FiGD: An Open Source Intellectual Property Violation Detector

Carson Brown, David Barrera, Dwight Deugo
The School of Computer Science, Carleton University
Ottawa, Ontario, Canada
carson@ccsl.carleton.ca, dbarrera@ccsl.carleton.ca, deugo@scs.carleton.ca

Abstract: *FiGD (Fingerprint Generator/Detector) is an open source Java application capable of detecting intellectual property violations in compiled Java programs without requiring access to the original source files. FiGD uses a modification of the n−gram method which is very accurate in discovering everything from blatantly copied source, to more advanced attempts of obfuscation (such as variable refactoring or white-space insertions). Our improvements to the algorithm allow us to increase the speed of detection and create small fingerprints which can be stored for future comparisons.*

1. Introduction

In recent years, Open Source Software (OSS) has seen a surge in popularity. It is now common to find OSS running on a variety of systems ranging from web servers [14] to super-computers to mobile phones. There are currently numerous OSS projects which have reached a level of maturity sufficient for use by governments and large corporations [15]. As this software makes its way into more areas, legal concerns begin to emerge. It is unclear who is at fault when an open source library in a commercial product fails. Open source licenses [12] can also be incompatible with each other, creating legal problems for companies developing OSS. These are legitimate concerns, but they are difficult to address if the origins of the code are unknown.

1.1 Problem

The problem we focus on in this paper is clone detection for software. We define a software clone as source code in a unknown project that has been copied (either fully or in part) from a known project. Clone detection is useful for pinpointing code theft, as well for general code auditing. Although clone detection has already been extensively researched, this paper focuses only on a small part of the problem which applies to software written in the Java programming language. We assume a black-box (A device or system whose workings are not understood by, nor accessible to, the user and is thus viewed in terms of its input and output characteristics) approach where we generally do not have access to the source code of the projects we are analyzing.

1.2 Motivation

The main motivation of this paper is to contribute to the Open Source philosophy. When open source software is stolen, any changes, improvements or otherwise, made by the intellectual property (IP) thief are unlikely to make it back into the community. Since the open source software development cycle relies heavily on developers contributing, IP theft can prove to be a dangerous threat to this particular ecosystem.

Another motivation is cost: we would like to make it affordable for companies or developers to audit their code for the existence of other OSS. As of October 2008, there are no known open source tools that (easily) allow this. Some available software packages allow source code comparison through simple string-matching, and others are designed to work with only specific programming languages. There are two commercial solutions, [13, 10], costing between $50,000 to $250,000 for annual subscriptions. Both of these companies also allow the end user to pay by the megabyte (Mb), but still at prices ranging from $3,000 to $25,000 for less than 100 Mb. Prices this high could prove to be a significant barrier of entry for small and medium-sized businesses.

1.3 Goals

Our goal is to write an application that will have the following functions:

- Generate a unique signature (fingerprint) from a Java ARchive (JAR) file.
- Search for similarities between a previously generated fingerprint and a new, unknown JAR file.
- Output relevant information regarding the matches found and percentage of certainty.

1.4 Objectives

Given the goals described in Section 1.3, our objectives are to focus not only on accuracy, but also on performance and system resources. The current string-matching approaches found in other projects [3, 4, 16] tend to be very precise but extremely slow, on the order of *O(n2)*. These approaches also assume access to the

original source code which is not always provided. Our objective is also to avoid using large amounts of memory while generating our fingerprints or performing a comparison. These concerns are of particular importance when fingerprinting large files (i.e., >5Mb).

As another improvement, we will also avoid looking at source code. We believe that since the source code is not always packaged within JAR files, it would be better to work without relying upon it, and base our comparison on compiled Java byte-codes (Java byte-codes are what the Java Virtual Machine (JVM) actually executes. It is the compiled version of source code, where each byte-code instruction is exactly one byte in length [7].).

The final objective is to make our fingerprint-detector immune to variable refactoring (to change all references to a variable, for example, to a different name). It is in this manner that our algorithm will still detect a match based on functionality, but not on semantics. Our algorithm is thus resistant to changing variable names, method names, or class names.

In order to generate small fingerprints, we will find and store parts of the JAR file which are highly representative of that file only. In essence, this technique closely resembles what is done in the anti-virus industry (and, in fact, in any signature-based detection environment) where the smallest matching string of a virus is used as a signature. Anti-virus software programs are able to rapidly look for thousands of signatures in a given file. We have created software that achieves similar behaviour at the Java method level.

1.5 Outline

The remainder of the paper is structured as follows: Section 2 describes certain basic concepts and terminology, as well as look at related projects that attempt to solve a similar problem. Section 3 explains our design strategy, including decisions that had to be made in order to reach our objectives. In Section 4 we present our results. Section 5 provides a conclusion and elaborates on future work.

2. BACKGROUND

Clone detection can usually be done either by string-matching the source code or looking at binary file signatures. The string-matching technique requires looking at small sub-strings in the file (called *n-grams*, where n is the number of characters in the string, or gram) and then try to identify those strings in a different file. This obviously requires a large amount of memory and processing, especially if a sliding window of the entire file is taken. For example, if the original file has 1000 characters in total (including white spaces and line termination), using *n*-grams of size 10 without a sliding window would give us 100 10-grams. If a sliding window were used, we would have 991 10-grams (1000−10+1). Continuing this example, we would need to search for occurrences of 991 strings in a new file.

Binary file signature matching provides the added benefit that the original source code is not required. This is useful, for example, in the anti-virus industry, where viruses and worms are packaged and distributed globally. There is a slight difference that prevents us from using this approach directly: source code may be slightly modified and rebuilt, producing a completely different binary file. For example we take the (extremely simple) method in Listing 1:

```
int method() {
    int i=10;
    return i;
}
```

Listing 1: Simple Method

This method would have a certain binary signature once compiled. However if we were to change its source to the following, Listing 2:

```
int method() {
    int i=10;
    i--;
    i++;
    return i;
}
```

Listing 2: Modified Simple Method

The binary signature may be completely different, even though the method has no changes in functionality (i.e., it still returns the value of i=10). This problem is of serious concern when considering OSS fingerprinting, as the source code is almost always easily available and can be changed and compiled by any software recipient.

2.1 JAR files and Class files

Our software will take as input any valid JAR file [5]. A JAR file is a file-type based on the popular "ZIP" file format. It was developed by Sun Microsystems, and it allows many files to be aggregated into one, with optional compression. JAR files contain the Java resources necessary to run Java programs. For this paper, we are interested in one set of resources called "Class" files [8].

Class files are Java's compiled files. A source file (usually ending in .java) will be compiled to produce one or more class files which are (for the most part) platform independent (excluding platform-specific system functions). Class files contain byte-code groups of Java's instruction sets that will be run (or interpreted) inside a JVM.

2.2 Related Software

Although there are countless papers on clone detection [2], software products that can detect clones of compiled Java programs are difficult to find. Many papers describe early prototypes of their algorithms and therefore have not yet released their software. Other papers describe

the best ways of comparing strings, but generally require access to source files. Software such as Simian [6], Clone Digger [3], CCFinderX [1] and Clone Doctor [4] are readily available, but also work only source files. The advantage of these tools, however, is that they should work on any type of source code (C, Python, Java, assembly, even plain text) since they are performing basic string matching techniques.

3. APPROACH

In this section describes at a high level our approach. We provide the main algorithms for FiGD as well as what decisions had to be made in order to achieve our goals.

3.1 Design

Rather than create a general purpose fingerprinting program, this project has the particular distinction of comparing JAR files. These have a known composition, both in compression and file structure. For the purposes of this project, we are looking for code reuse from one JAR to another. Thus, we have distilled our approach from the general case considerably: our fingerprint generator and detector considers only Java class files, and more specific still, considers only the byte codes of each method contained in these class files.

We have made this decision based on what we feel is representative of the uniqueness of a JAR file. When considering JAR files, we cannot guarantee the inclusion of source code (Java or otherwise), nor can we guarantee that any part of the comparison JAR file retains similar naming or folder structure of Java packages. What we can consider, however, is that the essence of a Java method will be retained, regardless of moving the method to another class or changing its name. That is to say, the method will still do the same thing.

This section has been broken down into two sections, comparing the two components of the project: the fingerprint generator, and the detector.

3.1.1 Fingerprint Generator

The fingerprint generator must first open the JAR file to be compared. All JAR files are created with the ZIP standard, and can be decompressed rather easily. In the Java API, the java.util.jar package contains many useful objects, including the JarFile and JarEntry classes. It is then possible to compute a listing of all files contained in the JarFile object, and a simple file type check allows for a complete listing of all class files.

The decompressing of the JAR file contents into class files is done through the JarResources class, adapted from a Java-World article [9]. We have modified the class to only decompress the JAR's class files into memory. Our fingerprint generator can then iterate over all class files, by requesting each class file individually from the JarResources object. This is done by writing the class out to a temporary file, which is later deleted upon the program's exit. FiGD incurs in a slight memory overhead due to the extensive utilization of objects as opposed to programming in a structural language such as C. This, however, proves to be a negligible performance limitation, since the JAR files we are testing usually fall within the 0Mb-50Mb file size range.

With the Java class file written out to a temporary file, we made use of another open source library to access the necessary methods. The org.netbeans.modules.classfile package [11] allows for direct access to the class file byte-codes, by loading the file as a ClassFile object, part of the NetBeans package. It is then possible to iterate through all methods of the ClassFile object, which are available as instances of the Method class. Each method can then be extracted as a list mof Java byte-codes using other classes found in the NetBeans package.

Rather than use the byte-codes for a whole method (which would increase the size of our fingerprint considerably), we decided on only storing a single *n*-gram per method. We first compute all *n*-grams of each method, then the most unique *n*-gram is selected to represent that method in the JAR file's fingerprint. The uniqueness of *n*-grams differ based on the size of n, but our testing has shown that using n a gram size of 10 (i.e., 10 byte-codes) strikes a good balance between accuracy and fingerprint size. Also, using larger values for n did not improve accuracy. By using this approach, the fingerprint size is linearly dependent on the number of methods found in the JAR file. It is this list of unique *n*-grams, as well as some statistical information—such as the number of methods, *n*-grams stored and total *n*-grams—that form the fingerprint of a JAR file.

3.1.2 Detector

Detection requires an original fingerprint as well as a comparison JAR file. The result returned from our detector contains both our certainty percentage that code from the fingerprinted JAR file is contained in the comparison, and also our calculation of how much of that original code appears. This is calculated by opening the JAR in much the same way as the fingerprint generator, save that our generator does not throw away non-unique *n*-grams but instead compares these to the representative *n*-grams of the fingerprint. This is done by first generating a list of *n*-grams for a given method in a class file. These are then compared to the *n*-grams in the fingerprint which have not already been matched by *n*-grams in the comparison JAR file. The list of *n*-grams generated by the detector are not stored for later use: the only *n*-grams stored in working memory are those that are being compared to the fingerprint. When a match has been found between the fingerprint and the comparison JAR (i.e.: both JAR files contain the same method) the next method in the comparison JAR is considered for detection.

The number of matches is stored, and used in the calculations of the detector's final result. The number of matches divided by the total number of *n*-grams in the

original fingerprint yields the percentage of the original JAR file in the comparison JAR file. The certainty of the final result is calculated by the percentage of the *n*-grams included in the original fingerprint divided by the total number of *n*-grams created from the JAR file.

3.1.3 Summary

The algorithms in Listing 3 and 4 detail the fingerprint generation and detection approaches from Sections 3.1.1 and 3.1.2.

```
Algorithm Generator
Input: Jar File
Output: Fingerprint
1.  G ← { θ }
2.  C ← {c | ∀ Class File c ϵ Jar File }
3.  for c ϵ C
4.      do M ← { m | ∀ Method m ϵ c }
5.          for m ϵ M
6.              do compute n-grams
                       from byte-codes of m
7.                     s ← n-gram of lowest count
8.                     add s to G
9.  add G to Fingerprint
10. return Fingerprint
```

Listing 3: Generator Algorithm

```
Algorithm Detector
Input: Fingerprint
Input: Jar File
Output: FingerprintResult
1. count  0
2. C ← { c | ∀Class File c ϵ Jar File }
3. for c ϵ C
4.  do M ← { m | " ∀Method m ϵ c }
5.      for m ϵ M
6.          do consider each n-gram g_c of m:
7.              if g_c ϵ Fingerprint
8.              then count ← count + 1
9.              remove g_c from considered
                        Fingerprint entries
10.             continue to next Method m
11. certainty ← count / Fingerprint_size * 100%
12. add certainty to FingerprintResult
13. return FingerprintResult
```

Listing 4: Detector Algorithm

3.2 Decisions Made

Over the course of creating the fingerprint generator and detector, we made a variety of design decisions. Our first implementation for creating fingerprints at the Java method level involved computing simple hashes of every method, which significantly reduced our accuracy when situations such those described in Section 2. This inability to catch "useless" modifications to the code in a method led us to desire a way of capturing the uniqueness of a method. We then implemented the *n*-gram implementation, and extracted only the first *n*-gram of lowest frequency. This involved some loss in accuracy, but it is our belief— proven through testing— that this loss is negligible when compared to the large decrease of the generated fingerprint's footprint.

Our experiments also show that using *n*-grams where n is 10 have shown to be the most representative. When n is set to lower values, accuracy of the algorithm suffers, as the *n*-grams represent very little of a method's structure. This loss of accuracy can be attributed to false positives when comparing the fingerprint to another JAR file. Similar to the method hashing described above, having large values of n leads to loss of accuracy, where truly equivalent methods are no longer detected as such. As the size of n increases, the algorithm approaches behaviour similar to the method hashing described above.

Our implementation of generating fingerprints and detecting similarities between JAR files compares based on the contents of class file methods. This means that "empty" or unimplemented methods are not considered. We are aware that our implementation cannot properly deal with interface classes, or the non-implemented abstract methods found in abstract classes. We do not believe this to be a fault in our design, as interfaces are by definition public, and abstract classes are still considered; only the abstract methods are ignored.

4. RESULTS

This section documents the results of testing FiGD on various, representative JAR files. The subsections below describe testing in both accuracy and performance during the implementation's construction and as a completed product in "real world" use cases.

4.1 Accuracy

For testing the accuracy of FiGD, we used two random JAR files found in the Eclipse JAVA IDE installation. We created fingerprints for each one, and then compared them to themselves using the detector. Both fingerprints were generated in under 3 seconds, and the output claiming a 100% match was displayed immediately after. A 99.999% certainty was also displayed in both cases, confirming that with high confidence, the files are fully identical.

One of the features of FiGD is that as soon as the first *n*-gram is matched for a given method, no further *n*-grams are compared for that method, since we assume we have found a cloned code segment. This greatly speeds up the detection phase when we know a priori that there is some kind of similarity between two files. If the files are completely different (i.e., zero matching methods), then our detector has to compare every single *n*-gram to the

fingerprint, which takes timeO(n · m), where n and m represent the number of methods in each JAR file.

Although false positives have not been extensively tested, we believe the chance of them occurring is small, since our *n*-grams are large enough to make each method signature reasonably unique.

4.2 Performance

For performance testing, we used a large JAR file (about 10Mb) and a small JAR file (about 300Kb). We saved copies of both JAR files with slight modifications. The modifications were simply to remove a random number of class files from each one. We then compared the original unmodified file to the modified variations. We obtained results in less than 5 seconds with FiGD reporting between 70% and 80% matches between the JAR files. This seems correct, as only ma small number of Class files were removed. The certainty percentage reported was still high at over 80% for both test cases, confirming that our algorithm is not only fast, but correct as well.

4.3 Real World Testing Results

Our various tests over the course of developing FiGD, made use of a variety of JAR files, including many from the Eclipse 3.4 Classic IDE plugin directory, chosen due to Eclipse's popularity. Two JAR files have been included in Table 2 from this software: org.eclipse.jface.text 3.4.0.v200806032000.jar and org.eclipse.jdt.ui 3.4.0.v20080603-2000.jar (abbreviated in the table due to file name length). The third JAR file used, commons-attributes-api-2.2.jar, is from the Apache Commons library. These three JAR files include mcompiled Java class files and are three representative sizes, mthe largest being included for "stress" testing. The tests have been performed on a Toshiba Satellite PSM40-SF300E laptop, with an Intel Pentium M processor (1.86GHz, 533MHz FSB, 32KB of L1 cache, 2MB of L2 cache), 1 GB of memory (2 x 512 PC2700 DDR SODIMM) running Ubuntu 8.10 GNU/Linux.

The JAR files used in Table 1 have been created especially for testing FiGD, and include small, easy to manage Java class files used in first-year programming assignments. These class files have been modified and compiled into various JAR files, as described in the table. "Original" is in reference to an original set of Java class files serving a particular purpose. For each of the test cases where files were modified, a significant number of changes were made—for example, more than 60% of all variable names were changed for the second test in Table 1. These tests show that FiGD is insensitive to aesthetic source-code changes such as variable name refactoring or source code comments.

Table 2 demonstrates more "real world" testing, involving real world JAR files. These files were deliberately chosen because they are not obviously related by purpose or content. The certainty percentages calculated are entirely dependent on how well FiGD can form a representative fingerprint on a given JAR file, while the inclusion percentage (Inc %) relies on the number of matched methods. These tests confirm our suspicions: that the JAR files are convincingly different. The only non-obvious data set is the last pair of tests comparing the two Eclipse-based JAR files. We believe these inclusion percentages to be correct, as both of these JAR files share a common Eclipse plug-in architecture, and likely do share similar code bases in this respect. These tests also confirm the worst-case running time calculated above, as these files have very few similarities, causing near quadratic run-times, executing over a few minutes on the test machine.

Table 1: Accuracy Testing

Description	Certainty	Inc (%)
Original compared to copy where method names were changed	100%	100
Original compared to copy where variables were renamed and comments added or removed	100%	100
Original compared to copy where additional class files were added	100%	100
Previous test in reverse	100%	61.6
Original compared to copy with methods and class files removed	100%	83.8
Above text in reverse	100%	100

Table 2: Performance Testing

Description	Certainty	Inc (%)	Time (ms)
commons-attributes-api-2.2.jar (35.9Kb) compared to itself	93.6%	100	220
org.eclipse.jface.text.jar (922.7Kb) compared to itself	88.5%	99.9	2137
org.eclipse.jdt.ui.jar (9.2Mb) compared to itself	97.6%	99.9	19304
commons-attributes-api-2.2.jar compared to org.eclipse.jface.text.jar	93.6%	0	3333
Previous test in reverse	88.5%	0	3643
commons-attributes-api-2.2.jar compared to org.eclipse.jdt.ui.jar	93.6%	0.6	28203
Previous test in reverse	97.6%	0	37660
org.eclipse.jface.text.jar compared to org.eclipse.jdt.ui.jar	88.5%	22.7	719561
Previous test in reverse	97.6%	3.0	820536

5. CONCLUSION

In this paper we have presented FiGD, an algorithm and implementation for detecting clones in compiled Java projects. Even when access to the source code is not available, FiGD is able to produce very accurate results in shorts periods of time by using a combination of previous approaches as well as custom optimizations. Source code for FiGD is released under the BSD license and is available by request. Included with the source code is full Javadoc documentation describing all methods and classes.

5.1 Review Goals and Contributions

Our main goals discussed in Section 1.3 are achieved with the design and implementation of our algorithm. We believe we are the first to approach the clone detection problem for software through a black box approach, giving the OSS community another tool for detecting IP violations.

5.2 Future Work

While we have shown that we are able to compare fingerprints quickly, there are still some possible optimizations that could be made in terms of generating each fingerprint. Making use of an advanced data structure (such as heaps) would provide us with faster searching than the current array-based implementation. This could theoretically reduce our worse-case running time for computing fingerprints to $O(n \cdot logn)$. Together with stored pre-computation of all the known JAR files previously fingerprinted and stored offline, we believe that FiGD would operate significantly faster. We would also like to do more work on finding optimal *n*-gram sizes and how they impact the accuracy of the detector. Finally, we would like to expand the fingerprint to also include source code and plain text files as opposed to only considering Class files, and include these findings into a more advanced detection schema.

6. REFERENCES

[1] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Trans. Software Eng. 28(7): 654-670, 2002.

[2] Clone Detection Literature - University of Alabama at Birmingham. http://students.cis.uab.edu/tairasr/clones/literature/). Accessed November 3, 2008.

[3] Clone Digger. http://sourceforge.net/projects/clonedigger/. Accessed November 3, 2008.

[4] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, Lorraine Bier: Clone Detection Using Abstract Syntax Trees. ICSM 1998: 368-377. 1998.

[5] JAR File Specification. http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html. Accessed November 3, 2008.

[6] Simian - Similarity Analyzer. http://www.redhillconsulting.com.au/products/simian/index.html. Accessed November 3, 2008.

[7] Wikipedia - Bytecode. http://en.wikipedia.org/wiki/Bytecode. Accessed November 2, 2008.

[8] Wikipedia - Class File. http://en.wikipedia.org/wiki/Class_(file_format). Accessed November 3, 2008.

[9] Arthur Choi. Java tip 49: How to extract java resources from jar and zip archives. http://www.javaworld.com/javaworld/javatips/jw-javatip49.html. Accessed October 30, 2008.

[10] Black Duck Software. http://www.blackducksoftware.com/protex. Accessed October 30, 2008.

[11] NetBeans.org. Classfile reader java documentation. http://bits.netbeans.org/dev/javadoc/org-netbeans-modules-classfile/. Accessed October 30, 2008.

[12] Open Source Licenses - Free Software Foundation. http://www.fsf.org/licensing/licenses/. Accessed October 31, 2008.

[13] Palamida Software. http://www.palamida.com/products. Accessed October 25, 2008.

[14] HTTP Server Project. http://httpd.apache.org/. Accessed October 25, 2008.

[15] Apache Tomcat. http://tomcat.apache.org/. Accessed October 25, 2008.

[16] P. Bulychev, M. Minea, Duplicate code detection using anti-unification, in: Spring Young Researchers Colloquium on Software Engineering, SYRCoSE 2008, 2008, p. 4