Title

J. Havelock, M. Mshana

Abstract: In this paper we address the problem of software package verification in the contexts of jar files. Where software package verification of a jar file is the problem of determining if two jar files originate from the same package. We propose a new method of representing a jar file in order to better compare jar files as well as a method of comparison. The representation of a jar file is called a fingerprint and consists of a fingerprint ID, the file name and an encoding of the content of the file. Using the fingerprints of two jar files, our method of comparison will produce a percent certainty that the jar files originated from the same package.

We outline our design of the fingerprint and how we compare two jar files to produce the percent certainty. As well as discuss the decisions that were made in order to design the fingerprint and the method of comparison. There are also improvements that are still being researched. We are looking at improving the fingerprint by encompass more of the class architecture.

To validate our design we test the fingerprint and method of comparison on over 50 jar files. We produce results by classifying each comparison as; a match, if the percent certainty grater 65%, as not a match, if the percent certainty less that 35%, or unknown, if the percent certainty grater 35% and less than 65%. The aim is to maximized both the number of true positives and false negatives. Our results are presented and discussed to develop ideas for improvement.

Index Terms: Jar files, Open source software, Software package validation, Jar file fingerprints

1. Introduction

1.1. Problem

The problem addressed in this paper is software package comparison; to construct a program that determines if a package (jar file) or part of a package contains the same content as another package. This comparison is done by creating a fingerprint, a representation of the content of the jar file. Typically the fingerprint is much smaller that the content of the jar file and may contain information on the structure and content of the jar file.

The program must adhere to an 3 interfaces. The first being the structure of of the fingerprint. The fingerprint will contain; an ID unique to the program, the original name of the file, and finally the encoded representation of the jar file. The second interface is for the structure of results of a comparison of two jar files. The results are comprised of two parts, the percent certainty and the comparison comments. The last interface is for the comparison program. This program contains the program ID, a function for creating a fingerprint from a jar file, and a function that returns the results of comparing a fingerprint to a jar file.

1.2. Motivation

The main reason for software package comparison is to address the issue of the copyright of software. The plagiarism of software is similar to that of written work, to prevent copying or modification beyond the scope permitted by the license, to assign proper credit to the creator of a piece of work, and to promote the production of these types of works.

There are many ways to compare a piece of written work to determine if it has been plagiarized. For the same reasons there are ways of comparing software. The idea of a fingerprint comparison is to, in the long run, create a database of fingerprints of all software so that they can be compared

to new software entering the world and hopefully eliminate the distribution of plagiarized software.

Comparing a piece of written work is done to prevent people from copying others without giving proper credit. Although is some ways written work is similar to software, it is also vary different. Written work like software may come in different languages, contain a portion of plagiarized content, or copy the overall idea. However, the distribution of software is much more complex due to the many forms it may be distributed in. It is for this reason we restrict our selfs to a specific form of software, jar files.

1.3. Goals

The overall goal of software comparison is to prevent plagiarism of software. However our goal is to prevent plagiarism in an open source domain to encourage the production and distribution of open source software. We do this in order to further the open source community. To direct our project towards the open source community we will look at the jar files, because the Eclipse Foundation is a major leader in the open source community and they use jar file plugins as their method of increasing their library.

The goal for this project is to create a program that adheres to the interfaces outlined in the problem statement that compares two jar files. The result of the comparison will contain a percent certainty that the second jar file contains content from the first. The idea of producing this result is not to declare that this software is plagiarized but to determine if a closer look needs to be taken at this software before allowing it to be distributed. This decision should take in to account the types of licences and their rights to modify and distribute code.

This project is not intended to ensure that the licences are abided by but only to compare two pieces of software.

1.4. Objectives

The fingerprint program is intended to determine the percent certainty of a jar files similarity to another jar file. Its primary intention is to find if a jar file has been modified and redistributed. There are many ways to modify or hide code. Our implementation of software package comparison focus is on the direct copying(copy and paste) of the files. This dose not include plagiarizing the idea or paraphrasing the code.

The intention of our implementation is to detect if some or all of the files in a jar file have been put into another possibly a larger jar file. We consider the possibility of renaming of content. The restructuring the jar file is also considered in the comparison; however, we consider the restructuring of the program itself to be paraphrasing or copying of the idea and will not be considered in our implementation.

1.5. Outline

In this paper, we give a brief background of the problem. Followed by our approach to the problem and the design and reason for the decisions made. We report the testing of our program on 50 different jar files origonaly from [1] . This is followed by a short discussion of the results and some conclusions about our project.

2. Background

This section covers the history behind the problem addressed. In the current world of open source software, the is a vast amount of software packages being made and injected into the world for free use and integration. Most of these packages are protected by the common open source licenses: MIT, BSD, GPL, LGPL, Eclipse and many more. A software package may be protected by a license that prevents users from modifying the content or code or may just be open to people to do anything with it so long as the license is present in any copy of the code made. In this project, we are trying to come up with a way to know if the package or part of the package has not being tempered with or changed by anyone else from where it was packaged by the author. This can just be for quality check or checking if the license is being voided.

Software package validation or as per this project known as Fingerprinting a software package is an existing research area which there has being a lot of contributions made. Though a lot of contribution has being made basing on security issues, like the MD5 encryption which encrypts a package sent over the internet and sends along the hash files with it so the person can run the algorithm again and check of the package what tempered with after being packaged by the sender.

Secure hash algorithm (SHA) is also an approach designed by the National Institute of science and Technology of the USA as a standard for information processing. Its some resembles with how MD5 works but its more improved. SHA is used in securing the jar packages made in Eclipse and any other IDE. How SHA works in securing the jar files is that all files to be packaged into a jar file are being run in the algorithm and SHA hashes are made for all the files and are stored in a manifest file of a jar file. When even the jar file is modified and repackaged again, these hashes are being recalculated again and saved and being compare with the original hashes. If the contents of the package was changed or modified the hashes would not match. We also used the SHA in composing the fingerprint since its a good indication of where the contents have being modified.

As a related work, the Eclipse foundation is also looking into this problem of know when a package has being modified or not. They are looking for a way of checking whether a build of a package from the submitter resembles with a build from a package which was submitted to the repository.

3. Approach

In this section, we cover the design made in developing the project and the decision made on implementing the designs.

3.1. Design

Our design covered the structural view of the jar file, Class Hierarchy, security features and license. The fingerprint produced at the end will be the encoding of different details we get from the covered sections. We believe that a combined pattern of these detail with maximize the probability of spotting the identity of a package.

In the structural view, we look at common details like content names, size and file type. We also looked into xml configuration files, getting distinct xml tags or Ids also brings the uniqueness to the fingerprint. In class hierarchy, we looked into different relations and inheritance that there can be within the class files in the package. Getting different relation and inheritance, methods and different object names will be used in encoding the fingerprint.

In security features, we try to look at different security features applied in the jar file. as previous discussed, SHA is usually used when a jar is being made and secured. The SHA hashes are being written in one of the manifest files with the jar files. We will get the hashes and save them as part of our encoding.

In the license option, we try to look for the case of those packages which will contain a license file of some type. Different licenses has different structures and quotes, we will take common tags of a license and try to get any specific information that can identify the author, community or organization involved with the work.

Implementation of extraction of all these details from the package will follow after the unzipping of the jar file. Different details are being stored in different data sets and are being encoded together to make up the string fingerprint which will be used in comparison with the other fingerprint that will be produced when another package is being brought up to be compared. The following section will go into implementation of all these design made.

3.2. Decision made

Implementation first started with implementing the fingerprint interface provided in class. These interfaces had defined data structures and methods to handle the jar file extraction and making the fingerprint. The data structure to handle the jar file had three variables: one for holding jar name, one for holding fingerprint ID and the string variable to hold the string made from that jar file.

Fingerprint ID is a unique identification being added to the finger print that will be as a mark that the fingerprint is made by us.

This project was implemented using java and being written using Eclipse. Extraction or unzipping of the jar file was done by calling an instance class called Unzip. This class uses the java.util.zip package and was written by Ian F. Darwin and is protected with Ian F. Darwin license. The license allows then modification of the work so long as the license is included in the new modified work. Since we have our main class in the project (FingerprintGeneratorDetector), we removed the main method from the Unzip class to our main method in FingerprintGeneratorDetector class. This is because java doesnt allow having more than one main class in a project. Hence with such modification we had obeyed the license of Unzip class and retain the license in it.

Within the Unzip class several sets where added so as to hold filenames, files size, file extensions and a variable to hold the number of files in the package.

Getting class hierarchy, methods and objects from class files was a big challenge to us. The class files in a package are already in a build state so reading it as a text file bring out random characters that cannot be of any importance to us. So we had to research of a way of reading a class file and we came up with two options: either use java reflection or java decompiler.

Java reflection API is used by programs which require the ability to examine or modify the runtime behaviour of applications running in the Java virtual machine[2]. With java reflection, one can get objects, methods and even variables being used in the classed being checked. The problem here is it checks the classes that are running in the virtual machine, our classes we want to check are in a disk. We came to find that with a special feature of java reflection called the reflection class loader it can be possible to load a class file into the running environment and make instance of it which can be used to used in checking for inheritance and objects.

Java decompiler is a tool used to get source code from the class files[3]. For the most of these decompiler we got online are either running in command line with the java virtual machine or are separate programs. This was a headache to us since its now hard to incorporate other runtime commands or programs into our project.

The extraction of class hierarchy, methods and objects has not being done yet due to time bounds and no clear solution has being obtained. But attempts to implement this section are still on and hopefully this part will be implemented in our project.

In implementing the security features part, we looked at the jar file manifest. A jar file has its specifications and the difference between a jar file and a zip file is that a jar files a used as application packages. JAR files are not just simple archives of java classes files and/or resources. They are used as building blocks for applications and extensions. The META-INF directory, if it exists, is used to store package and extension configuration data, including security, versioning, extension and services. As previous discussed in the design, jar files use SHA encryption in ensuring the files are not being modified.

As part of our fingerprint generation method, we go scan all the files in the META-INF directory for any SHA ID and extract it with its object name. The extracted details are being put into a set which will be used to encode the final fingerprint.

Our assumption here is that since security and validity of information is the main goal being research on in the world of software engineering, then all the software packages made will be secured with some encryption of some type. So we assume that all packages that will be run in this project will be secured. If its otherwise then the set for SHA Ids will be empty and still the encoding of final fingerprint will be done without them.

A jar files has a manifest file, usually an xml file (plugin.xml) which is being created as the jar is being extracted[4]. This xml has all the extension of the package and has all the packages, classes and any object used in the package. We found it interesting to get all the extension IDs and all objects and classes listed in it. This is more like an alternative to the class hierarchy approach. Since this file is being made whenever a jar file is being created or unzipped, we assume that if a package is to be changed and repackaged again then some object changed will also change in the xml configuration file. Our fingerprint generating method scans the xml file and takes it to a set which will also be used in the encoding of the final fingerprint string.

A license file may also be included in the jar file. Be it within the manifest directory of just as other files, it was of our interest to get in them and see if we can capture any unique feature of the license that may help in distinguishing between two packages being compared.

Encoding the final fingerprint, all the details being collected from different parts and sections of the jar file are being complied together in one string and saved in the fingerprint string variable in the Ifingerprint structure. The encoding has being simple to us so far, just appending all the details we have in one string and are being separated by a common string pattern which will be used in decompiling the fingerprint during comparison. We set a string separator to COMP5900, also after the final composition of the fingerprint; a fingerprint ID is being added to the string to mark our identity.

4. Validation and Results

In this section, we describe in detail the procedure for testing our chosen fingerprint and the method of comparison. The results of these tests are presented and discussed to give direction on where improvement is needed.

4.1. Validation Procedure

The validation of our designs of a fingerprint and method was done by running numerous tests on modified jar files from [1] to determine if our program could make the correct conclusion about the origin of the jar file.

To create the test jar files two main methods were used, modification with a java obfuscator and manual manipulation (unzip, modify, re-zip). The obfuscator that was use in this project is called ProGuard[5]. ProGuard has the ability to shrink, optimize, obfuscate, and preverify jar files. To create our test files we used the configurations for the obfuscator and the manual manipulation listed below.

TABLE I
TEST JAR CONFIGURATIONS

Obfuscator Configurations	Manual Manipulation
Keep names and Shrink sizes of files	Repackage (jars with signature only)
Change names and Shrink sizes of files	Change file sizes
Keep names and Flatten package hierarchy	Add files
Change names, Flatten package hierarchy and Shrink sizes of files	Remove files

Once the test files were created, the original jar files were compared with their versions of the test jar files. These test determine the rate and pattern of producing true positives and false negatives. Then the original jar files were paired with test jars of a different parentage to determine patterns for false positives. The percent difference used to designate a jar file as the same was set to 65 percent certainty that the jar files are the same. Similarity, the percent certainty for jar files not being the same file was set to 35 percent certainty of similarity. This means that there is and unknown region that is from 65-35 percent certainty that we must consider.

4.2. Results of Validation

The overall results are shown in table II. There are no false positives and no undecided for jar files that are not of the same parentage. The designed fingerprint and chosen comparison method

performs well with jar files that are different. The results for comparing a jar file with a modified version of its self are not as strong. The rate of true positive is only 60

TABLE II
CLASSIFICATION RESULTS FOR ALL CONFIGURATION

	Positive	Negative	Undecided
True	60%	28%	12%
False	0%	100%	0%

To look closer at the weaknesses in the fingerprint and comparison method we have divided the results according to the structure of our finger print. The fingerprint is based on three main comparisons; signature, XML tags and structure, and number of files and files sizes. Table III shows the results grouped according to the highest weighted attribute in the comparison. For example, if a comparison resulted in; No signature, 30% similarity of the XML tags, and 100% files size similarity, that the file would be grouped under file size.

TABLE III
CLASSIFICATION RESULTS GROUPED BY MOST COMMUNE ATTRIBUTE

Attribute	True Positive	True Negative	True Undecided
Signature	100%	0%	0%
XML tags	100%	0%	0%
Files size	48%	31%	21%

When the results are grouped according to the most prominent attribute it is obvious that relying on the size of the files is the lease accurate. In fact, when the files size is not the deciding attribute the classification is 100% accurate. This indicates that our fingerprint does well with signatures and when there is an xml file; however we need to look closer at the files in the jar file when there is no signature or XML file. It should be noted that all files that were grouped in the file group did not contain a XML file.

5. Conclusions

In conclusion our fingerprint preforms well when the jar file contains either an xml file or a signature. The good performance under these conditions is beneficial because most jar constructors (Ant, Maven and others build tools) produce an xml file[6], [7]. The fingerprint and comparison method can overcome many types of obfuscations; repackaging, changing file sizes, adding or removing files, flattening package hierarchy, and changing names of files, attributes and classes. These obfuscations may be done by unpacking the jar file, manually obfuscating the contents and then repacking the jar file, or by using a java obfuscator. While the fingerprint preforms well under certain conditions, there are areas that need to be improved on. When there is no XML file or signature the fingerprint only correctly classifies 48% of jar that have been obfuscated. To improve this result we are looking into comparing the structure of the classes with decompilers and java reflection.

References

[1] "Jar file download examples (example source code) organized by topic," 2009, http://www.java2s.com/Code/Jar/CatalogJar.htm, Date Accessed: Oct. 2010.

[2] "Trail: The reflection api," 1995, http://download.oracle.com/javase/tutorial/reflect/index.html , Date Accessed: Nov. 2010.

[3] "Jad (java decompiler)," 2010, http://en.wikipedia.org/wiki/JAD_(JAva_Decompiler) , Date Accessed: Nov. 2010.

[4] "Apache maven project," 2008, http://maven.apache.org/maven-1.x/using/jar.html , Date Accessed: Nov. 2010.

[5] E. Lafortune, "Proguard version 4.5.1," 2002-2010, http://proguard.sourceforge.net, Date Accessed: Nov. 2010.

[6] "Use ant to build a jar with version/build number," 2006, http://www.rgagnon.com/javadetails/java-0532.html, Date Accessed: Nov. 2010.

[7] “Jar file specification,” 1999, http://download.oracle.com/javase/1.3/docs/guide/jar/jar.html , Date Accessed: Oct. 2010.