Deploying Java Platform to Design a Framework of Protective Shield for Anti– Reversing Engineering

[1]Okonta, Okechukwu Emmanuel, [2]Wemembu, Uchenna Raphael, [3]Ojugo, Adimabua Arnold), [4]Ajani, Dele

[1] Department of Computer Science Federal College of Education (Tech) Asaba.
[2] Department of Mathematics Federal College of Education (Tech) Asaba.
[3]Department of Math/Computer Sc Fed University Petroleum Resources Effurun
[4] Department of Computer Science Federal College of Education (Tech) Asaba

Abstract

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 "byte-code" instead of native binary machine code. This byte-code 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 paper addresses the problem of reverse engineering attacks on software written in Java. It analyzes the present protective techniques used to protect software from such attacks, examines their limitations and provides a new tool that implements several anti-reversing techniques. This novel tool is code named KDefender and it drew its concept from ANTLR- ANother Tool for Language Recognition.*

Key words: JAVA, Anti-Reverse Engineering, byte-code, Re-Engineering, Obfuscators, KDefender

---

Introduction

The process of extracting knowledge or design blueprints from anything man-made is known as reverse engineering [19]. So in real terms, reverse engineering may be understood as a systematic methodology for analyzing the design of an existing device or system, either as a way to study the design or as a means 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" [12].

In the field of software engineering, 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 [2], code smell detection [20], re-documentation of programs [6], renewal of user interfaces [36], [38], migration of legacy code [9], translation of program from one language to another [8], and architecture recovery [28].

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" [11], and 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" [26]. The process of reverse engineering helps in understanding the logic of software which makes it easy to alter its behaviour 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 [24].

"To reverse engineer a software application, it is first necessary to gain physical access to it" [32]. 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 [12]. 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 but practically speaking it can also be misused.

Reverse Engineering Process

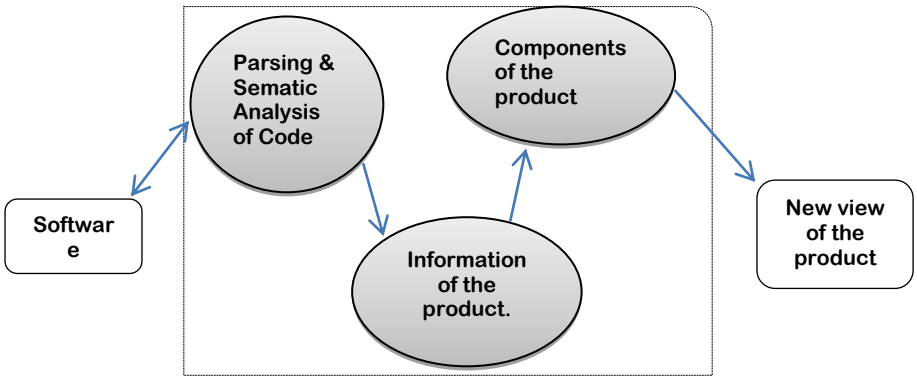

Figure 1 Reverse Engineering Process

Anti-Reverse Engineering

The protective techniques implemented in software in order to protect it from malicious attacks or blatant misuses are referred to anti-reversing techniques. It has become a challenge for the software industry 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 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 ideas and algorithms could simply be deemed as an inventive step over previous work. [26].

[19] states in his book "It is never possible to entirely prevent reversing" and [11] 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 [40]:

- Potency – How confused the de-compiler is?
- Resilience – Can it rebuff the de-compilation attempts?
- Cost – How much overhead does it cause?

Anti-Reversing Tools

Reversing is impossible without the right tool [19]. There are various software tools available in the market today; some are free while some cost thousands of naira. The tools available for reverse engineering include de-assemblers available for extracting assembly code from the executables, debuggers for dynamic analysis of code during execution, and de-compilers for generating high-level source code from the executables [11].

The most popular disassembling and debugging tools available include OllyDbg [46], IDA Pro , SoftICE), 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 byte code than it is to recover C/C++ code from machine code" [13]. Just a few of the various decompilers available include Jad [29], JODE [24], and Jdec [5].

A lot of research is going on in the software industry 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 [14], protecting the computing platform physically [17], encryption of executables [11], and watermarking [15].

Java Software: A Direct Threat

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 [13]. To understand why, we have to know the difference between Java byte-code and machine code.

- Machine code or processor instructions are a system of instructions and data executed directly by a computer's central processing unit [34]. These instructions are specific to the processor on which they are generated. Figure 2 illustrates this scenario.
- "Byte-code is a set of instructions that looks a lot like some machine code, but is not specific to any one processor" [31]. "It is the intermediate representation of Java programs just as assembler is the intermediate representation of C/C++ programs" [22]. Figure 3 illustrates the generation of byte-code.

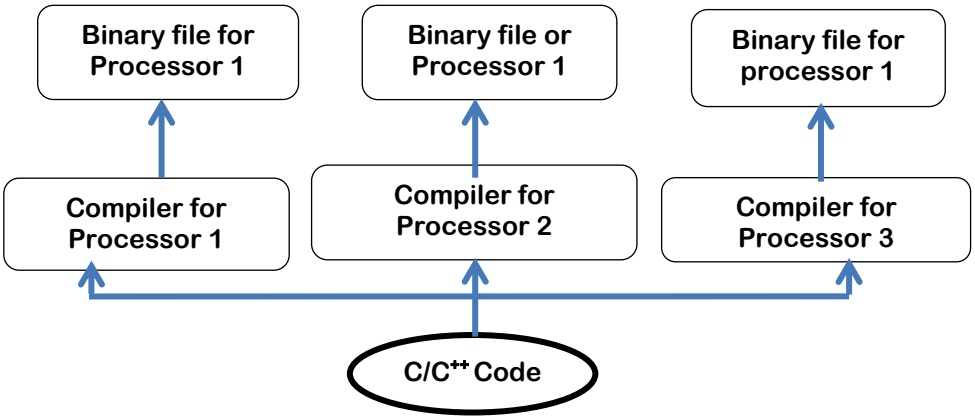

Figure 2: Machine code

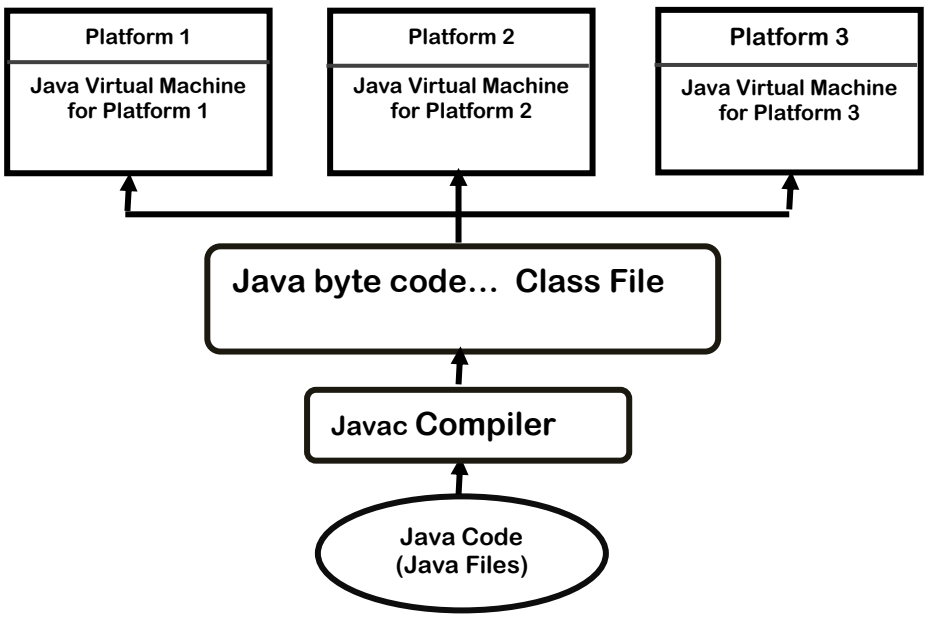

Figure 3: Generation of Byte Code

Java Byte code

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" [26]. 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 paper 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.

Previous Work done

A great deal of work and research has been done in the field of reverse engineering over the past 20 years [10]. 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.

Thus, most research during the nineties was focused on three main problems [10]:

- 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 [35], evaluating software design or examining software security [16]. [12] state 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 better replacement".

Relationship between Reverse engineering and Re-engineering

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: re-documentation and design recovery [10], as shown in Figure 4. "Re-documentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level" [12] 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" [7].

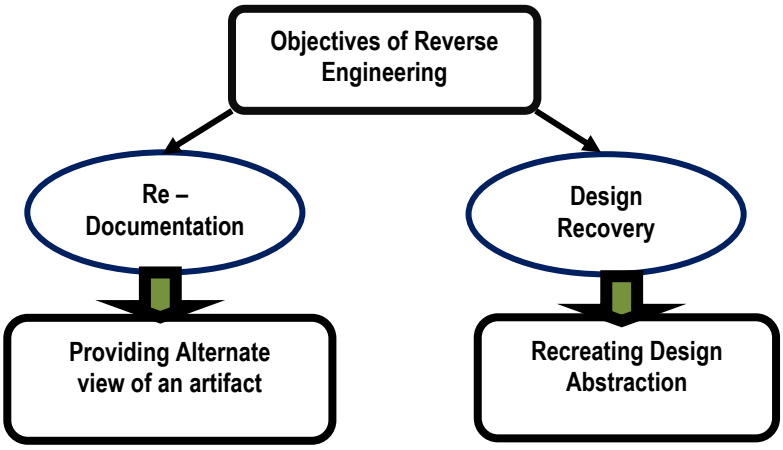

Figure 4 Objectives of Reversing Engineering

The argument given in support of this position 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.

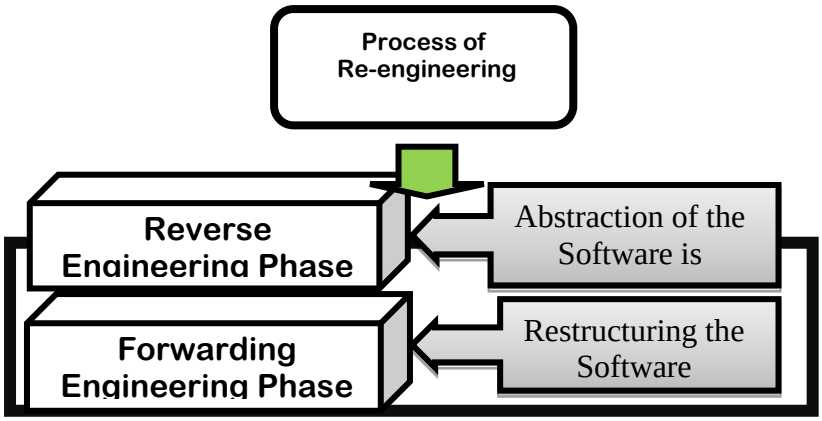

Figure 5. Re-engineering process Recovery Architecture Development

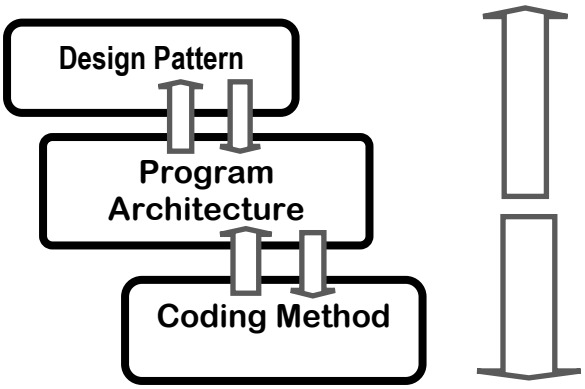

Figure 6. Architecture of Re-engineering

Figure 6 presents the Architecture Reengineering process [27]. 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. [12] gives 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 below shows the relationship between the three crucial life cycle stages.

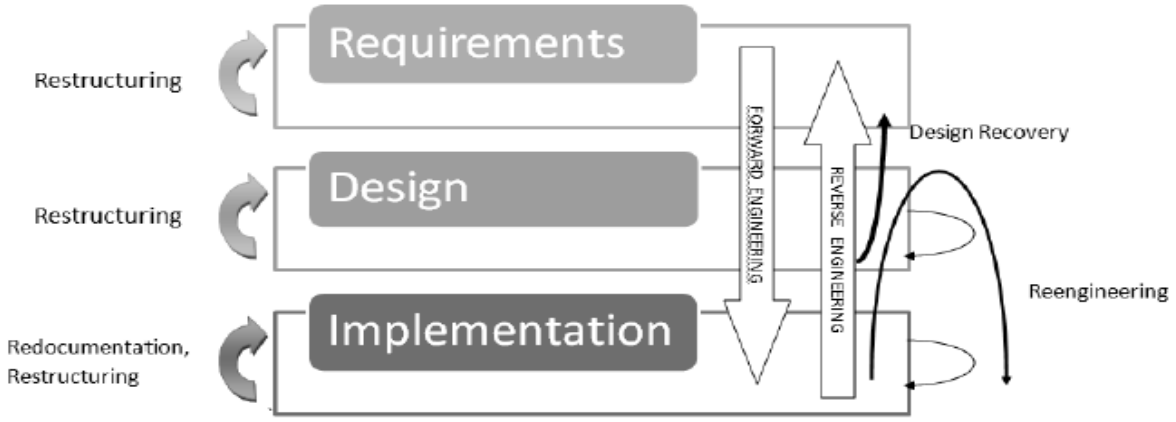

Figure 7 Relationship between the three main life cycles.

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 wanting in many programs where dynamic analysis was required [45]. Dynamic analysis is necessary in many situations and is widely used despite being expensive and incomplete. 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 [37].

Another event that inspired the research effort in the field of program analysis is the presence of clones in software systems [10]. The different techniques developed as an outcome include token-based [3], AST-based [4], and metrics-based [30] techniques.

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.

[43] 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. [45] 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 [2] and dynamic analysis techniques [23] were used to recover design patterns.

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 [10]. A number of such tools are available, like Rigi [39], CodeCrawler, Seesoft [18], and sv3D [21].

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 colour of the boxes" [10].

These advancements in the field of reverse engineering not only indicate the progress made, but also portray the pitfalls of reverse engineering. With the tools developed for the purpose of helping the software community, another set of people have been benefited – 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 original creators.

What more are we expecting?

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 de-compilers 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. And that means 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.

Anti-Reversing Techniques

To protect Java Code the software development community has been working on this problem 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

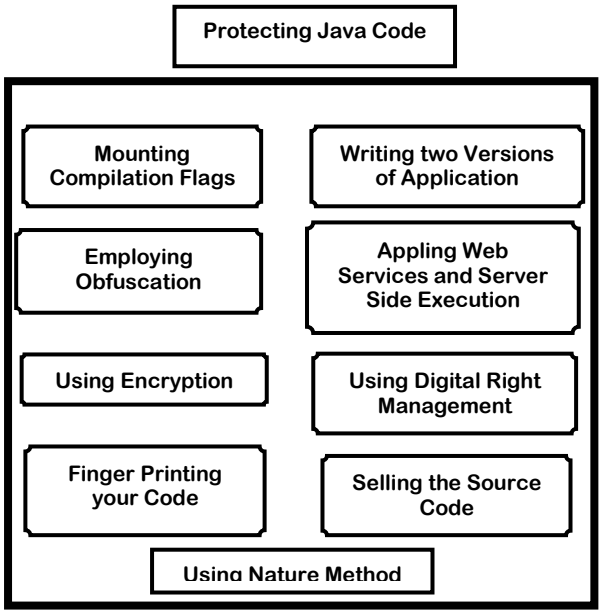

Figure 8 Protecting Java Code

Mounting Compilation Flags

The byte code generated by the compiler is affected by different types of compilation flags [40]. Use of the –g flag during compilation generates debugging tables that contain information about line numbers and local variables [25]. This information is very useful for the decompiler to retrieve the source code.

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. [40]

Applying Obfuscation

Obfuscated code is source or machine code that has been made difficult to understand for humans [41]. There are a number of techniques used to obfuscate code and it is the method used in this paper. The different techniques for obfuscation will be discussed briefly.

Encryption

Throughout the ages, mankind has turned to encryption when trying to protect secret transmissions" [40]. A common solution suggested for preventing the code from de-compilation is to encrypt the class files. These class files are not decrypted until before they are executed.

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. [40]

Fingerprinting the Code

Digital fingerprinting is a string of binary digits that uniquely identifies a file and 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. [40]

Selling Source Code

"If source code is so readily available, then why not just sell it at a higher price?" [40]. The de-compiler can be discouraged to decompile if you sell the source code directly to him.

Employing Native Methods

As we said earlier code written in Java is more difficult to protect than that written in C/C++. [40] 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.

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. These various obfuscation techniques can prove beneficial in protecting Java software from reversing attacks.. Obfuscation can be classified into three classes:

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.

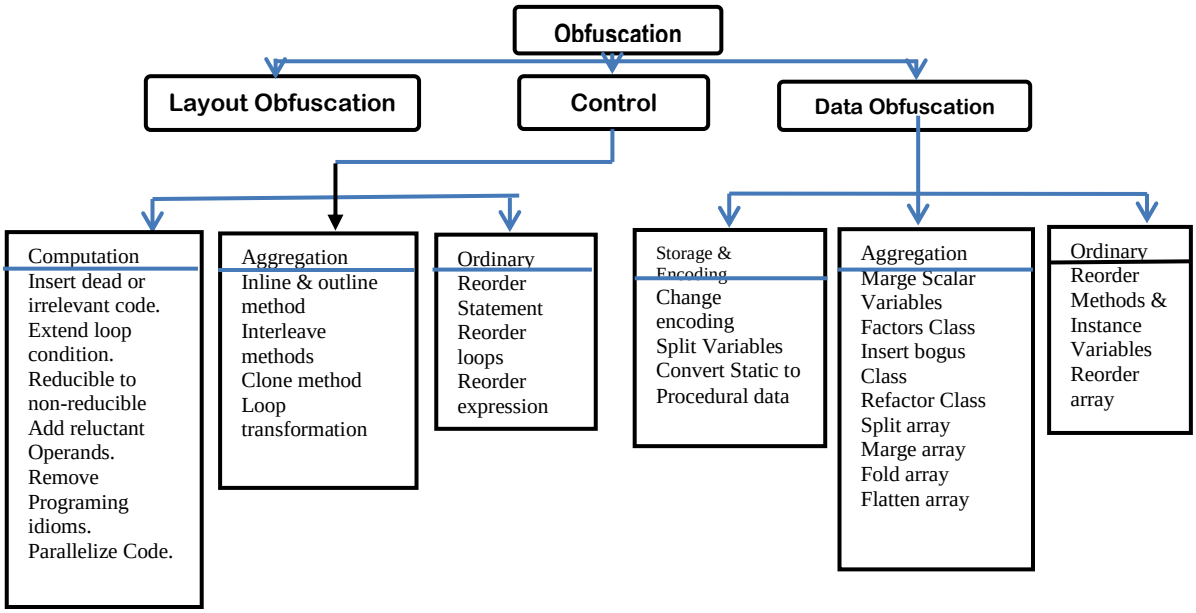

Figure 9 gives another classification of the obfuscation techniques [40], [14] based upon how the code is obfuscated.

Obfuscators available in the market work by scrambling the identifiers in the class file to make the decompiled source useless. The variables are renamed with automatically generated garbage variables which do not affect the code functionality as the class file uses pointers to methods and variables instead of actual names. It becomes difficult to understand the code but it is not impossible. A dis-assembler can be used to rename the variables in order to generate more meaningful names. [40]

Novel Framework Technique.

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 is code named **KDefender**, automates a number of obfuscation techniques. 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.

KDefender Functionality

Let us briefly outlines the functionality and features provided by KDefender. The tool analyzes Java code and applies various obfuscation techniques to the code to make it harder to reverse engineer. KDefender is a relatively small tool that uses an ANTLR [1] 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" [1]. As a proof of concept for our findings, KDefender was tested 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.

Techniques Implemented by KDefender

The KDefender code itself uses the data structures and then works based on the information generated by the parser. KDefender applies the following obfuscation techniques to a Java program: All the obfuscation techniques implemented by KDefender are adopted from suggestions made by [14] and [40] See figure 9

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 briefly discussed below

Control Obfuscation

The idea behind control obfuscation is to disguise the real control flow [32]. The control flow of the source code is altered to confuse anyone looking at the decompiled code [40]. [26] states, "The best obfuscators are capable of transforming the execution flow of bytecode by inserting bogus conditional and goto statements". [14] classifies control obfuscation into three different categories – computation, aggregation, and ordering. 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 [40]. For example, in the following example we have a simple if condition.

Before:

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

After:

```
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 Non-reducible

The Holy Grail of obfuscation is to create obfuscated code that cannot be converted back into its original format" [40]. We can devise some transformations that make the code non-reducible 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 [14]. 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;
}}

else {
while (condition1){
Statement2;
}}
```

In this algorithm, 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. [40]

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. [40] warns that using this technique all through the application has the potential to degrade its performance.

Framework Implementation

KDefender is implemented in C# and uses an ANTLR generated parser [1] for parsing the program. The IDE used for development is Microsoft Visual Studio.Net. 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.

KDefender 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 KDefender provides a prototype for a tool that implements most of these techniques in one place. KDefender 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 this paper. It is left as future work to enhance the capabilities of the tool to make it a commercially useful tool.

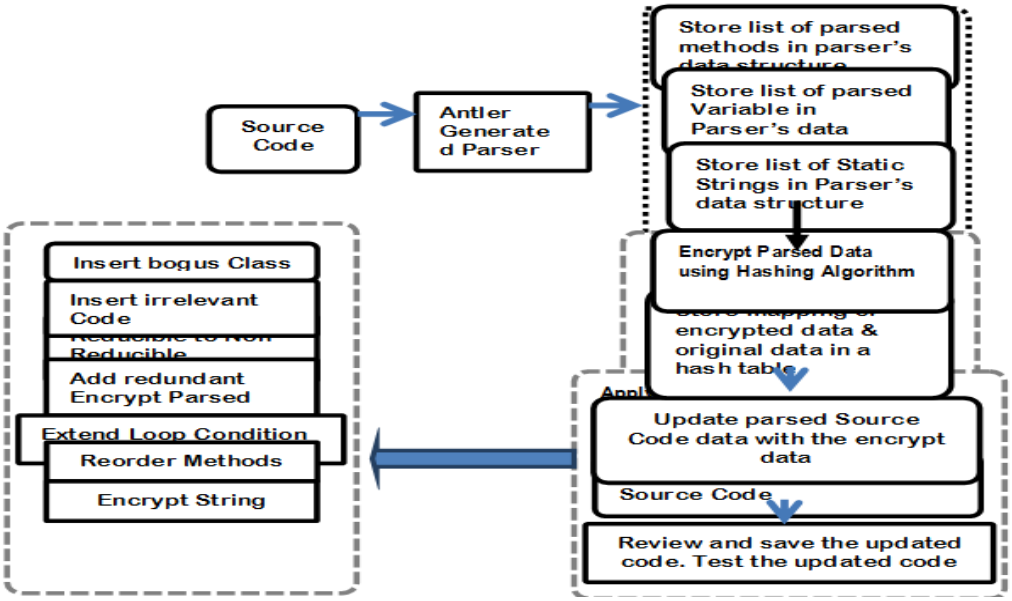

Figure 10: KDefender Implementation

Conclusion

With the availability of so many advanced tools and techniques, Java programs are vulnerable to reverse engineering attacks. The research described in this paper 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 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 can be applied to the java source code files and our tool generates an obfuscated version of the code as its output.

Recommendation

The current prototype of KDefender 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 framework implements several 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.

.

References

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

[2] Antoniol, G., Casazza, G., Penta, M.D., & Fiutem, R. (2001). Object-oriented design patterns recovery, Journal of Systems and Software, Volume 59, Issue 2,

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

[4] 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

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

[6] 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,

[7] 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:

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

[9] 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. 84

[10] 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,

[11] 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

[12] Chikofsky, E.J.; Cross II, J.H. (1990). Reverse Engineering and Design Recovery: A Taxonomy in IEEE Software [Electronic version]. *IEEE Computer*

[13] 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
[14] 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
[15] Collberg, C. & Thomborson, C. (2002). Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. IEEE traction on software engineering 8(28), pages 735- 746
[16] 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.
[17] 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
[18] 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
[19] Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering*. Indianapolis, Indiana: Wiley Publishing, Inc.
[20] 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
[21] Feng, L., Maletic, J.I., and Marcus, A. (2003). Source Viewer 3D (sv3D) – a framework for software visualization. In *proceedings of the 25th International Conference on Software Engineering* (ICSE 2003),
[22] 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
[23] 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
[24] Hoenicke, J. (2002). *JODE – Decompiler and Optimizer for Java*. Retrieved November 20, 2010, from http://jode.sourceforge.net/
[25] 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:
[26] Kalinovsky, A. (2004). Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering. Bedford, UK: Sam Publications
[27] 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
[28] Koschke, R. (2000). Atomic Architectural Component Recovery for Program Understanding and Evolution. *Ph.D. thesis*, University of Stuttgart, Germany.87
[29] Kouznetsov, P. (1997). Jad – *the fast Java de-compiler*. Retrieved December 3, 2010,
[30] Leblanc, C., Mayrand, J., and Merlo, E. (1996). 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

[31] Lemay, L. & Perkins C.L. (1996). Teach Yourself JAVA in 21 Days. Indianapolis, Indiana: Sams.net Publishing, Inc.
[32] Low, D. (1998). Java Control Flow Obfuscation. Master‟s thesis. University of Auckland, Auckland, New Zealand.
[33] 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:
[34] Machine code (2010). Wikipedia. Retrieved December 4, 2010, from http://en.wikipedia.org/wiki/Machine_code
[35] Memon, A.M., Banerjee, I., & Nagarajan, A. (2003). 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
[36] 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
[37] Moonen, L. (2001). Generating robust parsers using island grammars. *In Proceedings of the Working Conference on Reverse Engineering*, pages 13–22 88
[38] Moore, M. (1998). User Interface Reengineering, *Ph.D. thesis*, Georgia Institute of Technology, USA
[39] Muller, H.A., Storey, M.D., Tilley, S., and Wong, K. (1995). Structural re-documentation: A case study. IEEE Software, pages 46-54
[40] Nolan, G. (2004). *Decompiling Java*. Chapter 4 – Protecting Your Source: Strategies for Defeating Decompilers, pages 79 – 210. New York, USA: Springer-Verlag New , Inc.
[41] Obfuscated code (2010). Wikipedia. Retrieved November 4, 2010, from http://en.wikipedia.org/wiki/Code_obfuscation
[42] Parr, T. (2007). The Definitive ANTLR Reference. Building Domain-Specific Languages. The Pragmatic Programmers, LLC
[43] Potrich, A. and Tonella, P. (2005). Reverse Engineering of Object Oriented Code. Springer-Verlag, Berlin, New York
[44] Stamp, M. (2006). *Information Security: Principles and Practices*. New Jersey: John Wiley & Sons, Inc. 89
[45] Systä, T. (2000). Static and Dynamic Reverse Engineering Techniques for Java Software Systems. PhD thesis, University of Tampere, Finland
[46] Yuschuk, O. (2000). *OllyDbg*. Retrieved December 3, 2010, from http://www.ollydbg.de/