Fingerprint Identification of Open Source Software

Matthew Ng (mng@scs.carleton.ca), Andrew McCallum (amccall2@scs.carleton.ca),
School of Computer Science, Carleton University

Abstract

In the world of open source software there is a problem concerning the determination of the origin of code being contributed to open source projects. Does the modified program comply with license it was set for? Can this program be distributed for profit? In most cases people have to manually go through the programs to figure out licenses. The proposed solution will help lawyers, programmers, and whoever is asking to quickly identify known programs to unknown programs so an identity could be made, or licenses can be determined.

I. Introduction

A major problem that the open source community is being faced with is the recognition of contributions to a piece of open source software (OSS) that have come from other pieces of software. This may not seem like a huge problem, but with regards to software licensing (among other issues) it is a huge problem; not all licenses are compatible with each other. In the case of our project, the input provided consists of a JAR file from unknown origin and we must identify how similar it is to a JAR file that we have on file from a known source.

Potential reasons for why we would need to solve a problem like this: software licensing violations, copyright infringement (or other intellectual property violations), or simply to identify a mislabelled program. In open source software it is always important to keep track of what kind of licenses are being used within the software; certain licenses prevent software being utilised/included in other projects because it violates the terms of it license. There are lot of cases that prevent good software being released due to the fact that they violate terms of the software it was built off of. The solution to this problem will help teams to scrub the contributions to their OSS software to ensure that it is from a source that is compatible with its license and can be used in the existing software. In the case of the Eclipse Foundation, who has 100,000+ contributions, each one must be checked to ensure that no intellectual property has been infringed upon and that any license on the contribution is compatible with the existing license on the project.

It is also important to make sure that the proper contributors are being recognized for their works. Sometimes contributors will grab a piece of code, strip out a few variables/comments and/or rename some classes and call it their own work. Our solution is designed to sniff out these changes and provide the user with a percentage that indicates how closely the contribution resembles another piece of OSS software. Going back to the Eclipse example, with our solution they can take the contributed JAR file and compare it to known JARs that have already been deemed safe for use within the Eclipse software.

The objective is to create a fully functioning program using Java that will take the given interface classes, implement a solution around those and be able to generate fingerprints from a collection of known JAR files. A fingerprint is a unique identifier that will be generated from specific JAR files that are known to be from a trusted source. Just like a human fingerprint, every fingerprint of a JAR file is unique and no two unique JAR files can have the same fingerprint. Once fingerprints have been generated from JAR files of a known origin, they can be fed into the application and it will compare it with others to give a confidence level of certainty that the fingerprints match.

II. Background

As issues with mismatching of software licensing has been a problem since the invent of open source software, there are similar open source projects already available that will compare two JAR files and outline the differences between them.

Eclipse has a built in compare function for JAR files that will visually show the differences between contents of the JAR. However, this “easter egg” feature of Eclipse does not provide a solution to our fingerprint problem; it only visually shows the comparison and differences of a JAR file. The problem that needs to be solved for the purposes of our project is the generation of a confidence level (in percentage form) of how similar an unknown JAR file is to a known JAR file. Although this tool is half the solution of the problem, this would not be the ideal path to build off of to solve the problem.

In addition to the Eclipse functionality, there is also a third-party Java console application available called Jar Compare (jarc). As the name suggests, it will compare two JAR files together, however it will not run unless the full Java Development Kit (JDK) is installed. Since it will not run under the Java Runtime Environment (JRE), this application would not be ideal to build the solution for our project off of.

Finally, another piece of software that is available is the zdiff or zcmp command in Linux. These commands are available to compare compressed files.

III. Approach

The approach we are taking to solve this problem is to create a XML fingerprint of an unpacked JAR file. Using a utility class (JarUtils) we can manipulate the given JAR file and extract it to a temporary directory. This will allow us to get access to the files and their receptive contents/attributes that will be used to create our fingerprint. The XML fingerprint will consist of a serialised instance of the Fingerprint class, which in turn contains information on files from the JAR (i.e. file name, file sizes, MD5 hashes, file types, and class file information) using the respective class (FileInfo). The information being stored will be stored in primitive types like strings (i.e. file name, MD5 hash) and longs (file sizes). For things like file types and class files, they will be stored in Hash Maps. A "confidence comparison" function will go through all of the collected data, run tests upon it, and produce a confidence level of how similar the JAR of unknown origin is to a fingerprint generated from a JAR of known origin.

Execution will begin in the application class (FingerprintApp), with the program branching off depending on if the comparison or generation functionality is selected.

When selecting the generation functionality, a JAR filename is provided to the program, from which a fingerprint file is generated. In order to do this, the JAR contents must be extracted and analyzed (as well as the JAR file itself). To facilitate this information gathering, three classes were created for this purpose: JarInfo, FileInfo, and ClassInfo. The former is for information pertaining to the JAR file, whereas the latter holds Java class-specific information (i.e. methods, fields, etc.) and FileInfo holds general file information. In terms of general file information that's contained in the FileInfo class, the following are collected and/or calculated:

- file size,
- MD5 hash of the file contents, and
- file name.

ClassInfo and JarInfo complement that data; ClassInfo stores methods and fields contained in the class, while JarInfo stores a list of files (of type File) and metrics regarding file types (i.e. totals for each file type) contained in the JAR.

Once all of the files in the JAR have been iterated over and analyzed, the fingerprint can now be exported to XML. This functionality is facilitated by the XStream library, which serializes object instances to XML (and supports reading them back in). For an example of an exported fingerprint, refer to Appendix B – Exported Fingerprint.

IV. Results and Validation

For the comparison functionality, a fingerprint is provided along with a JAR file at execution time; the specified JAR will be tested with the supplied fingerprint. From the supplied JAR, an in-memory fingerprint is created, while the fingerprint supplied is imported using the previously mentioned XStream library. Once the fingerprint has been loaded and a second fingerprint for the JAR file has been created, the "confidence comparison" can begin!

Pseudo code for the overall confidenceCompare() is as follows (after calling confidenceCompare() in the Fingerprint instance):

```
• jarInfo.confidenceCompare()
  o compare MD5 hashes, if same then return 100%
  o compare filesize same, if same then return 30%
• if returned > 80%, then return
• for larger set of fileinfo instances
  o for smaller set of fileinfo instances
    ▪ fileInfo.confidenceCompare()
      • compare MD5 hashes, if same then return 100%
      • compare file sizes, return 70% if same
        o if file type matches then return 80%
      • if class, then classInfo.confidenceCompare()
        o return average based on number of fields and
          methods (plus applied weighting of 50%)
• if exactMatches > 60% (i.e. from MD5 matches), then
  return
• else, return average from fileInfo confidenceCompare()s
```

The confidence compare (i.e. how similar the JAR files are to each other) makes use of the IConfidenceCompare interface, with all classes that need to be involved in the comparison implementing it. The classes that implement (and are thus involved in the comparison) are JarInfo, FileInfo, ClassInfo, and most importantly, Fingerprint. An iteration of fingerprint comparison functionality involves the calling of confidenceCompare() on the following objects:

- all instances of ClassInfo & FileInfo (in both fingerprints),
- the single instance of JarInfo (in both fingerprints), and
- the instance of Fingerprint (on fingerprint imported from file).

In order to iterate through all of these instances, the "root" call of confidenceCompare() is in the Fingerprint object; all other calls to instances begin from that call. When the "root" confidenceCompare() is called, an instance of ConfidenceCompareComments is supplied to the method, which stores comments added along the full execution of the call. The execution hierarchy illustrates the order of the confidenceCompare() calls; it also indicates which objects hold others (i.e. ClassInfo instances are held within FileInfo instance). The execution hierarchy is as follows:

- Fingerprint
    - JarInfo
    - FileInfo
        - ClassInfo

The implementations of confidenceCompare() within each class run tests against the JAR-generated fingerprint data and generate a confidence of similarity percentage based on the test results. Each respective implementing class tests the information contained within its instance with the data contained within its JAR-generated counterpart. Where a specific counterpart in the JAR-generated fingerprint does not exist (i.e. for all the FileInfo instances), all instances are tested against each other.

Once the execution of the confidenceCompare() call in the "good" Fingerprint instance has returned, the FingerprintResult can be generated based on the percentage return value and all associated comparison comments collected on the full execution. When the FingerprintResult is generated it is written to the console, outputting all comments and finally, the confidence of similarity percent.

For quantitative examples of generation times and comparison examples, please refer to Appendix A.

V. Conclusion

The implementation that has been built for the purposes of the project is a good start at trying to solve the overall problem of software from unknown origin winding up in OSS and causing potential licensing issues.

There are many improvements that could be added to the implementation, such as threads while conducting confidenceCompare()s, as well as the addition of more test cases. This software is only as good as the test cases that are built into it.

References

[1] http://www.javalobby.org/java/forums/t19700.html
[2] http://www.extradata.com/products/jarc/
[3] http://download.oracle.com/javase/1.4.2/docs/api/java/util/jar/package-summary.html
[4] http://linux.about.com/library/cmd/blcmdl1_zdiff.htm
[5] http://xstream.codehaus.org/

VI. Appendix A – Quantitative Examples

The following table indicates the average time to compute fingerprints for a selection of well-known JARs:

Product	Jar	File Size (KB)	# Files	# Classes	Avg FP Gen Time	Avg Time/File	Test 1	Test 2	Test 3
Apache Ant	Ant.jar	1479	886	873	4.395s	0.005s	5.16	4.07	3.96
Apache commons IO 2.0	Commons-io-2.0.jar	156	109	104	1.960s	0.018s	2.12	1.84	1.20
Apache Commons Codec 1.4	Commons-codec-1.4.jar	158	35	30	0.604s	0.017s	0.63	0.62	0.56
Apache Commons Compress 1.1	Commons-compress-1.1.jar	57	82	77	0.716s	0.009s	0.76	0.71	0.68
Apache Commons CLI 1.2	Commons-cli-1.2.jar	41	27	22	0.530s	0.020s	0.55	0.55	0.50

The following table indicates the confidence comparison results (all tests were conducted on the fingerprint generated from commons-cli-1.2.jar):

Action	Expected Percent	Percent Result
Real Jar	100%	100%
Renamed Jar	100%	100%
Added extra file	> 90%	96%
Added extra class	>90%	96%
Removed a class	> 90%	96%
Removed a file	> 90%	96%
Renamed file/class	100%	100%
Supplied commons-io-2.0.jar	< 10%	1%
Supplied ant.jar	< 10%	1%

VII. Appendix B – Exported Fingerprint

The following is a basic class exported as a fingerprint:

```
            <ENTRY>
              <STRING>CLASS</STRING>
              <INT>1</INT>
            </ENTRY>
            <ENTRY>
              <STRING>MF</STRING>
              <INT>1</INT>
            </ENTRY>
          </FILE-EXTENSIONS>
        </JAR-INFO>
        <FILEINFO>
          <ENTRY>
            <STRING>HELLOWORLD.CLASS</STRING>
            <FILE-INFO>
              <FILE-SIZE>610</FILE-SIZE>
              <MD5-HASH>6A90D73DA2F0DEC1B805717192C324C9</MD5-HASH>
              <FILE-TYPE>CLASS</FILE-TYPE>
              <FILE-NAME>HELLOWORLD.CLASS</FILE-NAME>
              <CLASS-INFO>
                <METHODS>
                  <METHOD>
                    <CLASS>ORG.LAME.PROJECT.HELLOWORLD</CLASS>
                    <NAME>MAIN</NAME>
                    <PARAMETER-TYPES>
                      <CLASS>[LJAVA.LANG.STRING;</CLASS>
                    </PARAMETER-TYPES>
                  </METHOD>
    << ALL DEFAULT OBJECT METHODS HAVE BEEN OMMITTED >>
    </METHODS>
                <FIELDS/>
              </CLASS-INFO>
            </FILE-INFO>
          </ENTRY>
          <ENTRY>
            <STRING>MANIFEST.MF</STRING>
            <FILE-INFO>
              <FILE-SIZE>66</FILE-SIZE>
              <MD5-HASH>9CF97B6BB17EE0915001D76135789F80</MD5-HASH>
              <FILE-TYPE>GEN</FILE-TYPE>
              <FILE-NAME>MANIFEST.MF</FILE-NAME>
            </FILE-INFO>
          </ENTRY>
        </FILEINFO>
</FINGERPRINT>
```