Using Dependency Graphs to Detect Open Source Intellectual Property Violation

Gail Carmichael - gbanaszk@connect.carleton.ca and Siamak Sharif - SiamakSharif@hotmail.com

***Abstract*—Clone detection is important to the open source community, where projects are their source code are shared free of charge. The health of these communities is threatened when open source software is included in another product in violation of its license agreement. We developed a clone detector that improves on previous work by Brown et al [5]. We built a method dependency graph labelled with Brown et al's n-grams, used to concisely represent each method's byte code. Using an approximate graph matching algorithm, we efficiently report 100% similarity when comparing a Java Archive (JAR) file that has been modified without changing its meaning.**

I. Introduction

Open Source Software (OSS) provides users with both a product and the corresponding source code that created it, often free of charge. Users are allowed to modify or redistribute the product according to the open source license assigned to it. Companies that redistribute the product contrary to its license, either intentionally by hiding it inside their product, or unintentionally, become legally liable for license infringement. Furthermore, theft of open source intellectual property can threaten the health of open source communities, since those who would use it inappropriately are unlikely to contribute something in return and help the community grow. Safety can also be an issue when the origins of faulty software are unknown.

Possible legal action for license violation can be avoided by checking for inappropriate inclusion of open source software in a product before shipping it. With sufficiently fast techniques, a piece of software could be examined for this intellectual property violation as often as required.

One approach to solving this problem is to define a fingerprint that approximately represents a project without consuming much memory. A fingerprint for an open source project can be compared to a fingerprint defined for a piece of software suspected of intellectual property violation. A similarity percentage value is returned, and specific matches that warrant further investigation are reported. It is this approach that we focus on here.

More specifically, we examine a possible fingerprint format for Java ARchive (JAR) files. We improve upon the ideas of the FiGD intellectual property violation detector by Brown et al [5], who identify each method in a JAR by a representative n-gram based on that method's byte code. The number of common n-grams between two JAR files indicates the degree of similarity of those two projects. Only the compiled byte code is required for this method. While good results were obtained with this approach, the n-grams constructed are not robust for all types of code modifications.

We propose a modification to the process used by FiGD to create the n-grams representing individual methods. Our goal is to create an n-gram that will not change when, for example, a method is moved to another class, or a new argument is added to the method's signature. We explore the use of graphs built on the interdependency of methods to increase the uniqueness of the new n-gram, and to help reduce the occurrence of false positives when matching JAR files. In doing so, we aim to create a robust JAR comparison technique that is as accurate as FiGD, that remains accurate for new types of code modifications, and that continues to run efficiently.

Our approach is described in detail in Section III after the needed background is provided in Section II. Experimental results outline both the accuracy and running time performance of our fingerprint in Section IV, while a summary with suggestions for future work conclude the paper in Section V.

II. Background

Clone detection involves determining whether source code in software has been duplicated. This can be useful within a single software project to help keep maintenance costs low, but can also be used in the context of detecting inappropriate use of open source software in other projects.

Much research has been done on this topic [10], and Bellon et al provide an extensive comparison survey of clone detectors available as of 2007 [4]. Some clone detectors analyze source code using, for example, abstract syntax trees [3], frequent item sets [16], or a combination of lexical and local dependence analysis [7]. Semantic Design's Java CloneDR [6] displays code segments which are the same or almost the same. Other detectors work with compiled code, thus not requiring the source at all. Brown et al's FiGD [5] and University of Waterloo's Java Clone Detector [11] are examples of this.

We intend to build on the simplicity of Brown et al's FiGD intellectual property detector [5] using a collection of n-grams to represent all methods in a JAR file. These n-grams are computed by first obtaining the byte code for each method in the JAR. A sliding window of size n is passed over the byte code, moving one byte at a time. At each step, the contents of the window are added to a hash map that tracks how many times a particular window value has appeared. The most unique n-byte chunk is chosen to represent that method by finding the window value in the hash map that has the lowest count. Ties are broken by choosing the lowest window value in sorted order. A value of $n = 10$ was chosen as the most effective n-gram size. Once the n-grams are computed for all methods in one JAR file, the number of matching n-

grams in another JAR file is used to determine the similarity between those two JARs.

Dependency Finder [13] is an open source tool that finds and reports dependencies between classes, packages, and methods within a JAR file, as well as dependencies on outside libraries (including the standard Java libraries). This tool creates detailed dependency graphs that we use together with the above n-grams.

A common way to find the similarity of two graphs is to find their Maximum Common Subgraph (MCS). This NP-hard algorithm is used, for instance, in chemistry [12] and semantic web ontology [14] analysis. Given two graphs G_1 and G_2, the maximum common subgraph problem asks what the largest subgraph of $G_1$is that is isomorphic to some subgraph of G_2. Two graphs are considered isomorphic if there is a one-to-one mapping between every node in the first graph to a node in the second graph in such a way that all edges are preserved. We can imagine finding a way to overlay the the first graph so that all nodes and edges line up with their counterparts in the second graph. We explore this comparison technique for use with method dependency graphs.

The maximum common subgraph algorithm is implemented by SimPack [2], an open source package designed "for the research of similarity between concepts in ontologies or ontologies as a whole." The implementation of MCS is based on the algorithm described by Valiente in his book Algorithms on Trees and Graphs [15], and works on graphs that are not necessarily fully connected. We use this implementation in our experiments.

III. Approach

As mentioned above, our approach is based on Brown et al's FiGD intellectual property violation detector. The n-grams constructed as described in Section II can be used when comparing modified versions of a JAR file with the original, and result in high accuracy results for many cases. For instance, changing method names or variables, or adding or removing comments from source code, does not affect the reported similarity of the JAR files. However, moving a method from one class to another, or adding a superfluous argument to a method signature, does in fact change that method's n-gram and consequently the reported similarity of the JAR files. If enough such changes are made, FiGD might fail to detect the inappropriate inclusion of the modified JAR file.

In our approach, we consider the way methods interact with each other instead of as an unordered collection of individual methods. We build a graph with nodes that represent methods. A directional edge is created between two nodes if one method uses (depends on) the other. A node in this graph can be labelled with any string. For example, the n-gram generated from a method's byte code, the fully qualified name of the method, or a blank string may be used. These graphs can be compared using whatever graph comparison algorithm is most appropriate.

In the following subsections, the actual format of the fingerprint used in our experiments is described as well as the details of the graph created and compared.

A. Generating a Fingerprint From a Dependency Graph

The first stage in generating our fingerprint is creating a dependency graph from methods found in a JAR file's class files. We accomplish this with Dependency Finder. Given a JAR file name, Dependency Finder will generate a graph in memory that can be printed in plain text or XML format. While this tool can create graphs that include packages and classes as nodes and composition relationships as edges, we use it to obtain only method dependency graphs.

By default, Dependency Finder includes dependencies between programming elements within the JAR file and those outside; for example, many methods in the JAR will use methods from the standard Java libraries. Ignoring these external method calls would significantly reduce the size of the final graph, and potentially reduce the occurrence of false positives when matching projects that use external libraries in similar ways. However, since it is difficult to remove calls to these methods without changing the meaning of the method that is using them, they provide an excellent baseline for comparison between two JAR files. In our experiments, we compare results both with and without the inclusion of these methods.

The XML format used by Dependency Finder is simple and convenient. It lists methods as individual tags that are labelled with their fully qualified Java names, including package and class structure and argument types. Dependency edges are represented with 'inbound' and 'outbound' tags placed just inside the method they are associated with; these again are labelled with their fully qualified Java names.

The XML string obtained from Dependency Finder forms the core of our fingerprint. We also include a map with the XML that associates the unique fully qualified method names mentioned above with a custom label. This allows us to assign non-unique and potentially empty labels to the nodes in the dependency graph while ensuring that the graph can be rebuilt when the XML is read. The custom labels might be, as just one of many possibilities, the n-grams associated with the methods.

The advantage of this fingerprint format is that the dependency graph represented within is generic, and can be easily read into any type of graph object for later comparison. It is also not tied to a specific programming language, allowing for experimentation with options other than Java for the actual fingerprint comparison. However, the size of the fingerprint could easily be compressed, for instance by storing the bytes of the final graph object that would eventually be created from the XML. Thus, while we recommend using the string-based fingerprint while researching the best graph comparison techniques, we do suggest using a more space-efficient fingerprint once a technique has been chosen.

B. Using n-grams as Labels

As mentioned in Section III-A, the dependency graphs used in our fingerprints can be labelled using n-grams. This additional information about the nodes in the graph allows for comparison techniques that look at more than just the graph's structure. The advantage of n-grams is that they will not be sensitive to many modifications made to the source

code, whereas a simpler label, like the method's name, could easily change.

When using this type of label, we generate n-grams for methods when the fingerprint is created. This way, fingerprints for common open source projects can be stored in a database without their corresponding JAR files. We do not generate n-grams for external methods whose byte code we don't have access to, such as those in the standard Java libraries; instead, we simply use their fully qualified names as labels, since these cannot be changed.

We generate n-grams the way that FiGD did (described in Section II). However, we modify the byte code of a method first to ensure that common changes made to methods, such as moving them to another class, do not affect the final n-gram. We analyzed several example JAR files and studied the byte code specification [8], [1] to determine which byte codes should be ignored. Once these have been stripped out, the remaining byte codes are used to create the n-gram.

While these n-grams are less sensitive to common changes made to methods, they are also less unique. Using them with FiGD would result in many more false positives. As labels for a dependency graph, however, they complement the graph's inherent structural information. Together, these two pieces of information can help avoid false positives.

C. Comparing Two Predefined Fingerprints

Once fingerprints for two JAR files have been computed according to the method described above, they can be compared to each other to determine how similar the JAR files are. This involves recreating the graphs stored in XML in memory, and using graph comparison algorithms to determine their similarity.

The XML string is read using Java's built in SAX parser with a custom event handler. This handler can transform the graph stored in XML to any other type of graph. In our experiments, we use two different types of graphs: the `SimpleGraphAccessor` from SimPack, and a custom graph type described below. As a node is created for the target graph type, it is labelled with the value stored in the fingerprint's label map (described in Section III-A). This label might be a blank string or an n-gram. Graph edges are added when 'inbound' and 'outbound' tags are encountered in the XML.

Once graphs from two fingerprints have been loaded, they can be compared with any comparison algorithm. In our experiments, we first investigated the maximum common subgraph algorithm found in SimPack. The `MaxCommonSubgraphIsoValiente` object can be used to calculate the largest subgraph isomorphism between the two graphs. Once the maximum common subgraph has been found, we use its size to compute the similarity between the two graphs. We define similarity as the percentage of nodes in the maximum common subgraph that are also contained in the first JAR file's graph.

The maximum common subgraph algorithm considers only graph structure, since it finds the largest common isomorphic subgraph. This can be an advantage when one does not wish to include labels in the dependency graphs, avoiding the possibility that those labels might change when simple modifications are made to the JAR file. Results are promising in some initial experiments on some very simple JAR files. However, this algorithm is NP-hard and becomes too slow for practical use for realistically sized graphs created for real JAR files.

A possible alternative that we did not explore is to find the actual maximum common subgraph rather than an isomorphism. This requires that the graphs be labelled and that these labels be fairly stable even when small modifications to the JAR are made, but may have better running time performance.

Instead, we consider an approximate comparison of the dependency graphs. We load the graph stored in the fingerprint's XML into a very simple graph object. A hash map is created where lists of nodes with the same label are indexed by that label. Each node represents one method. Instead of storing references to other node objects to represent dependency edges, each node simply stores a list of its neighbours' labels. This format works well for local node comparisons but would not be used for algorithms that consider the overall graph structure.

To compare two of these simple graphs to each other, we loop through each node in the first graph, and using the hash map stored in the graphs, we find the list of nodes in the second graph with the same label. We then loop through this list and find the best match to the first graph's node based on how many neighbouring labels are the same. If there is a such a node, we compute a similarity score out of 1.0 by dividing the number of common neighbour labels by of the total number of neighbours. This score is added to a running total. The final sum of all nodes' scores will be at most the number of nodes in the first graph. The final similarity percentage is computed by dividing the sum of scores by the number of nodes in the graph. This algorithm is summarized in Algorithm 1.

This approximation relies on labelled nodes to avoid false positives. The less unique labels are, the more difficult it would be to use the neighbouring nodes as a distinguishing factor. We only experimented with neighbours at a depth of one in our analysis, but it may be worth exploring more deeply into the tree. The computation of the score for one node might involve assigning $1.0/n$ of the score to each of n levels explored.

This simple graph comparison algorithm does not consider as much of a graph's structural information as the NP-hard maximum common subgraph approach, but runs much more efficiently. If labels are relatively unique, and if methods generally depend on some constant number of other methods, then the running time will be on the order of the number of methods in the first graph. The running time will still be polynomial in the worst case, and therefore always outperform the maximum common subgraph algorithm in terms of efficiency.

Experimental results proving the accuracy and efficiency of this approach are presented next in Section IV.

IV. Experimental Results

Our approach, described in detail in the previous section, allows us to create fingerprints using a dependency graph

Algorithm 1 Simple dependency graph comparison.

Input: Graphs $G1$ and $G2$
Output: Similarity score out of 1.0

$total \leftarrow 0$

for each $node \in G1$
  $listOfNodes \leftarrow withLabel(getLabel(node), G2)$

  // based on most matching neighbour labels
  $bestMatch \leftarrow findBestMatch(node, listOfNodes)$

  $num \leftarrow commonNeighbours(node, bestMatch)$

  if $num = length(listOfNodes)$
    // Accounts for case that neither node has neighbours
    $total \leftarrow total + 1$
  else
    $total \leftarrow total + (num/length(listOfNodes))$
  end if
end for

return $total/numNodes(G1)$

labelled with n-grams for internal methods or fully qualified Java names for external methods. In this section, we present experimental results for our approach's accuracy for detecting intellectual property detection, and we present the running time performance.

Our first results, shown in Table I, were obtained by making a common change to a simple JAR file and comparing the new version to the original. The JAR file, about 35 KB in size, is an early but complete version of our code written for the creation of our fingerprint and the conversion of the fingerprint to a graph object. The running times noted include all stages of clone detection, beginning with unzipping the JAR file and ending with graph comparison. The actual graph comparison took 3 ms for all experiments in this set of results.

These results show that matching works as expected when the JAR file is compared against itself. When class files are added, the similarity remains at 100% because the original is completely contained in the copy, but when files are removed from the copy this is no longer true. Note that we do not see a similarity of $10/14 = 71.4\%$ in this case; this is because our similarity is not based on individual class files, but on the methods within. We also see a low percentage of similarity when comparing the JAR file to a completely unrelated one. This similarity might be even lower if we did not modify our byte code to handle the other types of changes, since this action makes the n-gram less unique for a particular method. However, 12.5% is low enough to be confident that the JARs are indeed unrelated.

The remaining results show that the JAR file's similarity is not affected by changes that could easily be made by somebody hoping to hide open source code in their own project. Brown et al showed that their FiGD intellectual property violation detector reports 100% similarity for the

Changes to copy of JAR file	Similarity (%)	Total Running Time (ms)
No changes	100.0	1512
Class files added	100.0	1496
Class files removed (4 out of 14)	59.8	1319
Method moved to subclass	100.0	1442
Static method moved to another class	100.0	1398
Extra argument added to a method	100.0	1370
Name of method changed	100.0	1384
Name of variable changed in method	100.0	1363
Comments added to method	100.0	1438
Unrelated JAR (org.eclipse.jface.text.jar)	12.5	10734

Table I
PERFORMANCE RESULTS FOR COMPARING A JAR FILES TO A VERSION OF ITSELF WITH COMMON CHANGES MADE (GRAPH COMPARISON RUNNING TIME 3 MS IN ALL CASES)

following types of changes:

- Original compared to copy where method names were changed
- Original compared to copy where variables were renamed and comments added or removed
- Original compared to copy where additional class files were added
- Original compared to copy with methods and class files removed

Our clone detector continues to report 100% for each of these cases. Brown et al did not report on moving methods to another class or adding a superfluous argument to the method's signature, and upon testing these scenarios, we found that the method's byte code and thus n-gram changes. With our modified approach, we eliminate byte codes that will change in these cases and are able to report 100% similarity.

Next, we focus on the accuracy and performance of our clone detector for real-world examples. Table II shows a series of comparisons between JAR files with a range of file sizes randomly chosen from the Eclipse plug-in directory. Although the size of a JAR file does not necessarily indicate how large and complex its dependency graph will be, a larger JAR file does tend to result in a larger graph. This is reflected in the increasing length of running times for the larger JAR files. Note that the graph comparison alone remains efficient even when the overall clone detection process can be slow in comparison, indicating that the approximation is fast but that the other stages would benefit from some optimizations.

The results for these realistic JAR files show that larger JAR files still give 100% similarity when matched against themselves. When matched against other unrelated JAR files, low percentages of similarity are generally obtained. However, in some cases the similarity of unrelated JAR files exceeded 20%. While it is reasonable to suggest that any percentages under, say, 30% could be taken to indicate that two JAR files are probably not related, this does indicate that modifying the byte code before computing a method's n-gram does lower

Description of Test	Similarity Result (%)	Similarity Result (%) [no external methods]	Comparison Running Time (ms)	Total Running Time (ms)
commons-attributesapi-2.2.jar (36 KB) compared to itself	100.0	100.0	3	1465
org.eclipse.jface.text.jar (949 KB) compared to itself	100.0	100.0	42	45239
org.eclipse.pde.ui.jar (4051 KB) compared to itself	100.0	100.0	195	415824
commons-attributesapi-2.2.jar compared to org.eclipse.jface.text.jar	20.9	9.0	4	8978
Previous test in reverse	0.8	0.3	14	39636
commons-attributesapi-2.2.jar compared to org.eclipse.pde.ui.jar	25.4	9.8	12	36740
Previous test in reverse	0.7	1.0	43	355647
org.eclipse.jface.text.jar compared to org.eclipse.pde.ui.jar	13.0	13.5	36	65096
Previous test in reverse	6.5	11.6	62	358719

Table II
PERFORMANCE RESULTS FOR SEVERAL REAL-WORLD EXAMPLE COMPARISONS

the uniqueness. Even so, Brown et al reported some results that also exceeded 20% for unrelated JAR files, so the lower uniqueness was likely offset by our use of dependency graphs to help distinguish n-grams.

Table II also includes a column indicating the similarity achieved when methods external to the JAR file were not included in the dependency graph. In some cases, not including these methods lowered the similarity for unrelated JAR files significantly, but in others, the similarity rose. From these results, it is not possible to conclude whether these methods should be included in general or not, and more research is required.

V. CONCLUSION AND FUTURE WORK

The problem of detecting open source intellectual property violation is important for legal and safety reasons as well as the prosperity of open source communities. It can be tackled in several different ways; we took the approach of generating fingerprints for open source Java byte code, which can then be compared to another piece of software's fingerprint. We generate dependency graphs using Dependency Finder, and compare these graphs using an approximate graph matching algorithm. Our fingerprints include n-grams based on modified byte code so they remain stable when common changes are made to an open source project before inclusion in another product. These n-grams are used to label the dependency graphs generated for the fingerprint.

Our results showed that our clone detection algorithm is able to report 100% similarity between JAR files that have been modified in a way that does not change their meaning. In particular, modifications that were not detected by previous approaches, such as moving a method to its superclass or adding a superfluous argument to its signature, no longer affect similarity results. We also report low similarities between two unrelated JAR files. The running time for the approximate graph comparison itself is very fast at only 195 ms for a 5 MB JAR file.

Future research directions include further investigation into an approximate maximum common subgraph implementation [9], a non-isomorphic subgraph matching algorithm using n-gram labels, and the benefit or drawback of including external methods in the dependency graph.

REFERENCES

[1] Java bytecode instruction listings. http://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions [accessed Apr 2010].

[2] Daniel Baggenstos, Beat Fluri, Antoon Goderis, Silvan Hollenstein, Manuel Kägi, Tobias Sager, Markus Stocker, and Michael Würsch. Simpack. http://www.ifi.uzh.ch/ddis/research/semweb/simpack/ [accessed Mar 2010].

[3] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[4] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.

[5] Carson Brown, David Barrera, and Dwight Deugo. FiGD: An open source intellectual property violation detector. In *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering*, pages 536–541, July 2009.

[6] Semantic Designs. Java clonedr. http://www.semdesigns.com/Products/Clone/JavaCloneDR.html [accessed Mar 2010].

[7] Yue Jia, Dave Binkley, Mark Harman, Jens Krinke, and Makoto Matsushita. KClone: A proposed approach to fast precise code clone detection. In *3rd INTERNATIONAL WORKHSOP ON IWSC'2009*, 2009.

[8] Tim Lindholm and Frank Yellin. The java virtual machine specification. http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html [accessed Apr 2010].

[9] Simone Marini, Michela Spagnuolo, and Bianca Falcidieno. From exact to approximate maximum common subgraph. pages 263–272. 2005.

[10] University of Alabama at Birmingham. Clone detection literature. http://students.cis.uab.edu/tairasr/clones/literature/ [accessed Mar 2010].

[11] University of Waterloo Software Architecture Group. JCD: Java clone detector. http://www.swag.uwaterloo.ca/jcd/index.html [accessed Mar 2010].

[12] John W. Raymond and Peter Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, July 2002.

[13] Jean Tessier. Dependency finder. http://depfind.sourceforge.net/ [accessed Mar 2010].

[14] Bach Thanh Le and Rose Dieng-Kuntz. A graph-based algorithm for alignment of owl ontologies. In *WI '07: Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pages 466–469, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Gabriel Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, Berlin, 2002.

[16] Vera Wahler, Dietmar Seipel, Jurgen Wolff v. Gudenberg, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM '04: Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, pages 128–135, Washington, DC, USA, 2004. IEEE Computer Society.