JAR Library Copy Identification Through Fingerprint Creation and Comparison

Grant McNeil, 100686111

***Abstract*—This paper is the description of an algorithm to test an unknown JAR library file against several fingerprints generated from other JAR files. This paper discusses the problem addressed, the algorithm used to solve the problem as well as extensive testing of the algorithm to ensure speed and accuracy.**

Index Terms – Introduction, Background, Approach, Results/Validation, Conclusion

I. Introduction

One of the things that makes the Java programming language great is the availability of user made code libraries. These code libraries are collections of Java classes contained in JAR files. Many programmers publish their code for others in the form of these JAR files, with some associated license specifying restrictions on their use. This greatly helps programmers by allowing them to use development time that would otherwise be spent on coding these classes elsewhere. This means they can use their time writing code that is more specific to the project they are working on. Essentially, the fact that there are so many JAR libraries available on the internet means that Java programmers rarely have to reinvent the wheel; that is, code something that has already been written.

Some of the licenses associated with code libraries are more restrictive than others. For example, any code released under the GPL license requires any code produced using that library to be released itself as open source software. This would mean any project produced using this code would have to forfeit its source, making itself essentially available for free, which would ruin any commercial viability.

The goal of this project is to help identify any code in a library which may come from a source with a more restrictive software license. The application produced from this project would allow software designers and developers to identify any code with a restrictive software license inside of a JAR library they've chosen to use with their project. This would allow them to know what libraries they can and cannot use in the case that they are looking to sell their project, and help them to avoid having to open source their work.

In order to identify code with restrictive licenses inside of a JAR library, this project will take known JAR libraries and be able to identify if any part of those libraries are contained within a library the user would wish to test it against. In order to achieve this goal, the project will make a fingerprint of known JAR files, and use these fingerprints to test against unknown JAR files. This way, you can use these fingerprints, instead of the JAR files themselves, to test two JAR libraries against each other and see if one is a partial or complete copy of the other.

II. Background

There has been some similar archive comparison software released by other developers. One example of another piece of similar software is JAR Compare.

JAR Compare is a tool designed to show the changes made from one release of a JAR library to another. It takes the classes from the JAR file, decompiles them, finds changed lines of code, and returns the result to the user. Though this is useful for its designed purpose, it doesn't quite fit as a solution to the problem targeted by this project.

III. Approach

The fingerprint taken from a JAR file is very simple. The fingerprint itself stores information about file names, file sizes and the depth of files inside the JAR library. When this fingerprint is tested against another JAR library, each file inside of the original fingerprint is tested against each file in the destination JAR file. The goal of this is to determine if the file inside of the fingerprint exists in the JAR library its being tested against. The program initially tests the file size, and if there is an exact file size match, it is returned that the program is 100% certain for that one file that it is a copy. If the two files don't have exactly the same file size, it then checks the file name and the folder depth inside of the JAR file. If the file name and depth are the same as the fingerprint, it considers this a 66% match. If only the file name is the same and not the file depth, its taken as a 33% match. These certainties are summed for each file in the fingerprint, and is then divided by the total number of files in the fingerprint. This way, if the unknown JAR file contains more files than the fingerprint, but also every file from the fingerprint unaltered and in full, this is still considered to be a 100% match.

The decision to store such a simple fingerprint in this manner was derived from two core assumptions. First, that the likelihood is very high that any included restricting licensed code inside of a JAR library would have been included in the JAR library not maliciously, but by mistake. The second is that with a simple fingerprint, you can parse through, store, and test

against other JAR files much more quickly than you would a more complex fingerprint and algorithm.

The first assumption being made that the creator of the JAR file simply wasn't aware or cautious enough while constructing the JAR file. Taking this into account, we can assume they wouldn't have decompiled and altered any of the class files themselves, which would mean they would maintain their initial file size. On the chance that the JAR library creator did modify a class file (for whatever reason), as long as their intention wasn't malicious, they would most certainly maintain the same file name and location within their new JAR library. Taking this into account you can use these three variables to determine whether a file from a fingerprint matches a file from an unknown JAR library.

The second assumption was that a user wouldn't just want to test two JAR libraries against each other, but instead test a JAR library against a large stored database of JAR library fingerprints. There are many JAR libraries available on the internet, many of which include code with restrictive licenses. Testing a JAR library against one fingerprint is simply not enough to ensure that the JAR library doesn't contain any code that would force the project to become open source. With a simple fingerprint, you'd be able to create fingerprints and test multiple JAR libraries much faster than you would be able to with a more complex fingerprint and testing algorithm.

Below is a table describing the different things you could do to a file within a JAR library, and how the Identifier handles each case (in terms of changing a file name, size, or depth):

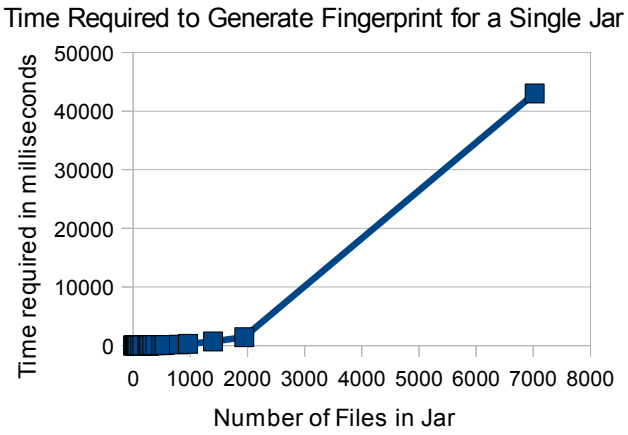
Change Name	Change Size	Change Depth	Certainty Returned
No	No	No	100.00%
Yes	No	No	100.00%
Yes	Yes	No	0.00%
Yes	Yes	Yes	0.00%
No	Yes	No	66.00%
No	Yes	Yes	33.00%
No	No	Yes	100.00%

As you can see, if two files are found to have identical file sizes, in all cases its considered to be a 100% match. Once the file size has changed, the certainty goes down significantly. A File with a changed file name and file size, regardless of depth, is undetectable by this solution. The only case in which the file name and size of a class would change, however, would be if the user packaging the JAR had malicious intent. These cases are not within the scope of this solution, have been ignored.

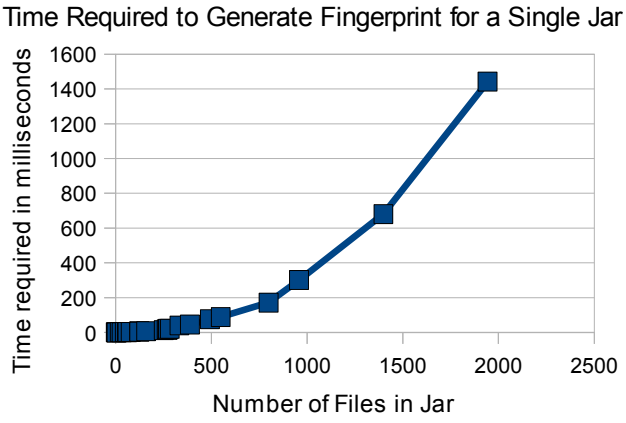
IV. Results / Validation

There are several steps to the process described in this project that can be extensively tested. Shown below are tests related to the speed to generate a single fingerprint, the speed related to generating a group of fingerprints, the time required to test a single unknown jar of a specific size, as well as the accuracy of the system in identifying copied JAR libraries.

The first test run was a check to see how long it took to generate a fingerprint against the number of files in a single JAR. The results of the first test are as follows:

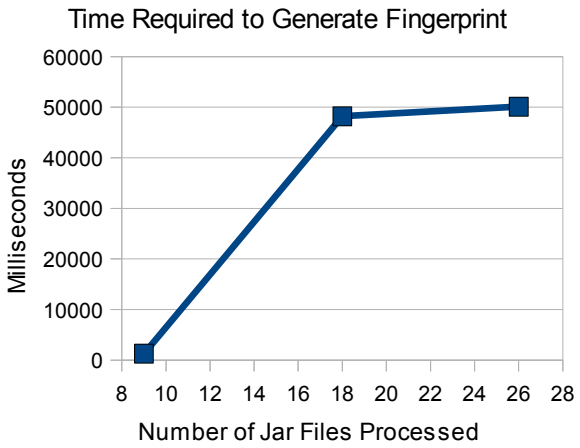


As you can see, there is an outlier. This was the file J2EE.jar, which contained over 7,000 entries. This is 5,000 files more than the next closest JAR library. With this entry removed, the graph appeared as follows:

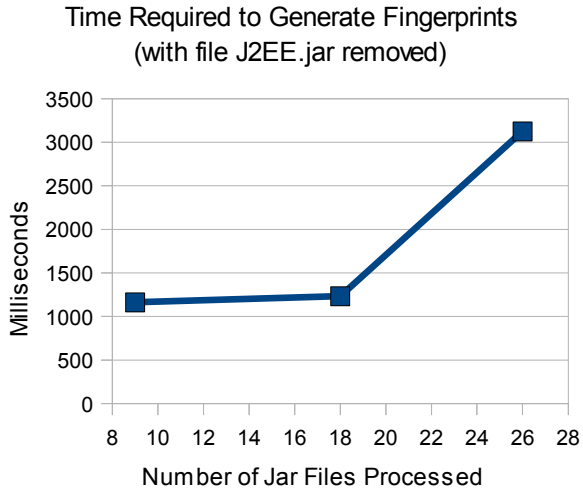


From this graph you can see that most JAR libraries had fewer than 500 files, and took a trivial amount of time to process. The more files added, however, the longer the process took, and the time seemed to increase exponentially. For most future tests, the file J2EE.jar was removed (unless otherwise specified).

The second test conducted was to show how long it would take to generate fingerprints from all 26 randomly selected JAR Libraries at once. The time elapsed for all libraries, including J2EE.jar, is showed in the graph below (time was recorded after 9, 18, and 26 entries processed).

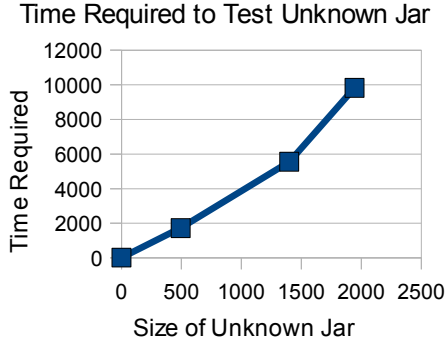


This graph shows a large increase in time between the first and second recordings. This was as a result of the J2EE.jar file. With this file removed, the results were as follows:



This graph shows the time to process all JAR files, with the J2EE.jar file removed. It is still easy to see a fluctuation based on the number of files held within a single JAR library. Of course, ideally you would only want to generate the fingerprints each JAR library once, and store them in some sort of flat file or database.

With this in mind, the following graphs show the time required to test the 25 fingerprints (J2EE.jar excluded) against the smallest, two mid-sized, and a large JAR file:



This shows that the time required to test the fingerprint database against an unknown JAR file seems to increasing fairly linearly for the first 3 data values, then exponentially as the size of the unknown JAR file increases significantly. This shows that the algorithm produced would work best when tested against JAR files of less than 1000 files. Of course, the time it takes to calculate the results may not be important as the accuracy determined within the results themselves.

The accuracy test was to test each JAR file from the 25 total JAR files (excluding J2EE.jar) against the other 25. This was to see if the correct JAR file was returned as the source of the unknown JAR file. In each case, the correct JAR file was returned from the detector. This shows that, while the system may take a significant amount of time to generate fingerprints, and test against large JAR files, in the end the generated results are worth the time waited.

V. Conclusion

Determining shared resources among two archives may seem like a simple task, but it grows more complex as the accuracy required and size of the archive increase. The solution shown in this paper would be best implemented while working with smaller JAR files, but when manipulating large numbers of JAR files with more than 1000 entries, this solution would be slow. Depending on the sample size, and the number of files held within each of the JAR libraries within a sample, it could easily take hours to determine the source of an unknown JAR library.

Future studies may wish to determine ways to calculate the certainty more accurately, and more quickly. It may even be possible to create fingerprints and generate results in significantly less time than was demonstrated by the algorithm shown in this paper. Also, future work may want to increase the sample size of the JAR files used, and increase the duration of testing to cross thousands of JAR files with thousands of entries.

References

[1] Jar Compare - http://www.extradata.com/products/jarc/

Authorship

Grant McNeil is a 4th year level Computer Science student at Carleton University. He learned BASIC programming at a very young age and branched out to learn Visual Basic, Java, C/C++, Python, and many other programming languages throughout his adolescence.