JAR File Fingerprinting Techniques

Hao Luo

COMP5900

October 2010

Abstract

Java Archive (JAR) file is a popular way to aggregate Java class files and resources into one package for distributing applications and/or libraries. JAR file format is developed based on the popular ZIP file format, while providing a convenient way to organize Java byte code for distribution in the open source community, JAR added extra layer of encapsulation and made it more difficult to tell the content of those binaries. However using code without knowing the source could potentially introduce license related issues in any project. In this article, we propose an approach to match JAR files based on the fingerprint generated from each JAR.

Chapter One

Introduction

The development of open source projects has been remarkable, mostly benefit from the utilization of the power of distributed peer review and the transparency of process. Many see Open Source Software (OSS) as not only a software engineering phenomenon, but as psychological, social, culture, political, economic, and managerial phenomenon as well [FEL07] . Generally speaking, open source software allows users to use and change or even distributing the software without any cost. Strictly speaking, whether a piece of software qualifies as OSS, and exactly what the users of software are allowed to do with it, are legal questions.The code written by developers are copyright protected, the authors of OSS can grant permissions on their creative work with conditions to the users, and they do so with software licenses.

1.1 Problem

Given a JAR file, assuming the byte code packaged inside are the most significant data. We want to find an unique representation of the JAR as its “fingerprint”, in a relatively time and space efficient way, such that by compute and compare the fingerprints of JARs we can decide whether two JARs are identical.

1.2 Motivation

Not all software licenses are created equally, this is true even in the open source world. Depending on the nature of the project, blindly using code libraries could lead to legal disaster later on. As the de factor distribution package in Java open source community, JAR files attached or submitted by contributors around world needs to be revisited to avoid license related traps. It is desirable to know if a JAR just submitted is actually the same as another JAR, which has been verified to have a license that is aligned with the project’s goal, used somewhere else in the project.

1.3 Goals

My goal is to generate a unique ID as the fingerprint of a given JAR, be able to tell whether two JARs are identical based on the fingerprints generated. The algorithm needs to be efficient enough to allow interactive process. Last but not least, in case the two JARs in comparison are different, still be able to provide a quantitative measurement on how similar or different the two JARs are.

1.4 Objectives

First I will read the JAR file contents into a utility class, then I will gather information about selected contents of concern from the class and generate the unique id based those information, and put the unique id into our fingerprint class. As for comparing JARs, different fingerprints means they are not identical, in this case a method will be supplied by me to show the similarity and difference between the JARs and the likely-hood of JARs been the same is also calculated as a percentage figure.

1.5 Outline

In the following sections of this paper, I will start by showing some related work in Chapter two, then describe my approach to the problem, as well as some decisions and assumptions made in Chapter three, followed by results of the some initial experiments in Chapter four. At the end, I will state my conclusion and possible ways to improve the process in the future in Chapter five.

Chapter Two

Related Work

Animal sniffer is an OSS that provides tools to assist verifying that classes compiled with a new JDK/API are compatible with an old JDK/API. It can check the classes and method signatures that your compiled code uses and verify that you have use only those classes and signatures available in the API you are targeting [ANI10] , wher the targeting API can be a JDK, or a collection of jar and class files, or a collection of other API signature files, or combination of these elements.

In [PIE11] they described an alternative way to attack the same problem I am trying to solve in this paper. By converting the the object relation map of the classes that a given JAR contains into a weighted graph, they obtained a fingerprint for that JAR. The the problem of matching JAR files based on fingerprint becomes graph isomorphism problem. They also model the graphs with matrices, then compute the the similarity using those matrices.

Chapter Three

Approach

3.1 Design

Three Java interfaces are given, which are IFingerprint.java, IFingerprintGeneratorDetector.java and IFingerprintResult.java. Implementaion class for each interface is described below.

3.1.1 JarFingerprint.java

JarFingerprint implements IFingerprint interface. File name of the JAR, which the fingerprint belongs to, is stored in a Java String. The generator ID to be passed in from the generator class, as well as the encoding which should be unique for different JAR are also Java String. In addition to the above required fields that are enforced by the interface contract, I created a Map<String, Long> to store information used to compute the unique encoding, the key is class name, and value is the Cyclic Redundancy Check (CRC) code associated with the class. The encoding is obtained by XOR all the CRC codes in the map. The special property of XOR will ensure the same set of CRC codes lead to the same result at the end, no matter in which order they are applied.

3.1.2 JarFingerprintResult.java

JarFingerprintResult implements IFingerprintResult interface. It has a float type field that represents the degree of certainty in percentage. A List is used to store all the comments generated during the JAR matching processing. The display string is not stored but rather computed every time it is requested.

3.1.3 JarFingerprintGeneratorDetector.java

JarFingerprintGeneratorDetector implements IFingerprintGeneratorDetector interface. The generator ID is a Java String which has default value “COMP5900-Hao”. This field is used to “stamp” every fingerprint produced by this generator. The method that generates the fingerprint will return a JarFingerprint object with all fields initialized and assigned a value from

the given JAR file, those values could be defaulted by the program or computed in the program. The other method that compares the given fingerprint and JAR and return the result wrapped in a JarFingerprintResult object. It will first check to see whether the generator ID in fingerprint matches with the generator's ID. If negative, matching process will terminate immediately and the result generated will have certainty percentage value of zero percent; otherwise, a fingerprint will be generated from the incoming JAR and the encoding of the given fingerprint and the newly generated one will be compared. Same encoding yells a result of one hundred percent match, or else a more complicated matching algorithm, which I call "Genealogy Check" and I will go over the details in the next subsection, is going to kick off. When it is done, returned result will contain a computed certainty percentage and comments created each step of the way.

3.1.4 Other Utility Classes

In order to load classes directly from a given JAR file, I used the following three classes found on http://www.javaworld.com [HAR97]. ***JarResources*** class maps all resources included in a Zip or Jar file. Additionally, it provides a method to extract one as a blob. ***JarClassLoader*** extends ***MultiClassLoader*** and it stores the data in the given JAR file passed to its constructor inside a JarResources object. It is responsible for loading a class by name from the given JAR.

3.1.5 The Genealogy Check

A brief description of the check algorithm is illustrated below.

Variable used:

- IncomingJar: the incoming JAR
- KnownFingerprint: the given fingerprint

Algorithm outline:

```
int numberOfClassInIncoming = 0;
int matchCounter = 0;
WHILE(IncomingJar has more class){
        numberOfClassInIncoming++;
        Name = IncomingJar.nextClass().getClassName();
        // grab the crc code for given class name
        CRC = KnownFingerprint.Map.get(Name);
        crc = IncomingJar.nextClass().getCrc();
```

```
        IF ( CRC == null) {
                addComment("Conflict can't be resloved");
        } ELSE IF ( CRC == crc) {
                matchCounter++;
                addComment("One class matched");
        } ELSE {
                CompareInheritanceStructure(Name);
        }
}
```

CompareInheritanceStructure method compares the class inheritance relation for the class specified by Name, by loading classes from both JAR (the incoming one and the one that corresoding to the fingerprint given) separately all the way up the inheritance tree and see whether they are the same class at each level of the tree, until either a conflict found(different classes found) and a mismatch recorded, or root of the inheritance tree (Java Object class) has reached and a match is recorded.

At the end, the number of matched classes will be divided by the MAX of the number of classes in incoming JAR and the number of classes in the JAR that the given fingerprint belongs to, to get our certainty percentage.

3.2 Decisions made

There could be many types of files existing in a JAR, but I limit our discussion to Java code, the other types of files such as images, properties file can also be creative work but are out of scope here. Only class files and their folder structures are chosen to produce the unique encoding.

Among all the information one can find in a JAR that are related to Java class files, potential candidates that can be used to compute a unique encoding are abundant. Since one of the goals of this paper is to allow interactive style of comparison in real time, CRC code is chosen , for its readiness to use but not for its cryptographic strength, to make the encoding generation a computationally lightweight task thus the program can be more responsive.

As mentioned in section 3.1.5, different encoding values means different fingerprints and therefore different JARs. Knowing two JARs are different was never enough, it only solves half of our problem, we want to measure how different or similar they are. Class inheritance relation is a good way to investigate whether two classes from two different JARs are the same or similar

in micro scope, provided that these two classes with the same name found to have different encodings. The assumption is that, although different versions of the same class will have different CRC due to code change or refactoring, their inheritance relation is a more stable characteristic since it rarely changes. The objective here is not to recognize adversary instrumentation of code but rather accommodate minor changes in different version of class files and be able to match such classes. It will resolve many conflicts result from different CRCs and will ensure high values of certainty percentage for similar JARs.

Chapter Four

Results/Validation

4.1 Unknown generator ID

1) generate fingerprint for commons-collections-3.2.1.jar

2) construct a empty fingerprint and assign a different generator id

Expected: Zero percent certainty due to unknown generator ID.

Observed:

```
##########################################
# The certainty percentage is [0.0%]
# See details below
##########################################
JARs involved [null] & [commons-collections-3.2.1.jar]
- This fingerprint with generator id [You don't know me] is UNKNOW to me.
```

4.2 Almost identical JARs

1) generate fingerprint for commons-collections-3.2.1.jar

2) generate a fingerprintresult for the above fingerprint and commons-collections-3.2.jar

Expected: High certainty percentage and details of matched and conflicted classes.

Observed:

```
######################################
# The certainty percentage is [98.25327%]
# See details below
######################################
JARs involved [jarsForTesting/commons-collections-3.2.1.jar] & [jarsForTesting/commons-collections-3.2.jar]
- conflict found in [org/apache/commons/collections/ArrayStack.class], checking inheritance structure...
        # [java.util.ArrayList] match
        # [java.util.AbstractList] match
        # [java.util.AbstractCollection] match
        # [java.lang.Object] match
# conflict resolved!
```

...

```
- conflict can not be resolved, [org/apache/commons/collections/DoubleOrderedMap$7.class] does not exist
- conflict can not be resolved, [org/apache/commons/collections/DoubleOrderedMap$8.class] does not exist
- conflict can not be resolved, [org/apache/commons/collections/DoubleOrderedMap$9.class] does not exist
```

...

4.3 Slightly different JARs

1) generate fingerprint for commons-collections-3.2.1.jar

2) generate a fingerprintresult for the above fingerprint and commons-collections-3.0.jar

Expected: Relatively high certainty percentage and details about match & conflicts.

Observed:

```
########################################
# The certainty percentage is [87.55459%]
# See details below
########################################
JARs involved [jarsForTesting/commons-collections-3.2.1.jar] & [jarsForTesting/commons-collections-3.0.jar]
- conflict found in [org/apache/commons/collections/ArrayStack.class], checking inheritance structure...
        # [java.util.ArrayList] match
        # [java.util.AbstractList] match
        # [java.util.AbstractCollection] match
        # [java.lang.Object] match
# conflict resolved!
- conflict found in [org/apache/commons/collections/Buffer.class], checking inheritance structure...
# conflict resolved!
```

...

```
- conflict can not be resolved, [org/apache/commons/collections/IteratorUtils$EmptyIterator.class] does not exist
- conflict can not be resolved, [org/apache/commons/collections/IteratorUtils$EmptyListIterator.class] does not exist
- conflict can not be resolved, [org/apache/commons/collections/IteratorUtils$EmptyOrderedIterator.class] does not exist
- conflict can not be resolved, [org/apache/commons/collections/IteratorUtils$EmptyMapIterator.class] does not exist
- conflict can not be resolved, [org/apache/commons/collections/IteratorUtils$EmptyOrderedMapIterator.class] does not exist
- conflict found in [org/apache/commons/collections/IteratorUtils.class], checking inheritance structure...
        # [java.lang.Object] match
# conflict resolved!
```

...

4.4 Exaclty the same JAR

1) generate fingerprint for commons-collections-3.2.1.jar

2) generate a fingerprintresult for the above fingerprint and commons-collections-3.2.1.jar

Expected: One hundred percent certainty match.

Observed:

```
########################################
# The certainty percentage is [100.0%]
# See details below
########################################
JARs involved [jarsForTesting/commons-collections-3.2.1.jar] & [jarsForTesting/commons-collections-3.2.1.jar]
# Encoding matched
```

4.5 Different JARs

1) generate fingerprint for commons-collections-3.2.1.jar

2) generate a fingerprintresult for the above fingerprint and commons-logging-1.1.1.jar

Expected: Low or zero percent certainty, with details of about match(is any) & conflicts.

Observed:

```
########################################
# The certainty percentage is [0.0%]
# See details below
########################################
JARs involved [jarsForTesting/commons-collections-3.2.1.jar] & [jarsForTesting/commons-logging-1.1.1.jar]
- conflict can not be resolved, [org/apache/commons/logging/impl/AvalonLogger.class] does not exist
- conflict can not be resolved, [org/apache/commons/logging/impl/Jdk13LumberjackLogger.class] does not exist
- conflict can not be resolved, [org/apache/commons/logging/impl/Jdk14Logger.class] does not exist
- conflict can not be resolved, [org/apache/commons/logging/impl/Log4JLogger.class] does not exist
- conflict can not be resolved, [org/apache/commons/logging/impl/LogFactoryImpl$1.class] does not exist
- conflict can not be resolved, [org/apache/commons/logging/impl/LogFactoryImpl$2.class] does not exist
- conflict can not be resolved, [org/apache/commons/logging/impl/LogFactoryImpl$3.class] does not exist
```

...

Chapter Five

Conclusion

5.1 Review goal and contributions

In this article, I have introduced a way to generate a fingerprint for a JAR, and to compare JARs based on their fingerprints. With the assumption of inheritance relation is more stable than actual code between software revisions, I also described an algorithm to obtain measurements on the level of similarity for two JAR files with different fingerprint. Implementations have been done and initial experiments shows that outcome conform with the expected results. And the comparing process has been very responsive, can definitely handle real time interactive operations.

5.2 Future Work

As noted in section 3.2, CRC code is not a safe again intentional mitigation on the class files. Cryptographic message digest like MD5 or SHA-1 can be calculated for each class file and stored in the fingerprint , and later be used as the basis to compute the encoding. This way, recompute the embedded checksum or trying to fiddle the file to make the checksum come out the same will be much harder to do, thus making the matching process safer against intentional hacking of the class files in JAR.

References

[FEL07] Perspectives on Free and Open Source Software, Edited by Joseph Feller, Brian Fitzgerald, Scott A. Hissam and Karim R. Lakhani, March 2007, ISBN-10: 0-262-56227-8, ISBN-13: 978-0-262-56227-0

[ANI10] Animal Sniffer, MOJO collection of plugins for Maven

[PIE11] Pierre from Adobe, Reducing JAR fingerprinting to graph isomofism, to appear in a conference 2011

[HAR97] Source code for class loading, Jack Harich, John D. Mitchell, 8/18/97 & 99.03.04, http://www.javaworld.com/