# Weighted Method Signatures Fingerprints

Stéphane Leblanc
School of IT and Engineering, University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, Canada, K1N 6N5
(001) 613-562-5800 #6699
slebl089@uottawa.ca

## ABSTRACT
A software fingerprint is a set of distinctive characteristics used to identify and compare programs. This paper presents a fingerprint technique based on the characteristics of method signatures. The proposed technique considers uncommon method signatures as being more effective than common ones to compare programs. The implementation of this technique is limited to Java programs, but it could be implemented for other programming languages as well. The experiments carried out to evaluate the technique demonstrated that it was credible, resilient and scalable.

## Categories and Subject Descriptors
K.5.1 [**Legal Aspects of Computing**]: Hardware/Software Protection - *Proprietary rights*

## General Terms
Legal Aspects

## Keywords
Keywords are your own designated keywords.

## 1. INTRODUCTION
Copyright infringement of software is a widespread illegal practice in our era. Detecting software copyright infringement is challenging because it is difficult to establish if there is a copy relationship between two programs. It follows that the copyright laws are rarely enforced even if the software copyright infringement has a huge economic impacts on the world of software development. This paper presents a technique that can be used to determine if a program is likely to be a copy of another. Consequently, the technique is a useful tool to detect copyright infringement.

## 1.1 Problem
The root cause of the problem is the impossibility to define what makes two pieces of software equal. Comparing something with a natural and durable identifier is simple. For instance, a bank account has an account number. An account number is a natural and durable identifier of an account because it will not change over time and will always refer to the same account. When two things do not have natural durable identifiers, they are usually compared according to their characteristics. For instance, there are no such things as date IDs (i.e. there are no natural durable identifiers for dates as is the case with bank accounts), but two dates with the same year, month and day are considered to be the same.

Unfortunately, programs do not possess any durable and unique identifiers and the intrinsic nature of all their characteristics is to change over time. Two versions of the same program do not have the same characteristics, but they are still considered to be the same program. On top of that, applying semantics-preserving code transformations, also known as obfuscation, can substitute some characteristics of a program with equivalent ones. Developers who do not want to share their source code often use these transformations to make the source code of their program almost impossible to read. Such transformations are an essential tool to protect the intellectual property of developers. On the other hand, these transformations can also act as a camouflage for developers who infringe copyrights. If a program is difficult to read, then it is also difficult to compare it with other programs. In this context, detecting software copyright infringement is not a simple task.

## 1.2 Goals
The proposed technique must be able to measure how likely a program is to be a copy of another and must exhibit the five following properties:

**Credibility**

"Let p and q be independently written sets of modules which may accomplish the same task. We say f is a credible measure if f(p) != f(q)" [1].

**Resilience to Natural Change**

Let p0 be a set of modules obtained by creating a derivative work from p. We say that f is resilient to natural change if f(p) = f(p0).

**Resilience to semantics-preserving transformations**

"Let p0 be a set of modules obtained from p by applying semantics-preserving transformations T. We say that f is resilient to T if f(p) = f(p0)" [2].

**Running Time Scalability**

Let n be the size of the set of modules p, we say that the running time of f is scalable if the average case running time complexity of f(p) is at least O(n) (linear).

**Memory Usage Scalability**

Let n be the size of the set of modules p, we say that the memory usage of f is scalable if the average case memory usage complexity of f(p) is at least O(n) (linear).

## 1.3 Objectives

Credibility, resilience and scalability cannot be completely reached simultaneously because they are conflicting goals. There are no doubts that considering only completely identical programs as being in a copy relationship would be credible. Such a technique is not likely to produce false positives. On the other hand, this technique would not be resilient. The slightest transformation would prevent this technique from working properly. Also, a simplistic technique that only compares the size of the files of two programs would be able to compare two large programs as effectively as two small ones but would be neither credible nor resilient. The first objective of the proposed technique is to provide good trade-offs between these conflicting goals.

Its second objective is to be cohesive. Techniques based on a specific measure (as opposed to those that summarize many kinds of measures) are easier to use in conjunction with other techniques. Since it is almost impossible for a technique to meet credibility, resilience and scalability simultaneously, being able to use the proposed technique easily with other existing techniques is definitely an asset.

## 1.4 Outline

The proposed technique is based on weighted method signatures (WMS) fingerprints. A fingerprint is a subset of all characteristics of a program. To be effective, fingerprints must contain distinctive characteristics that are not likely to change. Method signatures possess such characteristics; thus they constitute a solid base for the proposed technique. Some method signatures are more likely to be present in a class than others. For instance, a class often contains a parameterless method that returns `void`, but seldom contains a method that returns an array of `DateFormat`. The WMS technique gives more importance to rare method signatures than to common ones. Also, the WMS technique extracts data related to method signatures from Java binary files (.class). Therefore, the technique can only be used to compare Java programs. No data is extracted from the source code, the documentation or any other resource files because programs are often distributed in binary form only.

# 2. BACKGROUND

Many open source toolkits such as Stigmata [3], JBirth [4] and SandMark [5] use software fingerprints to compare Java programs. With its 16 birthmark techniques, Stigmata is the most comprehensive open source birthmark toolkit available for Java.

No technique found in these toolkits can be considered equivalent to the WMS technique. The Used Classes (UC) technique (and its variants) is the closest to the WMS technique. The UC technique is based on the concept of well-known classes, which is similar to the concept of identifiable types used in the WMS technique. The UC technique relies on the list of distinct well-known classes used by a class to identify and compare classes. Compared to the WMS technique, the UC technique can be considered a coarse-grained approach. The UC technique uses class-level attributes to compare programs while the WMS technique uses method-level attributes. Moreover, the UC technique ignores types that are not considered well-known such as primitive types and user-defined types. Furthermore, it does not take into account the fact that some types are used more frequently than others.

# 3. DESIGN

## 3.1 Identifiable Types

Types are identified by their full name (i.e. their namespace and their name). For example, `java.lang.String` uniquely identifies the type String from any other Java types. Semantics-preserving code transformations can easily alter the full name of types if they are not used from outside of a program. In contrast, they cannot easily alter types contained in the Java system libraries (i.e. the primitive types and the types with a namespace that starts with `.java` or `.javax`) because they can be used by any Java programs.

The WMS technique considers the types contained in the Java system libraries as identifiable. These types can be identified by their full name. On the other hand, other types are considered unidentifiable because they are likely to be altered by semantics-preserving transformations. The WMS technique can distinguish identifiable types from unidentifiable types but cannot distinguish one unidentifiable type from another. Arrays are also identifiable; therefore, `int`, `int[]` and `int[][]` are three distinct types.

## 3.2 Selectivity

The WMS technique uses selectivity points to measure the weight of a type. Some types are used more frequently than others in Java. The more often a type is likely to be used in a Java program, the less its selectivity. Some applications make more intensive use of some types than others. For instance, graphical user interface applications and command line applications do not use the same types as frequently. Prior to the implementation of the technique, a statistical analysis of the type used in 192 standard eclipse plugins was carried out to overcome this problem. The analyzed plugins were assumed to be a representative sample of Java programs in general.

Selectivity points are based on this statistical analysis. The collected data demonstrated that the most infrequently used type was proportionally used 139 thousand times less frequently than the most frequently used one. If selectivity points were proportionally assigned based on the likelihood of a type to be used, the results would be very credible. However, the results would not be resilient because changing only one very infrequently used type would have too much impact on the fingerprint. On the other hand, ignoring that some types are used less frequently than others would harm the credibility of the technique. Selectivity categories solve this problem by providing a trade-off between credibility and resilience. Arbitrary decisions based on the statistical data and motivated by the need to conciliate credibility and resilience were made to define the number of selectivity categories and the number of selectivity points assigned to each one. Selectivity categories are presented in Table 1.

**Table 1. Selectivity Categories**

| Category Name | Occurrence Per Type | Num. of Types | Total Occurrence | Selectivity Points |
|---|---|---|---|---|
| Very Frequent | >= 50,000 | 2 | 230,126 | 1 |
| Frequent | < 50,000 | 6 | 159,391 | 2 |
| Normal | < 5,000 | 21 | 30,816 | 4 |
| Infrequent | < 500 | 47 | 7,173 | 8 |
| Very Infrequent | < 50 | 314 | 2,595 | 16 |
| Total | | 390 | 430,101 | |

These selectivity points are the foundation of the WMS technique. It is not possible to compare fingerprints that use different selectivity categories or different selectivity points. For the sake of simplicity, the implementation of the WMS technique does not allow to change the selectivity categories.

## 3.3 Fingerprints

Method names, parameters names and method modifiers are not stored in the fingerprints because they are too vulnerable to semantics-preserving transformations. Only the return type, the parameters types and whether or not the method is static are taken into account. This information is then hashed for smaller memory consumption and faster comparison. Figure 1 illustrates the parsing of a method.

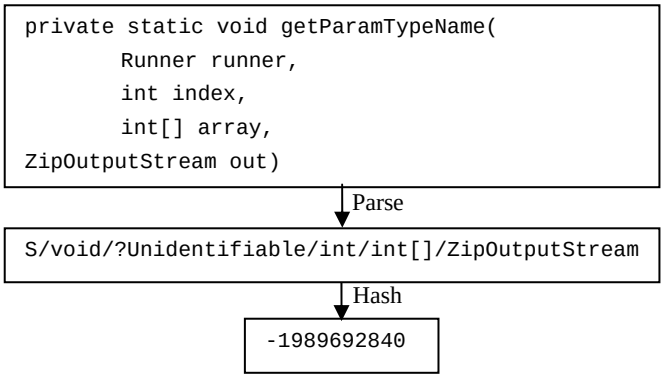


**Figure 1. Method Parsing**

The selectivity of each method is also stored in the fingerprints. The selectivity of a method is the sum of the selectivity of its parameters. The previous method selectivity is computed as follow:

```
S/void/?Unidentifiable/int/int[]/ZipOutputStream
=VeryFreq/VeryFreq/Freq/Normal/VeryInfrequ
=1+1+2+4+16
=24
```

Fingerprints also contain the selectivity of each class (which is the sum of the selectivity of its methods) and the selectivity of the Java archive (JAR) (which is the sum of the selectivity of its classes). Moreover, contextual information such as the JAR name and the name of classes are also included in the fingerprint. Contextual information is not used to compare fingerprints; therefore, it was excluded. However, the value added by the contextual information has proven to be worth the extra bytes. Figure 2 illustrates the content of a fingerprint.

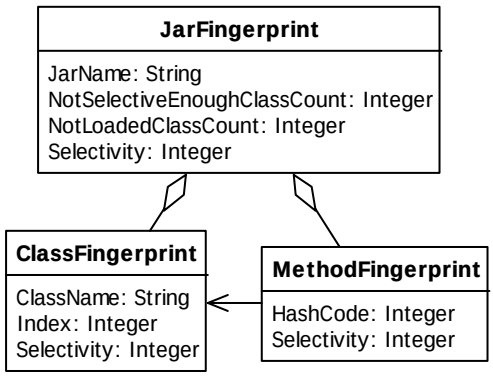


**Figure 2. Fingerprint Class Diagram**

## 3.4 Comparison

When comparing two JARs, the WMS technique produces a certainty percentage. This percentage represents how likely it is for the original JAR to be completely included in the compared JAR. If very strong evidence is found that 10% of the original JAR is in a copy relationship with the compared JAR, the certainty percentage will still be low because only a small portion of the original JAR was copied.

The certainty percentage is computed as follows. First, the method signature hash codes from two JARs are compared. The WMS technique does not consider similar methods; methods are either equal or different. Because the number of methods can be very large, the amount of time required for the comparison and the running time scalability of the WMS technique depend mostly on its ability to compare methods efficiently. The algorithm used to compare methods is inspired by the Merge-Join algorithm, which is widely used in relational database management system to quickly join large tables. The algorithm requires both method sets to be previously sorted by key (i.e. method signature hash code and class index). The method set is sorted when the fingerprints are created to avoid resorting for each comparison. The algorithm running time complexity is O(n) (linear) if all method keys are unique and $O(n^2)$ (quadratic) if all method keys are identical. The statistical analysis performed on Eclipse plugins demonstrated that the method signatures are very diverse. Therefore, comparing methods will tend to be linear.

The method comparison yields a matrix containing all potential matches between the original and the compared classes. Selectivity points are assigned to each potential class match. The selectivity of a class match is the sum of the selectivity of its matching methods.

To produce the certainty percentage, the next step is to identify which class matches are the best. In order to do so, class matches are sorted from the highest to the lowest selectivity. Then, the JAR match selectivity is obtained by summing up the selectivity of the best class matches. When the JAR match selectivity is computed, each original class can only be matched with one compared class and vice versa. The certainty percentage is

obtained by dividing the selectivity of the JAR match by the selectivity of the original JAR.

## 3.5 Contextual Information

In addition to the certainty percentage, the WMS technique also provides contextual information on the copy relationship between two compared JARs. The selectivity point ratio that yielded the certainty percentage and the reverse certainty percentage are part of this contextual information. The reverse certainty percentage is an estimate of how likely is the compared JAR to be completely included in the original JAR. This is useful when the original JAR is compared with less selective JARs.

Moreover, the name and high-level information on the JARs involved in the comparison are provided. Also the contextual information contains the 10 best class matches. The classes in this list are the ones to start with if further analysis is required to prove that copyright infringement has occurred. Also, this information is useful to detect the partial inclusion of a JAR into another one.

```
Certainty Percentage
    Original to Compared:                  98.5% (3802/3861)
    Compared to Original (Estimate):       96.8% (3802/3929)
Original JAR (Ori.jar)
    Selectivity:                           3861
    Number of Classes:                      237
    Not Selective Enough Classes:           167
Compared JAR (Zkm.jar)
    Selectivity:                           3929
    Number of Classes:                      237
    Not Selective Enough Classes:           167
Top 10 Class Matches
    (org.junit.Assert,gc): 100.0% (526/526)
    (junit.framework.Assert,n): 100.0% (293/293)
    (org.junit.runners.BlockJUnit4ClassRunner,ug): 100.0% (133/133)
    (junit.framework.TestSuite,cb): 100.0% (117/117)
    (junit.runner.BaseTestRunner,eb): 100.0% (117/117)
    (org.hamcrest.BaseDescription,jb): 100.0% (111/111)
    (org.junit.runner.Description,df): 94.8% (109/115)
    (org.junit.internal.runners.JUnit4ClassRunner,sg): 100.0% (104/104)
    (org.hamcrest.CoreMatchers,wb): 100.0% (95/95)
    (org.junit.runners.model.TestClass,hi): 93.1% (94/101)
```

**Figure 3. WMS Technique Output**

## 3.6 Decisions made

### *3.6.1 Low Selectivity*

The statistical analysis demonstrated that 66% of classes had less than 16 selectivity points and that their impact was only 14% of the total of selectivity points. Classes with low selectivity are numerous, have a small impact on selectivity and are likely to produce false positives when compared with other classes. Therefore, excluding them produces a positive impact on each of the goals. The price to pay is that the WMS technique cannot compare JARs that are not selective enough; the WMS technique would not have yielded accurate results for these cases anyway. Out of the 192 standard eclipse plugins used for the statistical analysis, only one was considered to be not selective enough.

A similar situation occurs when comparing two JARs yields matches with less than 16 selectivity points. These matches are considered to be an undesirable noise and are ignored by the comparison algorithm.

### *3.6.2 Generic Types*

Generic types are problematic from the perspective of the WMS technique because they are highly selective and also vulnerable to semantics-preserving transformations. Generics are all about type safety at compile time. Therefore, replacing them with unsafe types and using the appropriate type cast at runtime would preserve the semantics of a program. To overcome this problem the generic type arguments are ignored (e.g. `List`, `List<int>` and `List<List<CustomType>>` are all equal).

### *3.6.3 Tolerance to Missing Dependencies*

Java reflection is used to extract the method signature information from the Java binary files (.class). In order to load a class, Java reflection must be able to resolve all types used by this class. By convention, the WMS technique can resolve any types contained in JARs located in the same directory as the JAR from which the fingerprint is generated. If unresolvable types prevent a class from being loaded, it will be excluded from the fingerprint. A warning will notify users that some classes were not loaded.

# 4. RESULTS

A series of experiments were carried out to measure if each goal was met. The experiments were executed on a machine with an Intel Core i7 920 CPU (2.65GHz), 6 GB of RAM and running Windows Vista 64 bits.

**Table 2. Credibility Experiment JARs**

| JAR | | | | Classes | | | Methods |
|---|---|---|---|---|---|---|---|
| Category | Name | Size (KB) | Selectivity | Compared | Excluded | Not Loaded | Compared |
| Bit Torrent Client | Vuze [6] | 13,545 | 136,843 | 2,174 | 5,550 | 13 | 32,307 |
| Build Script | Ant [7] | 1,288 | 20,545 | 360 | 409 | 0 | 4,924 |
| Code Coverage | Cobertura [8] | 443 | 8,414 | 57 | 64 | 1 | 3,065 |
| Code Coverage | CodeCover [9] | 4,879 | 3,363 | 73 | 345 | 17 | 1,065 |
| Code Coverage | EclEmma [10] | 485 | 807 | 27 | 41 | 2 | 242 |
| DB Test Tool | DbUnit [11] | 587 | 8,248 | 175 | 196 | 14 | 1,306 |
| Mp3 Player | TMp3 [12] | 171 | 2,524 | 49 | 40 | 0 | 397 |
| Obfuscation | Proguard [13] | 658 | 13,502 | 231 | 310 | 1 | 3,590 |
| Obfuscation | Sandmark [14] | 5,127 | 44,608 | 817 | 1,103 | 3 | 10,151 |
| Object Mocking | EasyMock [15] | 109 | 3,463 | 32 | 46 | 5 | 481 |
| Object Mocking | JMock [16] | 235 | 1,486 | 26 | 48 | 0 | 246 |
| Programming Language | Jython [17] | 8,098 | 143,528 | 1,687 | 4,144 | 10 | 35,466 |
| Test Sequencer | Junit [18] | 238 | 3,833 | 70 | 167 | 0 | 717 |
| Text Editor | JEdit [19] | 3,902 | 44,388 | 677 | 455 | 0 | 5,929 |
| Text Editor | JExt [20] | 1,524 | 15,337 | 280 | 161 | 0 | 1,886 |

## 4.1 Credibility

Fifteen different JARs were used to measure the credibility of the WMS technique. Some of them accomplish similar tasks while some others were written for completely different purposes. A detailed list of the JARs used for the experiment can be found in Table 2.

In order to be credible, the WMS technique must be able to identify similar JARs without generating false positives. To measure this ability, all possible JAR combinations were compared using the WMS technique, leading to 225 comparisons (15*15).

The 15 comparisons where the JAR was compared with itself all yielded a certainty percentage of 100%. All of the 105 comparisons where the original JAR was compared with a smaller one yielded very low certainty percentage. These comparisons were not meaningful to measure credibility because the WMS technique will always yield lower certainty percentages in such cases. Therefore, these comparisons were not taken into account.

The 105 remaining comparisons did not generate any false positives. All comparisons yielded a certainty percentage under 34%, except one. The comparison between JExt and JEdit (two text editors) yielded a certainty percentage of 44.3%. However, the contextual information reported that the original and the compared class names were identical for the five best class matches. The fact that similar classes with the same name were identified provided strong evidence that the two sets of modules had not been independently written. Therefore, the result of the comparison between JExt and JEdit could not be considered a false positive.

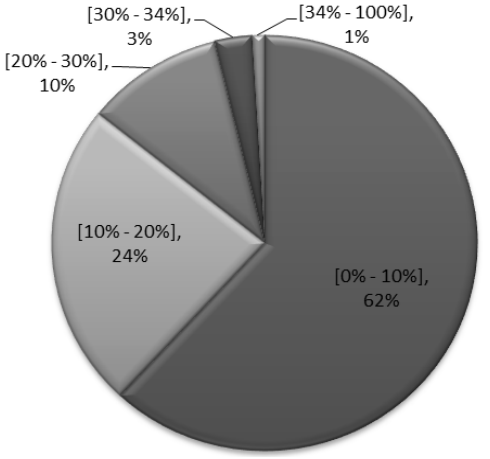


**Figure 4. Certainty Percentage Distribution**

## 4.2 Resilience to Natural Change

In order to be resilient to natural change, the WMS technique must be able to detect the copy relationship between an original work and a work derived from it. To measure this ability, seven significant releases of JUnit were compared using the WMS technique.

For each release (except one), the WMS technique was able to recognize strong similitude between the JARs. For the release of JUnit 4.0, the fact that the major digit was changed from 3 to 4 indicates that more than a minor revision was released. Also, the contextual information reported that the original and the compared class names were identical for the 10 best class matches. Moreover, each of the 10 best class matches had a certainty percentage of 100%.

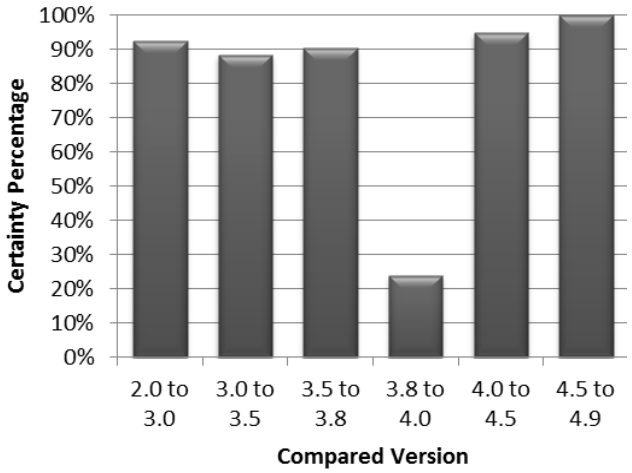


**Figure 5. Copy Relationship between JUnit Releases**

## 4.3 Resilience to Semantics-Preserving Transformations

If the WMS technique is resilient, comparing two JARs should not be affected by semantics-preserving transformations. To measure the impact of semantics-preserving transformations on the WMS technique, Pro Guard [14], Smoke Screen [21] and ZKM [22] were used to apply semantics-preserving transformations to JUnit. Then, the original JUnit JAR was compared with the obfuscated ones.

The WMS technique exhibited strong resilience to semantics-preserving transformations applied by the three obfuscators.

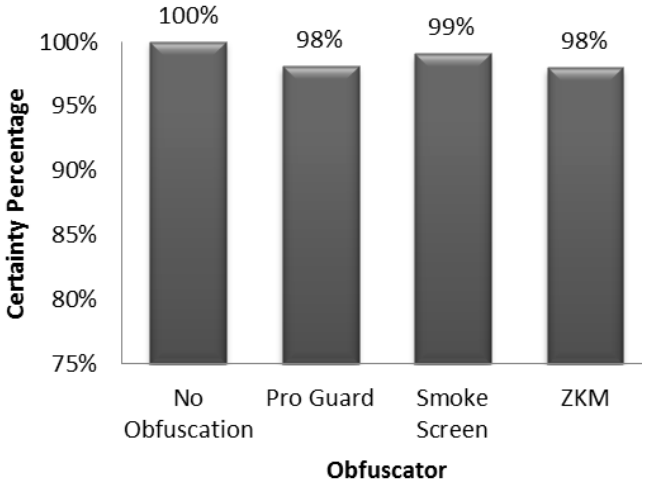


**Figure 6. Obfuscation Impact**

Nevertheless, being able to resist to well-known obfuscators does not mean that attacks on the WMS technique are impossible. Any semantics-preserving transformations that alter the method signatures such as reordering the parameters or promoting eligible instance methods to static would prevent the WMS technique from working properly.

Also, splitting a JAR into smaller ones or removing unused methods from the bytecode (a practice known as shrinking) would reduce the selectivity of the JAR. Such operations affect the accuracy of the certainty percentage yielded by the WMS technique. However, detecting a copy relationship in these cases would still be possible by using the reverse certainty percentage and the list of the 10 best class matches included in the contextual information.

## 4.4 Running Time Scalability

In order to be scalable, the time required by the WMS technique to compare two JARs must grow linearly when the size of JARs increases. Because most of the information contained in a JAR is not included in a fingerprint, the physical amount of memory required to store a JAR is not a good indicator of the size of a JAR. WMS fingerprints mostly contain information on method signatures. Thus, the number of methods contained in a JAR is a better size indicator from the perspective of the WMS technique. For the 225 comparisons performed in the credibility experiment, the relation between the number of methods to compare and the time required to perform the comparison was measured.

The results demonstrated that the average case running time complexity of the WMS technique tended to be O(n) (linear). Also, less than 10 seconds were required to compare the largest pair of JARs. Furthermore, the data gathered in the experiment revealed that the time required to compare two JARs was much shorter than the time required to create two fingerprints. The WMS technique could take advantage of the fast speed of the comparison when comparing multiple JARs. When the 30 fingerprints (2 sets of 15 fingerprints) were generated first and reused for all the comparisons, the WMS technique required 25 seconds to generate the fingerprints and only 12 seconds to perform the 225 comparisons.

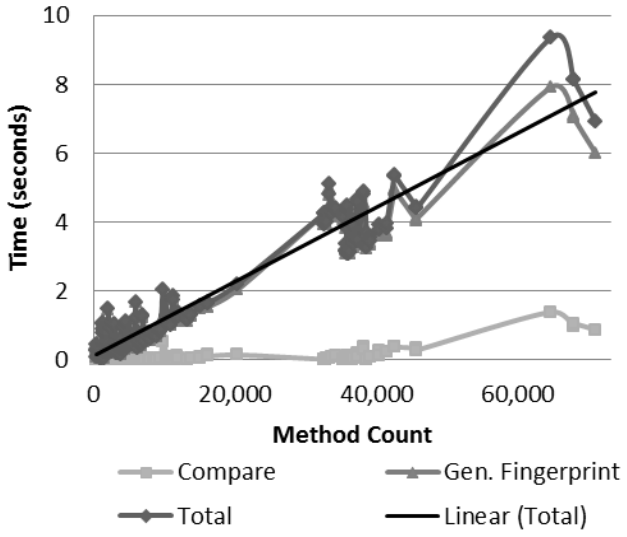


**Figure 7. Running Time Scalability**

## 4.5 Memory Usage Scalability

In order to be scalable, the size of the fingerprints must grow linearly when the size of the JARs increases. As explained in the previous experiment, the best size indicator of a JAR is the number of methods it contains. The relation between the number of methods contained in a JAR and the size of the fingerprint was used to measure the memory usage scalability of the WMS technique.

The results confirmed that the average case memory usage complexity of the WMS technique was O(n) (linear). However, the sizes of large fingerprints such as Jython and Azureus were close to one megabyte.

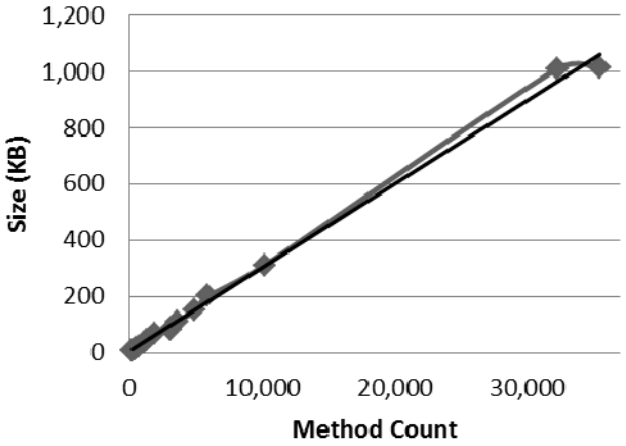


**Figure 8. Fingerprint Memory Usage Scalability**

# 5. CONCLUSION

A fingerprint technique that uses method signatures to compare JARs was presented in this paper. This technique recognizes that uncommon method signatures are more selective than common ones. The WMS technique cannot compare classes with low selectivity. Nevertheless, it is well suited to compare most JARs. The results of the experiments demonstrated that the WMS technique provided good trade-offs between the conflicting goals stated in the paper. The technique is cohesive enough that it could be included in a fingerprint framework such as the Stigmata Java Birthmark Toolkit.

# 6. REFERENCES


[1] Myles G. and Collberg C. 2005. K-gram Based Software Birthmarks. 2005 ACM Symposium on Applied Computing (March 13-17,2005, Santa Fe, New Mexico, USA) https://mailserver.di.unipi.it/ricerca/proceedings/AppliedComputing05/PDFs/papers/T07P02.pdf

[2] Ibid.

[3] Stigmata - http://stigmata.sourceforge.jp

[4] JBirth - http://se.naist.jp/jbirth

[5] SandMark - http://sandmark.cs.arizona.edu

[6] Vuze - http://www.vuze.com

[7] Ant - http://ant.apache.org

[8] Cobertura - http://cobertura.sourceforge.net

[9] Cobertura - http://cobertura.sourceforge.net

[10] CodeCover - http://codecover.org

[11] EclEmma - http://www.eclemma.org

[12] DbUnit - http://www.dbunit.org

[13] TMp3 - http://sourceforge.net/projects/tmp3
[14] Proguard - http://proguard.sourceforge.net
[15] Sandmark - http://sandmark.cs.arizona.edu
[16] JMock - http://www.jmock.org
[17] Jython - http://www.jython.org
[18] Junit - http://www.junit.org
[19] JEdit - http://www.jedit.org
[20] JExt - http://www.jext.org
[21] Smoke Screen - http://www.leesw.com/smokescreen
[22] ZKM - http://www.zelix.com