Session 3232

Gene Sequence Inspired Design Plagiarism Screening

Mark C. Johnson, Curtis Watson, Shawn Davidson, Douglas Eschbach

Purdue University / University of Illinois / Hewlett-Packard / Qualcomm

Abstract

Plagiarism of digital system designs has become increasingly convenient with the emergence of language-based design techniques. Detection and proof of plagiarism are similarly facilitated. This has long been an issue in computer programming courses and non-technical courses that rely heavily on text based assignments. However, until recently, digital design instruction was based on graphical design methods that did not adapt well to electronic cut-and-paste or web searches. Tools are needed to encourage and verify the originality of digital designs. Such tools exist for many programming languages and for essay text, but not for hardware description language (HDL) based digital design. In this paper, we present an implementation of HDL plagiarism checking that is similar to what is used to evaluate the similarity and ancestry of gene sequences. This form of plagiarism screening has been used for one semester in a digital integrated circuit design course. Other less effective and efficient methods were in use for two years. Results show a strong sensitivity to commonality between closely related source code files, even in the presence of a variety of obfuscation techniques.

Introduction

Plagiarism on the part of the few has long been a concern in most academic and professional disciplines. Copyright laws, patent laws, academic honor codes, and professional ethics codes all give evidence of the historic need to protect intellectual property (IP). In the public or commercial arena, the victim of IP theft usually has the burden of detecting, proving, and suing or pressing charges against the violator. In the classroom or instructional laboratory, the victims of IP theft (students) are not generally in a position to detect, prove, or prosecute the perpetrator. Academic honesty codes or honesty contracts encourage most students to fulfill their ethical obligations, but the codes do not guarantee complete compliance, nor do they provide a means of detection or proof. The course instructor and teaching assistants are in the best position to detect plagiarism since they all usually evaluate or at least have access to the complete set of student submissions. However, the logistics of checking for plagiarism can be prohibitive. Consider a class of fifty students. An exhaustive pair wise comparison of all student submissions for a single

assignment requires 1250 comparisons. Efficient automated pre-screening techniques, if available, can reduce this task down to human inspection of a small number of the most suspicious student submissions.

This paper focuses on techniques for detecting plagiarism in student digital designs, although the techniques could be easily adapted to most computer programming languages. In approximately the last ten years, digital design has shifted from schematic based design entry to the use of hardware description languages (HDLs) such as VHDL (Very high speed integrated circuit Hardware Description Language) or Verilog ™. The HDL approach makes it possible for students to create much more complex designs than before, but it also facilitates copying or transcribing design data from other sources. Computer programming instructors faced this problem decades earlier. Consequently, there are well established tools for plagiarism checking of computer code[1,2], but we have not been able to find any evidence of plagiarism checkers targeted to HDLs.

In the remainder of this paper, we will present an overview of textual plagiarism checking techniques, identify requirements peculiar to HDL code checking, describe our implementation of VHDL plagiarism checking, and present results for numerous test cases including artificial test cases, actual student submitted source code, and a mixture of both.

Survey of textual plagiarism checking

To see where the proposed techniques fit into the universe of existing plagiarism checkers, consider the following dimensions: 1. the type of documents to be compared, 2. the population and origin of source documents to be compared, and 3. the method of source code analysis.

Type of documents

Computer languages (including HDLs), written natural languages, and gene sequences all lend themselves to representation as a linear sequence of symbols. Consequently, many techniques to quantify the set of repeated character sequences between two documents may be broadly applicable. Bennett et al illustrated this[3] by using a well-known gene sequencing technique (using data compression techniques) to analyze the genealogy of chain letters. One of the basic principles of data compression algorithms is to eliminate redundancy. The file compression ratio becomes a measure of the redundancy within the file. One can then concatenate two files and compress the result to get a measure of the redundancy (i.e., similarity) between the two files.

Population and origin of source documents

Plagiarism checking is most easily done on a pair wise basis, but to be most useful in the classroom, one must be able to check an entire collection of student submissions, possibly even

including submissions from prior semesters. One well-known and widely used system is MOSS (Measure of Software Similarity) created by Alex Aiken[1]. MOSS is provided as an online service to which computer programming instructors can submit an entire group of student submissions for plagiarism checking. A report is returned to the instructor identifying the most suspicious sets of matches. MOSS is based on document fingerprinting techniques that should be applicable to any kind of text document, but at present MOSS does not support plagiarism checking for any HDLs. Another publicly accessible source code plagiarism checker with similar capabilities is Jplag[2].

A thorough check should also consider external sources of data such as textbooks or solutions accessible on the Internet. Tools have been developed and commercialized to search the Internet for textual sources matching a student submitted report. Several online services including MyDropBox.com, TurnItIn.com, CitationOnline.net, and Plagiarism.org are currently available, but all are focused on checking of narrative text rather than computer generated code.

Method of source code analysis

Computer languages including HDLs require a well-defined syntax and semantic structure in order to be useful. Using the syntax and semantic rules of a particular language, one can translate textual source code into a graph theoretical representation. This is how most high level language compilers work; they generate a graph on which can be optimized and ultimately re-expressed in terms of the machine language of the target computer. This might lead one to expect sub-graph matching algorithms to be useful in plagiarism checking. However, source code checking tools in the literature appear to rely on two general approaches: statistical analysis of program attributes such as the frequency of occurrence of particular operands, or substring matching on a tokenized representation of source code. A survey of plagiarism checking systems based on these two approaches was documented by Culwin et al[4].

A purely semantic approach to source code comparison will overlook many details that may point to plagiarism. Variable (or signal) names, indentation, and comments are all examples of information that is lost in the translation to a purely semantic representation. Identical variable names, and comments can be very strong indicators of plagiarism. On the other hand, a purely textual comparison may fail to recognize similar program structure in the presence of obfuscation techniques such as changes in variable names. A thorough source code comparison needs to consider both textual details and program structure. MOSS and JPlag strike a balance between literal comparison and structural comparison by performing a language dependent tokenization followed by either textual fingerprinting (MOSS) or fast substring matching (JPlag). Some literal information is lost, but it should protect against most obfuscation techniques.

In terms of the three dimensions described previously, our plagiarism technique is positioned as follows:

1. Type of documents: Our only intended target documents are VHDL source code, but the approach should be adaptable to any structured programming language or HDL. In addition, a subset of our approach could be applied to narrative text.

2. Population of documents: Document comparisons are done on a pair wise basis, but scripts are used to perform checks and aggregate the results for all student submissions of a particular assignment (50 to 100 students per semester; multiple semesters may be included). External sources of code such as texts or web sites are not included. However, external sources may be detected indirectly if more than one student refers to the same external source.

3. Method of source code analysis: Over several semesters, we have tried a variety of techniques including statistical analysis of code attributes and an exhaustive sub-string comparison of student submissions. However, our current and most successful approach is an application of file compression measurements (inspired by the gene-sequencer approach) to two versions of VHDL source code: a partially tokenized version and a literal version. One advantage of the file compression approach is that one can take advantage of existing fast compression programs such as gzip[5].

Requirements

Many of the requirements for plagiarism checking of HDL designs are the same as one would expect for hand-typed sequential programming languages, but there are two important differences. Synthesizeable HDL code is mostly declarative rather than sequential. By "synthesizeable", we mean that the code can be cleanly and automatically translated into a digital circuit netlist. Non-synthesizeable HDL code can be highly sequential, but in digital hardware design, our primary interest is in synthesizeable code. As a result, many HDL code statements or blocks of code can be reordered to obfuscate evidence of plagiarism without any effect on functionality. Synthesizeable HDL code can include blocks of sequential (order dependent) code, but the order of appearance of those blocks does not matter. The other important difference is that significant portions of HDL code may be automatically generated from graphical representations of a design, such as from block diagrams or state diagrams. Not surprisingly, automatically generated blocks of code from different students tend to have a higher degree of similarity than manually generated code.

Plagiarism of code can take many forms, but all of the cases we have encountered over several semesters fit the following scenarios:

- Some students willingly share source code as a way to reduce the effort required of each individual. In this case, each student's submission usually includes some code that is unique to that student. Often this is an act of desperation as an assignment deadline approaches. Usually the plagiarized portion of code is modified only slightly with some variable names changed or comments deleted. However, there is a tendency

for students to be careless in this situation, leaving unchanged much of the formatting, naming, and commenting.

- Some students work side by side, discussing nearly every step of the design. The two sets of resulting code end can up being nearly identical in structure, but the naming and formatting are different.

- Code is occasionally stolen as a result of carelessness: listings spooled to shared printers, unattended workstations left unlocked, and the like. As with shared code, there is usually an incomplete attempt at concealment, but in this case the student tends not to try to write any code of his or her own.

- Code is reused from prior semesters. As a matter of practice, we modify specifications for major design assignments from one semester to the next. Consequently, prior code cannot be used as-is, but substantial portions may be reusable. Artifacts of the prior semester's design tend to be retained and may be noticed by experienced teaching assistants.

- One student, not taking the course in question, does work for the student who turns in the work. The evidence in this situation does not lend itself to automated comparison of all student submissions. A style analysis of each student's submissions over the course of the semester might reveal suspicious activity, but we have not pursued this. The best clues are found in the behavior or lack of design knowledge on the part of the perpetrator.

- Plagiarism did not occur. Some small design assignments or sub-blocks within a larger design are so tightly constrained that numerous students independently arrive at virtually the same design. In addition, automatically generated code or commonly used code sequence may elevate the apparent degree of code similarity.

These scenarios lead to the following requirements for plagiarism screening:

- Compare both literal and structural features of student submissions.

- Maximize signal to noise, i.e., minimize features common to all students as well as code features that only serve to obscure the actual structure of the design. The features to be minimized differ for literal vs. structural comparisons.

- Use human observation. Some types of plagiarism are more easily observed by the instructor than by a program. In addition, instructor investigation is required to filter out false positives before taking any disciplinary action.

- Apply automated stylistic analysis methods to detect changes in the authorship of submissions from each student.

One obvious scenario has been ignored in these requirements: the use of external sources. However, our specifications for major design assignments are internally developed and modified each semester, thus reducing the usefulness of external code. This scenario is further mitigated by the possibility that more than one student may use the same external source; the plagiarism then becomes detectable by cross-comparison of student submissions.

Implementation

Our implementation of VHDL plagiarism screening addresses all of the requirements outlined in the last section except for stylistic checks and examination of external sources. The system consists primarily of a set of scripts (a mixture of perl and python) to perform the following tasks:

1. For each student's design, concatenate all files pertaining to that design.
2. For purposes of structural code comparison, filter out superfluous information (comments and extra white space) and create a partially tokenized version of the code. Keep a copy of the unfiltered code.
3. For each possible pair of students, concatenate the unfiltered source code into one file. Compress that file using gzip and compare the compressed file size to compressed versions of the individual student submissions. Compute the similarity metrics (defined below).
4. Repeat step 3 using the filtered version of the code. This will provide metrics for the similarity of code structure.
5. Accumulate similarity statistics, including a histogram and rankings of student pairs for each similarity metric.

Based on the rankings and histograms, the instructor can quickly identify suspicious cases to investigate further. We have created a character level cross-correlation program, called examiner, to provide a detailed listing of blocks of similar code. Based on this information and manual inspection of student code, the instructor is equipped to make decisions on where plagiarism has occurred, confront the students, and assess penalties.

Tokenization of VHDL code

In order to compare the structure of two VHDL source code files, one must eliminate any information in the files that won't affect the semantics of the code. This is accomplished by tokenization. Tokenization refers to the process of identifying syntactically significant elements and representing those elements in a form that is convenient for parsing. The parser in a compiler or HDL synthesis program exams the sequence of tokens to determine the semantic content of

the program. In the case of plagiarism screening, we compare token lists to identify similarities in the structure of two sets of source code.

In our case, the most convenient form for the token list is another text file where each token is represented by a short unique string. In that way, any program that compares (or compresses) text files can be used to compare the token list. The tokenized file is generated as follows:

1. All new lines, extra white space, and comments are eliminated.
2. Language keywords are replaced with short unique character sequences.
3. User defined names such as variable names are replaced with short generic character sequences.

Note: for screening purposes, it is not critical that all syntactic or semantic information is preserved. We take advantage of this by ignoring the uniqueness of user defined labels such as variable names in order to ensure that corresponding variables in two student submissions are assigned to the same token value. Consider the following VHDL statements appearing in two different student's files:

```
ack <= addressCorrect AND dataReceived;  -- student 1

acknowledge <= addressMatch AND datReceived; -- student 2
```

The signals in the previous statements may be functionally equivalent, but a simple string comparison will not recognize them as equivalent if uniqueness is preserved as shown below:

```
S1 <= S2 AND S3

S2 <= S3 AND S4
```

Instead, such statements are treated in the manner of:

```
SIG <= SIG AND SIG
```

Some information is lost this way, but the structure of each statement is preserved.

File similarity metrics

Our primary mechanism for automatically evaluating the similarity of two files is described by the following pseudocode:

```
file1 =     the first of the pair of files to be compared
file2 =     the second of the pair of files to be compared
file12 =    concatenation of file1 and file2
```

file1.gz =	gzip compression of file1
file2.gz =	gzip compression of file2
file12.gz =	gzip compression of file1cat2
size1 =	size of file1.gz
size2 =	size of file2.gz
size12 =	size of file12.gz
metricA =	(size1 + size2)/size12 - 1
metricB =	1 - (size12 - max(size1,size2))/min(size1,size2)

However, a bit more explanation is required to show why this works and to justify the choice of metrics. First, some understanding of gzip file compression is helpful. Numerous data compression programs (such as gzip and compress) have been developed based on the Lempel-Ziv and Lempel-Ziv-Welch[5] algorithms (LZ and LZW). LZ and LZW based algorithms build a dictionary of recurring substrings appearing in a file or document to be compressed. Instances of these substrings in the body of the document are replaced with a reference to the dictionary entry. Now, consider what happens in the extreme case where file1 and file2 are identical. When file12 is compressed to produce file12.gz, the size of file12.gz is only slightly larger than file1.gz (or file2.gz). The reason is that the second half of file12 (file2) can be represented entirely in terms of substrings from the first half of the file. As differences to file2 are introduced, the ability to represent file2 in terms of strings from file1 is reduced, leading to a larger file size for file12.gz.

One can judge the similarity of file1 and file2 by looking at the sizes of file1.gz, file2.gz, and file12.gz. The smaller file12.gz is relative to file1.gz and file2.gz , the more similar the files are. However, some method is needed to normalize the file size comparison so that we can rank all pairings of students in terms of the similarity of their submitted source code. Metrics A and B (metricA and metricB) were created for this purpose. Both metrics are designed to give a value of 0 if file1 and file2 have nothing in common, and a value of 1 if file and file2 are identical.

The first metric we considered, metricA, works as follows. If file1 and file2 have nothing in common (or if one of the files is empty), then the concatenation of file1 and file2 will not allow any additional compression beyond what is achieved by compressing the files individually. Hence, size12 will approximately equal size1 + size2, and metricA will be close to zero. It is not possible to speak in exact terms for at least two reasons: 1. there is some overhead associated with file compression, 2. VHDL syntax itself requires some commonalities between any pair of VHDL source code files. Nevertheless, let us consider the case of identical source code files. As discussed earlier, the sizes of file12.gz, file1.gz, and file2.gz will all be approximately equal. Hence, metricA evaluates to approximately 1.

The first metric has at least one shortcoming. It only produces meaningful results for files that are nearly the same size (or more accurately, that compress to nearly the same size). The value of metricA decreases in proportion to smaller file size divided by the larger file size, without regard to the redundancy between the files. A small file may exactly correspond to a substring of the larger file, but a low similarity metric will still be computed. Our solution to this problem is to

compute a metric that indicates how much of the smaller file is covered by strings in the larger file. This is accomplished by metricB. Let file2 be the smaller file. Then, if file2 is completely covered by file1, size12 should be only slightly larger than size1, resulting in a value of approximately 1 for metricB. If file2 has nothing in common with file1, size12 should be approximately equal to the sum of size1 and size2. This results in a value for metricB of approximately 0.

It should be noted that metricA and metricB are identical when evaluated for pairs of files that have the same compressed size (size1 = size2). Taking size1 = size2 and applying some algebraic manipulations, one can start with the formula for metric B and obtain the formula for metricA.

File cross-correlation report

Once the similarity analysis for a collection of source code files is complete, an instructor must then inspect the most suspicious pairs of source code to determine where plagiarism has occurred. Not all cases of file similarity constitute plagiarism. Small highly constrained design problems may tend to produce clusters of similar designs in the absence of student misconduct. Students may use example code from course notes or textbooks. There are numerous plausible situations that can result in similar code. Students are adept at identifying such situations. Manual examination of even a small set of suspected plagiarism cases can be tedious, so a program (examiner) was created to provide a detailed report of file similarities.

Examiner performs a brute-force character by character cross-correlation of two text files. To understand how it works, think of each file as one string of characters. Place the two strings side by side, initially with the first characters aligned to each other as illustrated below:

```
this is the first file to be compared
this is the second file to be compared
```

Scan through the characters of the first file while watching for matching characters below. In this case, you will find “this is the” as a matching string. Repeat this process for all possible alignments of the two strings. For example, when file1 is offset one character relative to file2, you will detect the matching string “file to be compared”.

```
 this is the first file to be compared
this is the second file to be compared
```

As this process progresses, examiner keeps track of what portions of each file are already covered by matching strings. Shorter matches may be replaced by longer matches, but no two matching strings are permitted to overlap. A user specified threshold specifies the minimum string length reported by examiner.

When the process finishes, a variety of reports can be generated. Probably most useful is a listing of the longest matching strings in the file. Other reports include a listing of differences and the percentage of file1 matched by file2.

Alternative implementations

The checking system described in this paper is not the only implementation considered or tested. The first system based on an attribute counting approach. When VHDL code is synthesized, the result is a netlist, a list of logic components and the wires (nets) connecting them. If two pieces of source code are functionally equivalent, one might expect them to produce the same or similar netlist. Consequently, the number of logic gates (nand, nor, etc) of each type should be the same between the two netlists. Unfortunately, the designs (and the synthesis constraints) have to be identical for this to be true. If only subsets of the designs match, one would have to perform maximal subgraph matching in order to identify the similarity. Worse, synthesis of VHDL code suffers from the butterfly effect, i.e., a slight change in the design or synthesis constraints can substantially change the resulting netlist. More cases of plagiarism were caught by observant TAs than by this system.

The second implementation was very similar to what is described in this paper, except that the examiner program, rather than file compression, was used to evaluate the degree of file similarity. This approach was effective but inefficient. In order to get run times down to a tolerable level (two or three days on a Sun enterprise 450) for 50 to 100 student submissions, the search window (range of possible relative file offsets) was constrained to as little as 100 characters. Pre-filtering of the files, similar to the tokenization described above, was used to reduce the size of the files to be compared. However, restricting the search window reduced the instances of plagiarism that could be detected.

Results

Initial testing was done using student source code samples from a simplified I^2C bus interface design project during spring 2003. I^2C is an industry standard synchronous serial bus interface used in a wide range of consumer electronic products. Transformations were applied to the source code samples in order to observe the behavior of the similarity metrics under a variety of conditions including plagiarism obfuscation techniques students might use.. Table 1 summarizes these initial test cases and selected results were presented in the interest of brevity.

Table 1. Initial test results for similarity metrics

Description	Original code Similarity metricA metricB [execution time]	Tokenized Similarity metricA metricB [execution time]	Comments
TEST01 Concatenation of the same file 1 to 16 times.			16x15/2 = 120 results were generated, a small subset were selected for presentation in the interest of brevity.
once. vs. 2 times	0.911 0.970	0.910 0.971	
once. vs. 8 times	0.811 0.978	0.826 0.980	
once vs. 14 times	0.752 0.980	0.771 0.974	
8 times vs. 14 times	0.718 0.870	0.749 0.883	
13 times vs. 14 times	0.692 0.824	0.726 0.844	
	[19 sec]	[18 sec]	
TEST02 For each of 8 student samples, a second version was created by re-ordering the code in non-destructive ways.			again 120 results were generated notice the large difference between the metrics for comparison of re-ordered files vs. metrics for source files from different students
# 1 vs. # 1 reordered	0.894 0.873	0.873 0.940	
# 1 vs. # 4	0.216 0.486	0.376 0.569	
# 4 vs. # 4 reordered	0.841 0.918	0.792 0.894	
# 4 vs. # 8	0.222 0.393	0.279 0.473	
	[5 min 39 sec]	[49 sec]	
TEST03 Each test case is composed of various concatenations of two different modules within the I^2C design			7 sample files were generated by various concatenations of t1 and t2, resulting in 21 comparisons. One would expect the similarity ratings to be very high for all but the t1 vs. t2 test case. Further investigation is needed.
t1 vs. t2	0.195 0.523	0.189 0.432	
t1 vs. t1+ t2	0.360 0.971	0.415 0.960	
t1 vs. t2+ t1	0.190 0.526	0.188 0.443	
t1 vs. t2+ t2+ t1	0.190 0.526	0.188 0.443	
t1+ t2 vs. t2+ t1	0.751 0.929	0.695 0.929	
t1+t1+t2 vs. t2+t2+t1	0.751 0.929	0.696 0.929	
	[4 min 14 sec]	[23 sec]	

Description	Original code Similarity metricA metricB [execution time]	Tokenized Similarity metricA metricB [execution time]	Comments
TEST04 Various signal renaming and re-ordering of signal assignments were applied to a sample code (the control unit for a simplified I^2C slave device).			6 sample files were generated and 15 comparisons were performed. Notice that in all cases for the original source code version, metricB consistently detected strong file similarity in the presence of signal name changes for original source code.
original vs. re-ordered signal names and assignments	0.859 0.925	0.774 0.874	
original vs. some signal names changed	0.817 0.901	0.723 0.843	
original vs. some other signal names changed	0.801 0.891	0.632 0.781	
original vs. all signal names changed	0.762 0.868	0.580 0.745	
original vs. all signal names changed again	0.682 0.812	0.511 0.681	
	[1 min 4 sec]	[7 sec]	
TEST05 A collection of 10 mostly unrelated perl, python, and shell scripts.			What if non-VHDL code crept in? 45 comparisons were run. The first two comparisons reported here had the highest similarity ratings. The third reported had the lowest ratings.
two very similar python scripts	0.781 0.885	0.763 0.870	
93 line perl script vs. 134 line perl script by the same person	0.222 0.510	0.210 0.528	
27 line shell script vs. 654 line python script	0.11 0.194	0.14 0.217	
	[6 min 27 sec]	[4 min 9 sec]	

Description	Original code Similarity metricA metricB [execution time]	Tokenized Similarity metricA metricB [execution time]	Comments
TEST06 6 test files were artificially plagiarized versions of the same original code (same files as TEST04). 14 additional files consisted of other students' designs for the same design specifications.	pool of 6 plagiarized files maximum 0.859 0.925 minimum 0.613 0.762 remaining non-plagiarized files maximum 0.352 0.628 minimum 0.134 0.275 [8 min 2 sec]	pool of 6 plagiarized files maximum 0.774 0.875 minimum 0.409 0.622 remaining non-plagiarized files maximum 0.491 0.778 minimum 0.230 0.389 [1 min 42 sec]	A total of 20 files were used, resulting in 190 comparisons. The purpose of this test was to see how well deliberately plagiarized files would stand out from among a pool of non-plagiarized designs.

We would consider a similarity metric to perform well if it distinguishes clearly between pairs of plagiarized files and non-plagiarized files. Based on that criterion, the original source code comparisons consistently performed better than the tokenized comparisons, even for cases TEST04 and TEST06 where tokenization should filter out many obfuscations. One possible contributor to the relatively poor tokenized performance is the loss of semantic information when signal names are all mapped to the same token. This could cause similarity metrics to increase for unrelated files. Another possible problem is that the tokenization itself compresses the files quite a bit. This may be reducing the amount of compression obtained by gzip. Since our similarity metrics depend on differences in level of file compression, tokenization may be reducing the differences that can be observed. In addition, since the files are much smaller to start with, the overhead and imperfections of file compression may become more significant. These possible problems are a subject for further investigation.

The relative merit of metricA and metricB is not entirely obvious. In general, metricA produced much better separation in the data for pairs of similar files vs. pairs of dissimilar files. However, test suites 1, 3, and 5 all demonstrate the strong effect of relative file size on the similarity ratings produced by metricA. This behavior causes metricA to substantially underestimate the similarity of differently sized files, and could result in files with common blocks of code to be overlooked. MetricB does not suffer from this problem.

The only test suite for which none of the metrics performed consistently well was TEST03. This suite tested similarity between files constructed of various orders of concatenation for two original blocks of code. The first metric, metricA, clearly suffers from the file size dependence

discussed earlier, both for the original and tokenized source code versions. However, certain concatenation orders caused unexpectedly low similarity measures. One would expect "t1 vs t1+ t2" and "t1 vs. t2+ t1" to produce nearly the same high similarity value. However, the second case produces a similarity rating very close to the comparison "t1 vs. t2". Fortunately, the case "t1+ t2 vs. "t2+ t1", which is more likely the kind of transformation that would occur in plagiarized student source, produces high similarity metrics. Since the students are all producing the same design, and since one of the similarity checks we do is for a concatenation of all source files in the design, it should not be possible to defeat the system by splitting the design hierarchy to achieve a "t1 vs. t1+ t2" effect.

The test cases were run on various machines ranging from a Sun Ultra-5 with 512M RAM to a Sun enterprise-450 with 4G RAM. No attempt was made to control the loading of the machines. Consequently, the execution times reported are included just to give some indication of the range of execution times that can be expected for a population of up to 20 source files to be compared. The source code used in the test suites ranged from 1.3kB to 28kB. Excluding TEST01, the largest sample source code file was 13.8kB.

Results for an actual student population

File compression based plagiarism screening was used for the first time in the course ECE495d ASIC Design Lab during the fall semester of 2003. The course syllabus and labaratory discussions were used to inform students of our expecations regarding originality of work and of the general means of verification. The system proved (unfortunately, in some respects) to be very effective at identifying suspicious student source code. The similarity metrics were not used directly as proof of plagiarism. Rather, teaching assistants examined the top ten most suspicious pairs of files based on each of the four metrics (metricA/original, metricB/original, metricA/tokenized, and metricB/tokenized). There was substantial overlap in the rankings for each metric, so that the total number of suspicious cases was not unreasonable to examine. In their examination, they looked for similarities that could not be reasonably explained by any means other than direct plagiarism or excessive collaboration. In particular, they focused on aspects of the source code that vary greatly from one student to another such as in-line comments and large complex blocks of code. In general, the teaching assistants found that as they moved down the similarity ranking lists, plagiarism became less and less evident. Pairs of files still considered to be suspicious were passed on to the instructor. A few marginally suspicious cases were eliminated after further inspection and discussion with the students. Some suspicious similarities turned out to be the result of using permitted code generation and code management tools (HDL Designer ™ from Mentor Graphics). In the remaining cases the students acknowledged either sharing source code files, collaborating extensively, or in one case, using code belonging to someone else that the students "found".

A variety of report generators and character mode graphing functions have been implemented including:

- showHistogram: displays two character mode histograms, one for the frequency of metricA values, and the other for metricB values. Depending on the input files selected, this will present either the original source or tokenized similarity rankings.
- showXYPlot: displays a character mode scatter graph for which each 'x' represents one or more instances of a particular metricA, metricB value pair landing within the corresponding bin on the graph.
- rankResults: creates a report that ranks each source code pair according to the average of the individual rankings for all four metrics. The rankings for each metric are listed
- examiner: (described earlier) compares two source code files and presents a variety of reports regarding file similarity. The most interesting report is a listing of the largest blocks of identical code in the original source code.

A sample XY plot is included in figure 1. The data presented correspond to the student source code submissions (original, not tokenized) for the control unit portion of a simplified I^2C slave controller design. The outliers are easy to observe. One can determine the identity of the outliers by looking at a sorted listing of results for each individual metric. A sample is included in table 2. In addition, one gets a sense of the distribution for the class even though the exact frequency of distribution is not presented, e.g., each 'x' may represent more than one data point. The corresponding histogram is not especially informative once you have examined the XY plot. The histogram merely serves to confirm that the bulk of the data points correspond to similarity metrics smaller than 0.5. Table 3 presents the first 10 entries in the report generated by rankResults. It is to be expected that the rankings differ significantly, especially between the tokenized and original source code metrics. This is because the tokenized similarity compares code structure whereas the original code similarity compares the literal text of the source code, including comments and indentation. However, the difference in rankings is not a significant as it might appear when one considers that there are 1326 data points (the number of possible pairings of 52 students).

The XY plot illustrates another characteristic of metricA and metricB. As discussed earlier, metricA and metricB are identical if the compressed file sizes are the same. When the file sizes are different, metricA will have a lower value than metricB. This is born out on the graph. In all cases, metricB values (Y axis) are greater than metricA. The obvious diagonal lower boundary corresponds to cases where the file sizes and consequently the metrics are nearly the same.

Execution time

The execution time for a population of students is primarily a function of two parameters, the size of the individual files to be compared and the number of students. File size only affects the file compression execution time. Assuming that student submissions can be bounded by a finite upper limit, the number of students is the critical factor. For a population of N files, N(N-1)/2 file comparisons are required. This causes the computational complexity to be $O(N^2)$ with respect to the number of students.

h: 40 w: 50, binX: 0.01283 binY: 0.01883,
X-Axis Metric: A, Y-Axis Metric: B

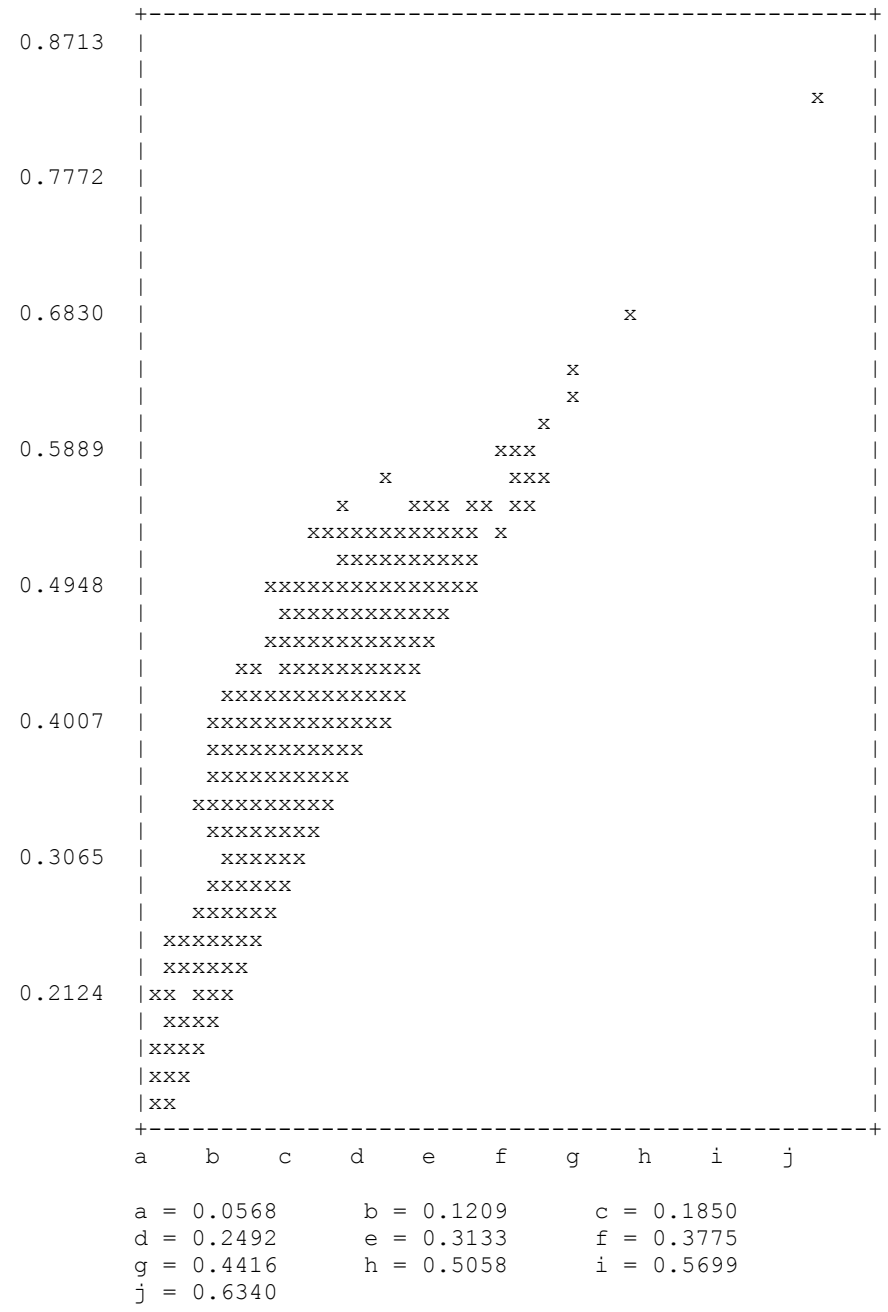

Figure 1. XY Plot of similarity metrics for original source code
Represents 52 student submissions on an I^2C slave unit controller design

Table 2. First 10 rankings of similarity for metricA and metricB
Based on 52 student submissions on an I^2C slave unit controller design.
Identification numbers have been remapped to protect student privacy.

Student pair	Original MetricA	Student pair	Original MetricB
pr14/pr13:	0.64818	pr14/pr13:	0.84013
pr01/pr48:	0.49167	pr01/pr48:	0.69470
pr27/pr47:	0.43306	pr27/pr47:	0.65940
pr48/pr57:	0.43036	pr48/pr57:	0.63667
pr04/pr35:	0.41044	pr04/pr35:	0.61271
pr01/pr57:	0.41013	pr35/pr56:	0.60129
pr31/pr49:	0.39803	pr56/pr55:	0.59742
pr02/pr49:	0.39525	pr40/pr49:	0.58996
pr04/pr20:	0.39363	pr00/pr49:	0.58612
pr56/pr55:	0.39271	pr20/pr35:	0.58554

Table 3. First 10 average rankings for all metrics
Based on 52 student submissions on an I^2C slave unit controller design.
Identification numbers have been remapped to protect student privacy.

Group Name	Avg. Rank	Tokenized A	Tokenized B	Original A	Original B
pr14/pr13	1.500	2	2	1	1
pr27/pr47	2.000	1	1	3	3
pr04/pr35	7.250	15	4	5	5
pr01/pr48	8.250	13	16	2	2
pr56/pr55	8.250	6	10	10	7
pr04/pr20	10.000	8	5	9	18
pr57/pr33	16.500	12	23	16	15
pr40/pr49	18.000	14	33	17	8
pr56/pr55	20.250	9	12	18	42
pr02/pr49	28.250	32	62	7	12

Execution times were presented in table 1 for the test suites used in initial testing. The results presented in Figure 1 for a population of 52 student submissions required 4 minutes and 35 seconds to perform 1326 original source code comparisons, and 3 minutes 5 seconds for 1326 tokenized code comparisons on a Sun enterprise-450 4 processor server with 4G RAM. There is no obvious square law relationship in the execution time data, but there are numerous factors affecting execution time that we didn't attempt to control including the selection of host machines, loading of the machines, and source file sizes.

Conclusions

In this paper, we have presented a source code plagiarism screening method that is efficient, effective, and can be implemented entirely with common scripting and file compression utilities. This is accomplished using file compression metrics, based on the same principle as what is used for screening of gene sequences. The method was tested for a variety of code obfuscation techniques. In the vast majority of cases, file similarity detection was not hampered by obfuscation techniques including signal (or variable) renaming or re-ordering of code. One case was found where block level reordering reduced the similarity measures to values typical of unrelated files. However, still other more complex re-orderings did not adversely affect the recognition of similar files. When the file compression method was used for the first time in the classroom, it proved to be very effective at identifying cases of plagiarism, i.e., most of the files automatically ranked highest in similarity were found to show strong indications of plagiarism when inspected by teaching assistants and the course instructor.

The screening method is efficient because it is able to take advantage of very fast file compression techniques. As a result, even though a large number of file comparisons are required for an entire class of students, screening results can be obtained in under 5 minutes for a class of 52 students (1326 file comparisons). Our previous implementations required several days of execution time for similar student populations. Given that the number of file comparisons grows as the square of the number of students, the current implementation should be practical for class sizes of at least 250 students.

While our technique was targeted for VHDL source code, only the tokenizer is dependent on the particular language being used. Since our test results don't indicate any particular advantage to the use of tokenized files over original source code, one could simply eliminate the comparison of tokenized files. However, intuition suggests that pre-filtering or tokenizing of the source code should help to reveal at least some cases of plagiarism. Since the execution time is short for our typical course enrollments, we will continue to use both approaches, even as we investigate more effective tokenized representations for similarity comparisons.

Bibliography

[1] Schleimer, S., Wilkerson, D.S., Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", Proc. SIGMOD 2003, June 2003, San Diego, CA.

[2] Prechelt, L., Malpohl, G., and Phillippsen, M., "Jplag: Finding Plagiarisms among a Set of Programs", Technical Report 2000-1, Universitat Karlsruhe, Germany.

[3] Bennett, C.H., Li, M, Ma, B., "Chain Letters and Evolutionary Histories", Scientific American, June 2003.

[4] Culwin, F., MacLeod, A., Lancaster, T., "Source Code Plagiarism in UK HE Computing Schools, Issues, Attitudes and Tools", Technical Report SBU-CISM-01-01, South Bank University, London, Sept. 2001.

[5] Nelson, M., Gailly, J-L, "The Data Compression Book", M&T Books, New York, NY 1995, ISBN 1-55851-434-1

MARK C. JOHNSON, Ph.D. Purdue University, 1998, M.S.E.E. Wichita State University, 1991, B.S.E.E. Purdue University, 1983. Dr. Johnson is currently the Manager of Digital and Systems Laboratories for the School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana. Dr. Johnson's primary research interest is in computer aided design for VLSI. His primary instructional interests are in digital system and VLSI design education.

CURTIS WATSON, B.S.E.E. Purdue University, 2003. Mr. Watson is currently a Masters Degree student at the University of Illinois, Champaign-Urbana. Mr. Watson worked for Dr. Johnson as an undergraduate lab assistant and an undergraduate teaching assistant from 2001-2003 in the ASIC Design Laboratory and Computer Architecture Prototyping Laboratory at Purdue University.

SHAWN DAVIDSON, M.S.E.E. Purdue University. Mr. Davidson is currently a design engineer with the Hewlett-Packard Company in Fort Collins, Colorado. Mr. Davidson worked for Dr. Johnson as a teaching assistant in the ASIC Design Laboratory at Purdue University.

DOUGLAS ESCHBACH, M.S.E.E. Purdue University. Mr. Eschbach is currently a design engineer with Qualcomm Incorporated in San Diego, California. Mr. Eschbach worked for Dr. Johnson as a teaching assistant and laboratory coordinator in the ASIC Design Laboratory at Purdue University.