Structure Based JAR Fingerprinting

SK Alamgir Hossain

***Abstract*—Usually Java software are releases as jar files under GPL or other licenses. Users may violate the terms and conditions of those licenses. In this paper we propose a structure based system for registering jar files called jar finger printing and then detecting copies, either complete copies or partial copies of the source code. We describe algorithms for such detection, and metrics required for evaluating detection mechanisms. We also describes implementation issues of a working prototype and present experimental results that suggest the proper settings for copy detection parameters.**

***Index Terms*—Software protection, Document fingerprinting, Plagiarism detection, Fingerprint, Jar**

I. Introduction

Steganography is the art and science of writing hidden messages in such a way that no one, apart from the sender and intended recipient, suspects the existence of the message, a form of security through obscurity. Steganography in the form of media watermarking and fingerprinting has also found commercial applications. In a typical application of image watermarking, a copyright notice identifying the intellectual property owner is imperceptibly embedded into the host image. Fingerprinting is a form of watermarking in which an individualized mark is embedded into a copy of the media. A typical fingerprint would include vendor, product, and customer identification numbers. This allows the intellectual property owner to trace the original purchaser of a pirated media object. Here our main interest is software finger printing specially Java jar file finger printing. Fingerprinting a jar file discourages intellectual property theft, or when such theft has occurred, allows us to prove ownership of the jar file.

Our contribution in this paper is three-fold. First, we present a complete finger printing and verification algorithm for Java jar file protection. Second, we introduces source level finger printing method called RLE [1] encoding. Third, we evaluates our methods in different metric and presents our developed prototype's results.

The remainder of the paper is structured as follows. Section II describes some background and well known related open source software. Section III describes our design methodologies and different decisions that we made to finger print and verify a jar file. We developed a prototype based on our proposed approach. We performed different experiment using this prototype and the evaluation results are in Section IV and finally we conclude and our future goal are describes in Section V.

SK Alamgir Hossain is with the School of Information Technology and Engineering, University of Ottawa, Ottawa, ON K1N 6N5, Canada (e-mail: shoss075@site.uottawa.ca).

II. Background

A. Related Open Source Software

Document finger printing and plagiarism detection is not new in software piracy protection. Different well known techniques are available now. One of the most familiar technique is k-gram or chunk based detection. A variety of techniques [2], [3], [4], [5] have been proposed based on this idea. One such example is Moss [6] which is an automated tool used to detect similarities between programs at the source code level. The technique used to identify similarities is called winnowing [7] which divides the file into k-grams. A hash of each k-gram is then computed and a subset of hashes is selected as the document fingerprint. This technique has proven to be quite successful at detecting plagiarism within student programs. However, one of the drawbacks of systems like Moss is that similarity is computed at the source code level. Often source code is unavailable.

Tamada, et al. [8], [9] have proposed a birthmarking technique specific to Java class files which is a combination of four individual birthmarks: constant values in field variables, sequence of method calls, inheritance structure, and used classes. These four birthmarks could be used individually but the combination yields a more believable and reliable technique. The Tamada technique relies on characteristics that are statically available and targets class level theft. A dynamic birthmark technique has also been developed which uniquely identifies a program based on a complete control flow trace of it's execution [10]. This technique targets whole program theft.

Another notable work is JPlag [11] [12], an alternative plagiarism detection service that operates in a similar way to MOSS [6] in that it runs on a remote computer. They do not give out accounts to "anonymous email addresses like Hotmail, Yahoo, Gmail, etc". JPlag is based on the "Greedy String Tiling" algorithm [13]. Due to the use of a "tokens", JPlag only supports Java, C#, C, C++, Scheme and natural language text [11]. Another improved work is YAP [14] which is based on the same algorithm but is designed to run locally. Although the code is released only non-commercial use is allowed as per the README in the download [14], and thus it is not Open Source either. One of the main limitation is it supports only single file submissions [15] instead of a jar file that contain multiple files. Also Plaggie [16] is similar to JPlag in functionality and UI, but is run locally like [14]. It is released under GPL, and supports only Java code.

SID [17] is a modification of a genome comparison algorithm. It works by comparing the amount of shared information between two programs. It is not open source, and files must be submitted to the server in a "carefully formatted zip file". Sherlock [18] and SIM [19] do not

advertise licenses, but make the source code and instructions for running the program on home machine. Sherlock formerly worked in two steps, first by taking signatures and then by comparing them, however it has since been modified not to. AC [15] is a GPL-licensed, stand-alone program that supports C, C++ or Java. It incorporates multiple similarity detection algorithms found in the scientific literature and displays results graphically [15]. The majority of these programs are unsuitable for this application as they compare the files.

Algorithm 1: Finger Print Generation Algorithm

Input: A valid Jar file, $jarFile$
Output: XML sting representing the finger print

```
1  begin
2      fileList ← extractJAR(jarFile)
3      decompile(fileList, destination)
4      xmlString ← null
5      append jar properties to xmlString
6      foreach Class File f in fileList do
7          append public class properties to xmlString
8      end
9      rle ← null
10     foreach Java source File f in fileList do
11         while read every line of f do
12             remove all white space
13             α ← length of line
14             foreach Character c in line do
15                 if c is in the stripSymbols then
16                     rle +← c α
17                     α ← 0
18                 end
19                 else
20                     α +← 1
21                 end
22             end
23             append rle to xmlString
24         end
25     end
26     return xmlString
27 end
```

III. Approach

In this section we presents our jar finger printing approach and the verification approach. Here in Section III-A we discuss the design of our method and next in Section III-B we discuss some of the decision that we need to make for the finger printing and verification process.

A. Design

The finger printing and verification are perform in two steps. In the first step the finger print is created from the provided jar file. The finger print contain some information that will be used in the next step.

The finger print generation step is further divided into some smaller steps which are shown in Figure 1. As the jar file is a compressed file format so at first the system need to extract

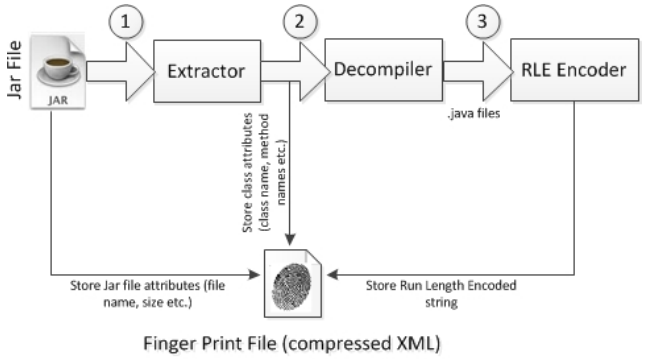

Fig. 1. Finger print generation process.

the jar file. A jar file may contain any types of file. It may contain XML file, Image file, Class file, Java file, text file etc. For our finger print generation process we use jar file properties, class file properties and Java source structure. As in a jar file the most important file is class file so here our main attention is on class files. From the jar file the system will collect the jar file properties like jar file name, jar file size, last modification date, number of files inside the jar file etc. When intruder misuse a jar file this attributes are help full for the similarity check. After extraction the system will calculate each class files properties like class file method names, it's super class name, package name, variables name etc and will store in to the finger print file. In the next step each class file decompile and generate the Java source file. Always it is not possible to generate the Java source code from the Java byte code. But most of the time it's possible to get significant amount code. After getting the source code from the decompiler the system will calculate the run length encoding of the corresponding source code. Run-length encoding (RLE) is a very simple form of data compression in which runs of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most useful on data that contains many such runs. Here in our method we consider some selected symbol for the run. All other symbols, characters are continuous character for the count. This RLE code string is very shorter than the original source code. The main benefit of this RLE code that it can detect a simple change of code as it store the structure of the code. As the RLE string may be big so we compress the string with standard Gzip or gunzip compression tools for reducing the finger print file. Those tools can compress a text file more than 50:1 ratio. The finger printing generation algorithm is shown in Algorithm 1. In the algorithm line number 2 and 3 will extract and decompile the jar file. Jar file properties and class file properties will be calculated in line 5 and 6 to 8 respectively. In line 10 each Java file read one per line and remove the white space then produce a nonstop string and then convert to RLE.

Algorithm 2: Finger Print Verification Algorithm

Input: A valid finger print file FP generated by Algorithm 1 and a valid Jar file $jarFile$ to test
Output: Percentage of match

1 **begin**
2 $oldFP \leftarrow$ fetch the content from finger print file FP
3 $newFP \leftarrow$ Algorithm 1 ($jarFile$)
4 $\beta \leftarrow$ compareJarProperties($oldFP$, $newFP$)
5 $classPro \leftarrow$ null
6 $rle \leftarrow$ null
7 **foreach** *xml file tag* $f1$ *in* $oldFP$ **do**
8 $f2 \leftarrow$ searchFileTag($newFP$, $f1$)
9 $classPro$ +$\leftarrow$ compareClassProperties($f1$, $f2$)
10 rle +$\leftarrow$ Longest Common Subsequence of RLE of $f1$ and $f2$
11 **end**
12 $\gamma \leftarrow$ average $classPro$
13 $\delta \leftarrow$ average rle
14 return $\frac{\beta+\gamma+\delta}{3} \times 100$
15 **end**

In the verification process the system need two files one is the finger print file and another is a Jar file that need to verify. The verification steps and the algorithm are shown in Figure 3 and in Algorithm 2 respectively. In Algorithm 2 the jar file need to extract, decompile and generate the finger print. This process will perform line 3. Now there will be two finger print one is the finger print that generated from the original jar file and another is generated from the jar that we need to verify. In line 4 the Jar file properties information stored in the finger prints will be compare. Next in line from 7 to 11 the class properties and the RLE string will be compare. For RLE compare the algorithm will use Longest Common Subsequence [20] string matching algorithm. Finally in line 14 the percentage of match result will be returned. The RLE comparison process are illustrated in Figure 2 where the green down arrow indicate the source code in the original jar file and the red down arrow indicate the partial copy of the original source code and the final result shown in that figure. The RLE string match algorithm will return for this example is 15+8+13+15+7 = 58 that a length of code 68 or 85% code are common.

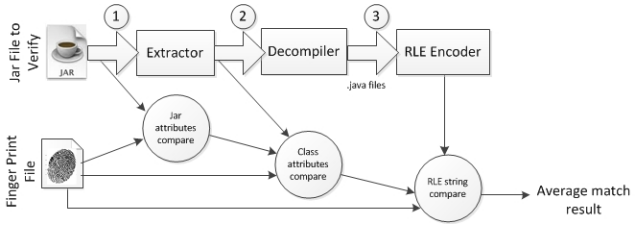

Fig. 3. Finger print verification process.

B. Decisions Made

In this section we presents some of the decisions that we need to make for the whole finger printing and verification process. Here in our proposed method we use some metric

TABLE I
METRICS USED FOR FINGER PRINT GENERATION

	Metric
1	Name of Jar
2	Size of Jar
3	Last modification
4	Number of files in the Jar
5	File names in the jar
6	Modifier names of class files
7	Package name of class files
8	Super class name of class files
9	Inner class names of class files
10	Interface names of class files
11	Name of constructors of class files
12	Name of variables of class files
13	Name of methods of class files
14	Run Length Encoding of Java source codes

that are shown in Table I. Where 1-5 are indicating jar file properties, 6-13 are Java class file properties and 14 is the RLE. In our approach we used "; {}() = []" as the strip symbol for the RLE encode. The more symbol we use the more perfect match result will be found but the time and space will be more. So we need to balance to determine the perfect list of strip symbols. From the empirical estimation we selected those symbol for our system as a stripping symbols. For the decompile process different decompiler are available but we suggest to use JODE [21] decompiler because it is open source and also the performance is satisfactory. The content of the finger print file is a XML as shown in Figure 4. For our method we also consider that if A and B two jar file then in A, x% of code from B not equal to in B, x% of code from A. Because the percentage calculation depends on the size of the original Jar file. It is true when we say that what percentage of code match in a Jar file compare with an original Jar file. The results depends on the original Jar file. For example if a Jar file copy 1KB of code from each of two Jar file of size 10KB and 100KB then result will be 10% source copy form 10KB and 1% copy from 100KB file.

```
<?xml version="1.0" encoding="utf-8"?>
<FP>
  <!-- Finger Print Generator ID -->
  <id>__FPGenV1.0__SK_Alamgir_Hossain__Universite_of_Ottawa__</id>

  <jarname></jarname><!-- Jar file name -->
  <jarsize></jarsize><!-- Jar file size -->
  <lastmodified></lastmodified><!-- Last modification date time -->
  <numoffile></numoffile><!-- Number of files inside the Jar file -->
  <filenames></filenames><!-- File names in side the Jar file -->
  <similarities>
    <file name=""><!-- RLE code here -->
      <class modifiers="" name="" package="" superclass="">
        <innerClasses></innerClasses>
        <interfaces></interfaces>
        <constructors></constructors>
        <variables></variables>
        <methods></methods>
      </class>
    </file>
    <file name="">...</file>
    <file name="">...</file>
  </similarities>
</FP>
```

Fig. 4. XML Specification of Finger Print.

IV. Results and Validation

Based on our proposed method described in Section III we developed a prototype system (Figure 5) which was developed

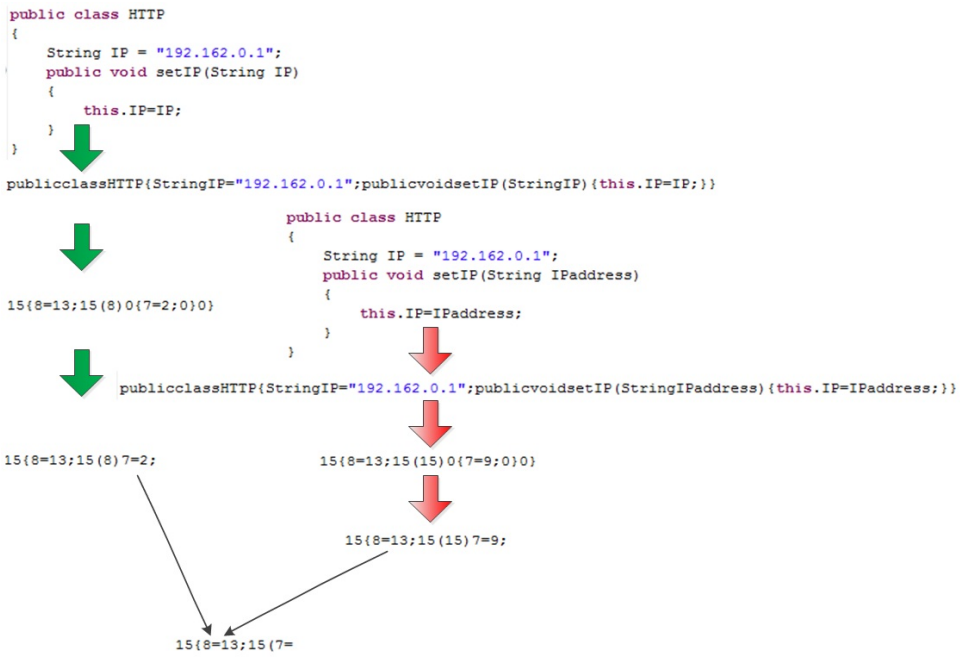

Fig. 2. Step by step Run Length Encoding Creation Process.

TABLE II
JAR FILES FOR TESTING

Jar ID	Jar Name	Jar ID	Jar Name
1	axiom-api-1.2.8.jar	10	knownjar.jar
2	axiom-impl-0.95.jar	11	log4j.jar
3	eclipse.jar	12	mail.jar
4	http.jar	13	tomcat-jni.jar
5	jakarta-oro-2.0.8.jar	14	unknownjar1.jar
6	javamail-pop3-1.4.jar	15	unknownjar2.jar
7	jboss-j2ee.jar	16	unknownjar3.jar
8	jboss.jar	17	XmlSchema-1.3.1.jar
9	junit.jar	18	xwork-2.1.2.jar

by using JAVA 1.6 and Eclipse editor, and design a test case contains a list of 18 jar files as shown in Table II. We perform different experiment to identify the validity of our system.

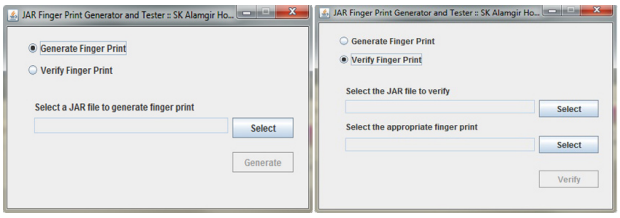

Fig. 5. User interface of our developed prototype.

A. Overall Results

The Jar file tested fell into two groups. First group was in standard commonly used jar downloaded from [22] and the other group was created by using Eclipse that contain almost identical sources. We used this jars to identify the source level comparison that whether our method correctly identify the similarities. Table II shows the jar files that we used in our experiment. In the experiment we at first created finger print of each of the jar file listed in Table II in batch mode. In the next step we compare each jar file with the others finger print generated in the first steps in batch mode. The comparison results shows in Table III. Table III shows only those results that returned a similarity percentage of over 30%. For the validity of our approach we also compare some finger prints with it's original jar file, that is some entries in Table III shows 100% match result. Here we were not considering those results which percentage of match less than or equal to 30% because none of the expected comparison returned a result of less than or equal to 30% similarities at this level appear to be identical and uninteresting likely as a result of both jars using libraries or similar reasons. The overall distribution is shown in Figure 6. It is clear that the vast majority of comparisons are uninteresting, with over half of the comparisons generated falling in the range of 0 to 10%.

TABLE III
COMPARISON RESULTS OVER 30%

	Known Jar	Unknown Jar	Match (%)
1	axiom-api-1.2.8.jar	axiom-api-1.2.8.jar	100
2	javamail-pop3-1.4.jar	mail.jar	91
3	knownjar.jar	unknownjar1.jar	97
4	knownjar.jar	unknownjar2.jar	88
5	knownjar.jar	unknownjar3.jar	77
6	log4j.jar	log4j.jar	100
7	log4j.jar	mail.jar	33
8	mail.jar	javamail-pop3-1.4.jar	36
9	mail.jar	log4j.jar	33
10	unknownjar1.jar	unknownjar2.jar	90
11	unknownjar1.jar	unknownjar3.jar	77
12	unknownjar2.jar	unknownjar3.jar	77
13	unknownjar3.jar	knownjar.jar	100
14	unknownjar3.jar	unknownjar1.jar	96
15	unknownjar3.jar	unknownjar2.jar	84
16	xwork-2.1.2.jar	xwork-2.1.2.jar	100

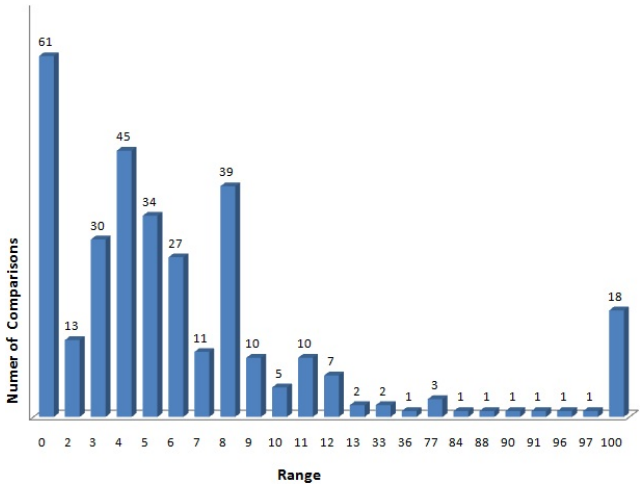

Fig. 6. Comparison rating distribution.

Fig. 7. Different metric and percentage of runtime for generating finger print.

B. The Effect of Different Comparison Factors

In this section we presents the effect of different factors of the comparison process of our algorithm. The effect of different metric contributed in the finger print generation process are shown in Figure 8. The Jar file numbers are the same as in Table III. From this figure it is clear that lower than 10% time need to calculate the properties (Jar, class, RLE) and the rest of the time required to compress and decompile the file.

We also tested the Jar files (Table III) and plot the Jar file size with the corresponding finger print file size and the result shows in Figure 8. Based on this figure we can say that finger print file size is not consistent because the size not only depend on the Jar file attributes or class attributes but also depends on how many Java class file inside the Jar as our algorithm mainly concern about the Java class files or byte code available inside the Jar file.

Figure 10 shows the effect of Jar file content in generator outputs. In our experiment we used 3 sets of jars with 10 files in each Jar and their size was identical for each group but the content was different. In the first group all the files in the jars were class file only and the detector output shown in Figure

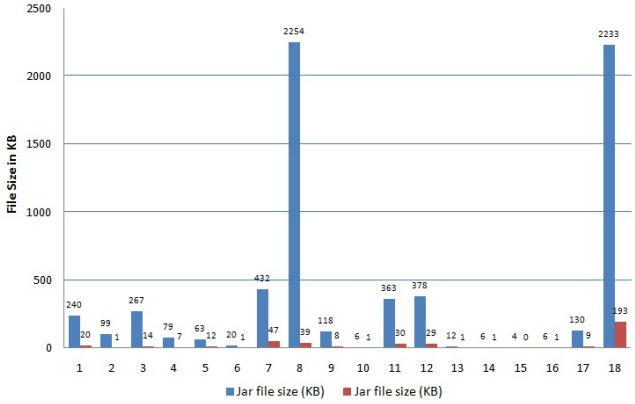

Fig. 8. Comparison of Jar file size and generated finger print file size

10(a). In the second group each jar has 5 class files and other 5 non class files. The detector result shown in Figure 10(b). In the third group each jar has no class file, all files are non class files and the detector result shown in Figure 10(c). From this three figure we can say that the performance of our algorithm is depend on the type of files inside the jar. If all the files are class file the algorithm need more time to process. The Figure 10(c) the FP size and output time is almost constant because as there is no class file inside so there will be no de-compilation time and also need not to calculate the class file properties and also the RLE.

C. Running Time

The running time of our proposed system is approximately linear. As shown in Figure 9, the length of time to finger print the jar typically increases with size. Although there are some exception available. In Figure 9 the straight line indicating the average time to run the finger printing algorithm discussed in Section III-A .

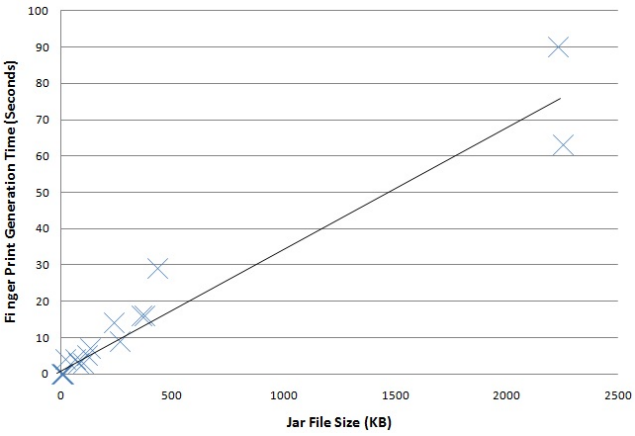

Fig. 9. Average Time to fingerprint a Jar File (Extreme Values Removed).

V. Conclusion and Future Work

A. Conclusion

In this paper we presented a method for finger printing a jar file and later on verify other jar files with this finger

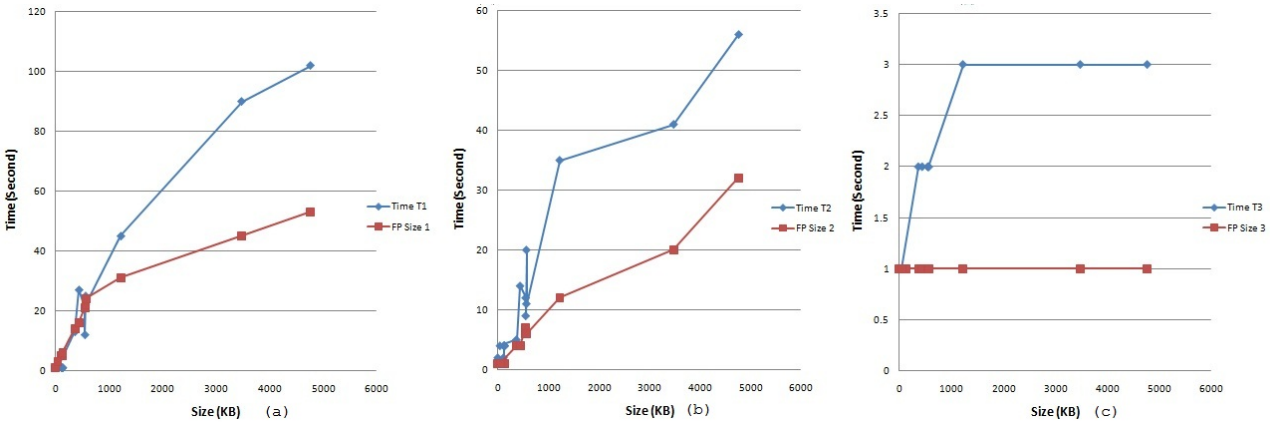

Fig. 10. Finger printing jar files and the results in different conditions.

print to get how similar the new jar file with the original jar file. The developed system can be used for copy protection in Universities or companies. From the experiment it is clear that our method show good performance without some exceptions. Although our system has some limitations like it need a good decompiler for decompiling the class files. But practically it's not possible to regenerate the source code completely from Java byte code. As normally jar file may contain large collection of files so it will be very time consuming with our system. But most of the time need to decompress and decompile the jar file. In our future work we want to address the issues into more details. We also need more experiment in different uncertainty conditions. However, we believe that our proposed techniques will remain as a motivation for further research in this area.

ACKNOWLEDGMENT

I would like to thank Dwight Deugo, Professor, Carleton University, Ottawa. Without his valuable feedbacks it was not possible to finish my work properly.

REFERENCES

[1] RLE, "Run length encoding, http://en.wikipedia.org/wiki/run-length_encoding," Last Access, November 08, 2010.
[2] N. Heintze, "Scalable document fingerprinting," in *Proceedings of USENIX Workshop on Electronic Commerce*, 1996.
[3] S. Brin, J. Davis, and H. Garcia-Molina, "Copy detection mechanisms for digital documents." in *ACM SIGMOD international conference on Management of data*, 1995., p. 398409.
[4] A. Z. Broder, "On the resemblance and containment of documents," in *In Compression and Complexity of Sequences*, 1998, p. 2129.
[5] U. Manber, "Finding similar files in a large file system," in *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994, p. 110.
[6] MOSS, "A system for detecting software plagiarism." in *http://theory.stanford.edu/ aiken/moss/. Last accessed October 31th, 2010.*
[7] S. Schleimer, D. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proceedings of the 2003 SIGMOD Conference*, 2003.
[8] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, "Detecting the theft of programs using birthmarks," in *Information Science Technical Report, Graduate School of Information Science, Nara Institute of Science and Technology*, 2003.
[9] H. Tamada and M. Nakamura, "Design and evaluation of birthmarks for detecting theft of java programs," in *IASTED International Conference on Software Engineering*, 2004, p. 569575.
[10] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in *Information Security Conference*, 2004.
[11] JPlag, "Detecting software plagiarism," in *https://www.ipd.uni-karlsruhe.de/jplag/. Last accessed October 31st, 2008.*
[12] L. Prechelt, G. Malpohl, and M. Philippsen, *Finding Plagiarisms among a Set of Programs with JPlag*. Journal of Universal Computer Science, 2002, vol. 8, no. 11.
[13] P. L, M. G, and P. M, "Jplag: Finding plagiarisms among a set of programs." in *http://page.mi.fu-berlin.de/prechelt/Biblio/jplagTR.pdf. Last accessed October 31st, 2010.*, March 28th, 2000.
[14] P. D. . YAP, "http://luggage.bcs.uwa.edu.au/accessed november 1st, 2010."
[15] AC, "An anti-plagiarism system for programming assignments." in *http://tangow.ii.uam.es/ac/. Last accessed November 1st, 2010.*
[16] Plaggie, "http://www.cs.hut.fi/software/plaggie/. last accessed november 1st, 2010."
[17] S. P. Detection., "http://genome.math.uwaterloo.ca/sid/. last accessed november 1st, 2010."
[18] T. S. P. Detector, "http://www.cs.su.oz.au/ scilect/sherlock/. last accessed november 1st, 2010."
[19] T. software and text similarity tester SIM., "http://www.cs.vu.nl/
[20] LCS, "http://wordaligned.org/articles/longest-common-subsequence," Last Accessed, November, 2010.
[21] JODE, "Java optimize and decompile environment , http://jode.sourceforge.net/, accessed november, 2010."
[22] java2s, "http://www.java2s.com/," Last Accessed, November, 2010.

SK Alamgir Hossain SK Alamgir Hossain received the B.Eng. degree in computer science and engineering from Khulna University, Khulna, Bangladesh. Currently he is studying M.C.S degree in Computer Science, University of Ottawa, Ottawa, ON, Canada. He is also working in the Multimedia Communications Research Laboratory (MCRLab), School of Information Technology and Engineering. From 2008 to 2009, he was a Lecturer with the Computer Science and Engineering Discipline, Khulna University, Khulna, Bangladesh. Before join to Khulna University he worked a few years with JAXARA IT Ltd as a Software Engineer. He has authored or coauthored more than 7 publications including refereed journals and conference papers. His research interests include Ambient Intelligence and Humanized Computing, Virtual reality with Haptic, Smart environment and Telesurveillance System.