COAT: Code ObfuscAtion Tool to evaluate the performance of code plagiarism detection tools

Sangjun Ko, Jusop Choi, Hyoungshick Kim
Department of Computer Science and Engineering, Sungkyunkwan University
Suwon, Republic of Korea
Email: {kkosazz, cjs1992, hyoung}@skku.edu

***Abstract*—There exist many plagiarism detection tools to uncover plagiarized codes by analyzing the similarity of source codes. To measure how reliable those plagiarism detection tools are, we developed a tool named Code ObfuscAtion Tool (COAT) that takes a program source code as input and produces another source code that is exactly equivalent to the input source code in their functional behaviors but with a different structure. In COAT, we particularly considered the eight representative obfuscation techniques (e.g., modifying control flow or inserting dummy codes) to test the performance of source code plagiarism detection tools. To show the practicality of COAT, we gathered 69 source codes and then tested those source codes with the four popularly used source code plagiarism detection tools (Moss, JPlag, SIM and Sherlock). In these experiments, we found that the similarity scores between the original source codes and their obfuscated plagiarized codes are very low; the mean similarity scores only ranged from 4.00 to 16.20 where the maximum possible score is 100. These results demonstrate that all the tested tools have clear limitations in detecting the plagiarized codes generated with combined code obfuscation techniques.**

***Keywords*—Code plagiarism, Code similarity, Code obfuscation**

I. Introduction

Source code plagiarism is an act of copying someone's source code without giving credit to that code. The popularity of source code plagiarism is a serious concern in both academia and industry [1], [2]. For example, according to the results of the survey at a university, about 72.5% of university students admitted to cheating at least once during their graduate studies [3].

To deal with this problem, several source code plagiarism detection tools were introduced [4]–[7]. A typical procedure for detecting plagiarized codes is to analyze the structure of a source code (e.g., control flow graph, data structures) and compare it with the structure of another source code in order to measure the similarity of those two source codes. However, it is still questionable whether the existing plagiarism detection tools are indeed effective in identifying the plagiarized codes generated by code obfuscation techniques. Our work is originally motivated by this question.

To evaluate the performance of source code plagiarism detection tools, we developed a tool named Code ObfuscAtion Tool (COAT) that takes a program source code as input and produces another source code that is exactly equivalent to the input source code in their functional behaviors, but with a different structure. COAT particularly uses the eight techniques for code obfuscation. We used COAT to test the performance of the four popularly used source code plagiarism detection tools: Moss[1], JPlag[2], SIM[3], and Sherlock[4]. We highlight our main contributions as follows:

- We implemented COAT as the first fully automated and working tool to evaluate the performance of source code plagiarism detection methods by automatically generating functionally-equivalent obfuscated source codes based on the eight obfuscation techniques to modify the structure of program code.
- We showed the feasibility of COAT on the four popularly used source code plagiarism detection tools (Moss, JPlag, SIM and Sherlock). In our experiments, the mean similarity scores between the original source codes and the obfuscated plagiarized codes only ranged from 4.00 to 16.20, with higher scores indicating higher similarity to their original source codes where the maximum possible score is 100.

The rest of the paper is organized as follows. In Section II, we first introduce the four source code plagiarism detection tools that we used in experiments. We briefly explain the overview of COAT in Section III. We present the eight code obfuscation techniques used in COAT in Section IV. In Section V, we describe the experiment procedure and summarize the main results. In Section VI, we discuss the limitations of COAT and related work is covered in Section VII. Finally, our conclusions are in Section VIII.

II. Code plagiarism detection tools

In this section, we first describe the four representative program plagiarism detection tools which were tested with the obfuscated codes generated by COAT. We will present the experiment results in Section V.

A. Moss

Moss (Measure Of Software Similarity) is a source code plagiarism tool developed by researchers at Stanford university. Moss provides the result file as an HTML file that shows the percentage of code similarity between the two input files (i.e., source codes) and the parts to be identified as similar codes in each file. A high level procedure of Moss is as follows [8]: (1) Moss removes comments, variable names and

[1]https://theory.stanford.edu/~aiken/moss/
[2]https://jplag.ipd.kit.edu/
[3]https://dickgrune.com/Programs/similarity_tester/
[4]http://www.cs.usyd.edu.au/~scilect/sherlock/

all empty spaces, and then converts all uppercase characters to the corresponding lowercase characters; (2) it divides the given source code into k-grams where a k-gram is a contiguous substring of length k which can be comprised of codes and calculate the hashes of those k-grams [9]; (3) it chooses a subset of the hash values for the fingerprint of the source code; and (4) it compares this fingerprint with another source code's fingerprint. Moss has been popularly used because it supports a variety of programming languages (e.g., C, C++, Java, C#, Python, Javascript, Haskell, Lisp, Matlab and VHDL).

B. *JPlag*

JPlag was developed by Karlsruhe university and also provides the locations of similar parts between codes like Moss. A high level procedure of JPlag is as follows [10]: (1) JPlag removes comments, variable names, function names and all empty spaces; (2) it converts the source code into several string tokens; and (3) token strings are compared, respectively, to measure the similarity of codes. For token string comparison, Greedy String Tiling as introduced by Michael Wise algorithm was used [11]. JPlag supports not only programming languages (Java, C, C++, C# and Scheme) but also normal texts.

C. *SIM*

SIM was developed by Vrije university Amsterdam. SIM provides not only the percentage of code similarity between the two input files (i.e., source codes) but also the locations of similar parts between those files. Similar to JPlag, SIM uses string tokens for the comparison of given source code files. SIM supports not only some programming languages (C, Java, Pascal, Modula-2, Lisp and Miranda) but also normal texts.

D. *Sherlock*

Sherlock was originally developed by Rob Pike and modified by Loki Patrick in which its source code (written in C language) is available in public. Unlike other tools, Sherlock only provides the percentage of code similarity between the input files. Sherlock supports all types of programming languages and normal texts.

III. Overview of COAT

In this section, we describe how COAT was designed to evaluate the performance of source code plagiarism detection tools. Figure 1 shows the overall architecture of COAT where N is the number of obfuscation APIs used.

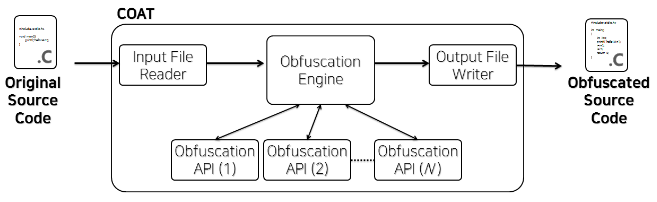

Fig. 1: Overall architecture of COAT.

The high level architecture of COAT is simple. COAT takes a program source code as input and produces another source code that is a functionally equivalent to the input source code, but with a different structure based on selected obfuscation techniques. To selectively use obfuscation techniques, we developed application programming interfaces (APIs) for obfuscation techniques. We particularly considered the eight representative obfuscation techniques. We will present the details of those techniques in Section IV. Naturally, additional obfuscation techniques can also be implemented as APIs and then incorporated into COAT.

To show the feasibility and effectiveness of COAT, we implemented COAT using Python and tested the obfuscated source codes generated by COAT with four plagiarism detection tools described in Section II. The experiment results will be presented in Section V. We note that the current COAT implementation only supports the source codes written in C language.

IV. Code obfuscation techniques used in COAT

Intuitively, if a code expression is changed with another code expression yielding the same result, the similarity score of any plagiarized detection tool can be lower. We applied the following eight code obfuscation techniques that can be categorized into code replacement, dummy code insertion and unnecessary code deletion.

A. *Modifying the order of function parameters*

For functions, COAT changes the order of the parameters in each function because changes to the order of parameters inside a function do not affect the behaviors of the function in C language. For example, COAT replaces `func(int a, char b)` with `func(char b, int a)` in both function descriptions and function calls through the entire code.

B. *Modifying loop conditions*

For loop statements such as `While` and `For`, COAT replaces their loop conditions with other equivalent expressions.

For example, COAT replaces the loop condition of "`i<10`" with another equivalent expression of "`i<=10-1`". Obviously, those expressions for loop condition have the exactly same context. COAT provide many different loop conditions having the same context.

C. *Modifying variable declarations*

For variable declarations, COAT replaces those expressions with other equivalent expressions.

```
int a;
int b;
int c;
```

(a) Before

```
int a,b,c;
```

(b) After

Fig. 2: Example of modifying variable declarations.

Figure 2 shows an example of such modifications. Figure 2 (a) and (b) show the codes before and after applying COAT, respectively. Those declaration statements have the exactly same context. COAT merges the variable declaration statements into a single line, and vice versa.

D. Modifying composite data types

For composite data types such as `struct`, COAT transforms a structure A into another (nested) structure B that contains the structure A. In the obfuscated source code, the new structure B is used instead of the original structure A.

```
struct A {
  int a;
  float b;
};
```

(a) Before

```
struct B {
  int A_a;
  float A_b;
  struct B A;
};
```

(b) After

Fig. 3: Example of modifying composite data types.

Figure 3 shows an example of such modifications. Figure 3 (a) and (b) show the codes before and after applying COAT, respectively. The newly defined nested structure is functionally equivalent to the original structure. The member `A` of `struct B` is itself a structure of type `struct A`. The members `a` and `b` of `struct A` can be accessed via the instance of `struct B`.

We note that the implementation of this obfuscation technique is a bit tricky because developers have their own programming styles in using composite data types. In our prototype implementation, many different source codes were intensively tested to cover corner cases.

E. Replacing the numeric values of array elements with arithmetic expressions

For arrays, COAT replaces the numeric values of array elements with equivalent arithmetic expressions. For example, COAT adds a randomly chosen value to each array element and then subtracts that value from each array element when the element is accessed.

F. Inserting dummy statements

COAT randomly inserts dummy statements without changing any functional behavior of the original program.

```
i=0;
j=0;
```

(a) Before

```
i=0;
j=1024;
j=0;
```

(b) After

Fig. 4: Example of inserting dummy statements.

Figure 4 shows an example of the insertion of a dummy statement. Figure 4 (a) and (b) show the codes before and after applying COAT, respectively. The inserted assignment "`j=1024;`" does not change any functional behavior of the program. We used several dummy code gadgets consisting of sequential statements (e.g., `i++; i=i-1;`) that do not change the program behavior.

COAT also declares new dummy variables at random. For example, the statement of "`int dummy=0;`" can be inserted.

G. Printing empty strings

COAT randomly calls output functions (e.g., `printf`, `fprintf` and `puts`) with an empty string. For example, COAT inserts the statement of "`printf("");`" to the source code at random.

H. Deleting whitespace and comments

Since C programming language uses curly braces ($\{ \cdots \}$) to define a block of code and ignores whitespace, COAT freely inserts various forms of whitespace (e.g., indentation, tap and space) and/or deletes some of existing whitespaces.

Also, COAT can freely place comments in the source code and/or displace existing comments from the source code because comments are ignored and not compiled by the compiler.

V. Experiments

In this section, we evaluate the performance of the four popularly used source code plagiarism detection tools (Moss, JPlag, SIM and Sherlock) with the obfuscated codes produced by COAT. Our goal was not only to find the best source code plagiarism detection tool but also to analyze the limitations of the existing tools.

For experiments, we used the dataset consisting of the source codes for the three programming assignments at a university. We gathered 20, 18 and 31 source codes (`Original`), respectively, from the first, second and third assignments. Next, we used COAT to generate the corresponding obfuscated codes (`Obfuscated`) from the collected 69 source codes.

We compared the properties of the obfuscated codes (`Obfuscated`) produced by COAT with the original source codes (`Original`) to show that the obfuscated codes by COAT can be executed without incurring significant overhead compared with the original source codes.

We first analyzed the differences in lines of code (LOC) between `Original` and `Obfuscated` (see Table I). The mean LOC of `Obfuscated` is about 1.5 times larger than that of `Original` across all assignments.

TABLE I: Lines of code (LOC) between `Original` and `Obfuscated` (μ: mean, σ: standard deviation).

Code type	*First assignment*		*Second assignment*		*Third assignment*	
	μ	σ	μ	σ	μ	σ
`Original`	151.85	54.94	272.94	70.31	214.64	73.60
`Obfuscated`	229.10	74.32	416.88	92.34	318.54	120.67

Next, we also measured the differences in the execution time between `Original` and `Obfuscated` (see Figure 5).

In Figure 5, the difference between the execution time of `Original` and that of `Obfuscated` is small. A majority (about 90.62%) of cases are less than 0.01 seconds and the maximum time difference is less than 0.06 seconds.

Based on those experiment results, we may conclude that the overheads incurred by COAT is not practically huge even though the mean LOC of `Obfuscated` was rather increased.

TABLE II: Results of similarity score between the original source code and the obfuscated code produced by COAT.

Obfuscation	*First assignment*								*Second assignment*								*Third assignment*							
	Moss		JPlag		SIM		Sherlock		Moss		JPlag		SIM		Sherlock		Moss		JPlag		SIM		Sherlock	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Parameter (A)	90.08	10.66	98.88	3.02	97.95	2.50	57.60	15.84	91.67	5.07	99.50	1.42	94.17	3.34	73.78	10.49	88.23	11.09	99.68	1.15	92.61	7.14	62.84	18.98
Loop (B)	68.70	13.22	100.00	0.00	91.35	4.84	73.50	16.92	92.17	3.27	100.00	0.00	95.33	2.56	83.72	11.09	95.42	4.07	100.00	0.00	98.29	2.47	80.06	17.78
Declaration (C)	90.45	9.75	94.80	3.19	96.80	1.69	77.40	15.93	95.17	2.11	95.06	2.90	96.67	1.20	86.78	11.76	96.29	2.20	95.06	3.44	97.42	1.39	80.81	16.57
Composite (D)	79.05	25.51	96.95	5.67	90.30	16.87	89.70	14.56	**58.06**	10.87	86.76	7.24	**69.50**	9.48	56.56	10.24	71.58	9.67	69.90	21.36	85.71	6.86	84.00	8.79
Array (E)	**60.10**	14.98	100.00	0.00	**76.75**	10.95	67.55	11.98	63.00	13.61	100.00	0.00	74.89	12.76	72.61	15.64	**71.03**	22.66	100.00	0.00	**80.81**	17.15	70.71	20.33
Dummy (F)	63.20	15.61	**44.70**	21.17	86.75	7.05	70.30	15.51	70.89	6.95	**44.18**	12.87	83.78	5.88	63.33	15.65	71.45	11.44	**45.29**	13.70	86.90	6.97	63.42	17.28
Printing (G)	82.30	13.87	100.00	0.00	96.60	3.07	75.70	14.67	95.56	2.41	100.00	0.00	97.17	1.50	89.22	11.68	94.03	3.29	100.00	0.00	96.68	1.75	83.06	17.05
Whitespace (H)	93.75	13.94	100.00	0.00	100.00	0.00	**35.90**	12.34	99.00	0.00	100.00	0.00	100.00	0.00	**11.83**	8.31	98.81	0.90	100.00	0.00	100.00	0.00	**18.13**	11.26
All Combined	4.40	3.93	6.40	6.90	12.15	6.80	16.20	6.11	5.17	4.09	11.44	8.20	11.61	4.77	4.00	3.97	4.06	5.44	10.16	8.27	9.32	7.31	4.45	3.82

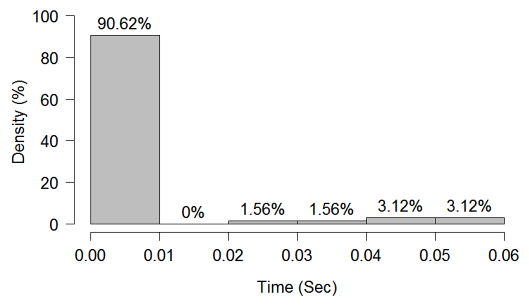

Fig. 5: Histogram of execution time differences between `Original` and `Obfuscated`.

For evaluation, we ran each of source code plagiarism detection tools (Moss, JPlag, SIM and Sherlock), respectively, to measure the similarity scores between the original source codes and the obfuscated codes produced by COAT, with higher scores indicating higher similarity to their original source codes. In the following sections, we present the experiment results and discuss our observations from those results.

A. *Effectiveness of code plagiarism detection tools*

To show the effectiveness of obfuscation techniques, each of the obfuscation techniques described in Section IV was individually applied. That is, for an input source code, we generated nine obfuscated sources codes with the following techniques: “Modifying the order of function parameters (Parameter)”, “Modifying loop conditions (Loop)”, “Modifying variable declarations (Declaration)”, “Modifying composite data types (Composite)”, “Replacing the numeric values of array elements with arithmetic expressions (Array)”, “Inserting dummy statements (Dummy)”, “Printing empty strings (Printing)”, “Modifying whitespace and comments (Whitespace)” and “All eight obfuscation techniques together (All Combined)”. Table II shows how code obfuscation techniques affect the similarity scores of the tools used in the experiments. For each dataset and tool, the lowest similarity score except for “All Combined” is presented in bold font to highlight the most effective obfuscation technique.

The results were somewhat inconsistent for the effectiveness of the individual techniques. Moss and SIM have similar effects; for the first and third assignments, “Replacing the numeric values of array elements with arithmetic expressions (Array)” was the most effective obfuscation technique while “Modifying composite data types (Composite)” was the most effective one for the second assignment. The technique of “Inserting dummy statements (Dummy)” was particularly effective for JPlag although it was also a generally effective way across source code plagiarism detection tools. Interestingly, the technique of “Modifying whitespace and comments (Whitespace)” was effective against Sherlock only. We surmise that the underlying differences of tools may explain this. Sherlock processes the input source codes without ignoring whitespace and comments while other tools generally remove all whitespaces and comments.

Although each obfuscation technique showed rather limited effectiveness in lowering the similarity scores between the original source codes and the obfuscated plagiarized codes, significantly low similarity scores could be achieved by a combination of those techniques. When all obfuscation techniques were applied together (i.e., “All Combined”), the mean similarity scores between the original source codes and the obfuscated plagiarized codes only ranged from 4.00 to 16.20 where the maximum possible score is 100. These test results demonstrate a clear limitation of all plagiarism detection tools tested in detecting the plagiarized source codes generated by COAT with combined code obfuscation techniques.

B. *Accuracy of detection of plagiarized codes*

The similarity scores measured by source code plagiarism detection tools can be used for detecting plagiarized codes; when a similarity score is greater than a chosen threshold value, we consider an input source code as a plagiarized code from another input source code.

Therefore, the performance of detection will be certainly affected by a chosen threshold value. To find the best plagiarism detection tool and/or their optimal threshold, we evaluated their performance with varying the detection threshold.

TABLE III: Results on the accuracy (Acc.) and F-measure (F-m.) of the plagiarized code detection tools.

Threshold	*First assignment*								*Second assignment*								*Third assignment*							
	Moss		JPlag		Sim		Sherlock		Moss		JPlag		Sim		Sherlock		Moss		JPlag		Sim		Sherlock	
	Acc.	F-m.	Acc.	F-m.	Acc.	F-m.	Acc.	F-m.	Acc.	F-m.	Acc.	F-m.	Acc.	F-m.	Acc.	F-m.	Acc.	F-m.	Acc.	F-m.	Acc.	F-m.	Acc.	F-m.
0	0.375	**0.545**	0.595	**0.597**	0.500	0.666	0.537	0.683	0.472	**0.641**	0.650	**0.729**	0.500	0.666	0.361	**0.530**	0.258	**0.410**	0.764	**0.766**	0.467	0.637	0.483	**0.652**
5	0.400	0.368	0.555	0.529	0.571	**0.677**	0.579	0.693	0.561	0.532	0.687	0.713	0.633	**0.720**	0.155	0.247	0.448	0.369	0.773	0.758	0.646	**0.667**	0.373	0.381
10	0.420	0.147	0.523	0.386	0.530	0.561	0.610	**0.698**	0.515	0.056	0.659	0.594	0.614	0.613	0.191	0.170	0.528	0.291	0.714	0.628	0.659	0.587	0.402	0.139
15	0.436	0.081	0.454	0.154	0.442	0.385	0.545	0.588	0.462	–	0.605	0.413	0.499	0.356	0.220	–	0.518	0.167	0.651	0.480	0.543	0.261	0.479	0.058
20	0.456	–	0.459	0.084	0.370	0.137	0.503	0.376	0.490	–	0.568	0.278	0.447	0.091	0.337	–	0.479	–	0.560	0.226	0.521	0.118	0.496	–
25	0.485	–	0.479	0.087	0.384	0.075	0.458	0.155	0.490	–	0.518	0.103	0.475	–	0.409	–	0.489	–	0.514	0.062	0.527	0.120	0.500	–
30	0.485	–	0.473	–	0.383	–	0.442	–	0.500	–	0.518	0.103	0.491	–	0.478	–	0.494	–	0.514	0.062	0.511	0.061	0.500	–
35	0.485	–	0.480	–	0.405	–	0.475	–	0.500	–	0.490	–	0.491	–	0.492	–	0.494	–	0.498	–	0.498	–	0.500	–
40	0.485	–	0.485	–	0.427	–	0.489	–	0.500	–	0.500	–	0.498	–	0.500	–	0.494	–	0.498	–	0.498	–	0.500	–
45	0.485	–	0.488	–	0.438	–	0.493	–	0.500	–	0.500	–	0.498	–	0.500	–	0.494	–	0.498	–	0.498	–	0.500	–
50	0.485	–	0.488	–	0.450	–	0.493	–	0.500	–	0.500	–	0.500	–	0.500	–	0.494	–	0.498	–	0.498	–	0.500	–

For evaluation, the plagiarized code detection task was transformed into a binary classification problem. The pairs of a source code and its plagiarized code are considered as Positive (P) and the pairs of independent source codes as Negative (N). True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) can be summarized as below:

- TP: Pairs of a source code and its plagiarized code *correctly* classified as pairs of a source code and its plagiarized code;
- FP: Pairs of independent source codes *incorrectly* classified as pairs of a source code and its plagiarized code;
- TN: Pairs of independent source codes *correctly* classified as pairs of independent source codes;
- FN: Pairs of a source code and its plagiarized code *incorrectly* classified as pairs of independent source codes.

To evaluate the performance of source code plagiarism detection tools, we use four measures as follows:

- **Accuracy**: the proportion of correctly classified pairs; $(TP + TN)/(P + N)$
- **Precision**: the proportion of pairs classified as pairs of a source code and its plagiarized code that actually are such pairs; $(TP)/(TP + FP)$
- **Recall**: the proportion of pairs of a source code and its plagiarized code that were accurately classified; $(TP)/(TP + FN)$
- **F-measure**: the harmonic mean of *precision* and *recall*; $(2 * Precision * Recall)/(Precision + Recall)$

In each programming assignment dataset, the number of negatives is much smaller than the number of positives; for example, in the first assignment, the number of negatives is 20 while the number of positives is 190 (= 20 $\times$ 19/2). To deal with the problem caused by unbalanced data distribution, the positives were randomly undersampled until its number is equal to the number of positives. Each source code plagiarism detection was tested with the obfuscated source codes generated by COAT (with the option of "All Combined") where its threshold value varying from 0 to 50. Table III shows the results on the accuracy and F-measure of the plagiarized code detection tools for each programming assignment.

Overall, except for the first assignment, JPlag produced the best results. In the second and third assignments, JPlag achieved 0.729 and 0.766 in F-measure when the threshold value is 0. In the first assignment, however, JPlag did not outperform other tools. Interestingly, Sherlock performed well with the source codes in the first assignment. We surmise that the properties of the source codes in the first assignment may explain this. Since the first assignment was relatively simple and the source codes contain only few comments, the technique of "Modifying whitespace and comments" used by COAT does not significantly degrade the performance of Sherlock.

Probably, a small threshold value (e.g., 0 or 5) may be preferred because all tools except for Sherlock performed well in the first assignment when their threshold values are 0 or 5. This is because most test cases for plagiarized codes generated by COAT have a similarity score which is less than 20.

VI. Limitations

COAT was actually designed to consider automatic plagiarism detection methods only. Therefore, the obfuscated source codes generated by COAT could easily be identified by manual inspection. We conducted a pilot study with a small number of participants to test the stealthiness of the obfuscated codes generated by COAT. In this initial pilot study, all participants can easily identify the obfuscated codes. In fact, the generation of obfuscated source codes against manual inspection will be a very challenging problem that may require sophisticated natural language processing algorithms.

Also, COAT does not seem to be effective in generating obfuscated binary codes. To show this, we used a popularly used binary comparison tool called Bindiff[5] to compare the binary codes complied from a source code and its obfuscated source code (generated by COAT). Bindiff provides not only the similarity score for the entire program but also that for each function in the program. To ignore unnecessary parts such as compiler-generated metadata, we used the mean similarity score of all functions rather than the similarity score for the entire program. In this analysis, we found that the mean similarity score is always greater than 0.8 for any pair of a code and its plagiarized code. Based on these results, we can conclude that the plagiarized codes generated by COAT can easily be detected by a static analysis of binary codes.

[5]https://www.zynamics.com/bindiff.html

VII. Related work

The plagiarized code detection problem has been popularly studied [1], [2], [4] over the past two decades. Measuring software similarity can generally be used for various applications such as malware detection and identifying vulnerable codes. Here we focused only on plagiarism detection.

To detect plagiarized codes, many different techniques were introduced. For example, Ji et al. [12] proposed a heuristic method that extracts sequences of predefined keywords from the target source codes and compares them by using a sequence alignment algorithm to measure the similarity of the codes. However, the performance of such detection methods can simply be degraded by changing the structure of the programs (e.g., with inserted dummy codes). Kim et al. [13] particularly discussed which obfuscation techniques would be effective in degrading the performance of sequence alignment-based matching algorithms. Granzer et al. [14] also raised several challenges in source code plagiarism and discussed the limitations of automatic source code plagiarism detection methods without considering the semantics of programs.

In general, code obfuscation techniques make the program analysis difficult. Barak et al. [15] theoretically proved that a perfect black box obfuscation is provably impossible. Collberg et al. [16] summarized various code obfuscation techniques and their applications. Recently, Schrittwieser et al. [17] categorized software obfuscation techniques and analyzed their effectiveness against the state-of-the-art of de-obfuscation methods. In this paper, we extend their work by developing a framework to generate various obfuscated codes against code plagiarism detection tools. We particularly applied our implementation to the four popularly used code plagiarism detection tools (Moss, JPlag, SIM and Sherlock) and found that plagiarized codes generated by combining several code obfuscation techniques cannot practically be identified by those tools.

VIII. Conclusions

Detecting plagiarized codes still remains an important problem. Many source code plagiarism detection tools were introduced in academia. However, there is no practical method to evaluate the performance of those tools. To address this problem, we presented COAT that takes a program source code as input and produces another functionally-equivalent source code as output, but with a different structure. For the first prototype implementation, the eight reasonable code obfuscation techniques were incorporated into COAT.

To show the feasibility of COAT on source code plagiarism detection tools, we collected 69 real programming codes used in a university and applied COAT to generate obfuscated source codes from the collected source codes. Next, we tested the performance of the four popularly used source code plagiarism detection tools (Moss, JPlag, SIM and Sherlock) with the generated obfuscated (plagiarized) codes and original source codes together. Interestingly, the test results demonstrated clear limitations in the practice of existing source code plagiarism detection tools. In the experiments, when all obfuscation techniques were applied together, the mean similarity scores between the original source codes and their plagiarized codes only ranged from 4.00 to 16.20 where the maximum possible score is 100. This implies that better program analysis tools should be developed to detect plagiarized codes generated by combining several code obfuscation techniques.

In future work, we plan to increase the size of the dataset and analyze any changes in performance. It would also be interesting to deploy COAT for real-world applications such as source code repository services and evaluate its performance and effectiveness.

Acknowledgment

This research was supported by IITP (No. B0717-16-0116) and ITRC (IITP-2017-2015-0-00403). The authors would like to thank all the anonymous reviewers for their valuable feedback.

References

[1] M. Joy and M. Luck, "Plagiarism in programming assignments," *IEEE Transactions on Education*, vol. 42, no. 2, 1999.

[2] D. Chuda, P. Navrat, B. Kovacova, and P. Humay, "The issue of (software) plagiarism: A student view," *IEEE Transactions on Education*, vol. 55, no. 1, 2012.

[3] D. Sraka and B. Kaucic, "Source code plagiarism," in *Proceedings of the Information Technology Interfaces 31st International Conference on*, 2009.

[4] Y. C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011.

[5] V. T. Martins, D. Fonte, P. R. Henriques, and D. da Cruz, "Plagiarism detection: A tool survey and comparison," in *OASIcs-OpenAccess Series in Informatics*, 2014.

[6] J. Hage, P. Rademaker, and N. van Vugt, "Plagiarism detection for java: a tool comparison," in *Computer Science Education Research Conference*, 2011.

[7] A. Bin-Habtoor and M. Zaher, "A survey on plagiarism detection systems," *International Journal of Computer Theory and Engineering*, vol. 4, no. 2, 2012.

[8] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the ACM SIGMOD International Conference on Management of data*, 2003.

[9] C. D. Manning, H. Schütze *et al.*, *Foundations of statistical natural language processing*. MIT Press, 1999, vol. 999.

[10] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *J.UCS. The Journal of Universal Computer Science*, vol. 8, no. 11, 2002.

[11] M. J. Wise, *Running karp-rabin matching and greedy string tiling*. Basser Department of Computer Science, University of Sydney, 1993.

[12] J.-H. Ji, G. Woo, and H.-G. Cho, "A source code linearization technique for detecting plagiarized programs," in *ACM SIGCSE Bulletin*, 2007.

[13] H. Kim, W. M. Khoo, and P. Liò, "Polymorphic attacks against sequence-based software birthmarks," in *Proceedings of the 2nd ACM SIGPLAN Workshop on Software Security and Protection*, 2012.

[14] W. Granzer, F. Praus, and P. Balog, "Source code plagiarism in computer engineering courses," *Journal on Systemics, Cybernetics and Informatics*, vol. 11, no. 6, 2013.

[15] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," *Journal of the ACM*, vol. 59, no. 2, 2012.

[16] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proffing, and obfuscation: Tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, 2002.

[17] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys*, vol. 49, no. 1, 2016.