Fingerprint Generator and Detector Implementation for JAR Files
<Draft>

Naif A. Alzahrani

Graduate School of Computer Science
Carleton University
Ottawa, Ontario, Canada
E-mail: nalzahra@connect.carleton.ca

Java archived file (JAR) is a kind of compressed java package file which contains mainly compiled java classes and other necessary resources such as XML files, text files, and picture files. Source code files are usually not included in the JAR file, especially for commercial applications. JAR files are always reused in open source software. Some developers adopt external code to their application without considering license constraints. Others reuse it after modifying the code to some level to appear as new code not related to the original source. In both cases, reusing the open source code should follow the license rules and constraints. This paper will explain the implementation of the fingerprint generator and detector program for JAR files. The implementation works through JAR file and deals with Java compiled files only to collect selected metadata attribute values of all methods to be included in the fingerprint for the JAR file.

***Index Terms*— byte code, code cloning, copy detection, JAR, Software licensing violations, software similarity**

I. INTRODUCTION

In the last decade, open source software (OSS) became more popular, robust, dependable, and cost effective. Many companies adopted OSS in their businesses. OSS requires low hardware specification and works with high performance stability compared to other commercial software.

<To be developed later>

1) Problem

The main issue in reusing and adopting OSS is the violation of license constraints and how it can be merged and integrated with other software licensed under a different license type. In fact, some OSS is licensed under a specific license and cannot be integrated with other licenses. For instance, any software licensed under general public license (GPL) must remain under the same license and the code should be available for others to use.

Some developers do not understand this point and they reuse OSS in company products that are licensed differently. This violation of OSS licenses raises a legal issue.

The problem that this paper focuses on is how companies and developers can avoid using open source illegally by comparing their applications with other open source programs and identifying the similarities.

2) Motivation

Many companies invest in developing commercial software aiming to get revenue and a reputation. These companies test their application before release. If OSS has been used in developing such a product without considering the open source license constraints, a company's investment and reputation may be lost due to illegal use of OSS. The motivation for this paper is to solve part of this complicated problem by helping companies, as well as developers, to determine the similarity between two Java archived files (JAR). Getting a similarity percentage will support companies in evaluating the final product that adopts OSS before release. Also, it will save company investments and reputation from any illegal concerns related to OSS licensing violations.

3) Goal

The goal is to develop a Java program that does the following:

- Generate a unique fingerprint from the original JAR.
- Compare target the JAR file with the original JAR file.
- Calculate the certainty present.
- Generate the required comments.

The purpose of the fingerprint is to act as an information container for an identifier that is used in the compression process. It has an ID that is unique and different from other fingerprints. The size of the generated fingerprint must be manageable to reduce the time of compression.

Certainty and similarity present result will show how two JAR files are similar to each other. A high percentage indicates that both JAR files have many classes and methods that are similar; on the other hand, a lower percentage means that fewer classes and methods are similar or the same.

In addition, this program should show the end user interesting comments during and after the compression process. These include the status of each step in the compression and the total number of class files and number of methods in both JAR files.

4) Objective

In order to generate a small-sized fingerprint, the implementation of the fingerprint will focus only on the object code files (.class) of a JAR file. The following steps must be completed:

- Do some research on code cloning to deeply understand other compression techniques and the latest research in this field.
- Study the structure of Java object code.
- Find a Java library to read and parse object code files.
- Design the fingerprint structure.
- Find a way to calculate the similarity factor and percentage.

5) Outline

The work in this paper is provided in the following sections. Section II gives some history and background knowledge in the same research area – cloning detection technique. Section III illustrates the design of the program that detects the similarity of two JAR files. Section IV presents some results and compares them with the goals and objectives that are

identified in previous sections. Section V concludes the paper and suggests some future work.

II. BACKGROUND

This section describes the knowledge and work related to project implementation.

1) Related work

Java programs are compiled into a platform-independent format as byte code (.class file). These files contain important information about the original code [2]. To read byte code files and extract the information about the original code, a developer uses one of the following methods:

- Interprets class files by reading metadata [9].
- Reverse engineers the byte code to java source code [2] [8].

The first method – interpreting the byte code (.class) files – is done by reading the metadata of each class in a JAR file, such as return type, input parameters, access flag, and other information about each method in the class file. Many open source Java libraries are customized to read and interpret byte code such as [9].

Expert developers can access byte code, modify and correct the code even without the original source code. Qing et el, study and the byte code file structure and they developed a strategy to read and modify the byte code. In fact, this strategy is limited since it can modify some metadata values.

The reverse engineering method is done by De-compiler application [8]. These applications transform the byte code to an instruction set or Java source code that can be recompiled again. Memone et el, proposed two byte code obfuscation techniques to prevent the De-compilers from generating the correct source code.
<To be developed later>

III. APPROACH

The following section will illustrate the design and decisions that were made based on [2] [4].

1) Design

A JAR file contains different file types, such as java source code, text file, images, XML files, and class files. Developers can modify the JAR file easily except for class files that need special skills as well as advanced tools for modification. The approach of implementing a fingerprint generator and detector will focus on byte code (.class files).

As mentioned before, the main goals are to generate a string-based fingerprint and get a similarity factor after matching to the JAR file. The design of the fingerprint generator and detector are summarized in the following points:

- Ignore all file types and focus on byte code (.class files) only in order to generate small size fingerprint
- Collect some selected values from metadata of class files that are not changeable or not easy to change.
- Build up a fingerprint as a string containing the data collected in the previous step for each byte code file in JAR.

A. Pseudocode for fingerprint generator

The following is Pseudocode for the generator:

```
Get JAR file
If the passed file is not JAR then
        Print error message and exit
Else
        Unzip JAR file
        For each .class file
                For each method on .class file
                        Read method metadata attributes
                        Add attribute value to Array List
        Build up the fingerprint with unique ID
        Return fingerprint for JAR file
```

B. Pseudocode for fingerprint detector

```
Get JAR file
Get the fingerprint
Set certainty percentage value to zero
Set similar method counter value to zero

If fingerprint ID is not valid then
        Print error message and exit.
Else
        Pass JAR file to fingerprint generator

        Parse fingerprint one in array list one.
        Parse fingerprint two in array list two.

        For each value in the array list one
                Read collected metadata attribute
                For each value in the array list two
                        Read collected metadata attribute
                        Compare values
                        If match then
                                similar method +=1
        Calculate certainty percentage.
        Add total number of methods to the comments.
        Return comments and certainty percentage.
```

1) Decision made

In order to implement a fingerprint generator and detector that are based on the byte code files, we have to carefully select the metadata attributes. In fact, some attribute values are easy to change even without the original source code. As mentioned above, [2] is an example of one strategy developed to modify byte code files. Using this strategy, experts can modify some metadata values of the byte code to refine or modify the code of any Java class file without having the source code of that class. However, this strategy is limited to

some modifiable attributes of metadata. As a result, we decided to select non-modifiable metadata for each method in the .class file. The fingerprint will be a string containing a metadata attribute value for each method. The following table shows the main metadata selection decision in our implementation.

Metadata attribute name	Result Description
Name_index	Refers to a string in the constant pool.
Attripute_count	Number of entries in attribute table of the method.
Max_stack	Size of stack required by the method's code.
Max_local	Number of local variables required by the method's code.
Code_length	The method's executable byte codes length.
Exception_table_length	The length of the method exception table.

Table 1 : Description of method metadata

IV. RESULTS

This section shows the results of the fingerprint generator and detector in different scenarios. The following table summarizes the results in each scenario.

Scenario	Results
Original and target JAR files are the same	Figure 1 shows that the Certainty percent = 100 number of similar function =1120 number of functions in 1st JAR = 1120 number of functions in 2ed JAR = 1120
Two different JAR files	Figure 2 shows that the Certainty percent = 23.07 number of similar function=3 number of functions in 1st JAR = 1120 number of functions in 2ed JAR = 13
Two identical JAR files, each one containing only one .class. In the second file we modify the following in one method only • Method name • Return data type • Input parameter data type.	Figure 3 shows that Certainty percent = 33.33 number of similar function=1 number of functions in 1st JAR = 3 number of functions in 2ed JAR = 3

```
-----Final comments-----
Total number of functions in .class files of first JAR file is : 1120
Total number of functions in .class files of second JAR file is : 1120
Number of matched functions in second JAR file is : 1120
Certainty Percentage is : 100.0
```

Figure 1: Results of first scenario

```
-----Final comments-----
Total number of functions in .class files of first JAR file is : 1120
Total number of functions in .class files of second JAR file is : 13
Number of matched functions in second JAR file is : 3
Certainty Percentage is : 23.076923
```

Figure 2: Results of second scenario

```
-----Final comments-----
Total number of functions in .class files of first JAR file is : 3
Total number of functions in .class files of second JAR file is : 3
Number of matched functions in second JAR file is : 1
Certainty Percentage is : 33.333336
```

Figure 3: Results of third scenario.

<To be developed later>

V. CONCLUSION

1) Review goal and contribution

To summarize, the implementation of the fingerprint generator and detector show a positive result. In fact, selecting specific metadata has an impact on the overall accuracy. Including only non-modifiable attributes in the fingerprint increase the accuracy and reduce the execution time.

There were four goals set in the previous section:

- Generate a unique fingerprint from original Java archived file (JAR): this goal successfully achieved since we defined unique ID for generated fingerprint
- Compare target JAR file to original JAR file: from the result section we can see the positive results of JAR files compressions. In fact, scenario three shows how the smart modification can also be identified.
- Calculate certainty and similarity present
- Generate required comments. As it can be seen on the result section the implementation shows the certainty percentage based on number of matched methods and the total number of methods in each JAR file.

2) Future work

The implementation of the current version will match the selected metadata of each method as one block. For instance, if all metadata are matched in both methods the program will consider it, but if one value is different the current version will not consider it as a matched method. So developing an existing version to include a partially matching technique is one of the most important points for future work.

<To be developed later>

REFERENCES

[1] Java class file format. http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html. Accessed: October 10, 2010.

[2] Jan M. Memon, Shams-ul-Arfeen, Asghar Mughal, and Faisal Memon, "Preventing Reverse Engineering Threat in Java Using Byte Code Obfuscation Techniques," 2nd International Conference on emerging Technologies. page 689-694. 2006.

[3] Open Source Initiative Licenses - http://www.opensource.org/licenses/ Accessed: November 1, 2010.

[4] Qing Su, Yongfeng Cao, and Guoda Liang Su, Q, Cao, Y and Liang, G., "The Strategy of Java Class File's Modification," 2009 Second International Workshop on Computer Science and Engineering. page 322-326.

[5] DJ Java Decompiler. http://members.fortunecity.com/neshkov/dj.html Accessed: October 1, 2010

[6] Java Class File Specification. http://en.wikipedia.org/wiki/Class (file format) Accessed: October 10, 2010.

[7] Free Software Foundation - http://www.fsf.org/licensing/licenses/ Accessed: November 1, 2010.

[8] JD-GUI. Java De-Compiler. http://java.decompiler.free.fr/ Accessed: October 1, 2010.

[9] Java class library. http://www.amosshi.net/site/freeinternals.org/product/javaclassfilelibrary/javadoc/. Accessed: October 15, 2010.