Tree Based Approach to JAR Identification

Lester Mundt
Carleton University
1125 Colonel By Drive
Ottawa, Ontario, Canada
lmundt@gmail.com

ABSTRACT

This document describes an approach used to compare Java JAR files against pre-generated fingerprints. The fingerprints can be stored to compose a library of known JAR files. The same tool can then be used as a comparison tool allowing users to identify if a JAR is from a known source. This allows project owners to ascertain the source of a suspect JAR which in turn helps prevent unintended intellectual property violations. The algorithm used in this implementation uses the structure of the JAR's contained packages and classes as a means of identification which bypasses many potential obfuscation approaches that can be applied to JAR files.

General Terms

Algorithms, Experimentation, Legal Aspects, Verification.

Keywords

JAR, obfuscation, software fingerprint, trees, tree comparison, ProGuard, Open Source.

1.INTRODUCTION

Developers now have access to a large body of code the size of which was unimaginable in the time preceding the internet. While this provides a significant learning potential for developers, copyright holders of many products are faced with significant risk. The risk that concerns these copyright holders is the inclusion of intellectual property (source or binary code) that could imperil or otherwise challenge the ownership of their own products. Even in open source projects where one might think the ownership of intellectual property becomes unimportant it has to be remembered that each open source body of work is distributed under a license and different open source licenses may actually be incompatible [1] Thus knowing the true identity of all the potential sources of intellectual property in a product become paramount.

1.1.Problem

There needs to be ways to identify if additional sources of intellectual property are original or copied so it can be determined if the inclusion of the specific piece of intellectual property poses issues. Relative to software there are really two avenues to look at binary files and source code.

Open Source Software, November 15, 2010, Ottawa, Ontario, Canada.

For the purposes of this implementation we will focus on one small portion of the identity problem. We will confine ourselves to the Java programming language and the granularity of identification will be JAR files. To expand, instead of focusing on a single piece of source and trying to deduct if classes, functions or even smaller measurements of code have been plagiarized we will look at the unit of distribution in Java the JAR and attempt to identify if it is a copy of a known source.

1.2.Motivation

For both the open source and closed source communities there are significant reasons to want to know if intellectual property contained in their own works poses any threat to their own rights in regards to original works. Failure to do this can have significant financial risk [2]. Reliable methods of identifying the source of intellectual property can prove invaluable.
In fact there are currently companies that charge exorbitant amounts for the the service[3,4].

1.3.Goals

To help solve the problem an application will be written that can do two things:

I. Produce a fingerprint from an arbitrary jar.
II. Compare an arbitrary jar to a preexisting fingerprint and present the likelihood of them being the same.

These two functions can be combined to allow large libraries of fingerprints to be generated and then automated tests that could compare jars of unknown origin against the library.

With the extensibility and breadth of a library comparison envisioned two other non-functional requirements arise:

I. The fingerprints should be small.
II. The comparisons should be fast.

Additional goals are to keep accuracy fairly high and false positives need to be kept to an absolute minimum.

1.4.Objectives

We determined that with the goals in mind we wanted to develop an algorithm that would work in real world settings. We hypothesized that if a JAR has the potential for intellectual property conflicts with the rest of a project there are three likely ways in which it got there with each with varying degrees on the likelihood of being able to detect it:

1. It is a legitimate mistake and the developer who added the JAR to the project doesn't realize that the inclusion of the JAR presents a potential problem. In this case the jar would be in it's original form and easily matched.

2. The developer who added the JAR wasn't sure about the inclusion of the JAR and has made some attempt to hide the inclusion. Small modifications such changing the name of the JAR, refactoring packages to not include identifying names, etc may have occurred. This in turn increases the difficulty of determining the JAR's identity slightly.

3. Someone has gone to great efforts to make the identity of the JAR obscured. The developer adding it either did this themselves or received a JAR altered to such a degree they had been unable to determine the JAR could pose an intellectual property issue. Most likely the JAR will have been ran though a professional obfuscator causing massive changes to the JAR. This presents the most significant challenges to identifying the JAR's origin.

The three previous scenarios present several additional objectives in addition to the original goals. The algorithm should be robust in regards to obfuscation. To identify the accidental inclusion we wanted to be able to identify unmodified JARs with 100% certainty.

1.5.Outline

The rest of the document will be as follows. Section two will provide some background information that pertains to the problem domain and to the implementation domain. Section three will illustrate the approach I had taken detailing features of the design and the decisions made in the implementation. Section 4 will provide some results from testing the implementation. Section 5 will review the goals and how close the implementation came to reaching them. Additionally Section 5 will provide ideas for future work.

2.BACKGROUND

2.1Technical Issues and Constraints

JARs and obfuscation methods and Java.

2.2Previous Research

2.3Related Open Source Software

Expand

3.APPROACH

With speed and a small fingerprint size being the two of the primary objectives an examination of JARs, and classes was discussed. Originally we envisioned a tree structure to contain class nodes in quickly processable way and to develop class based comparison upon some comparable and unique attribute set of the classes was envisioned.

We then wondered if the structure of the JAR itself might present enough of unique signature. It would help speed as the we would only have to process the JAR itself and we could skip introspection of the classes.

Using tree structures would involve direct comparison potentially in the form of subtree detection. The trees involved were to be reflective of the hierarchical structure within the JAR so the trees couldn't be balanced or binary as the original structure would be lost.

It was then realized that the normal method of detecting similarity between two trees would be subtree detection, testing if one tree fit inside the other and comparing their sizes to see if they were identical or if one was a subtree of the other and how many nodes were in common. In our unbalance non-binary tree the comparison would involve backtracking which would slow the comparison down considerably. Additionally small changes to the structure would cause the comparisons to fail.

With robustness against obfuscation an additional goal this form of comparison would be too brittle. However we realized that using the structure of the JAR could make the comparison immune to some of the methods of obfuscation that obfuscators regularly use. The renaming of classes of methods wouldn't matter. Partial flattening was identified as the obfuscation method that would pose the most significant problem for a structure based method. So a comparison method that could circumvent this issues would be highly desirable.

We augmented the tree structure so a level of the tree could be examined as a single list. A level consists of all nodes at the same depth in the tree. In a normal tree structure this would be accomplished by traversing all branches and returning nodes at the specified depth. This level approach allows us to iterate over the tree and the number levels is simply the height of the tree.

We then developed a level comparison algorithm that would allow tree of different sizes to be compared.

Finally it had been observed by us that many JARs do not distribute source and we wanted to avoid the speed issues with using decompilers so it was decided to use the JARs and the binary class files they contain. This lead us to using the using a custom class loader to dynamically add the classes contained in the JAR at runtime.

3.1Design

The fingerprinting is executed in a fairly simple way. In response to our decision to forgo the use a decompiler the most significant challenge was creating a custom class loader to load all the classes contained in the JAR at runtime.

As the JAR is processed a tree structure is built representing the hierarchical package structure inside the JAR

Table 1. Algorithm for generating a fingerprint

```
Input: A JAR file (jar)

Output: ASCII string representing the
hierarchy of the JAR
```

```
Open jar
Create empty tree
  foreach entry in tree
    Get parent
    if entry->parent exists in tree
      Add node(entry,parent) to tree
    else
      Find parent in jar
      Add node(entry, parent) to tree
  endfor
```

```
Input: A JAR file (jar)

Output: ASCII string representing the
hierarchy of the JAR
```

```
Find
Input: A name that for a new node (parent)
       A JAR file (jar)
       A tree (tree)
Output:A new node that is a parent
```

```
Get grandparent
if grandparent exists in tree
  Add node( parent, grandparent) to tree
else
  Find grandparent in jar
  Add node( parent, grandparent) to tree
return node
```

The comparison is done between a fingerprint and a JAR.

Add comparison stuff here

4.RESULTS

4.1.Test Setup

All tests were executed with a selection of jars that are indicative of what can be found on a Java developers machine. However server and client JARS are JARs of our design that we have the ability to modify for any special cases we wanted to test.

All tests were ran on a MacBook Pro with a 2.1GHz Core 2 Duo Intel Processor, 4GB of RAM, running on OSX 10.6 and a 1.6 JVM.

4.2.Fingerprint Size and Speed

The first test was to look at the size of the resulting fingerprint and the speed in which it can be generated.

Table 1. Fingerprint sizes and creation speed

JAR Name	JAR Size(bytes)	Fingerprint Size (%)	Time (ms)
androidprefs.jar	2705	25.101663586	22
client.jar	59276	8.7860179499	213
Java2D.jar	396061	2.0102963937	408
javax.mail_1.4.0.jar	320972	5.6297745598	723
junit.jar	121204	4.9008283555	180
org.apache.commons.lang_2.1.0.jar	222427	4.604207223	384
org.apache.commons.lang_2.3.jar	259489	4.4314017165	290
server.jar	44946	9.1687803142	48
swt.jar	1388610	4.7955869539	1020

The fingerprints seem to be a reasonable size with one notable exception. Androidprefs.jar is the smallest JAR used and since the content of the fingerprint file is ASCII and uses package names the encoding is a significant percentage of the size of the JAR. As the JAR size increases the size penalty imposed by the uncompressed ASCII become much less significant.

The speed appears to have O(n) growth with swt.jar being four times as large as the org.apache.commons.lang_2.3.jar and taking about four times as long to generate the corresponding fingerprint. The growth order continues with server.jar taking one tenth as long as Java2D.jar and being approximately one magnitude smaller.

4.3.Fingerprint Comparisons

Appendix 1 shows the compete data set for comparisons we will only show a subset here. We have chose some illustrative cases that we will examine here.

Table 2. Androidprefs.jar comparison table

Fingerprint compared against	Relative JAR Size (%)	Time (ms)	Match (%)
androidprefs	1.00	14.0	100.00
client	21.91	30.0	72.50
Java2D	146.42	56.0	24.36
javax.mail_1.4.0.v201005080615	118.66	64.0	28.10
junit	44.81	18.0	36.27
org.apache.commons.lang_2.1.0	82.23	19.0	70.00
org.apache.commons.lang_2.3.0	95.93	12.0	70.00
server	16.62	8.0	74.00
swt	513.35	187.0	72.50

Androidprefs.jar is the smallest JAR tested. As illustrated in Table 2. the next smallest JAR is server.jar and it's more than 16 times as large.

This table starts to show that just using the structure of the JAR may be insufficient to avoid the possibility of false positive since we have 5 other jars with a 70.00% match that are completely unrelated to androidprefs. Interestingly size doesn't seem to matter as server.jar the next smallest jar has a similar match percentage as swt.jar the largest JAR size.

Finally by examining the comparison times we not that that speed illustrates a quick time that has a linear best fit but fluctuates likely due to system overhead as task switches.

Table 3. Swt.jar comparison table

Fingerprint compared against	Relative JAR Size (%)	Time (ms)	Match (%)
androidprefs	0.01	283.0	70.00
client	0.27	175.0	56.10
Java2D	1.78	129.0	38.89
javax.mail_1.4.0.v201005080615	1.44	140.0	34.30
junit	0.54	152.0	48.77
org.apache.commons.lang_2.1.0	1.00	130.0	100.00
org.apache.commons.lang_2.3.0	1.17	134.0	96.86
server	0.20	129.0	54.13
swt	6.24	153.0	47.87

The org.apache.commons.lang_2.1.0.jar was compared against the fingerprints of the other JARs. The apache JAR was chosen because there is a slightly newer version of it with a similar structure amongst the test set as well. The other version of the apache JAR produced a match percentage of over 95 percent which illustrates that algorithm works on similarly structured JARs. We also see the same result of 70 percent match percentage with the android JAR that was discussed in the analysis of Table 1. this is expected because the algorithm just tries to identify the percentage of commonality between the fingerprint and the tested JAR.

Table 4. Apache-commons_2.1.jar comparison table

Fingerprint compared against	Relative JAR Size (%)	Time (ms)	Match (%)
androidprefs	0.00	928.0	72.50
client	0.04	725.0	67.21
Java2D	0.29	795.0	26.83
javax.mail_1.4.0.v201005080615	0.23	739.0	20.68
junit	0.09	798.0	36.27
org.apache.commons.lang_2.1.0	0.16	741.0	47.87
org.apache.commons.lang_2.3.0	0.19	766.0	48.43
server	0.03	745.0	60.16
swt	1.00	776.0	100.00

The swt.jar is the final JAR selected for analysis. This jar was chosen because it is the largest of the JARs and it's and ideal candidate to refute the possibility that the comparison algorithm is based primarily on size. It can be observed in Table 3. that the swt JAR actually produces the highest match percentages against the smaller libraries. This seems to makes sense since in such a large library it is entirely feasible that portions of it could have structures identical to some of the smaller JARS. The time stays consistent because the other JARs are at best a fifth the size of swt JAR and thus are negligible to the amount of time that it takes to process their fingerprints the bulk of the time would be spend processing the swt JAR.

4.3.Obfuscated Fingerprint Comparisons

In this test the Open Source obfuscation tool ProGuard was used to alter JARs. ProGuard was used with default obfuscation settings enabled for Test 1. This setting completely destroys the class name, eliminates unused classes, and changes the size of the compiled jar dramatically. Test 2 had the ProGuard setting aggressively attack the package structure in addition to the previous changes. Test 3 adds to the previous settings by attempting to merge interfaces altering the class inheritance structure. All tests are the fingerprint of the original JAR against the obfuscated JAR.

Table 5. Swt.jar comparison table

JAR	Test 1 Match(%)	Test 2 Match(%)	Test 3 Match(%)
server	78.51	46.5	46.51

Table 5. shows that 100 percent match of a the previous self tests drops to just under 80 percent when a obfuscation tool is used. Internal examination of the obfuscated JAR showed all class names had been changed and several classes were removed.

Further customization of the obfuscation procedure oriented towards the destruction of the original structure reduces the match percentage below 50 percent. At this point it is well in the realm of almost any JAR compared to any JAR as per the capabilities of the comparison approach used in this paper.

5.CONCLUSION

5.1.Goal Attainment

While we had postulated that the internal JAR structure of packages and classes could be used to generate a fingerprint and easily compared using tree based data structures it seems that the most aggressive settings of obfuscators can defeat our detection algorithms.

The speed of the algorithm seems quite good. Comparisons seem to happen in O(n) time according to the observed data.

The accuracy of comparing a JAR with it's own fingerprint is 100% in all cases.

The accuracy against obfuscation is quite high with standard setting on with setting enabled that aggressively alter the package hierarchies there is a significant loss of accuracy.

Additionally with only the structure as the form of comparison the potential for false positives seems to rise. In initial tests two completely unrelated jars can produce a result indicating similarity of up to 70 percent.

5.1. Contributions

The quick comparison speed and minimal fingerprint size make the tree based approach a candidate to augment other detection algorithms. Potentially this approach could be used as a first pass to quickly eliminate a significant portion of potential candidates and reduce the number of in depth comparisons needed.

5.1. Future Work

Further tests with a larger variety of obfuscators would help to identify the capabilities of current obfuscators and also identify further weaknesses in the algorithm.

One potential avenue to decrease the likeliness of false positiveness and to increase robustness against obfuscation could be to augment the tree structure with the inheritance hierarchy of the classes us both parent classes and interfaces.

Additionally having weighting on the level comparisons could reduce the amount of false positives.

Finally experimenting with exchanging the level bases comparisons for subtree comparisons while decreasing the speed of the comparisons might provide an increase in accuracy.

6. REFERENCES

[1] Brown, Carson, Barrera, David and Duego, Dwight. 2009. FiGD: An Open Source Intellectual Property Violation Detector. *The School of Computer Science, Carleton University*. DOI= www.ccsl.carleton.ca/~dbarrera/personal/figd.pdf.

[2] *SCO-Linux Controversies*. Wikipedia. DOI= . http://en.wikipedia.org/wiki/SCO-Linux_controversies

[3] Palmida Software. Accessed November 15, 2010. *Palamida Software*. DOI= http://www.palamida.com/products.

[4] Black Duck Software. Accessed November 15, 2010. *Black Duck Software.* DOI= http://www.blackducksoftware.com/protex.

Appendix 1. Complete comparison data

JAR Name	JAR Size (bytes)	Fingerprint Name	Fingerprint Size (bytes)	Time(ms)	Result (% match)
androidprefs	2705	androidprefs	679	14	100.00
androidprefs	2705	client	5208	30	72.50
androidprefs	2705	Java2D	7962	56	24.36
androidprefs	2705	javax.mail_1.4.0.v201005080615	18070	64	28.10
androidprefs	2705	junit	5940	18	36.27
androidprefs	2705	org.apache.commons.lang_2.1.0	10241	19	70.00
androidprefs	2705	org.apache.commons.lang_2.3.0	11499	12	70.00
androidprefs	2705	server	4121	8	74.00
androidprefs	2705	swt	66592	187	72.50
server	44946	androidprefs	679	23	74.00
server	44946	client	5208	21	83.17
server	44946	Java2D	7962	22	28.17
server	44946	javax.mail_1.4.0.v201005080615	18070	27	28.96
server	44946	junit	5940	23	39.19
server	44946	org.apache.commons.lang_2.1.0	10241	21	54.13
server	44946	org.apache.commons.lang_2.3.0	11499	27	53.51
server	44946	server	4121	22	100.00
server	44946	swt	66592	41	60.16
client	59276	androidprefs	679	157	72.50
client	59276	client	5208	106	100.00
client	59276	Java2D	7962	112	28.06
client	59276	javax.mail_1.4.0.v201005080615	18070	84	34.53
client	59276	junit	5940	101	36.27
client	59276	org.apache.commons.lang_2.1.0	10241	62	56.10
client	59276	org.apache.commons.lang_2.3.0	11499	74	55.24
client	59276	server	4121	68	83.17

JAR Name	JAR Size (bytes)	Fingerprint Name	Fingerprint Size (bytes)	Time(ms)	Result (% match)
client	59276	swt	66592	194	67.21
junit	121204	androidprefs	679	93	36.27
junit	121204	client	5208	54	36.27
junit	121204	Java2D	7962	97	40.39
junit	121204	javax.mail_1.4.0.v201005080615	18070	152	52.16
junit	121204	junit	5940	69	100.00
junit	121204	org.apache.commons.lang_2.1.0	10241	172	48.77
junit	121204	org.apache.commons.lang_2.3.0	11499	186	48.77
junit	121204	server	4121	97	39.19
junit	121204	swt	66592	139	36.27
org.apache.commons.lang_2.1.0.v201005080500	222427	androidprefs	679	283	70.00
org.apache.commons.lang_2.1.0.v201005080500	222427	client	5208	175	56.10
org.apache.commons.lang_2.1.0.v201005080500	222427	Java2D	7962	129	38.89
org.apache.commons.lang_2.1.0.v201005080500	222427	javax.mail_1.4.0.v201005080615	18070	140	34.30
org.apache.commons.lang_2.1.0.v201005080500	222427	junit	5940	152	48.77
org.apache.commons.lang_2.1.0.v201005080500	222427	org.apache.commons.lang_2.1.0	10241	130	100.00
org.apache.commons.lang_2.1.0.v201005080500	222427	org.apache.commons.lang_2.3.0	11499	134	96.86
org.apache.commons.lang_2.1.0.v201005080500	222427	server	4121	129	54.13
org.apache.commons.lang_2.1.0.v201005080500	222427	swt	66592	153	47.87
org.apache.commons.lang_2.3.0.v201005080501	259489	androidprefs	679	147	70.00

JAR Name	JAR Size (bytes)	Fingerprint Name	Fingerprint Size (bytes)	Time(ms)	Result (% match)
org.apache.commons.lang_2.3.0.v20100508 0501	259489	client	5208	157	55.24
org.apache.commons.lang_2.3.0.v20100508 0501	259489	Java2D	7962	267	35.53
org.apache.commons.lang_2.3.0.v20100508 0501	259489	javax.mail_1.4.0.v201 005080615	18070	187	32.97
org.apache.commons.lang_2.3.0.v20100508 0501	259489	junit	5940	260	48.77
org.apache.commons.lang_2.3.0.v20100508 0501	259489	org.apache.commons.lang_2.1.0	10241	174	96.86
org.apache.commons.lang_2.3.0.v20100508 0501	259489	org.apache.commons.lang_2.3.0	11499	210	100.00
org.apache.commons.lang_2.3.0.v20100508 0501	259489	server	4121	149	53.51
org.apache.commons.lang_2.3.0.v20100508 0501	259489	swt	66592	166	48.43
javax.mail_1.4.0.v201 005080615	320972	androidprefs	679	478	28.10
javax.mail_1.4.0.v201 005080615	320972	client	5208	399	34.53
javax.mail_1.4.0.v201 005080615	320972	Java2D	7962	254	57.06
javax.mail_1.4.0.v201 005080615	320972	javax.mail_1.4.0.v201 005080615	18070	242	100.00
javax.mail_1.4.0.v201 005080615	320972	junit	5940	245	52.16
javax.mail_1.4.0.v201 005080615	320972	org.apache.commons.lang_2.1.0	10241	241	34.30
javax.mail_1.4.0.v201 005080615	320972	org.apache.commons.lang_2.3.0	11499	257	32.97
javax.mail_1.4.0.v201 005080615	320972	server	4121	265	28.96
javax.mail_1.4.0.v201 005080615	320972	swt	66592	234	20.68
Java2D	396061	androidprefs	679	460	24.36
Java2D	396061	client	5208	193	28.06

JAR Name	JAR Size (bytes)	Fingerprint Name	Fingerprint Size (bytes)	Time(ms)	Result (% match)
Java2D	396061	Java2D	7962	163	100.00
Java2D	396061	javax.mail_1.4.0.v201005080615	18070	180	57.06
Java2D	396061	junit	5940	274	40.39
Java2D	396061	org.apache.commons.lang_2.1.0	10241	161	38.89
Java2D	396061	org.apache.commons.lang_2.3.0	11499	155	35.53
Java2D	396061	server	4121	151	28.17
Java2D	396061	swt	66592	231	26.83
swt	1388610	androidprefs	679	928	72.50
swt	1388610	client	5208	725	67.21
swt	1388610	Java2D	7962	795	26.83
swt	1388610	javax.mail_1.4.0.v201005080615	18070	739	20.68
swt	1388610	junit	5940	798	36.27
swt	1388610	org.apache.commons.lang_2.1.0	10241	741	47.87
swt	1388610	org.apache.commons.lang_2.3.0	11499	766	48.43
swt	1388610	server	4121	745	60.16
swt	1388610	swt	66592	776	100.00