Cheating Cheating Detectors

Christian Collberg, Ginger Myles, Michael Stepp
Department Computer Science
University of Arizona
Tucson, AZ 85711
{collberg,mylesg,steppm}@cs.arizona.edu

Technical Report TR04-05

March 3, 2004

Abstract

In this paper we present a new cheating technique that is successful at defeating cheating detectors and could become popular with students. The idea is to use obfuscating code transformations (such as those found in the SANDMARK tool) to apply a sequence of minor code transformations to a copied programming assignment. This purpose is to produce a copy that will defeat detection. We show that this technique is successful in defeating common plagiarism detectors such as Moss.

This paper is offered as a cautionary tale to the Computer Science teaching community. With the advent of powerful code transformation tools it will become necessary to develop correspondingly more powerful cheating detectors, or to revert back to manually testing for plagiarism.

1 Introduction

The lure of a high paying industry jobs has brought many to the field of Computer Science. However, in order to be eligible for one of these jobs a student needs to successfully graduate with a Computer Science degree. This, in turn, requires the student to spend many long hours completing difficult and time-consuming programming assignments. Unfortunately, some students try to get around this part of their educational experience by simply copying other students' code. In order to avoid detection the student often makes minor changes to the program, such as changing comments and identifiers. To the untrained or overworked eye this technique is often sufficient to prevent the student from being caught. In order to stop such cheating attempts a variety of plagiarism detection tools have been constructed. A popular such tool is Moss [1].

In this paper we present a new cheating technique that is successful at defeating Moss and could become popular with students. This particular technique relies on code obfuscation algorithms such as those found in the SANDMARK framework [3, 2]. Code obfuscation is a tool that was developed to aid in the prevention of software piracy by applying semantics preserving transformations to programs. As we will see, students can use SANDMARK or similar tools to apply a sequence of minor code transformations to a copied programming assignment, in order to produce a copy that will defeat detection by Moss.

The goal of code obfuscation is to transform a program is such a way that an attacker is unable to understand it. To accomplish this degree of obfuscation a series of simple transformations is applied to the program. In order for a student to use code obfuscation as a successful cheating technique the transformations must be minor enough that they preserve much of the readability of the program. Because obfuscating transformations are designed to produce unreadable code a student who uses this technique is unlikely to obtain a perfect score on the assignment, unless the grading is based strictly on the results the program produces. However, if a C- student applies a sequence of minor

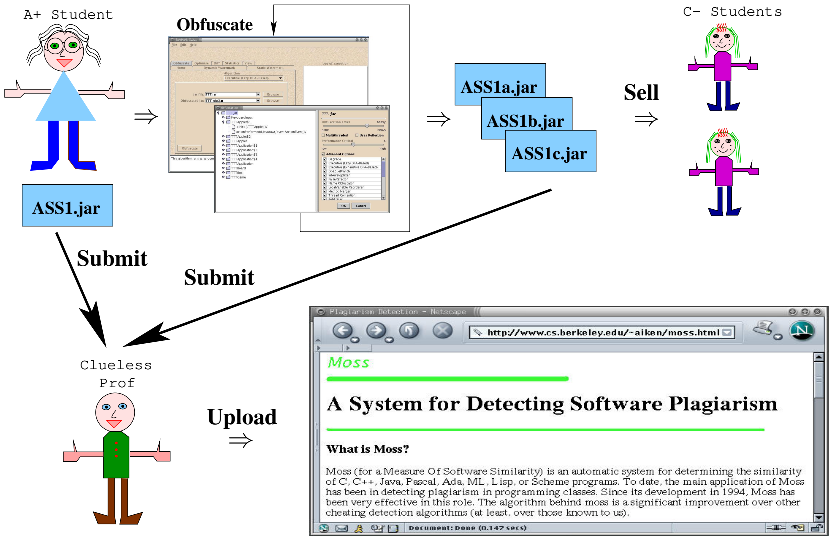

Figure 1: Cheating Moss.Ginger: add more stuff here explaining the figure

code obfuscations to code copied from an A student, he might possibly obtain a B on the program without getting caught cheating. This idea is illustrated in Figure 1.

This paper is offered as a cautionary tale to the Computer Science teaching community. We show that using tools readily available on the web, students can bypass commonly used cheating detectors. In particular, we show that even very subtle transformations (that are unlikely to affect a student's programming style grade) are enough to prevent Moss from detecting that cheating has taken place.

The remainder of this paper is structured as follows. In Section 2 we describe the algorithms used by the Moss plagiarism detector. In Section 3 we describe the SANDMARK code obfuscation tool. In Section 4 we describe the experiments we performed, exercising Moss with programs transformed by SANDMARK. In Section 5 we discuss future work and in Section 6 we summarize our results.

2 Moss

Moss is an automated tool commonly used in both undergraduate and graduate Computer Science programs to detect similarities between programming assignments. To use Moss, an instructor collects all the submitted assignments (along with similar assignments from previous years) and submits them to the Moss server. Moss compares the programs pair-wise for similarity and generates a web page of results. For each pair of programs Moss reports the number of lines that match and the degree of similarity. In addition, the instructor is able to view the two offending files to see the areas of the program that Moss found to be similar.

The algorithm used in Moss [5] is more sophisticated than in other plagiarism detection tools. A technique called *winnowing* is used to locate matching sequences between two files. Each file is divided into k-*grams* which are contiguous substrings of length k. A hash of each k-*gram* is then computed and a subset of the hashes is selected as the fingerprint of the document. Previous algorithms selected the set of hashes by retaining only those that are $0 \bmod p$. However, this selection technique provides no guarantee that a match will be detected between two documents.

To use the winnowing algorithm a window of size w of w consecutive hashes is defined. From each window the minimum hash value is selected. The hash values selected using this technique comprise the fingerprint of the document. To apply this technique to software programs each fingerprint also contains positional information. For each pair of programs the two lists of fingerprints are sorted and compared. The matches are presented to the user

Figure 2: The SANDMARK-tool.

---

ordered by number of matching fingerprints.

3 SandMark

SANDMARK [3, 2] is a tool for the study of software protection techniques. It contains a large number of obfuscating transformations that can be used to protect the intellectual property (such as important algorithms, system architectures, etc.) of a software application. SANDMARK works on Java bytecode. Specifically, it reads a Java jar-file (a collection of class files) as input and produces a new, obfuscated, jar-file as output. SANDMARK can be used either through a graphical user interface (see Figure 2) or a more traditional command line.

SANDMARK also contains a number of software watermarking algorithms. Like obfuscation algorithms these, too, are code transformations, but they are used to embed a unique identifier (such as a credit card number) into a Java program. Watermarking can be used to trace software pirates.

There are many similar (but much less sophisticated) tools available on the web. Zelix Klassmaster [7], for example, is a commercial code obfuscation tool.

SANDMARK offers a wide variety of obfuscating transformations. Essentially every aspect of a Java program can be affected by these transformations. This includes the class hierarchy, local variables, method bodies, method arguments, etc. For example, there are transformations that will merge and split classes and methods, transformations that will box integers (turn a Java `int` into a `java.lang.Integer`), split boolean variables into multiple parts, insert bogus control flow statements, etc.

We will next present experiments that show that some very simple obfuscating transformations are enough to defeat the algorithms used by Moss.

4 Experiments

To test the Moss system we performed a variety of different manipulations on the same input program, and compared the results against the original. Our test program was a simple Java set-implementation, with a `find`, `remove`, and `add` method, along with a public constructor. The souce code for the experiment can be seen in Figure 3. The eight manipulations we perfomed are described below.

a) Compile/decompile: This manipulation simply tested the effect of compiling the program with `javac` and then decompiling it with a commercial Java decompiler [6]. No other manipulations were performed. This test is of particular interest because most of the other manipulations must perform a decompilation as well, since the SANDMARK tool only acts on class files. Figure 4 demonstrates the transformation that occured using this manipulation.

b) Interleaving methods: This manipulation combined the `find` and `remove` methods of the `Set` class into one larger method. The new method used an extra input parameter to distinguish which of the two methods' bodies should be executed. The code from the bodies of `find` and `remove` remained mostly intact inside the larger method.

c) Identifier name obfuscation: All the identifiers in the program were converted into shorter, less user-friendly ones. The point of this common obfuscation is to remove any hints about the purpose of a variable or method that may appear in its name (i.e. `numElements` or `getSize()`). The only other semantic changes to the code were a result of the decompilation process.

d) Static method bodies: Each of the three methods was replaced by a static counterpart, which took an additional `Set` object parameter. Then three non-static wrapper methods were added to invoke the static methods by passing them their normal parameters along with the `this` reference. This left the external interface to the class the same, but changed the implementation by adding a level of indirection. It also introduced three extra methods that were not present in the original program.

e) Swap if/else bodies: This manipulation took every `if`-statement and negated its condition, then swapped the bodies of its `then` and `else` branches. This left the semantics of the program intact, but changed the relative positions of certain blocks of code. Thus a strict line-by-line comparison of this code with the original would likely find no similarities within the `if` statement whatsoever. The code transformation that occured can be seen in Figure 5.

f) Method merger: This is similar to Interleaving methods, except that it merges *all* methods with the same signature into one larger method that is parameterized by an extra input variable, rather than just merging pairs of methods. (Note: this manipulation only acts on static methods, so the program was first run through the Static method bodies transformation.)

g) Optimization: We used the BLOAT [4] Java optimizer as another manipulation to see how different optimized code would be compared to the original.

h) Smart renaming: Our final manipulation was to do a "smart" reassignment of variable names. For each identifier that was written in camelback style (i.e. parts of an identifier are identified by capitalization), the constituent words were reversed to create a new camelback identifier (e.g.. `removeElement` becomes `elementRemove`). The remainder of the program was left intact. This type of reassignment of variable names more closely models a student's attempt to copy someone else's code and then put in a minor effort to differentiate the two to avoid detection. The modifications made between the `removeElement` and `elementRemove` methods are shown in Figure 6.

Among all of these manipulations, the only one for which the Moss system could detect any similarities at all was Smart renaming. For the two files compared using this transformation, Moss detected a 98% similarity. This seems appropriate due to the small textual difference and the complete lack of semantic difference between the source files. Many of the others, however, still contained large regions of similar code that were clearly evident with casual inspection by a person.

```
class Set{
        private Object[] data;
        private int count;

        public Set(){
                data = new Object[100];
                count = 0;
        }

        public void add(Object o){
                if (o==null)
                        return;
                Object found = find(o);
                if (found==null){
                        if (data.length==count){
                                Object[] temp = new Object[data.length+100];
                                for (int i=0;i<count;i++)
                                        temp[i] = data[i];
                                data = temp;
                        }
                        data[count++] = o;
                }
        }

        public Object find(Object o){
                if (o==null)
                        return null;
                for (int i=0;i<count;i++){
                        if (o.equals(data[i]))
                                return data[i];
                }
                return null;
        }

        public void remove(Object o){
                if (o==null)
                        return;
                for (int i=0;i<count;i++){
                        if (o.equals(data[i])){
                                for (int j=i;j<count-1;j++)
                                        data[j] = data[j+1];
                                count--;
                                return;
                        }
                }
        }
}
```

Figure 3: Example code prior to obfuscation.

5 Future Developments

It should be noted that most security work is a cat-and-mouse game. As the cat gets cleverer at detecting and pursuing the mouse, so must the mouse get cleverer at avoiding detection and pursuit. It is reasonable to believe that plagiarist (and the tools they use or develop) will improve as the tools to detect plagiarism improve.

Consider, for example, a student who is aware that their instructor is using Moss to detect cheating. He can turn to SANDMARK or similar tools to transform his copied program into something for which Moss cannot detect cheating. Once the instructor is made aware that students are using SANDMARK to cheat he/she can assign TAs to manually check programs for artifacts that indicate that SANDMARK has been used. The TAs may, for example, look for unusually inverted tests, since these are produced by SANDMARK's Swap if/else bodies transformation. As a countermeasure, sophisticated students can then use Moss itself to find the subtlest sequence of transformations that will defeat Moss. The idea is to transform the original program P_0 into obfuscated program P_1 and then submit $\langle P_0, P_1 \rangle$ to Moss to see if it reports similarities. If it does, then transform P_1 into P_2 using some other transformation, and again

```
class Set{
    ...

    public Object find(Object o){
        if (o==null)
            return null;
        for (int i=0;i<count;i++){
            if (o.equals(data[i]))
                return data[i];
        }
        return null;
    }

    ...
}
```

$\xRightarrow{\mathcal{T}}$

```
class Set{
    ...

    public Object find(Object o){
        int int2;

        if(o != null){
            int2 = 0;
            while(int2 < count){
                if(o.equals(data[int2]))
                    return data[int2];
                else
                    ++int2;
            }
        }
        return null;
    }

    ...
}
```

Figure 4: The `find` method before and after compile/decompile.

```
class Set{
    ...

    public void remove(Object o){
        if (o==null)
            return;
        for (int i=0;i<count;i++){
            if (o.equals(data[i])){
                for (int j=i;j<count-1;j++)
                    data[j] = data[j+1];
                count--;
                return;
            }
        }
    }

    ...
}
```

$\xRightarrow{\mathcal{T}}$

```
class Set{
    ...

    public void remove(Object o){
        int int3;
        int int2;

        if(o != null){
            int2 = 0;
            while(int2 < count){
                if(o.equals(data[int2])){
                    for(int3 = int2; int3 < count - 1; ++int3)
                        data[int3] = data[int3 + 1];
                    --count;
                    return;
                }
                else
                    ++int2;
            }
        }
    }

    ...
}
```

Figure 5: The `remove` method before and after the `swap if/else` obfuscation.

query Moss for similarities. Incrementally, the student can find a final sequence of transformations $\langle P_0, P_1, \ldots, P_n \rangle$ that is both successful at cheating Moss and where P_n has as few suspicious artifacts as possible.

The question then arises are there no ways for instructors and honest students to protect themselves from cheating students. One possible technique is to use software watermarking to embed each student's identification number into their program. When two similar programming assignments are submitted it may then be possible to detect which student wrote the original one, and which one was copied and obfuscated. We believe this idea is promising and hope to investigate it further.

```java
class Set{
    ...

    public void addElement(Object o){
        if (o==null)
            return;
        Object found = findElement(o);
        if (found==null){
            if (dataArray.length==countElement){
                Object[] temp =
                    new Object[dataArray.length+100];
                for (int i=0;i<countElement;i++)
                    temp[i] = dataArray[i];
                dataArray = temp;
            }
            dataArray[countElement++] = o;
        }
    }

    ...
alphaenum}
```

$$\xRightarrow{\mathcal{T}}$$

```java
class Set{
    ...

    public void elementAdd(Object o){
        if (o==null)
            return;
        Object found = elementFind(o);
        if (found==null){
            if (arrayData.length==elementCount){
                Object[] temp =
                    new Object[arrayData.length+100];
                for (int i=0;i<elementCount;i++)
                    temp[i] = arrayData[i];
                arrayData = temp;
            }
            arrayData[elementCount++] = o;
        }
    }

    ...
}
```

Figure 6: The `add` method before and after `smart renaming`.

6 Summary

We have presented results that show that code obfuscation can be used as a successful method for cheating on programming assignments. Through our experiments we discovered that minor code obfuscations can be applied to a program in such a way that the student may still be able to obtain a respectable grade. In addition, when the Moss cheating detector is run on the transformed program no similarity is detected between the original and the copy. The fact that current code obfuscation tools can be used so successfully for this purpose is disconcerting. Through this study we hope that others will become aware of the possibility that cheating detectors can be cheated. We also hope that the community will begin to develop better cheating detection techniques in anticipation of dishonest students catching on to the use of SANDMARK and similar tools.

References

[1] Moss – a system for detecting software plagiarism. http://www.cs.berkeley.edu/ aiken/moss.html.

[2] Christian Collberg. SANDMARK. `http://sandmark.cs.arizona.edu`.

[3] Christian Collberg, Ginger Myles, and Andrew Huntwork. SANDMARK — A tool for software protection research. *IEEE Magazine of Security and Privacy*, 1(4):40–49, 2003.

[4] Nathaniel Nystrom. Bloat – the bytecode-level optimizer and analysis tool. `http://www.cs.purdue.edu/homes/whitlock/bloat`, 1999.

[5] Saul Schleimer, Daniel Wilderson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2003.

[6] Ahpah Software. Sourceagain. `http://www.ahpah.com`.

[7] Zelix Software. Zelix klassmaster. `http://www.zelix.com/klassmaster/`.