Deducing Similarities in Java Sources from Bytecodes

Brenda S. Baker
Bell Laboratories
Lucent Technologies
700 Mountain Avenue
Murray Hill, NJ 07974
bsb@bell-labs.com, `http://cm.bell-labs.com/~bsb/`

Udi Manber
Department of Computer Science
University of Arizona
Tucson, AZ 85721
udi@cs.arizona.edu, `http://glimpse.cs.arizona.edu/udi.html`

Abstract

Several techniques for detecting similarities of Java programs from bytecode files, without access to the source, are introduced in this paper. These techniques can be used to compare two files, to find similarities among thousands of files, or to compare one new file to an index of many old ones. Experimental results indicate that these techniques can be very effective. Even changes of 30% to the source file will usually result in bytecode that can be associated with the original. Several applications are discussed.

1 Introduction

Java bytecode, particularly in the form of applets, is geared to become the common way to execute programs through the web, and not only by traditional computers. Network computers, even appliances, are expected to be controlled by Java bytecode, which will arrive through the net transparently and for the most part, automatically. There is obviously a great need to be able to control all these programs. There will be problems of security, management of updates, portability, handling preferences, deletions, and so on.

This paper introduces several techniques to help one aspect of these problems. We show that it is possible, given a large set of bytecode files, to identify most of those that originate from similar source code. In addition, an index can be built from any number of bytecode files, such that given a new bytecode one can very quickly find all the old ones that are similar to it. Our tools also can detail in a convenient way just which sections of the files are similar.

Our techniques make it possible to identify bytecode files that came from the same source without knowing the actual source even if significant parts of the source files are different. The files do not even have to be of similar sizes. For example, the original source code files may be different versions of the same program or different programs containing similar sections.

Finding similarities from compiled code is much more difficult than finding similarities in source code or text, because a small change in the source code can produce completely different binaries when they are viewed just as sequences of bytes.

To accomplish these goals, we adapt three tools designed to find similarity in source code and text to work with bytecode files. Siff [24] [1] is designed to analyze large number of text files to find pairs that contain a significant number of common blocks ("fingerprints"), where a block corresponds to a non-trivial segment of code, usually a few lines.

[1] The original name was SIF, but at some unknown point during the development an extra f was added to make it more natural.

Dup [5], searches sets of source files to look for sufficiently long sections that match except for systematic transformations of names, such as variable names, and can deal efficiently with a few million lines of source code. The UNIX utility diff (described in [21]) uses dynamic programming to identify line-by-line changes (insertions or deletions) from one file to another, and is useful for detailed comparison of two files and for automated distribution of patches.

None of the programs mentioned above was designed to deal with binaries. The only work that we know of that may have applied one of these to binaries is .RTPatch [27], a tool for creating patches (differences between files) for updating files. This tool is claimed to work for arbitrary files, even binaries; while no technical description is given of their methods, it seems likely that it applies the diff algorithm to arbitrary bytes rather than to hashes of lines of text. We are not familiar with any work that can find similarities in large number of binaries.

Our adaptations of siff and diff do not work directly on bytecode files, or even on disassembled bytecode files. Bytecode files contain many indices into tables, and values of most indices can be changed by a slight (even one character) change to a Java source file. Therefore, we encode disassembled bytecode files in a "normal form" which takes into account positional values that are less affected by small changes than absolute values of indices. This encoding, based on a technique first used in dup, enables the programs to find the hidden similarity in bytecode files. Since this technique is already incorporated into dup, dup can work on disassembled bytecode files; however, additional simple preprocessing improves its performance.

There are other tools for finding similarities in text or source code, as well. However, tools based on style metrics, such as [7, 19, 25], or data flow graphs [17] would require decompilation of bytecode files in order to be applied. Some other tools based on fingerprints, such as [16, 20, 10], chunks of text [9, 30, 34], or visualization via a graphical user interface [11] may be adaptable to byte code files using the same techniques that we use for siff, dup, and diff.

To search for similar files in a large set of bytecode files, we run siff on the encoded disassembled bytecode files and dup on the preprocessed disassembled bytecode files. We show that combining the output from siff and dup is more effective than either individually at finding similar files while keeping false positives low. We then use dup and diff on the similar pairs to examine the similarities in more detail.

Our programs are fast and can analyze thousands of bytecode files in minutes. A new file can be compared to an index of thousands of existing files in a second or two. The number of false positives is kept to a very small minimum. Our programs are written in C and run under UNIX. Detailed experimental results are given later in the paper.

We foresee several applications for our tool.

program management: When people have numerous Java classes, and get many more on a regular basis, there will be a great need to organize them, often not according to the original "plan." Being able to tell which classes are similar can be very helpful. Sometimes a similarity will identify the source. Knowing that a very similar class is already installed on one's disk may be helpful in deciding what to do with the new class. It can also help with version control, etc. In some cases, especially for programs that perform mostly arithmetic operations, it may be possible to identify programs that implement the same algorithm, even when they are written by independent writers. For example, when we ran experiments on thousands of arbitrary class files, we discovered two MD5 programs with 78% similarity in their bytecode files.

Plagiarism detection: Our approach will identify stolen code if only minor changes are made to it (including any amount of syntactic changes such as changing variable names). However, the advance in sophistication of obfuscators [13, 12], especially for Java, would allow someone to hide any code segment pretty well, and our methods (and very possibly any other method) will not be able to identify it. One may be able to identify that obfuscators were used, on the other hand, which may be good enough (e.g., in a classroom).

Program reuse and reengineering: It will obviously be useful to know which classes are similar when trying to reuse them. Identifying versions is a good example of that.

Uninstallers: If you finally obtain the exact code you wanted, then you probably want to get rid of other code that is very similar.

Security: Detecting that a class is similar to a known bad class can be very helpful.

Compression: It may be possible to store only differences among classes that are very similar. But first one must detect all such similarities. This could be especially useful for low-storage devices.

Clustering: Small similarities are quite typical among programs written by the same person. Our tools cannot by themselves cluster everything, but they can help a clustering program identify many candidates.

Miscellaneous: There was recently discussion in the press about Sun Microsystems allegedly using a copy of a popular benchmark program directly in its Java compiler. We believe that our program would have discovered that easily and could be used to continuously monitor for cases like that.

The rest of the paper is organized as follows. The next section describes the three similarity tools that we adapt to Java bytecode files. Section 3 describes how we process Java class files to make them suitable for comparisons by dup, siff, and diff. In Section 4 we present experimental results. These include experiments with siff alone, siff and dup together, and diff alone. Section 5 describes related work. We end with conclusions and future research.

2 Tools for finding similarity

This section describes the three tools (diff, siff, and dup) that we adapt to finding similarity in Java bytecode files. All three were designed originally for source or text files.

2.1 Siff

Siff[24] is a program to find similarities in text files. It uses a special kind of random sampling, such that if a sample taken from one file appears in another it is guaranteed to be taken there too. A set of such samples, which are called in siff *approximate fingerprints*, provides a compact representation of a file. With high probability, sets of fingerprints of two similar files will have large intersection, and the fingerprints of two non-similar files will have a very small intersection. Siff works in two different modes: all-against-all and one-against-all. The first mode finds all groups of similar files in a large file system and gives a rough indication of the similarity. The running time is essentially linear in the total size of all files, which makes siff scalable. (A sort, which is not a linear-time routine, is required, but it is performed only on fingerprints, and therefore does not dominate the running time unless the total size of all files is many GBytes.) The second mode compares a given file to a preprocessed *approximate index* of all other files, and determines very quickly all files that are similar to the given file. In both cases, similarity can be detected even if the similar portions constitute as little as 25% of the size of the smaller file (for smaller percentages, the probability of false positives is non trivial, so although siff will work, its output may be too large).

2.2 Dup

Dup [4, 5, 6] looks for similarity of source codes based on finding sufficiently long sections of code that almost match. Dup's notion of almost-matching is the *parameterized match* (*p-match*): two sections of code are a p-match if they are the same textually except possibly for a systematic change of variable names; e.g. if all occurrences of "count" and "fun" in one section are changed to "num" and "foo", respectively, in the other. For a threshold length (in lines) specified by the user, dup reports all longest p-matches over the threshold length within a list of files or between two lists of files. It can compute the percentage of duplication between two files or the percentage of duplication for the cross-product of two lists of files. It can also generate a profile that shows the matches involving each line in the input and a plot showing where matches occur. Dup has been found useful for identifying undesirable duplication within a large software system, looking for plagiarism between systems, and for analyzing the divergence of two systems of common origin.

Using dup, one may choose to base a notion of similarity on the existence of matching sections over a threshold length, on the percentage of common code resulting from these matching sections, or on some combination of the two.

The key idea that enables dup to identify p-matches is to replace tokens such as identifiers by offsets to remove the identity of the identifier while preserving the distinctness of different identifiers. In particular, the first occurrence of an identifier is replaced by a 0, and each later occurrence of the same identifier is replaced by the number of tokens since the previous one. For example, `u(x,y)=(x>y)?x:y;` and `v(w,z)=(w>z)?w:z;` are both encoded as `0(0,0)=(6>6)?5:5;` because u and v occur in the same positions, as do the pair x and w and the pair y and z; the numbers 6 and 5 represent the number of tokens (single symbols, in this case) between successive occurrences of the same identifier token. More generally, if max, arg1, and arg2 are tokens, the same encoding represents `max(arg1,arg2)=(arg1>arg2)?arg1:arg2;`.

Dup computes longest p-matches via a data structure called a *parameterized suffix tree (p-suffix tree)*. The p-suffix tree is a compacted trie that represents offset encodings of suffixes of the token string, but only uses linear space because only the offset encoding of the entire input is stored. At each access to an offset, the previous context is used dynamically to determine whether this is the first use of this parameter within the suffix being processed. The algorithms for building a p-suffix tree and searching it for duplication are described in [5, 6].

2.3 Diff

Diff is the oldest tool for finding commonality between files. Given two text files, diff reports a minimal length edit script for the differences between the two files, where edit operations include insertions and deletions. Many algorithms for minimal edit scripts have appeared in the literature. A popular version of diff today is the GNU implementation based on an algorithm of Myers [26]. Because of the quadratic worst-case running time, diff can be slow for comparing large amounts of code with many differences.

Graphical user interfaces such as gdiff (which runs on SGI workstations under UNIX) make it convenient to look at output from diff by aligning identical sections of two files, side by side. Variants of gdiff exist for other platforms; a list is given in [15].

2.4 Combining the tools

The three tools use three very different notions of similarity. In this paper, we show that a combination of the three is more powerful than any of them individually.

For searching large numbers of files, siff and dup can be used together either to broaden the notion of similarity (by taking the union of pairs of files found by siff and dup) or to make it more stringent (by taking the intersection of the pairs reported as similar by dup and siff).

Siff finds similarities that are not found by dup because occasional differences may disrupt the longer matches looked for by dup; on the other hand, dup finds some files that have long matches but aren't reported by siff because the overall percentage of similarity in the files is too low. For example, if a block of code is added to an otherwise unmodified file, the percentage of similarity might fall below the percentage of similarity selected for siff to report, but dup would find that the rest of the file was unchanged.

For more detailed analysis of files found to be similar, dup, diff, and gdiff are useful. Dup provides a list of matching sections of code, a profile showing for each line what other lines match (based on matches over threshold length), and scatter plots for visualization. Diff and gdiff provide an alignment of the two files that enable looking at the differences line-by-line, and are especially effective when the number of differences is very small.

3 Adapting the tools to Java bytecode files

This section describes how we adapt dup, siff, and diff to handle Java class files. A class file, usually obtained by compiling a Java file, is a stream of bytes representing a single class in a form suitable for the Java Virtual Machine [22]. We use the terms "class files" and "bytecode files" interchangeably.

3.1 Information in Java class files

The first items encoded in a class file are a magic number that identifies the file as a class file, the minor version number, and the major version number. A Java Virtual Machine of a particular major and minor version number may run code of the same major version number and the same or lower minor version number. The version handled currently by our system is major version 45, minor version 3, described in [22].

The remainder of a class file is mainly tables of structures. Of these, the "constant pool" and the method table are important to the design of our tools.

The constant pool contains information about all the constants used in the class, e.g. strings, integers, fields, and classes. This table is very important as many other parts of the class file (including code for methods) contain indices into the constant pool. For example, a reference to a different class includes the index for the string that names that class.

For each method, the method table contains the code and other information such as the number of local variables, exception-handlers, the maximum stack length, indices into the constant pool representing the method name and a string describing the method type, and optional tables aimed at debuggers, such as line number relationships between bytecode and source and information relating local variable names to the code.

3.2 Disassembly of Java class files

The first step in processing a Java class file is to disassemble it. Disassembling a class file requires keeping track at each point of how far the parsing has gotten in the conceptual hierarchy of tables and structures, based on tables in the disassembler derived from [22]. We wrote our own disassembler, although others have been implemented previously; a list appears in [32].

After disassembling the file, we extract the code of each method for further analysis. The disassembled code contains opcodes and arguments for a sequence of assembly-language-level instructions. Figure 1 shows an example of a section from the middle of a disassembled bytecode file, with comments added to identify the numerical opcodes. Since opcodes can have variable numbers of arguments, we put one opcode or argument per line. A character at the start of the line identifies the type of item on the line. Opcodes, indices into the constant pool, indices into the local variable table, signed integers, unsigned integers, and jump offsets are identified by 'o', 'c', 'v', 'i', 'u', and 'j', respectively. Jump offsets are translated from numbers of bytes in the class file to numbers of lines in the disassembled file.

```
o182            #invokevirtual
c106
o153            #ifeq
j+17
o025            #aload
v4
o180            #getfield
c253
o025            #aload
v5
o182            #invokevirtual
c102
o025            #aload
v4
o180            #getfield
c253
o025            #aload
v4
o182            #invokevirtual
c260
o025            #aload
v5
o180            #getfield
c253
```

Figure 1: Disassembled bytecode

Dup can be run on disassembled bytecode if it is provided with an appropriate lexical analyzer, though performance is improved by undoing the jump offsets before running dup. (See the next section.)

Running siff or diff on the disassembled file without further preprocessing does not produce useful information. For example, changing a 4 to a 5 in two places in a 182-line Java file resulted in over 1100 lines of diff output on the disassembled bytecode file, and less than 1% similarity reported by siff.

The reason that siff and diff fail is that indices into the constant table or local variable table may change with slight changes to the Java file, due to additions, deletions, or reorderings of the constant pool and/or local variable table. Such indices typically occur frequently in the code, as in the example of Figure 1. When the files mentioned in the previous paragraph are preprocessed as described shortly, diff reports only changes in two lines (containing opcodes referring to a constant of 5 rather than 4) and siff finds 98% similarity.

3.3 Preprocessing for dup

With an appropriate lexical analyzer, dup can be run on disassembled bytecode files. However, before running dup, it is preferable to undo the jump offsets already present in the disassembled code by changing jumps into a "goto label" form. Dup will compute the offsets itself for the labels. The dynamic way in which dup calculates offsets relative to suffixes means that when two otherwise identical sections of code contain jumps to earlier points and the jumps cross insertions or deletions, these jumps will not cause mismatches, as would happen with a precomputed fixed encoding. Thus, this preprocessing enables dup to find longer p-matches.

Our lexical analyzer for the disassembled bytecode files (without jump offsets) breaks up the input into two classes of tokens: parameter tokens and non-parameter tokens. Offsets are computed for parameter tokens but not for non-parameter tokens. Parameter tokens include indices into the constant pool, indices into the local variable table, labels for jumps, and signed and unsigned integers; the various types of parameter tokens are distinguished so that the offsets are computed separately and tokens of different types will not be matched to each other in parameterized matches. The non-parameter tokens include opcodes.

3.4 Further preprocessing for siff and diff via offsets

Even though absolute values of table indices in bytecode files may change with slight changes to the Java source, there are still hidden similarities in the bytecode files. In particular, the corresponding uses of indices maintain the same *positional* relationship.

Consequently, we use the same offset encoding that is used in dup. In the context of disassembled bytecode, what corresponds to the "identifiers" are the indices into the constant pool or local variable table. (Jumps are already encoded as offsets in the bytecodes.) Siff and diff are then run on the offset-encoded files.

We treat each index into the constant pool or local variable table as a parameter to be replaced by an offset. The first occurrence of each index is encoded as 0, and thereafter each use of an index is encoded as the negative of the number of lines since the previous use (if any) of the same index. Offsets for indices into tables are negative to be consistent with jump offsets, which are negative for a jump to a preceding line and positive for a jump to a later line. The offsets are calculated independently for the constant table and the local variable table. The example of Figure 1 is shown in Figure 2 after calculating offsets for the entire file from which this section was extracted.

```
o182
c-26
o153
j+17
o025
v-10
o180
c-10
o025
v-10
o182
c-26
o025
v-8
o180
c-8
o025
v-4
o182
c-26
o025
v-12
o180
c-8
```

Figure 2: Disassembled bytecode after calculating offsets

This encoding decreases reliance on the absolute value of the indices but preserves the information as to whether indices for different instructions are the same or different. For example, if two files are the same except that indices in one file are always one larger than indices in the other, the encodings of the two files will be identical. The next section describes experiments demonstrating that this encoding enables siff and diff to work effectively.

4 Experiments

4.1 Experiment 1: Random changes

In the first experiment, we took one Java program and made many different random changes to it. These changes included addition of statements (e.g., "newvariable = 43;") in random places, and substitution/deletion of statements (e.g., changing complex conditions in "if" and "while" statements to "i $<$ 1"). We varied the number of changes and the ratio between additions and deletions. We ran these tests on two different Java programs. For each run, we measured the similarities of the source code (using the original siff) and the similarities of the bytecode files. The results, shown in Table 1, consistently show that the bytecode similarities are close to the source code similarities. For each of the two programs we partitioned the tests into three groups according to source similarities: 90-100%, 80-89%, and 70-79%. The results are averages for each group, showing the number of trials, the average similarity for source and bytecode, and the maximal difference between them in all the trials.

Furthermore, we also ran siff on all the variants of the two programs together, and found no false positives.

4.2 Experiment 2: a large set of bytecode files

In the second set of experiments we took 2056 Java bytecode files (from 38 collections of files from many different sources) and ran tests on all of them at the same time (allowing for at least 50% similarity). The goal was to look for similar files from different collections.

Siff reported 634 ordered pairs of files with similarities of at least 50%. We define similarity of two files as the percentage of one file that is contained in the other. As a result, similarity is an asymmetric relation: for example, if a file A is contained in another file B twice its size, then A is 100% similar to B, but B is 50% similar to A. We use ordered pairs here for this reason.

Of the 634 pairs, 591 were between files in the same collection, and 43 were between files in different collections. Next, we use dup to aid in analyzing which of these similar pairs represent interesting relationships.

For the same 2056 Java bytecode files, dup reported 92 ordered pairs of files to have at least one common code section of 200 lines or more, in comparison to the 634 ordered pairs reported by siff. Table 2 shows the breakdown as to how many ordered pairs were reported by siff alone, by both siff and dup, and by dup alone.

The goal of the experiment was to look for similarity in files from different collections (out of the 38 collections we downloaded). The second line of the table gives the breakdown with respect to similarities between files from different collections. The initial analysis was in terms of ordered pairs, since similarities can be asymmetric, but for ease of discussion, the third line shows the corresponding number of unordered pairs.

Of the 9 different-collection (unordered) pairs reported by both siff and dup, we believe that 8 pairs are originally from the same source, based on the pairs of files having the same name. Of the 8 pairs, four pairs are identical files, and four are in the range of 57%-100% similar according to siff and 45% to 97% similar according to dup (based on just the code sections of at least 200 lines that match.)

The remaining different-collection (unordered) pair of files reported by both siff and dup are a pair of programs from different sources (cryptiX, developed by Wolfgang Platzer, and part of JavaFaces, developed by John Thomas) to compute MD5. Siff found them to have 78% similarity (and 86% in the other direction), and dup found them to have a single common code segment of 1336 lines. The common code segment corresponded to 60 identical lines in the Java files. There was no similarity otherwise in the Java files, but siff found additional similarities in the bytecode files. The 60 identical lines are different from the corresponding lines of the MD5 RFC [29], but semantically equivalent. Interestingly, a search of the WWW turned up a third Java program with the same 60 lines. Possibly two of these programs borrowed from the third, or all three copied a description of MD5 other than the RFC.

The different-collection (unordered) pair reported only by dup looks at first glance as if it surely must come from a common source, because dup reported a common code section of 1521 lines, representing 85% of one file and 28% of the other. We didn't have

program	range of source similarity percent	number of trials	average source similarity percent	average bytecode similarity percent	max difference percent
Program 1	90-100	59	93.2	88.7	9
	80-89	76	84.4	78.7	9
	70-79	35	75.9	71.9	7
Program 2	90-100	17	92.9	91	9
	80-89	25	83.4	80.2	8
	70-79	6	76.2	74.2	5

Table 1: Similarity found by siff for corresponding Java files and bytecode files

	siff only	both	dup only
ordered pairs	552	82	10
ordered pairs, different collections	25	18	2
unordered pairs, different collections	23	9	1

Table 2: Similarities reported by siff and dup for 2056 bytecode files

the java source to compare. Upon inspection of the bytecode files, however, the common code turned out to be the initialization of an array of size 256. Since the stored values (retrievable from the constant pool of the bytecode files) appeared unrelated, the similarity is probably coincidental, merely an artifact of how the compiler generates code for a series of 256 array assignments.

For the pairs reported only by siff, the Java source was not available. However, almost all result from comparing a very small file (500 bytes or less in most cases) to a large file; the small file was found to be similar to the large one, but not the other way around. In addition, the names of these pairs indicate that the purposes of the files are unrelated. We conclude that these are false positives. The exceptions were matches whose names included H or V (apparently for horizontal and vertical); the H/H and V/V pairs were reported by both siff and dup, but the H/V and V/H pairs only by siff. The names of another two pairs of matching files related to buttons and checkboxes, but the remaining pairs of files appear to be totally unrelated based on the subject matter indicated by the names.

To summarize, we found many similarities and very few false positives by combining the information from dup and siff. The different-collection pairs reported by both dup and siff all appear to be valid instances of similarity. The one reported only by dup was not; some of the ones reported only by siff were not, especially for small files. Overall, the number of false positives was very small: at most 18 out of more than 2 million pairs.

4.3 Experiment 3: False negatives

The first two experiments indicate that our tools can effectively discover similarities while minimizing false positives. But the question of false negatives – that is, how many similar pairs we missed – remains unresolved. There were none in the first experiment, but it was too limited.

Since there are no other tools to compare to, there is no guaranteed way for us to measure false negatives. Nevertheless, we believe that the following "blind" experiment gives a reasonable indication.

We asked a friend who was familiar with the goals of our work, but not with the techniques we use, to randomly pick 10 programs from a set of 765 java programs (a subset of the 2056 programs for which we had the source), make random changes to them, compile them (possibly under a different version of the compiler) and give the set of bytecode files in a random order. We then ran siff and dup to see how many of his changes we can detect.

To make the test even more blind, our tester actually made changes to 12 programs, and added to our original test some additional programs (as well as removed some programs). siff running with a threshold of 65% similarity discovered 9 of the 12. Dup, with a threshold of 100 common lines, discovered 8. Together, they discovered 10. All 12 were found by siff when run with a 25% similarity threshold and by dup when run with a 50-line threshold. (On the other hand, decreasing the threshold to 25% increases the number of ordered pairs siff finds among the 2056 original files from 634 to 1430. Still reasonably small compared to the total of over 4 million possible pairs, but clearly the number of false positives grows when the threshold is decreased.)

Out of the two programs that were missed by both siff and dup, one is particularly interesting. It involved relatively few changes to the source, but they made the bytecode file very different. On close inspection, we found that the main culprit was a move of a segment of code, which resulted in a bytecode file with many jumps around the relocated code. For siff, the offsets of all these jumps were affected by the relocation, resulting in different fingerprints. For dup, the relocated code broke up long matches in both places, which mattered since the bytecode files were small - only four times the threshold length.

To validate the usefulness of the offset encoding for diff, we ran diff on the disassembled code with and without the offset encoding. The results are shown in Table 3. The pairs varied in file size and how many changes were made. To get a relatively size-invariant measure, we use the length of the diff output (in lines) divided by the sum of the file sizes, for each type of file. (Pair 6 is special: different versions of the JDK compiler were used to generate two class files from the same java file. Consequently, no value is given for this measure for the java file.) Our second measure is the average length of blocks of identical lines reported by diff. The last column contains the sum of the sizes of the offset-encoded disassembled files, which is the same as the sum of the file sizes without the offset-encoding.

For most of the pairs, the data in Table 3 show a significant improvement from using offset encodings for diff. In a few instances, the values are about the same with and without offsets. Pair 11 is the only instance where diff found significantly less similarity with offset encodings, but just for the second measure. Note that pairs 3, 10, and 12 were the ones not discovered by siff with a 50% threshold, as discussed above, and pairs 9-12 were the ones not discovered by dup with a 100-line threshold.

To validate our approach of using offset encodings for siff, we also ran siff on the disassembled code without applying the offset encoding. With a 25% threshold of similarity, siff discovered only three of the 12 pairs (4, 5, and 10), a much worse performance than that described above for the offset encoding.

4.4 Running times and scalability

For the 2056 files, siff used 41 seconds of elapsed time and 4 seconds of user time while dup used 1 minute 55 seconds of elapsed time and 1 minute 7 seconds of user time (running under IRIX 5.3 and using one of 12 150 MHZ IP19 Processors, with data cache size 16 Kbytes, instruction cache size 16 Kbytes, secondary unified instruction/data cache size 1 Mbyte, and main memory size 1280 Mbytes). Encoding the 2056 bytecode files took 7 minutes of elapsed time for siff and 8 minutes (independently) for dup. (Most of the preprocessing could be shared.) Siff also enables the creation of an index so that new files can be compared with an index of files processed earlier. Comparing one 50K bytecode to all the 2056 files in the index (within 50% similarity) takes a couple of seconds for encoding the new file and thereafter the processing by siff is essentially instantaneous (0.2 user time and between 0:00 and 0:01 elapsed time). The size of the index depends on the amount of sampling done and the precision of results; for the experiments used here the index occupied about 5% of the total size of all files.

Dup is much greedier in space than siff, and consequently will not scale to processing huge numbers of bytecode files at one time. If the goal were to process an enormous number of bytecode files, say to provide a registry service for the World Wide Web as has been proposed for text files [9], then siff should be used to screen for similarities that should be checked subsequently by dup.

5 Related Work

Java bytecode files are relatively easy to decompile into Java. Programs can be rewritten in structured form by analyzing the flow of control, and the names of classes, fields, and methods are available from the class file. In fact, at least four bytecode decompilers have been implemented: Mocha (by Hanpeter Van

pair	diff-size/total-lines as %			ave. ident. block size in lines		sum of file sizes in lines
	java	offset	no-offset	offset	no-offset	offset = no-offset
1	13	6	57	185	3	2740
2	14	17	68	18	2	1040
3	30	24	61	8	2	2098
4	14	14	21	173	20	1603
5	59	40	42	10	10	988
6	-	2	34	198	5	6041
7	19	13	32	22	6	8789
8	23	24	59	36	12	941
9	14	8	45	61	4	265
10	24	32	31	7	8	662
11	50	55	56	7	24	428
12	18	41	67	6	2	1037

Table 3: Results of running diff with and without the offset encoding

Vliet, now deceased, and according to [31], taken over as part of Borland's JBuilder [8]), WingDis [33], DejaVu [18], and Krakatoa [28]. A discussion of these appears in [14]. These produce quite readable code except for variable names, and even those may be available (perhaps inadvertently) from the class file if the compiler generated optional information aimed at debuggers.

Because class files are relatively easy to decompile readably, Java applications are potentially vulnerable to theft, even if they are distributed only as bytecode files. An unscrupulous person could use one of the decompilers mentioned above to decompile the bytecode files of an application, modify the resulting source, recompile into bytecode files, and distribute the bytecode files.

In fact, a number of "obfuscators" have been written to make decompilation less successful. These include Crema (by Van Vliet, and like Mocha, also reportedly taken over by Borland's JBuilder[8]), Hose-Mocha, by Mark LaDue, HashJava (now renamed SourceGuard [1]), by K.B. Sriram, and Jobe, by Eron Jokipii; for discussion of these, see [23, 32]. Early obfuscators either changed symbol names or added no-ops to bytecode files in such a way that particular decompilers crashed. New obfuscators (e.g., [13, 12]) employ much more sophisticated techniques, some of cryptographic strength, to change the appearance of code. These techniques not only make it harder to decompile, but they also make it possible to change the appearance of bytecode files that come from the same source. Our methods will not be able to defeat such techniques, although it may be possible to detect that they are used.

Another technique for enabling detection of stolen code is steganography, the art of hiding information such that the information can later be detected [2]. For object code, insertions of extra no-op sequences of instructions would slow it down, but Intel reports that it has successfully replaced code fragments by equivalent ones to customize security code [3]. However, this is not common practice.

For Java bytecode files, the single Java Virtual Machine means that the variety of platforms is not an issue. Multiplicity of compilers can be dealt with by having the owner of the code compile the source files with each of the available Java compilers and compare the resulting bytecode files with the bytecode files that were suspected to be stolen. Currently, the number of different compilers is not large, so this is manageable. Code compiled by different releases of the same compiler will probably still be very similar, although not necessarily identical. In our experiments we found that, given the same source files, jdk 1.0.2 and jdk 1.1.x generated class files that rarely differed in any significant way.

6 Conclusions and Future Work

Our experiments have validated our approach by showing that our tools are able to deduce similarity in Java source from similarity in the bytecode files. We can make the rate of false positives very low while keeping false negatives reasonably minimal. Our positional encoding proves to be a very powerful technique. Since the three tools siff, dup, and diff are based on different techniques, one may detect similarity when another one misses it. Local modifications within sections of code will reduce the percentage of similarity found by siff, but dup will still find long matches in the unchanged sections. Name changes of variables would not be a problem since the compiler turns variable references into table indices, which we have shown our methods to handle despite change in absolute values. Reordering major sections of code, e.g. by changing the order of methods, will have little effect on siff and dup, though it will greatly affect diff.

A major advantage of looking at Java class files rather than at binaries for other programming languages is that there is just one version of the Java Virtual Machine for all platforms (or at least that is Sun's intention), while binaries of other languages are different for different platforms. Extending our techniques to binaries is a natural step to take, although binaries present several additional problems: They are strongly tied to the architecture, there are many compiler and even more optimization and code restructuring programs, and the information in binaries as not as well organized as in bytecode files. Nevertheless, we believe that it will be possible, at least in some degree, to identify similar binaries.

It would be helpful to have a graphical user interface that links the output of siff and dup to a decompiler, allowing a developer to see the evolution of two similar programs clearly.

7 Acknowledgements

Peter Bigot was our blind tester, and we thank him for a thankless job well done.

8 Availability

Contact Brenda Baker, bsb@bell-labs.com, for information about licensing the software from Lucent Technologies.

References

[1] 4thpass Software Corporation. Protect your Java investment. `http://www.4thpass.com/`, Apr. 7, 1998.

[2] Ross Anderson and Fabien Petitcolas. On the limits of steganography. In *IEEE J-SAC*, to appear.

[3] D. Aucsmith. Tamper resistant software: An implementation. In *Information Hiding*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333. Springer-Verlag, 1996.

[4] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering*, pages 86–95, 1995.

[5] Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, Feb. 1996.

[6] Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Computing*, 26(5):1343–1362, Oct. 1997.

[7] H.L. Berghel and D.L. Sallach. Measurements of program similarity in identical task environments. *SIGPLAN Notices*, 9(8):65–76, August 1984.

[8] Borland. Jbuilder. `http://www.borland.com/jbuilder/`, Oct. 23, 1997.

[9] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD)*, 1995.

[10] Andrei Broder, Steve Glassman, Mark Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Proceedings of the Sixth International World Wide Web Conference*, pages 391–404, April 1997.

[11] Kenneth Ward Church and Jonathan Isaac Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, 2(2):153–174, June 1993.

[12] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *IEEE International Conference on Computer Languages 1998*, Chicago, IL, May 1998.

[13] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, Jan. 1998.

[14] Dave Dyer. Java decompilers compared. *JavaWorld*, July 16, 1997. `http://www.javaworld.com/javaworld/jw-07-1997/jw-07-decompilers.html`.

[15] Luis Fernandes. Are there any apps that can display files in parallel, highlighting (in color) the differences between them? `http://www.ee.ryerson.ca:8080/~elf/xapps/Q-XI.html`, April 3, 1997.

[16] Nevin Heintze. Scalable document fingerprinting. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, Nov. 18-21, 1996.

[17] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, June 1990.

[18] Innovative Software. OEW for Java. `http://www.isg.de/OEW/Java/`, Oct. 2, 1997.

[19] H.T. Jankowitz. Detecting plagiarism in student PASCAL programs. *Computer Journal*, 31(1):1–8, 1988.

[20] J. Howard Johnson. Substring matching for clone detection and change tracking. In *Proc. International Conf. on Software Maintenance*, pages 120–126, 1994.

[21] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[22] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1997.

[23] Qusay H. Mahmoud. Java tip 22: Protect your bytecodes from reverse engineering/decompilation. *JavaWorld*, Jan. 2, 1997. `http://www.javaworld.com/javatips/jw-javatip22i.html`.

[24] Udi Manber. Finding similar files in a large file system. In *Proc. 1994 Winter Usenix Technical Conference*, pages 1–10, Jan 1994.

[25] T.J. McCabe. Reverse engineering, reusability, redundancy: the connection. *American Programmer*, 3(10):8–13, Oct. 1990.

[26] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[27] PocketSoft. .RTPatch Professional, Feb. 23, 1998. `http://www.pocketsoft.com/products.html`.

[28] Todd Proebsting and Scott A. Watterson. Krakatoa: Decompilation in java (does bytecode reveal source?). In *USENIX Conference on Object-oriented Technologies and Systems*, June 1997.

[29] Ronald Rivest. The MD5 message digest algorithm. RFC 1321, Apr. 1992. `http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1321.txt`.

[30] Narayanan Shivakumar and Hector Garcia-Molina. Building a scalable and accurate copy detection mechanism. In *Proceedings of 1st ACM International Conference on Digital Libraries (DL'96)*, March 1996.

[31] Eric Smith. Mocha, the Java decompiler. `http://www.brouhaha.com/~eric/computers/mocha.html`, Dec. 28, 1997.

[32] Thomas Traber. Tools for working with bytecode: Bytecode assemblers, disassemblers and dumps. `http://www.oasis.leo.org/java/development/bytecode/00-index.html`. Part of Java Oasis, `http://www.oasis.leo.org/java/00-oasis.html`.

[33] Wingsoft. Wingsoft Products Information. `http://www.wingsoft.com/products.shtml`.

[34] Tak Yan and Hector Garcia-Molina. Duplicate removal in information dissemination. Technical report, Stanford Computer Science Dept., 1995. submitted for publication.