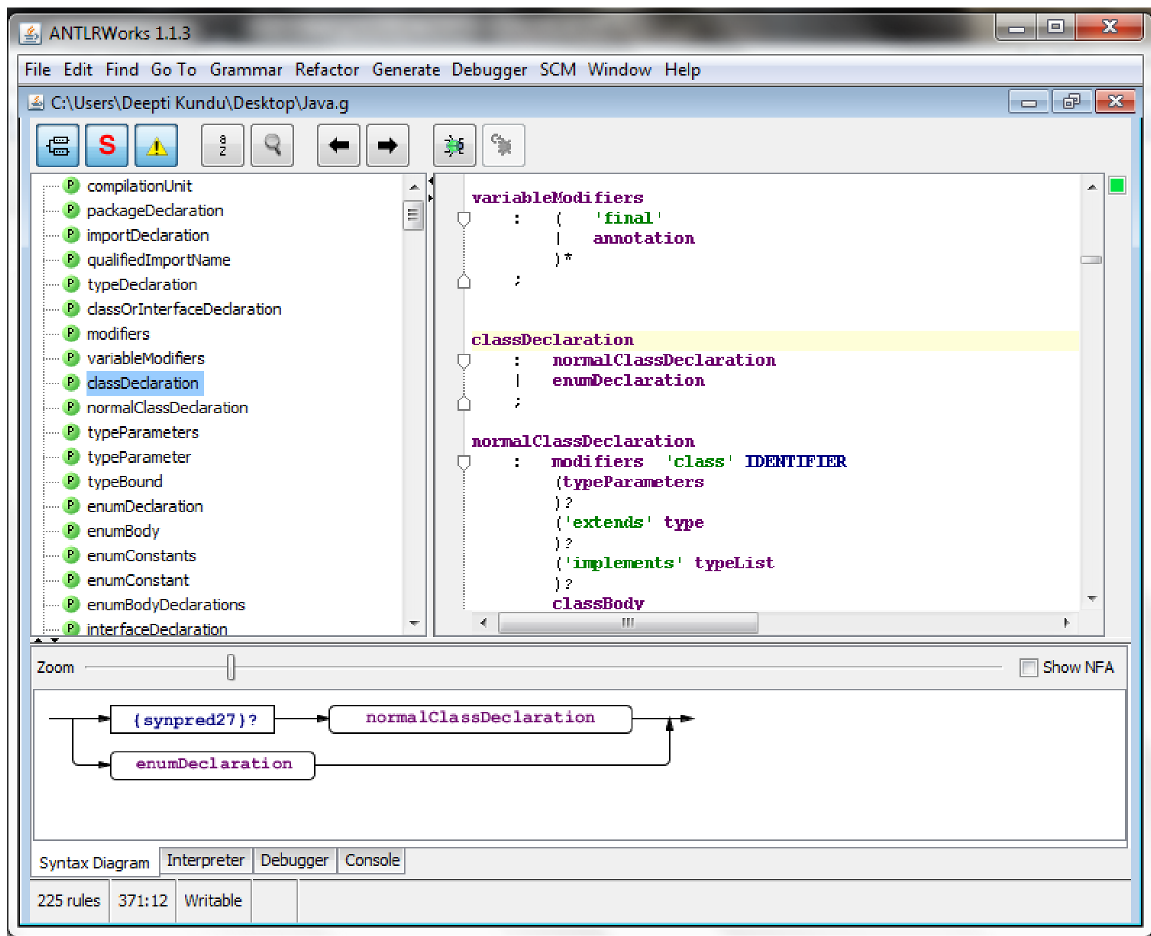

Figure 37 ANTLRWorks Interface

We used ANTLRWorks to generate a parser in C#. The code generated by ANTLRWorks is integrated into Microsoft Visual Studio 2005. ANTLRWorks takes Java grammar as input and generates parser and lexer classes in C# thus making it easy to integrate the parser.

We declare data structures in the grammar to capture information about the target class. Figure 38 shows a code snippet of one such addition. The code added to the parser is to capture the required information about the code at runtime. When the Java code is

parsed, we capture information about the methods, variables, etc. and store it in data structures, as explained in Section 5.4.1.

```
importDeclaration
@init{String ids = "";}
        :       'import' 'static'? id=Identifier{ids+=$id.Text;} (id='.'{ids+=$id.Text;}
                        id=Identifier{ids+=$id.Text;})* (id='.'{ids+=$id.Text;} id='*'{ids+=$id.Text;})? ';'
                        {imports++;importArrayList.Add(ids); classProperty += "\r\nImported Packages:  "+ids;}
        ;
```

Figure 38 Code Snippet

APPENDIX B: Terminology

AST-based clone detection:

The AST-based clone detection technique was suggested by (Baxter et al., 1998). In this technique, the source code is parsed to build an abstract syntax tree (AST) and the subtrees are compared to detect clones.

Clones (in software systems):

“Copying code fragments and then reuse by pasting with or without minor modifications or adaptations are common activities in software development. This type of reuse approach of existing code is called code cloning and the pasted code fragment (with or without modifications) is called a clone of the original.” (Cordy & Roy, 2007)

Code Smell:

The software undergoes a lot of changes during its life cycle. This may introduce some undesirable design flaws in the code. These design flaws which are introduced in the system during the maintenance phase of life cycle are called code smells. (Roperia, 2009)

Metrics-based clone detection:

“Metrics-based techniques gather a number of metrics for code fragments and then compare metrics vectors rather than code or ASTs directly. One popular technique involves *fingerprinting functions*, metrics calculated for syntactic units such as a class, function, method or statement that yield values that can be compared to find clones of these syntactic units.” (Cordy & Roy, n.d.)

Reducible & Nonreducible:

The bytecode that cannot be converted back to its original code is termed as nonreducible. For example inserting a goto statement in bytecode shall make the bytecode nonreducible as the equivalent of goto statement is not available in Java language. The reducible code is the bytecode that is converted back to its original source code with the help of a decompiler.

Token-based clone detection:

“A clone detection technique, which consists of the transformation of input source text and a token-by-token comparison.” (Inoue et al., 2002) The source code is converted into a sequence of tokens using lexical analyzer and then these sequences are matched to detect clone.

APPENDIX C: JShield Example

We use a java program of simple calculator that we use here to demonstrate the functionality of JShield and to verify that the logic of the target program remains unaltered after obfuscation using JShield. The original code of simple calculator is given in Table 12 (Program adopted from (Swartz, 2007) under MIT License).

Table 12 Simple Calculator: Before Obfuscation

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.UIManager;

class Calculator extends JFrame {

      private static final Font BIGGER_FONT = new Font("monspaced",
Font.PLAIN, 20);

      private JTextField _displayField;
      private boolean    _startNumber = true;
      private String     _previousOp  = "=";
      private CalcLogic _logic = new CalcLogic();

      public static void main(String[] args) {

            try {

      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName(
));
            } catch (Exception unused) {
                  ;
            }

            Calculator window = new Calculator();
            window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            window.setVisible(true);
      }
```

```
    public Calculator() {
        _displayField = new JTextField("0", 12);
        _displayField.setHorizontalAlignment(JTextField.RIGHT);
        _displayField.setFont(BIGGER_FONT);
        JButton clearButton = new JButton("Clear");
        clearButton.setFont(BIGGER_FONT);
        clearButton.addActionListener(new ClearListener());
        ActionListener numListener = new NumListener();
        String buttonOrder = "789456123 0 ";
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(5, 3, 2, 2));
        for (int i = 0; i < buttonOrder.length(); i++) {
            String keyTop = buttonOrder.substring(i, i+1);
            JButton b = new JButton(keyTop);
            if (keyTop.equals(" ")) {
                b.setEnabled(false);
            } else {
                b.addActionListener(numListener);
                b.setFont(BIGGER_FONT);
            }
            buttonPanel.add(b);
        }

        ActionListener opListener = new OpListener();
        JPanel opPanel = new JPanel();
        opPanel.setLayout(new GridLayout(5, 1, 2, 2));
        String[] opOrder = {"+", "-", "*", "/", "="};
        for (int i = 0; i < opOrder.length; i++) {
            JButton b = new JButton(opOrder[i]);
            b.addActionListener(opListener);
            b.setFont(BIGGER_FONT);
            opPanel.add(b);
        }

        JPanel clearPanel = new JPanel();
        clearPanel.setLayout(new FlowLayout());
        clearPanel.add(clearButton);

        JPanel content = new JPanel();
        content.setLayout(new BorderLayout(5, 5));
        content.add(_displayField, BorderLayout.NORTH );
        content.add(buttonPanel   , BorderLayout.CENTER);
        content.add(opPanel       , BorderLayout.EAST  );
        content.add(clearPanel    , BorderLayout.SOUTH );


    content.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));
        this.setContentPane(content);
        this.pack();
        this.setTitle("Simple Calc");
        this.setResizable(false);
        this.setLocationRelativeTo(null);
    }
```

```
    private void actionClear() {
        _startNumber = true;
        _displayField.setText("0");
        _previousOp  = "=";
        _logic.setTotal("0");
    }

    class OpListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            if (_startNumber) {
                actionClear();
                _displayField.setText("ERROR - No operator");
            } else {
                _startNumber = true;
                try {
                    String displayText =
_displayField.getText();
                    if (_previousOp.equals("=")) {
                        _logic.setTotal(displayText);
                    } else if (_previousOp.equals("+")) {
                        _logic.add(displayText);
                    } else if (_previousOp.equals("-")) {
                        _logic.subtract(displayText);
                    } else if (_previousOp.equals("*")) {
                        _logic.multiply(displayText);
                    } else if (_previousOp.equals("/")) {
                        _logic.divide(displayText);
                    }

                    _displayField.setText("" +
_logic.getTotalString());

                } catch (NumberFormatException ex) {
                    actionClear();
                    _displayField.setText("Error");
                }

                _previousOp = e.getActionCommand();
            }
        }
    }

    class NumListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String digit = e.getActionCommand();
            if (_startNumber) {
                _displayField.setText(digit);
                _startNumber = false;
            } else {
                _displayField.setText(_displayField.getText() +
digit);
            }
```

```
        }
    }

    class ClearListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            actionClear();
        }
    }

    public class CalcLogic {
        private int _currentTotal;

        /** Constructor */
        public CalcLogic() {
            _currentTotal = 0;
        }

        public String getTotalString() {
            return "" + _currentTotal;
        }

        public void setTotal(String n) {
            _currentTotal = convertToNumber(n);
        }

        public void add(String n) {
            _currentTotal += convertToNumber(n);
        }

        public void subtract(String n) {
            _currentTotal -= convertToNumber(n);
        }

        public void multiply(String n) {
            _currentTotal *= convertToNumber(n);
        }

        public void divide(String n) {
            _currentTotal /= convertToNumber(n);
        }

        private int convertToNumber(String n) {
            return Integer.parseInt(n);
        }
    }}
```

The execution of the original program yields a simple calculator that performs all basic mathematical operations. The screenshot for the output I given in Figure 39 below:

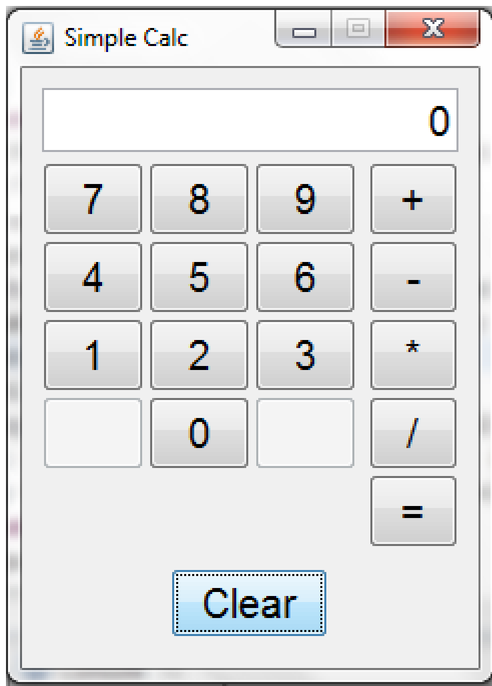

Figure 39 Simple Calculator Output

Table 13 shows the code after the program is obfuscated using JShield.

Table 13 Simple Calculator: After Obfuscation

```
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.UIManager;

class Calc extends JFrame {
	static emacglocnds OBJ_AHNAGBBQIB = new emacglocnds();
	 static int v_addIfCondition = 10;

	private static final Font BIGGER_FONT = new Font("monspaced",
Font.PLAIN, 20);

	private JTextField _displayField;
	private boolean   r = true;
```

```
    private String    te   = "=";
    private lldgfn _logic = new lldgfn();

    public static void main(String[] args) {

        try {

    UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName(
));
        } catch (Exception unused) {
            ;
        }
        Calc window = new Calc();
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setVisible(true);
    }

    public Calc() {
        _displayField = new JTextField("0", 12);
        _displayField.setHorizontalAlignment(JTextField.RIGHT);
        _displayField.setFont(BIGGER_FONT);
        JButton clearButton = new JButton("Clear");
        clearButton.setFont(BIGGER_FONT);
        clearButton.addActionListener(new ClearListener());
        ActionListener numListener = new NumListener();
        String tdi = "789456123 0 ";
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(5, 3, 2, 2));
        for (int i = 0; i < tdi.length(); i++) {
            String tdklt = tdi.substring(i, i+1);
            JButton b = new JButton(tdklt);
            if (v_addIfCondition%2 == 1 || tdklt.equals(" ")) {
                b.setEnabled(false);
            } else {
                b.addActionListener(numListener);
                b.setFont(BIGGER_FONT);
            }
            buttonPanel.add(b);
        }

        ActionListener tdklgbm = new OpListener();
        JPanel tdklgbdn = new JPanel();
        tdklgbdn.setLayout(new GridLayout(5, 1, 2, 2));
        String[] tdklgbdjh = {"+", "-", "*", "/", "="};
        for (int i = 0; i < tdklgbdjh.length; i++) {
            JButton b = new JButton(tdklgbdjh[i]);
            b.addActionListener(tdklgbm);
            b.setFont(BIGGER_FONT);
            tdklgbdn.add(b);
        }

        JPanel tdklgbdjdaaq = new JPanel();
        tdklgbdjdaaq.setLayout(new FlowLayout());
```

```
            tdklgbdjdaaq.add(clearButton);

            JPanel tdklgbdjdaajf = new JPanel();
            tdklgbdjdaajf.setLayout(new BorderLayout(5, 5));
            tdklgbdjdaajf.add(_displayField, BorderLayout.NORTH );
            tdklgbdjdaajf.add(buttonPanel    , BorderLayout.CENTER);
            tdklgbdjdaajf.add(tdklgbdn        , BorderLayout.EAST  );
            tdklgbdjdaajf.add(tdklgbdjdaaq     , BorderLayout.SOUTH );


      tdklgbdjdaajf.setBorder(BorderFactory.createEmptyBorder(10,10,10,
10));
            this.setContentPane(tdklgbdjdaajf);
            this.pack();
            this.setTitle("Simple Calc");
            this.setResizable(false);
            this.setLocationRelativeTo(null);
      }

      private void lldgn() {
            r = true;
            _displayField.setText("0");
            te   = "=";
            _logic.setTotal("0");
      }

      class OpListener implements ActionListener {


      public String getPassword(int ID){
      String t_password = "";
      t_password = OBJ_AHNAGBBQIB.returnPassword(ID);
      if(!t_password.equals("User not found"))
            return t_password;
      else
            return "";
      }      public void actionPerformed(ActionEvent e) {
       int v_userID = 3;
      if(v_userID == 3){}

                  if (r) {
                        lldgn();
                        _displayField.setText("ERROR - No operator");
                  } else {
                        r = true;
                        try {
                              String tdklgbdjdaajbi =
_displayField.getText();
                              if (te.equals("=")) {
                                    _logic.setTotal(tdklgbdjdaajbi);
                              } else if (te.equals("+")) {
                                    _logic.add(tdklgbdjdaajbi);
                              } else if (te.equals("-")) {
```

```
                                _logic.subtract(tdklgbdjdaajbi);
                        } else if (te.equals("*")) {
                                _logic.multiply(tdklgbdjdaajbi);
                        } else if (te.equals("/")) {
                                _logic.divide(tdklgbdjdaajbi);
                        }

                        _displayField.setText("" +
_logic.getTotalString());

                } catch (NumberFormatException ex) {
                        lldgn();
                        _displayField.setText("Error");
                }

                te = e.getActionCommand();
            }
        }
    }

    class NumListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String tdklgbdjdaajbcr = e.getActionCommand();
            if (r) {
                _displayField.setText(tdklgbdjdaajbcr);
                r = false;
            } else {
                _displayField.setText(_displayField.getText() +
tdklgbdjdaajbcr);
            }
        }
    }

    class ClearListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            lldgn();
        }
    }

    public class lldgfn {
        private int tdklgbdjdaajbcqb;

        /** Constructor */
        public lldgfn() {
            tdklgbdjdaajbcqb = 0;
        }

        public String getTotalString() {
            return "" + tdklgbdjdaajbcqb;
        }

        public void setTotal(String n) {
            tdklgbdjdaajbcqb = convertToNumber(n);
```

```
        }

        public void add(String n) {
                tdklgbdjdaajbcqb += convertToNumber(n);
        }

        public void subtract(String n) {
                tdklgbdjdaajbcqb -= convertToNumber(n);
        }

        public void multiply(String n) {
                tdklgbdjdaajbcqb *= convertToNumber(n);
        }

        public void divide(String n) {
                tdklgbdjdaajbcqb /= convertToNumber(n);
        }

        private int convertToNumber(String n) {
                return Integer.parseInt(n);
        }
}}
```

The output of the program after obfuscation is shown in Figure 40 below:

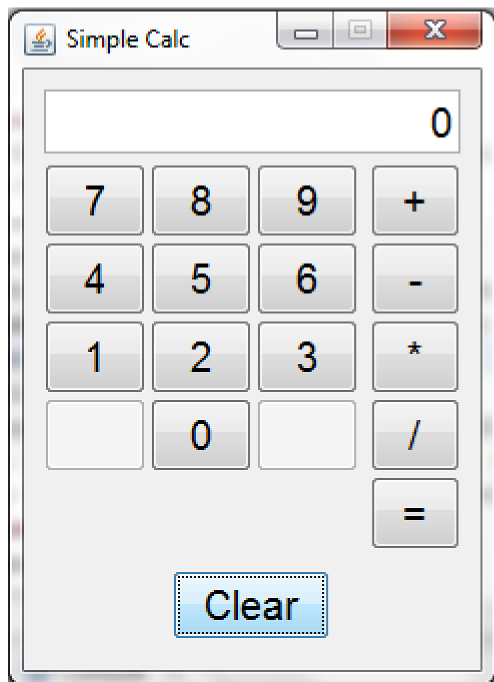

Figure 40 Obfuscated Calculator Program Output

Memory Size and Runtime Performance:

There is negligible change in the memory size of the program. For example, in case of simple calculator the memory size before obfuscation is 5,014 bytes. After the obfuscation is done, the size of the file is 5,094 bytes. Although the size of the file itself doesn't change much, the addition of the bogus class increases the size of the program. The runtime performance of the program remains unaffected as well. We tested the runtime performance by running a loop that executed 1000000 times. The time of execution before and after the obfuscation was noted to be same.