Reducing Effects of Plagiarism in Programming Classes

Kevin W. Bowyer
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, Indiana 46556-5637 USA
kwb@cse.nd.edu

and

Lawrence O. Hall
Department of Computer Science and Engineering
University of South Florida
Tampa, Florida, 33620-5399 USA
hall@csee.usf.edu

ABSTRACT

Large programming classes are traditionally an area of concern for maintaining the integrity of the educational process. Systematic inspection of all program solutions for evidence of plagiarism can be done using an automated tool. The ``Measure Of Software Similarity" tool developed by Alex Aiken at the University of California at Berkeley analyzes a set of programs to detect evidence of "duplicates." However, experience in applying this sort of plagiarism detection in a large programming class indicates that the main long-term effect may be to simply shift the source of plagiarism. This possibility leads to considering the reason for fighting plagiarism and then to exploring additional techniques aimed at reducing the perceived motivation for plagiarism.

Keywords: computer programming, plagiarism, cheating, academic integrity, grading.

1. INTRODUCTION

Probably every faculty member who regularly teaches a programming course encounters plagiarism. Programming courses are often taught with a substantial portion of the grade determined by the solutions for programming assignments done as homework. Plagiarism on assignments presents a serious problem for the integrity of the educational process. In a recent survey of 242 undergraduates at Duke University, nine percent revealed that they had copied another student's computer program at least once while at Duke (Bliwise, 2001). Students appear to take this type of cheating lightly, as only forty percent of the respondents characterized it as "serious" (Bliwise, 2001).

Instances of plagiarism are most often detected on an ad hoc basis. The grader may notice that two programs have the same idiosyncrasy in their input/output behavior, or the same pattern of failures for certain test cases. With suspicions raised, the programs may be examined further and the plagiarism discovered. Obviously, this scenario leaves much to chance. Especially in large classes, program solutions may be graded by teaching assistants. The larger the class, and the more graders involved, the lower the chance that any given instance of plagiarism will be detected. For students in the class who are aware of various instances of cheating, which instances are detected and which are not may seem to be essentially random. Cheating that is widely known among the students and yet not detected by the faculty can undermine the good students' confidence in the educational process.

The standard "brute force" attempt at cheating on a program assignment is to obtain a copy of a working program and then change statement spacing, variable names, I/O prompts and comments. This type of plagiarism can be detected through systematic comparison of all pairs of program solutions. It is possible to do this manually, but it requires an impractical level of effort to be done routinely for large classes. Also, more sophisticated plagiarism efforts (e.g., rewrite main program and change the order of appearance of the functions) may survive a quick manual inspection. There is a clear need for an automated tool for this task, and various such tools have been developed (Joy 1999; Verco 1997; Wise 1996). We have used the Measure of Software Similarity (MOSS) tool and feel that it offers an excellent solution (Aiken 1995). Sections 2 and 3 of this paper outline our experience using MOSS and handling incidents of detected plagiarism (Bowyer 1999).

It is important to note that the type of evaluation done with MOSS can only detect plagiarism that exists among the set of program solutions. It cannot detect the type of cheating where a student in the class gets a student outside the class to supply the program solution. Another approach to combating plagiarism is to try to stop plagiarism incidents from occurring. Plagiarism on program solutions is grade-motivated and generally occurs outside of class. This suggests basing the grade solely on programming efforts made in-class under faculty supervision. Also, students who commit plagiarism often were working on a solution of their own, but perceived that they were hopelessly behind schedule. Sections 4 through 6 describe our experience with alternative course procedures based on these observations, with the goal of minimizing the effects of plagiarism on the integrity of the educational process.

The particular course that is the context for our experiences reported here is named Program Design. It is a required course for entry into the undergraduate majors in Information Systems, Computer Science, and Computer Engineering at the University of South Florida. The language used in the course is C. Course sections range in size from 80 to 120. Department policy calls for an F in the course for a first incident of academic dishonesty. A second incident may result in dismissal from the Department. Students are typically informed of the policy both in the syllabus and in a separate handout.

2. USING THE MOSS TOOL

The MOSS tool (Aiken 1995) makes it possible to objectively and automatically check all program solutions for evidence of plagiarism. It works with a wide variety of languages, including C, C++, Java, Pascal, ADA and others. The MOSS script for the client end should run on UNIX systems that have perl, uuencode, mail and either zip or tar. A comment in the MOSS script states – "Feel free to share this script with other instructors of programming classes, but please do not place the script in a publicly accessible place." Accordingly, and in deference to possible copyright issues, we do not reproduce any of the script in this paper. Aiken does not supply explicit information about the algorithm(s) used to detect cheating. In keeping with his desire that the inner workings be confidential, we do not speculate on the algorithms involved.

Program files to be submitted to MOSS can be in any subdirectory of the directory from which the MOSS command is executed. For example, to compare all programs in the current directory on a UNIX system, assuming that the programs are written in C and that MOSS is in the current directory, the following simple command could be used:

```
moss -l c *.c
```

The system allows for a variety of more complicated situations. For example, it allows for a "base file" that contains a program outline or partial solution handed out by the instructor. The degree of similarity between programs that is traceable to this base file should be factored out of similarity rankings of the programs. Also, MOSS allows for the programs that are to be compared to be composed of sets of files in different directories.

The MOSS command results in the programs being sent to a server at UC – Berkeley. The server sends email sent back to the login name that invoked the MOSS command, giving a web address for the results. In our experience, sending 75 to 120 C programs of a few hundred lines each, the results are available the same day. The return email from the server currently states that the results are kept available for fourteen days. Figure 1 shows the MOSS results page for some program pairs in actual plagiarism incidents in a section of our class. Program file names have been changed to hide the individuals' identities. For each listed program pair, the results summary lists number of tokens matched, number of lines matched, and percent of each program source found as overlap. A "token" here is just a name or operator in the program, as found by the early stage of a compiler. The degree of program overlap is found in terms of the percentage of tokens in the program. The percentage of overlap may vary as measured for each program if one has been altered in a way that changes its total number of

```
Moss Results

Sun Mar 14 15:24:02 PST 1999

Options -l c -m 10
  ----------------------------------------------------------------------
[ Text Report | How to Read the Results | Tips | FAQ | Contact Moss |
Submission Scripts | Credits ]
  ----------------------------------------------------------------------
      File 1             File 2       Tokens Matched Lines Matched
 mike_wolf.c (79%)  mike_fox.c (80%)            463           139
 bill_smyth.c (86%) bill_smith.c (88%)          456           133
 jane_white.c (59%) jane_blanco.c (68%)         354           111
 john_doe.c (100%)  john_deer.c (100%)          220            49
  ----------------------------------------------------------------------
Any errors encountered during this query are listed below.
```

Figure 1 – Example of MOSS Program Comparison Results.

tokens.

In our experience, anything over 50% mutual overlap is highly suspicious. However, the threshold for suspicion may depend on the size of the programming problem, the amount of hints given in class, and many other factors. We would strongly recommend that accusations of plagiarism cannot be made purely on the basis of MOSS ratings. It is essential for the instructor to consider the similarities in the particular pair of solutions in the context of how the course has been taught before reaching a final decision

Clicking on a program pair listed in the results summary brings up side-by-side frames for the two programs, along with a list of ranges of lines of source code that "pair up" in the two programs. The paired sections of the programs are given color-coded highlighting. The user can scroll through the programs, or click on a listed range of lines to jump straight to that section.

Relatively sophisticated attempts at plagiarism are readily detected using MOSS. Multiple similar sections of code separated by sections with substantial differences are still found and given color-coded highlighting. Functions may be given different names, and placed in a different order in the program and they are still matched up. Students who have changed all variable names, the statement spacing, the comments, the function names and the order of appearance of the functions stand out just as readily as students who turn in exact duplicate programs!

To summarize, detection of possible program plagiarism is made relatively simple using MOSS. In our experience, the real difficulties for the faculty member are now shifted to (1) processing the cases of plagiarism through the grading and appeals process, and (2) designing course policies and procedures that reduce the students' perceived pressure to cheat and to make the learning process more effective.

3. HANDLING CASES OF PLAGIARISM

In the first semester that we used MOSS, in one section of about 80 students, a total of ten received an F for plagiarism! The incidents of suspected plagiarism were handled as follows. First, an email was sent to the students requesting (1) a written summary of any information that might help in understanding why the programs were rated as highly similar, and (2) a time when the students could meet with the professor. Importantly, no accusation of plagiarism is made at this point. In perhaps 10% of the incidents, this e-mail elicited a confession of plagiarism from one of the students. In another perhaps 20% of the incidents, the first response to the e-mail was a denial, but then a confession came before the scheduled meeting. The rest of the incidents resulted in a meeting with the professor. In these meetings, reviewing the program similarities resulted in a confession in all but one case. In this one case, the two students admitted talking about the program and agreed that their programs were strikingly similar, but insisted that there was no plagiarism, even when it was pointed out that the programs contained identical non-functional elements such as un-needed curly brackets, `const` values passed to functions and not used, and so on.

In cases where it appeared that one student copied another's program without their consent, only the one student who plagiarized received an F. In cases where it was clear that one student intentionally gave their

program to another, both received an F. In the one case where both students denied the plagiarism after meeting with the instructor, both students were assigned an F. The university handbook provides for several levels of appeal. In our experience, about half the F grades due to plagiarism are appealed. Most appeals are not on the basis of denying the plagiarism, but arguing that the penalty of an F for the course is too harsh. Additional premises sometimes offered were that it would hurt the student's GPA, chances of getting into grad school, and/or chances of getting a desired job. The first level of appeal is to the Director of Undergraduate Studies in the Department. The next level is a committee of students and faculty from across the college. The final level is the university-wide Dean of Undergraduate Studies. Only one of the ten incidents mentioned earlier was appealed as far as the University level. In all cases, the plagiarism decision was upheld. One student who received an F due to plagiarism in this section re-took the course the next semester, plagiarized again, and was dismissed from the Department. Some other students re-took the course in subsequent semesters and did well.

Each plagiarism incident typically requires several hours of the professor's time. Examining the MOSS comparison results is a small part of this. Additional time is spent communicating with the students, meeting with some of the students, documenting the incident in memos to the various appeals groups, and appearing before appeals committees to explain the incidents.

One particular incident provides a strong caution against jumping to a decision based solely on the MOSS results. In this incident, two students had very similar program solutions, well above the threshold that would be suspicious. However, after investigation, it appears that both had discovered the same way to adapt an example in the textbook into a solution for the assignment. Thus, the two program solutions were constrained to be highly similar by design. In this incident, there was no plagiarism. The students had simply utilized a novel observation from their assigned reading. If students re-use code from an outside source, we would expect some standard form of attribution in comments. However, we would not be as critical when students adapt an example from assigned reading or class discussions.

1. UNDETECTABLE PLAGIARISM

In the first semester we used MOSS, ten of approximately 80 students received an F for plagiarism. However, in a section of over 140 students the next semester, only nine received an F for plagiarism. The rate of detected plagiarism decreased, presumably indicating that the level of actual plagiarism decreased, as it became understood that all programs are carefully checked. However, one incident suggested another interpretation. Two students whose programs were nearly identical insisted that they had not cheated from each other. One student eventually indicated that a friend who was not in the class had written the program. This third person was not even currently a student at the university. But this third person had a second friend in the class, and had also provided the program solution to that student! So we detected the plagiarism only because two students turned in the same program actually created by a third person.

The "ghost author" phenomenon is likely to be more widespread than just the incidents we have ccidenttally discovered. We have noted the phenomenon of students who consistently receive near-perfect scores on program assignments yet also consistently receive substantially lower scores on in-class quizzes which require writing short program segments. Of course, some students may have "test anxiety" and naturally perform below their "true" level on in-class quizzes. Also, some students may be getting help or coaching at a level that is not plagiarism, but that does pre-empt some of the learning experience for them. But we suspect that some are regularly getting "help" at a level that constitutes plagiarism.

It is helpful to consider this problem in the context of the purpose of grading in the course. A student's final grade should reflect their programming ability relative to some objective standard. It certainly should not reflect the ability of someone from whom they have plagiarized. But it also should not reflect the degree or quality of outside help that they have received. Students may get many innocent explanations of "this is how you write a loop" or "this is how you do a selection sort" from friends and acquaintances. Some of these explanations may apply fairly directly to the homework solutions. The problem occurs when the learning has not occurred; that is, when the student does not understand the help that they have received well enough to generalize to their own independent solution of similar problems.

2. EXPERIENCE WITH UN-GRADED PROGRAM ASSIGNMENTS

In a recent semester, the course was taught with the homework program solutions having zero weight in the course grade. The course operated as follows. We gave the same number (six) of program assignments as in previous course offerings. These assignments start out fairly simple and increase in difficulty.

Solutions to the first assignment might require 30-40 lines of code, and the last assignment 150-300 lines of code. While the solutions were not "graded" in the sense of counting in the final grade, they were "evaluated." Students invoke a "hand-in" script that compiles their program, runs it with test data sets created by the instructor and teaching assistants, and then sends an email back to the student. The email contains (1) output from compiling their program, (2) the program's output for each test case, and (3) a specification of the preferred output for each test case. Students can invoke the hand-in script as often as they want. Thus they can incrementally develop and or debug their program against the standard embedded in the script.

Along with the program assignments, there were five quizzes and a final exam. Questions on the quiz are correlated to the material covered in the class and applied in the program assignments. Thus the students saw loops on the quiz after they used loops in the program assignment, arrays after they used arrays, and so on. Quiz questions were of the form – "write a function to do the following task." Example quiz questions are listed in Figure 2. Thus the students **were** graded on their programming ability, but as demonstrated on in-class quizzes rather than out-of-class program assignments.

```
Example question from first quiz:
  Write a program that will read in 20 integer values, compute
  the average, and print an appropriate message giving the
  average value.  Use a for loop.

Example question from second quiz:
  Write the function definition for the prototype
    float find_max (float data[], int N);
  The function should return the maximum value in the N items in
  the data array.

Example question from third quiz:
  Write a function for the prototype
    int above_thresh (float data[], float threshold, int size);
  The int value returned is the number of items in data that are
  greater than the value of threshold, and the size parameter
  gives the number of values in the data array.

Example question from fourth quiz:
  Write the function for the prototype
    int binary_search (int data[], int lo, int hi, int look_for);
  The function should return the index of the value look_for in
  data, or -1 if look_for is not found. The function should
  operate recursively.  Assume data is in ascending order.

Example question from fifth quiz:
  Write the recursive function for the prototype
    int number_of_nodes (node *front);
  The parameter front points to a singly-linked list with nodes:
    typedef struct node_tag {
      char name[80]; float rating; struct node_tag *next;} node;
  The list is kept without a dummy header node.  The function
  returns a count of the number of nodes currently in the list.
```

Figure 2 – Example Programming Questions from the In-Class Quizzes.

This experience with un-graded program assignments turned out quite pleasant, but was ultimately regarded (by the professor) as a failure. One the positive side, it eliminated many elements of the class that tend to frustrate students. If a program was not working "perfectly" by the submission deadline, there was no penalty. There were no "gotcha" test cases that the student failed to anticipate and resulted in a lower grade. Also, the number of programming questions brought to faculty and teaching assistant office hours greatly decreased.

Judging from the professor's perspective, students

seemed quite happy with the course run in this manner. And, not surprisingly, there were no incidents of plagiarism. The failing of this approach was that because the programs did not count in the final grade, many students simply did not work seriously on the programs. For example, only about ten percent of the class used the hand-in script available for self-evaluation on the last program assignment in the semester.

3. EXPERIENCE WITH "LADDER" GRADING

Positive aspects of having the programs not count in the final grade and providing a hand-in script for the students to use in self-evaluation of their program solutions were (1) removing the influence of "outside programming help" on the course grade, and (2) allowing the students to stay focused on developing their solution to a well-defined standard. The problem was that without the program assignments concretely linked to the course grade, too many students opted not to work on the programs. Thus in the following semester, a "ladder" grading system was used. Students could still run the hand-in script for an assignment as many times as desired. The result of the last run before the assignment deadline was the basis of an S/U grade for the assignment; in general, more than one test case not working meant a U. The course grade was then determined using a ladder based on quiz averages and number of S program assignments: A = 90+ quiz average and S on all programs, b = 80 to 89 quiz average and S on 5 of 6 programs, C = 70 to 79 quiz average and S on 4 of 6 programs, and so on.

This approach to the course seems the best of those tried. Students are motivated to work on each assignment. At the same time, "outside help" on assignments cannot distort the class grades, since the letter grade is based on the students' programming ability as demonstrated on in-class quizzes. The ability to run the hand-in script as many times as desired before the deadline seems to have reduced motivation for plagiarism; through the first four assignments in a class of 80 students no plagiarism has been detected.

3. SUMMARY AND DISCUSSION

MOSS is a major innovation for identifying possible plagiarism in programming courses. Regardless of other course policies and procedures, we recommend routine use of MOSS to screen for evidence of plagiarism. The use of a hand-in script that can be run as often as liked before the assignment deadline is also an unqualified positive. It eliminates student resentment over "gotcha" test cases and encourages them to focus on creating a solution to a given specification. It also appears to reduce the frequency of plagiarism. This might be the result of helping to keep students focused on the problem, and so reducing the frequency of students finding themselves at a hopeless dead end in their programming efforts. As a final element, the use of a "ladder" approach to course grading eliminates the influence of outside help on the final grades in the course.

Most programming courses are organized on the premise that students should work individually on the programming assignments. Williams discusses an approach to teaching programming classes that actually requires students to collaborate (Williams, 1999). The approach is related to what is called the "Extreme Programming" methodology, which incorporates elements of what Weinberg called "ego-less programming" (Weinberg, 1971). Williams describes an approach to teaching programming classes in which students are paired together for the entire semester for purposes of completing the program assignments, but take the exams individually. The students were instructed to meet together to design, implement, and test the program assignment. Williams reports subjective evidence for students learning faster and implementing higher quality programs in this approach. The approach of course runs into the problem of how to assign credit when the two students do not contribute equally, as well as other administrative and organizational issues. We plan to experiment with some variant of this approach in a future semester.

8. REFERENCES

Aiken, Alex, 1995. www.cs.berkeley.edu/~aiken/MOSS.html

Bliwise, Robert .J., 2001. "A matter of honor." Duke Magazine, May-June 2001, pp. 2-7.

Bowyer, Kevin and L.awrence Hall, 1999. "Experience using MOSS to detect cheating on program assignments," Frontiers in Education, 13b3, 18-22.

Bowyer, Kevin, 2001. Ethics and Computing: Living Responsibly In a Computerized World (second edition), John Wiley / IEEE Press.

Joy, M. and M. Luck, 1999. Plagiarism in Programming Assignments, *IEEE Transactions on Education* 42 (2), 129-133.

Verco, K.L. and M. J. Wise, 1997. Plagiarism a la Mode: A Comparison of Automated Systems for Detecting Suspected Plagiarism, *Computer Journal* 39 (9), 741-

750.
Weinberg, Gerald, 1971. The Psychology of Computer Programming. Dorset House.
Williams, Laurie, 1999. "But isn't that cheating," Frontiers in Education, 12b9, 26-27.
Wise, M.J., 1996. YAP3: Improved Detection of Similarities in Computer Programs and Other Text, *SIGCSE Bulletin* 28 (1), 130-134.

Kevin Bowyer is the Schubmehl-Prein Professor and chairman of the Department of Computer Science and Engineering. He received an Outstanding Undergraduate Teaching award from the USF College of Engineering in 1991, and Teaching Incentive Program awards in 1994 and 1997. He has served as Editor-In-Chief of the IEEE Transactions on Pattern Analysis and Machine Intelligence, and as North American Editor of the Image and Vision Computing Journal. He was elected as a Fellow of the Institute of Electrical and Electronics Engineers in 1998.

Lawrence Hall is a professor in the Department of Computer Science and Engineering. He has taught programming classes for over 15 years. He is the electronic editor for the IEEE Transactions on Systems, Man, and Cybernetics Part B and on the editorial boards of the IEEE Transactions on Fuzzy Systems, and the International Journal of Data Analysis. He has published a number of papers in the fields of artificial intelligence, machine learning, and soft computing.

Information Systems & Computing
Academic Professionals

STATEMENT OF PEER REVIEW INTEGRITY

All papers published in the Journal of Information Systems Education have undergone rigorous peer review. This includes an initial editor screening and double-blind refereeing by three or more expert referees.

ISSN 1055-3096