Analysis of Java Code Protector

September 2005 – December 2005
Submitted to Lockheed Martin Co.

Principal Investigators

Students
Sean McGinnis   ECE
Matthew Sargent ECE

Advisors
Dr. Gina Tang
Dr. Ravi Ramachandran
# TABLE OF CONTENTS

1. PROJECT GOAL ........................................................................................................3  
   1.1. BACKGROUND ..............................................................................................3  
   1.2. PROJECT SCOPE .......................................................................................4  

2. CODE PROTECTION ...............................................................................................4  
   2.1. OBFUSCATION .........................................................................................4  
      2.1.1. Control Flow Obfuscation .................................................................6  
      2.1.2. Variable and Class Renaming .........................................................7  
      2.1.3. String Encryption ............................................................................7  
      2.1.4. Class Splitting and Class Coalescing ............................................7  
   2.2. ENCRYPTION ..............................................................................................8  
   2.3. TAMPER PROOFING ................................................................................8  

3. JAVA CODE PROTECTOR ....................................................................................8  
   3.1. INSTALLATION ..........................................................................................8  
   3.2. USAGE ......................................................................................................9  
   3.3. FEATURES ................................................................................................10  
   3.4. PROBLEMS .............................................................................................10  

4. FUTURE RESEARCH ............................................................................................12  

5. ACKNOWLEDGMENTS ......................................................................................13  

REFERENCES .........................................................................................................14
1. Project Goal

The goal of this project is to evaluate the performance of Chainkey, Inc.’s *Java Code Protector (JCP)*. Specifically, qualitative measurement and statistical analysis will be used to study the JCP’s obfuscation, encryption, and integrity verification functions against state-of-the-art protection techniques available for the Java language.

1.1. Background

The use of Java is growing due to its power and flexibility. However, the language Java is designed to be compiled into a platform independent bytecode format. Much of the information contained in the source code remains in the bytecode, which can be easily accessed via decompilation. More deadly is the fact that attackers can decompile such bytecode and extract proprietary algorithms and data structures from it. Failing to deal with such malicious act of reverse engineering poses a significant threat to software industry.

There are various technical defenses against different attacks on the intellectual property of software, such as obfuscation, encryption, and tamper-proofing. Obfuscation is a viable protection against reverse engineering which transforms a program into a functionally identical one that is much more difficult to understand [1, 8]. Encryption, involving strong encryption algorithms and secret keys, shield vital information from those who are trying to gain access illegally. The security of the keys is crucial to the strength of the encryption. A defense against tampering is tamper-proofing, which fails the code whenever integrity violations of the original software are detected [3].
Analysis of Java Code Protector

JCP is a newly developed tool that combines all of the aforementioned techniques for software protection. It is our intention in the project to study this tool and provide an analytical profile for comparing and evaluating its efficiency against other similar toolsets.

1.2. Project Scope

This project is scheduled to be completed within two semesters. In the first semester, the major focus was to review last year’s Java Obfuscation study and to evaluate JCP’s interface in terms of ease-of-use and ease-of-learning. Typically, we would analyze the JCP’s obfuscation function and compare it against two obfuscators studied last year: Preemptive Solutions Inc’s DashO-Pro and Zelix’s KlassMaster. After developing competency with JCP, we will start the second phase of the project with a focus on the JCP’s encryption and integrity functions. The main task is to run the same test code used in the first semester through JCP and then to analyze the effectiveness of the various levels of protection (i.e., the performance of individual functions and their combinations).

2. Code Protection

As stated before, there are many different ways of software protection which can be used independently or in combination. In this section, a detailed review of each of these protection techniques is presented.

2.1. Obfuscation

Obfuscation transforms source code into more obscure one with a view to making the logic and data structure of the code as meaningless as possible even after being reverse engineered. The main idea is that reverse engineering will be worthless, when the cost of understanding and/or stealing intellectual property of code becomes prohibitively high, especially if the task is more
arduous than that of rewriting the program. Obfuscation can be performed in a variety of ways, such as restructuring control flows, renaming variables and classes, and encrypting strings, etc. An example of such transformations is given in Figures. 1 and 2. Figure 1 shows the original code that uses a simple \textit{for} loop to output each letter of the alphabet. The same code is obfuscated using Dash-O Pro. The transformed code is then shown in Figure 2. It is clear to see that the obfuscated code is much more complicated than the original one due to added overhead. For instance, the simple \textit{for} loop is reconstructed with several \textit{goto} statements and \textit{case} statements with labels, which makes it harder to follow the code sequence.

```java
import java.io.PrintStream;

class forloop {
    static {
        forloop();
    }

    public static void main(String arg[]) {
        for(int i = 65; i <= 90; i++)
            System.out.printf("%c\n", new Object[] {
                Integer.valueOf(i)
            });
    }
}
```

Figure 1: Simple Java code

While many different obfuscation techniques are developed, it is of importance to evaluate and compare their strength. Currently, the quality of an obfuscating transformation is analyzed according to four criteria: cost, potency, resilience and stealth [2]. The cost of the obfuscation is based on the additional execution time/space that a transformation incurs on obfuscated code. The potency measures how obscure a transformation adds into source code. The resilience of code deals with how easy it is for a de-obfuscating program to undo obfuscation. A highly resilient code can make it impossible to de-obfuscate code. To be stealthy, an obfuscated code must blend in with the original code as much as possible, so that the obfuscated parts cannot be
picked out easily. A measure of each of these parameters except cost is subjective. Therefore, the evaluation of overall efficiency of an obfuscator is also subjective due to its dependence upon the aforementioned factors [9].

2.1.1. Control Flow Obfuscation

Control flow obfuscation is to restructure the computation routines of code. This can be done using many different types of loops and conditional statements to 1) break up computations that logically belong together, 2) merge ones that do not, or 3) add irrelevant statements that do not contribute to the actual computation sequence. For such transformations that alter the control flow of code, a certain amount of computational overhead will unavoidable. An example of control flow obfuscation can be seen in Figure 2. It shows different types of looping and conditional statements added after obfuscation.
2.1.2. Variable and Class Renaming

For easy debugging, a variable or class is usually defined using a descriptive identifier. For example, a variable for time would be named *time* and a class that would count the change would be named *countChange*. Variable and class renaming method transforms such descriptive names into ones that are hard to determine the meaning of variables/classes they represent. In this fashion, the variable *time* and the class *countChange* in the early example would be renamed into a01010 and a00011.

2.1.3. String Encryption

String encryption is a method of converting a string of characters into something that is virtually unreadable. This encryption must be done using an encryption algorithm along with a key, and then when the code is run the key must be used to decrypt the strings. For example, instead of the code reading “Your data is”, it will read “aj129jdnvkl” which will make it harder to figure out what part of the code it refers to. When the code is run, however, it will still output the intended string.

2.1.4. Class Splitting and Class Coalescing

Many Java programs use classes to perform a certain operation that needs to be used in the main code. These classes are usually defined outside of the main code and called within the code. There are two types of obfuscation transformations for classes: class splitting and class coalescing [4]. The former is to replace a class in the program with a number of classes; and the latter meld two or more classes with a single class. Both transformations make classes less recognizable and harder to understand.
2.2. Encryption

Encryption is the process of scrambling information by applying a mathematical function with an encryption key in such a way that it is extremely difficult for anyone other than an intended recipient to retrieve the original information. To decipher the scrambled information, a decryption method along with a decryption key has to be used. In this process, the encryption key and the decryption key may not be the same. It is obvious that encrypted information will be unsafe while the secrecy of cryptographic keys does not withstand attacks. To protect partial key exposure or cryptanalysis, multiple encryption schemes need to be used [6]. The similar method is used in JCP, called multi-layer encryption [5].

2.3. Tamper proofing

Tamper proofing will disable some or all of code functionality once it detects any unwanted modifications. Such modifications include both manual code changes by end users and automated changes by a virus [7]. A straightforward tamper-proofing technique is to compare software program to the original one in some way, such as checksums, right before the program runs [7]. However, an attacker can easily remove or patch around them. The oblivious hashing technique has also been used for tamper-proofing. A new dynamic stack-tracing approach introduces extra hashing instructions to monitor the top of the stack to check if the running program has been tampered [7].

3. Java Code Protector

3.1. Installation

The Java Code Protector (JCP) does not come with a console based installation interface and a graphical user interface (GUI). Its installation requires Java Development Kit (JDK) and Java
Analysis of Java Code Protector

Cryptography Extension (JCE) to be obtained from http://java.sun.com, and Apache Ant from http://www.apache.com. After extracting a zip file for JCP, the installation directory named JavaCodeProtector_1.3.2 is acquired. Two files in this directory need to be edited according to the computer setup, one is anttask_codeprotect.bat and the other is codeprotect.xml. The former is a MS-DOS batch file that helps set classpath to locate files required to run the JCP program and other associated programs. The latter is a XML build file that integrates JCP with Apache ant. The file specifies the locations of directories and files that are needed to run JCP. Typically, the directories include the installation directory, the source code directory where the original Java code to be protected is located, and the project directory under which JCP creates the corresponding protected code and works. In addition, the codeprotect.xml defines the protection levels of JCP on Java code, which will be elaborated in further detail in Section 3.3.

3.2. Usage

To prepare an aforementioned project directory, several Ant with different targets need to be run in the console sequentially. These targets are prepare1, complement_constructor, prepare2, and prepare3. While executing Ant with the target prepare1 under the installation directory, a new project directory with several sub-directories is created. These newly created directory/sub-directories store all necessary libraries and resources for the compilation of original Java code, and other document files to be created during the protection process. The Ant commands with other targets are then run under the project directory. The complement_constructor target brings up a graphical wizard, where users can request and renew a license key, and the JCP program checks the validity of a key. Upon requesting a license key, a request file with the data on your registration and computer identification is generated and needs to be mailed to ChainKey for a valid license. Note that the key is specific to the computer that the request is mailed from and only works with the computer. The key and the key-request files must both be stored in the
**Analysis of Java Code Protector**

`licensee_module` sub-directory under the project directory. After importing the valid key, a file called `javadoc_argument_file_api` under the project directory must be edited with all API packages and/or classes of the Java code listed one per line. The targets `Prepare2` and `Prepare3` are used for setting up the API classes. When the preparation is completed, JCP protects the Java code via executing Ant with the target `main`. A resultant archive is then generated after the Java code being obfuscated, encrypted, and tamper-proofed.

### 3.3. Features

The unique feature of JCP is the combination of obfuscation, encryption and tamper-proofing. In addition, JCP can be used along with a third-party obfuscation program to make code extra secure. There are three important parameters that control the protection strength of JCP, which are tamper-checking density, tamper-checking size, and layers of encryption. The tamper-checking density is about how frequent tamper checking is conduct in percentage, which is defined as a basic option named `tcdensity` and whose value defaulted at 5 can be modified in `codeprotect.xml` file. For instance, `<property name="tcdensity" value="5"/>`. The higher the percentage, the slower the program runs. The tamper-checking size is another basic option named `tcsize` that specifies how many classes to be checked in one routine of tamper-checking. Its value is a positive integer less than or equal to the total number of classes in your program. By default, it is 10. The last parameter is the number of layers of encryption. The bigger the number is, the stronger the protection. However, it takes more time to protect the code during the build process. The default value is 100, but the minimum is 50. Following the same fashion, the values of these parameters can be modified in `codeprotect.xml` file.

### 3.4. Problems
Unfortunately, in the course of this project, a lot of problems arose with the Java Code Protector. The detailed information regarding the problems and their solutions are presented below.

**Problem I. XML syntax errors**

XML is a text-based markup language, where elements are delimited by angle brackets identifying the nature of the content they surround. A tag is just a generic name for an element. Each element is marked with an opening tag that looks like `<element>` and a close tag with a slash placed before the element’s name, such as `<element>`. If an element does not enclose any content, it is known as an “empty tag” which can be created by ending it with `/>` instead of `>`. For example, `<element/>`.

While editing the `codeprotect.xml` file, simple mistakes were made by missing the extra slash in the empty tags. This error was quickly corrected.

**Problem II. String literal error**

As shown in Figure 3, an error occurred when we executed Ant with the target `main`. The error message was very vague with little detailed information regarding the reason and potential solution. We had worked closely with the ChainKey customer service and helped them identify this serious problem with the JCP software, although our communication via emails was difficult due to different time zones. The reason for this error was misconfiguration of reserved names for J2SE 5.0. The bug was eventually fixed by ChainKey on December 2005. An upgraded version was released then.

**Problem III. API error**
Analysis of Java Code Protector

Using the sample code provided by ChainKey, we tested the new version of JCP (Version 1.3.2). Unfortunately, an error related to inconsistent API occurred when we executed Ant with the target `prepare 2` as shown in Figure 4.

The reason for this error was that the `javadoc_argument_file_api` file was not edited correctly. After modifying this file according to the ChainKey’s suggestion, we were able to pass the build process with the target `prepare 2`. However, we could not complete the whole protection process to test the JCP’s functionalities since our license key was not recognized by the new version of JCP. We are working with the ChainKey for the solution now.

```
[protector] exit status: failure
[protector] elapsed time: 0 min. 20.110 sec.
[protector] failed to protect the code. Please fix the problem and try again.
[protector] the details of the error is shown below.
[protector] if the problem persists, please send the relevant information to support@chainkey.com for trouble shooting. we'd appreciate your feedback.
[protector] error details: failed to adjust a string literal
[protector] if you need assistance, please don't hesitate to contact us at:
[protector] support@chainkey.com
[protector] your feedback is important to improve our products.
[protector] thank you.
```

Figure 3: String Literal Problem

4. Future Research

The Java Code Protector program is currently under construction, and new updates and fixes have been released recently. We will continue to work closely with the ChainKey Company and try to get JCP working correctly, so it would be doable for us to test its functionalities against other protection techniques. Meanwhile, it is worthwhile to look into other protection programs and analyze their performance strength. One of such programs is the yGuard Java Bytecode
Analysis of Java Code Protector

Obfuscator (http://www.yworks.com/en/products_yguard_docs.htm). This freeware uses Ant to run the necessary configurations and creates patches to easily help developers integrate obfuscation into their development process. Smokescreen (http://www.leesw.com/smokescreen/index.html) is a new obfuscator which uses a GUI to perform obfuscation on classes, methods, and fields as well as control flow obfuscation and string encryption. Another program that is worth investigating is the JCloak obfuscator (http://www.force5.com/JCloak/ProductJCloak.html). It has several important features, including automatic reflection handling, incremental obfuscation, and automatic serialization handling, etc.

```
[Protector] ------------------------------- Result
[Protector] -------------------------------
[Protector] Exit status: Failure
[Protector] Elapsed time: 0 min. 30.434 sec.
[Protector] Failed to protect the code. Please fix the problem and try again.
[Protector] The details of the error is shown below.
[Protector] If the problem persists, please send the relevant information to support@chainkey.com for trouble shooting. We'd appreciate your feedback.
[Protector] Error details: Inconsistent API and source (API Class: Index)
[Protector] If the api class is an inner class, it is against the restriction of the current version. Please remove the entry from the API document (typically under the "api" directory), and try again.
[Protector]
[Protector] If you need assistance, please don't hesitate to contact us at:
[Protector] support@chainkey.com
[Protector] Your feedback is important to improve our products.
[Protector] Thank you.
```

Figure 4: API Problem

Overall, the project will continue next semester with the analysis of not only the JCP program but also other efficient obfuscation tools. A new system of metrics is expected to be developed in judging the protection strength of obfuscation transformations.

5. Acknowledgments

We would like to thank Lockheed Martin for funding this research project and its personnel on this project for their time and efforts to help us configure the Java Code Protector.
References


