Needle: Detecting Code Plagiarism on Student Submissions

Yanyan Jiang
jyy@nju.edu.cn
Nanjing University

Chang Xu
changxu@nju.edu.cn
Nanjing University

ABSTRACT

Code plagiarism is one of the most prevalent academic dishonesty activities in programming practicums. Automated code plagiarism detection plays an important role in preventing plagiarism and maintaining the academic integrity. This paper describes a novel code plagiarism detection algorithm needle, which is based on the network-flow approximation of editing distances between programs. This paper also presents the effectiveness and efficiency evaluation of the algorithm, the lessons and experiences learned from applying needle in practice, and discussions of the future challenges.

CCS CONCEPTS

• **Applied computing → Learning management systems**;

ACM Reference format:
Yanyan Jiang and Chang Xu. 2018. Needle: Detecting Code Plagiarism on Student Submissions. In *Proceedings of ACM TUR-C 2018 (SIGCSE China), Shanghai, China, May 19–20 (TUR-C'18),* 6 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Code plagiarism is one of the most prevalent academic dishonesty activities in programming practicums [15], and can be surprisingly severe without proper mechanisms to prevent against [1].

Code plagiarism detection tools help an instructor identify potentially plagiaristic program copies among students' submissions. Development of such tools dates back to early 1980s [7], and has undergone active research. Such a tool calculates the similarity between a pair of programs using token sequences [11, 14] or dependency graph features [2, 6, 9].

On the other hand, existing code plagiarism detection techniques can be tricked by "smart" students by applying certain semantics-preserving changes that drastically change the syntactical and dependency graph features [3, 12]. It is a challenge to identify such plagiaristic code clones effectively and efficiently. Background and related work are briefed in Section 2.

This paper presents our technical solution to code plagiarism detection, as well as our field study of applying code plagiarism detection tools in the teaching practice.

TUR-C'18, May 19–20, Shanghai, China

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The major technical contribution of this paper is a novel algorithm needle (and its implementation) for detecting code plagiarisms in program binaries, such that it can be easily applied to programs written in `C/C++`, `Java`, or `Go`. The needle algorithm approximates the editing distance between programs (to what extent a program binary can be "embedded" into another) in polynomial time. The algorithm not only resists to simple program changes (variable renaming or local program semantics changes), but also resists to many types of semantics-preserving changes that drastically change syntactic and dependency-graph features like function merging and splitting. Sections 3.1–3.3 elaborate on this algorithm.

We evaluated needle (in Section 3.4) and deployed the tool in the course teaching practice. We conducted empirical studies and report our experiences of (1) a case study of an anonymous programming assignment without plagiarism prevention (in Section 4.1), and (2) the three-year field deployment of needle in the "Principles and Techniques of Compilers" course (in Section 4.2). The findings are highlighted in the following and further discussed in Section 5.

Code plagiarism is a problem that *"everyone realizes but everyone chooses to ignore."* Without proper mechanism to control, code plagiarism could *totally corrupt* a programming assignment: 65 of 79 (82%) submissions are plagiaristic in the case study, where 42 are direct copies of over 99% code similarity.

The experiences of our three-year field study show that deploying code plagiarism detection tools can greatly alleviate the issue, but code plagiarism cannot be easily eliminated. There are still ~20% of the students risked themselves to plagiarize, and many made program changes to trick the plagiarism detection tool.

The maintenance of academic integrity, being orthogonal to the curriculum design, remains *nearly blank* in our curriculum system. We hope that the results presented in this paper could serve as a wake-up call to draw further attention to this issue.

2 BACKGROUND

2.1 Code Plagiarism of Students

Plagiarism is a commonly occurring issue in academic courses that students take advantage of others' work for higher grades. Students may copy others' code (and often make semantics-preserving changes or obfuscations to it) and receive unethical grades and thus threaten the integrity of the academic system. It is crucial to recognize and penalize such behavior in the class [11, 13, 15]: not only for maintaining academic integrity, but also for reducing plagiarism in the subsequent programming assignments in the same course.

Students usually choose plagiarism because they either lack the ability or the time to complete the task. Following the results of Wager [15], a plagiaristic copy is usually obtained by performing the following types of changes or obfuscations:

(T1) Change comments, names, or cases.
(T2) Reformat or reorder code fragments.

```
1  int f(int *a, int n) {
2    int s = 0;
3    for (int i = 0; i < n; s += a[i++]);
4    return s;
5  }
6  int g(int *a, int n) {
7    int i = 0, sum = 0;
8    while (i < n) { sum = sum + *(a + i); i++; }
9    return sum;
10 }
```

Figure 1: Two semantically equivalent but syntactically different functions.

(T3) Add or delete redundant elements.
(T4) Refactor standard constructs, such as type of loops and structure of functions.

For code changes beyond T4 (e.g., independently re-implementing function components) it is impossible even for a human to reach a verdict of plagiarism. Furthermore, realizing that such kind of plagiarism is much less frequent in practice, we consider such kind of changes out of the scope of this paper.

A student may simultaneously apply multiple types of changes, and may apply further changes to a specific part of the program (e.g., by adding some irrelevant functionalities). All these factors contribute to the challenge of detecting code plagiarism in practice.

2.2 Related Work

Code plagiarism can be automatically detected. Such an idea dates back to early 1980s [7]. A typical plagiarism detection tool summarizes a program using syntactic features like token sequences [14] or fingerprinting of program dependency (e.g., control- or data-dependency) graphs [2, 6, 9], and measures the similarity between a pair of programs by calculating their summary distance.

For example, Moss [14] summarizes a program as its n-gram token distribution; GPlag [9] summarizes a program by its dependence graph. The similarity between programs is considered as a subgraph isomorphism problem, and is solved by a heuristic search. Existing code clone detection techniques [2, 6] can also be applied to code plagiarism detection. These tools have been studied [8] and improved [4, 5, 11].

Existing techniques work well for T1 and T2 types of plagiarisms. However, when a student simultaneously applies local changes (T1–T3) and splits larger functions into smaller ones (T4), the similarity measured by existing techniques [9, 14] will drastically drop. Therefore, we present our editing distance approach to code plagiarism detection in the following.

3 AUTOMATED CODE PLAGIARISM DETECTION

3.1 Observations and Insights

To automatically identify plagiaristic copies that may subject to changes, we leverage the following observations:

(1) *Compiler optimization is good at normalizing functionally equivalent but syntactically distinct code fragments* [10]. This is because a compiler optimizer always searches for the "optimal" instruction sequence while preserving the program semantics, and is resilient to local minor semantic changes by its nature.
If we only consider the types (i.e., opcodes) of the instructions in a compiled binary (ignoring the operands), changes of types T1–T2 will have zero impact to the output, and even many changes of types T3–T4 will have only minor impacts as long as the plagiaristic program is still clean and readable. Consider the functions `f` and `g` in Figure 1. Compiled by a modern `clang` compiler, the opcode sequences of `f` and `g` only differ in one of 45 places (97.8% places are identical).
(2) *The plagiaristic copy $\hat{P}$, though may subject to changes, obfuscation, or even additional functions, usually properly "contains" the one being copied (P)*, otherwise it would take substantial efforts of code refactoring or re-implementation of functional modules. In other words, there usually exists an embedding of instructions in the binary of P to the ones of $\hat{P}$.

Therefore, we argue that code plagiarism detection could be done by measuring the degree of "*embedding*" between two program *binaries* as a discrete optimization problem, which is elaborated on in the following.

3.2 Problem Formulation

3.2.1 Preliminaries. We detect potentially plagiaristic code clones from a set of program binaries $\mathcal{P}$ (the students' submissions). Each program $P \in \mathcal{P}$ is a list of functions $P = \{f_1, f_2, \ldots, f_n\}$. A function $f_i \in P$ ($i \in [n]$) is a sequence of machine instructions. We use $|f_i|$ to denote the instruction sequence length of f_i, and use $f_{i,j}$ to denote the j-th instruction in f_i ($1 \le j \le |f_i|$).

Given programs $P_1 = \{f_1, \ldots, f_n\}$ and $P_2 = \{f'_1, \ldots, f'_m\}$, we would like to quantify to what extent P_1 can be "embedded" into P_2. Easier P_1 can be embedded into P_2, or more likely P_1 is a modified version of P_2 (and thus is susceptible of being a plagiaristic clone of P_2). We first define the *editing distance* between programs, and then relax such a definition for a tractable algorithm.

3.2.2 Editing Distance Between Programs. The editing distance $d(P_1, P_2)$ is defined as the minimum cost of transforming P_1 to a program that, when written flattened as an instruction sequence, is identical to that of P_2. The following editing operations are allowed:

(1) Reordering of two functions with zero cost.
(2) Deletion of an instruction $f_{i,j}$, costing c_d.
(3) Modification of an instruction $f_{i,j}$ (either instruction type or operands), costing c_m.
(4) Insertion of an arbitrary instruction at position j of function f_i, costing c_i.

Setting $c_d = c_m = 1$ (deletion and modification have a unit cost) and $c_i = 0$ (insertion of padding instructions has zero cost) reflects the idea of measuring to what extent P_1 can be embedded into P_2, and works well in practice.

Suppose that P_1 is a plagiaristic copy of P_2. If $P_1 = P_2$, it is trivial that $d(P_1, P_2) = 0$. Changing comments, names, cases, and the order of variables/functions does not affect $d(P_1, P_2)$, and thus the definition of editing distance over binaries by its nature is resilient to T1–T2 types of changes. For changes of type T3–T4, they may slightly impact the binary, and $d(P_1, P_2)$ reflects how much changes are done. Figure 2 displays a real plagiaristic code

```
1  void P1() {
2  ...
3    puts("Game␣Over");
4  ...
5  }
6  void P2() {
7  ...
8    puts("G"); puts("a"); puts("m");
9    puts("e"); puts("␣"); puts("O");
10   puts("v"); puts("e"); puts("r");
11 ...
12 }
```

Figure 2: Plagiaristic code fragments in practice.

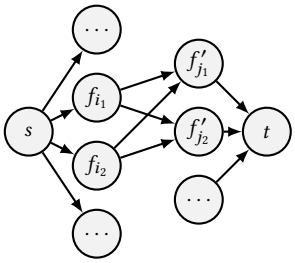

Figure 3: The flow graph G constructed by P_1 and P_2. Each edge (u, v) has capacity $c(u, v)$ and weight $w(u, v)$. The maximum weight network flow from source vertex s to sink vertex t denotes to what extent P_1 can be embedded into P_2.

fragment (detected by our tool) in which a functional component is manually expanded. Setting $c_i = 0$ yields a small editing distance between P_1 and P_2.

3.3 The needle Algorithm

Though the editing distance is effective in detecting code plagiarism, calculating $d(P_1, P_2)$ is NP-Complete. The computational cost would be unaffordable for medium-sized programs of ~10^4 instructions. The needle algorithm is an approximation of the editing distance that can be computed in polynomial time.

To approximate $d(P_1, P_2)$, we first define the similarity between a pair of functions $f_i \in P_1$ and $f'_j \in P_2$ as

$$\sigma(f_i, f'_j) = \max_{k \in \{1,2,\ldots,|f'_j|\}} \mathrm{LCS}(f_i, f'_j[k : k + \omega]),$$

where $\mathrm{LCS}(s_1, s_2)$ denotes the longest common subsequence of s_1 and s_2 (an instruction equals another if their opcodes are identical), and $f'_j[k : k + \omega]$ denotes the subsequence of f'_j whose indexes are in range $[k, \min\{k + \omega, |f'_j|\}]$. Intuitively, $\sigma(f_i, f'_j)$ is the longest common subsequence that is constrained in a fixed window size ω. A larger σ denotes that more instructions in f_i have corresponding equivalent instructions in a consecutive instruction sequence of length ω in f'_j, and thus denotes a smaller distance d.

Then, we compute the similarity between P_1 and P_2 by computing to what extent all functions $f_i \in P_1$ can be embedded into the functions in P_2. To allow multiple functions in P_1 to be embedded into a single function $f'_j \in P_2$, we construct a weighted flow network graph $G = (V, E)$ with capacity function $c : E \mapsto \mathbb{R}$ and weight function $w : E \mapsto \mathbb{R}$. Let $V = \{s, t, \ell_1, \ldots, \ell_n, r_1, \ldots, r_m\}$ in which s is the source and t is the sink, as shown in Figure 3.

For each $i \in [n]$, construct an edge (s, ℓ_i) where $c(s, \ell_i) = |f_i|$ and $w(s, \ell_i) = 0$; for each $j \in [m]$, construct an edge (r_j, t) where $c(r_j, t) = |f'_j|$ and $w(r_j, t) = 0$; for each $(i, j) \in [n] \times [m]$, construct an edge (ℓ_i, r_j) where

$$c(\ell_i, r_j) = \sigma(f_i, f'_j) \text{ and } w(\ell_i, r_j) = \frac{1}{1 + e^{-\alpha \cdot \frac{\max\{\sigma(f_i, f'_j), \sigma(f'_j, f_i)\}}{\min\{|f_i|, |f'_j|\}} + \beta}}.$$

A logistic function with constants α and β is used to normalize the weights such that "successful" embeddings between a pair of functions will have a weight near 1, while failed embeddings will have a weight near 0.

The similarity between P_1 and P_2 is thus defined as

$$\sigma(P_1, P_2) = \frac{\mathsf{MaximumWeightFlow}(G, c, w)}{\sum_{i \in [n]} |f_i|},$$

the optimal global instruction embedding (denoted by the maximum weight network flow) divided by the number of instructions in P_1. Intuitively, each unit of flow in $(\ell_i, r_j) \subseteq E$ "embeds" an instruction in $f_i \in P_1$ into another instruction in $f'_j \in P_2$. Only those edges with large $w(\ell_i, r_j)$, indicating that f_i can be embedded into f'_j, have significant contributions to the final similarity score, and a larger $\sigma(P_1, P_2)$ indicates a smaller $d(P_1, P_2)$.

The algorithm is implemented as a command line tool `bincmp` in which $\omega = \frac{3}{2}|f_i|$, $\alpha = 2$, and $\beta = 1/2$. To detect code plagiarism in a set of programs, a driver program takes a set of program binary paths as arguments, invokes `objdump` to extract instruction sequences, strips out system and common libraries, and finally manages a queue of parallel `bincmp` comparisons. In practice, invoking `bincmp` usually costs a few seconds of time. Analyzing hundreds of submissions (by a pairwise comparison) takes less than an hour of time on a multiprocessor machine.

3.4 Evaluation

3.4.1 Experimental Setup. The evaluation tries to answer the following two questions:

(1) (*Effectiveness*) Can needle effectively detect code plagiarism in programming assignments?
(2) (*Efficiency*) Does needle cost a reasonable amount of resource in plagiarism detection?

The evaluation is conducted over a set of students' 37 submissions in a lab of the "compiler design and implementation" course. In the lab, students need to parse a `C--` (a simplified dialect of the `C` programming language) program using `flex` and `bison` and produce a syntax tree. This code plagiarism detection task is particularly challenging because students' code is mixed with automatically generated code by `flex` and `bison`, which may have a negative impact on the precision of plagiarism detection.

To evaluate the effectiveness, we manually inspected the programs (with the aid of program similarity scores), and conduct face-to-face interviews for those suspect students of plagiarism. To evaluate the efficiency, the running time information is logged and analyzed.

We also implemented a fast and scalable code clone detection algorithm based on the fingerprint of control flow graph [2] (named

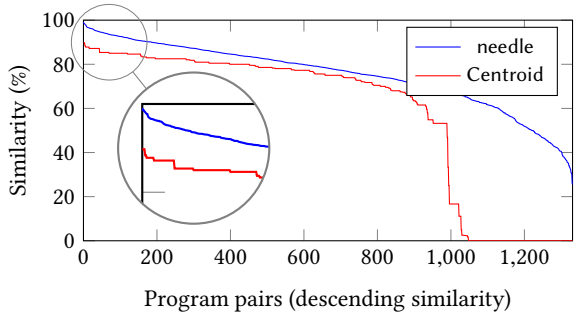

Figure 4: The similarity score distributions of needle and Centroid in the evaluation.

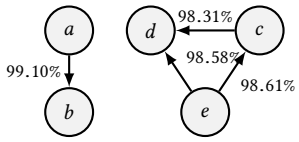

Figure 5: The top-4 ranked program pairs output by needle exactly match the manual plagiarism checking results.

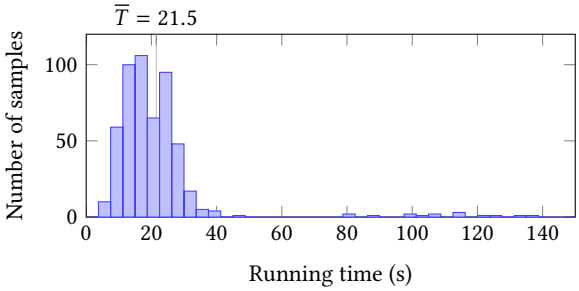

Figure 6: Efficiency evaluation results of needle.

Centroid) for comparison. The fingerprint (centroid) of a function only slightly changes over a set of program modifications, and therefore plagiaristic P_1 and P_2 tend to have many pairs of functions $f_i \in P_1$ and $f'_j \in P_2$ with a close centroid.

3.4.2 Results. Similarity scores generated by needle and Centroid are shown in Figure 4. Each curve plots the similarity scores of program pairs in a descending order. The overall similarity scores are high because a student's code is mixed with `flex` and `bison` generated codes. A typical student's work consists of ~500 lines of code, while the corresponding generated code is around 5,000 lines. It is clear that needle produces a smoother score distribution compared with Centroid.

We manually inspected all submissions (and carefully inspected code pairs that have a high similarity score produced by either tool) and identified five students involved in plagiarism (further confirmed by a code quiz). The top-4 ranked scores of the needle output (Figure 5) exactly matched these manually confirmed plagiaristic code pairs among the five students, indicating that needle is effective in code plagiarism detection. On the other hand, these four pairs are ranked in the 3rd, 262nd, 153rd, and 81st position,

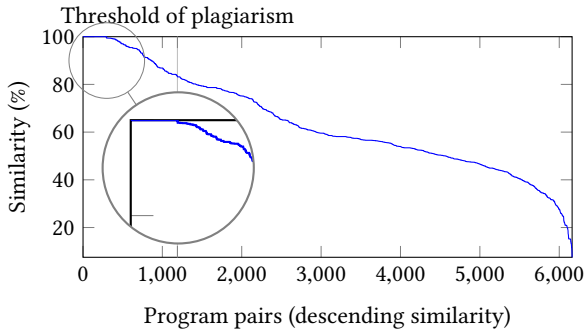

Figure 7: The similarity score distribution in the case study.

respectively by Centroid. We found that the plagiaristic copies $\{c, d, e\}$ had undergone extensively modifications, both syntactically (changing coding styles) and semantically (e.g., exchanging APIs like `strncmp` and `strcmp`, or adding functional code), which may greatly affect the effectiveness of existing plagiarism detection tools [2, 9, 14].

The efficiency evaluation results of needle are shown in Figure 6. Though the results (21.5s per pair on average) are not comparable with Centroid (<0.01s per pair), such a cost is usually affordable because plagiarism detection can be done offline by a multiprocessor machine. Considering that the programs are relatively large (~5000 lines) and needle is highly effective against detecting code plagiarism even if syntactic and semantical changes are made, such a trade-off would be beneficial in practice.

4 CODE PLAGIARISM DETECTION: AN EMPIRICAL STUDY

4.1 Case Study

We applied needle to all 79 submissions in an anonymous programming lab. (The labour work is ~500 lines of code.) No plagiarism prevention mechanism is applied in the lab, and the teaching assistant grades the submissions merely based on lab reports. This programming assignment contains neither skeleton code nor code generator, and two pieces of independent work usually have a similarity below 70%. Figure 7 shows the similarity distribution.

To our surprise, 65 of 79 (82%) submissions are plagiaristic in our manual inspection[1]. Among them, 398 pairs received a similarity score greater than 99%, covering 42 out of 79 (53%) of the submissions. Such a similarity indicates a direct copy with negligible changes. The plagiaristic copies of the least similarity score differ in some places, but there are many places of shared code and identical helper functions.

Being somewhat shocked, we interviewed the teaching assistant and students. The TA gave us the following feedbacks:

> "...Looking at their lab reports, I knew that many plagiarized. However, it would be impossible to handle such many cases ..."

And a student (participated in that class) agreed with our results:

[1]We manually checked all submissions in the ascending order of similarity (ignoring program pairs that are both marked as plagiarism) and stopped at the first false positive.

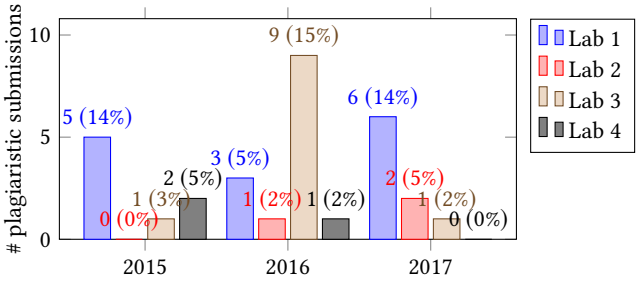

Figure 8: Statistics of confirmed code plagiarism submissions in the field study.

> "...There are some self-motivated students who worked on the assignment. However, it's too easy to get a working copy from the Internet. Some simply copied it, and the others submitted modified versions ..."

These results based on a particular set of submissions may not be generalizable to other classes. However, they do reveal that the code plagiarism issue could be much more severe than one thinks. The results also served as one of our motivations to deploy code plagiarism detection tool in practice.

4.2 A Three-Year Field Study

4.2.1 Deploying Code Plagiarism Detection Tools in Class. Realizing that code plagiarism, without proper mechanisms to prevent, can greatly threaten the academic integrity, we deployed the plagiarism detection tools in the recent years of the "Principles and Techniques of Compilers" course teaching:

(1) We let the students know in advance that code plagiarism will be detected by a computer program (and the tool is resilient to many semantics-preserving code changes) and will be seriously penalized in the class. We also offer "honest grades" that a student receives partial grades even if the submission cannot pass any test case that checks the lab requirements.
(2) We deployed both tools (needle and Centroid) for plagiarism detection. Particularly, Centroid is used to detect cross-year code plagiarisms because it scales better than needle.
(3) We manually inspected the machine-produced results to identify high-confidence plagiaristic submissions, and have face-to-face interviews (with code quiz) with these students for a final verdict.

Each year's compiler design and implementation course contains four programming labs in which students write code to translate a `C--` program into syntax tree, intermediate representation code, and finally a runnable MIPS32 binary.

4.2.2 Experience Report. The statistics of our confirmed code plagiarism submissions are shown in Figure 8. In the code interview, all students suspected for code plagiarism acknowledged except for one case. Though we have strong evidence that two programs are copies (almost identical through T1–T2 types of changes), the student denied plagiarism and claimed that the lab is done independently. We consider this case plagiarism in our study.

```
1  cur->lineno = temp->lineno;          |  head->number_signal = 0;              1
2  strcpy(cur->type, type);             |  head->line = temp->line;              2
3  cur->isLexical = 0;                  |  strcpy(head->type,type);              3
4  cur->children = temp;                |  head->child_left = temp;              4
```

```
1  $$->is_root=1;                       |  $$->final=0;                          1
2  $$->no_leaves=1;                     |  $$->num_children=1;                   2
3  $$->leaves[0]=(Node*)$1;             |  $$->children=(Node**)malloc           3
4  if(exit_error==0)                    |     (sizeof(Node*)*$$->                4
5    {print_tree($$,0);}                |           num_children);
                                        |  $$->children[0]=(Node*)$1;            5
                                        |  if(!wrong)                           6
                                        |    printNode($$,0);                   7
```

```
1  temp->line = a->line;                |  p_node->left_child = temp;            1
2  temp->lChild = a;                    |  p_node->line = temp->line;            2
3  while(num > 1){                      |  for(int i=0;i < num-1;++i){           3
4    a->rChilds = va_arg(list,          |    temp->right_child =                 4
           node*);                      |      va_arg(valist,struct              5
5    a = a->rChilds;                    |           Node*);
6    num--;                             |    temp = temp->right_child;           6
7  }                                    |  }                                     7
```

Figure 9: Plagiaristic code snippets (left v.s. right).

It is a surprise that even if we made strong claim in class that code plagiarism will be seriously penalized and the existence of honest grades, *many* students still risked themselves for unethical grades. In a single lab, there can be as many as 15% project groups involved in plagiarism. In a semester, ~20% of the project groups have academic dishonest issues. Since the plagiarism detection tools may have false negative reports (only program pairs of a high similarity are manually checked), this number could be even higher.

Ten of 31 plagiaristic submissions are code clones from previous semester's submissions. Such code is obtained either from personal contact or the public domain. Despite we emphasize that publishing code is prohibited in class, we still found students making their code publicly available on `github` or personal blogs.

Almost all submissions underwent various kinds of code changes to make the copies looking drastically different at first glance. Components that could be modified without affecting the overall functionality (assertions, debugging messages, etc.) are most extensively changed in a plagiaristic copy. Some plagiarizers even take time to apply semantics-preserving local changes (T4 changes like statement moving, loop refactoring, or even changing some underlying data structures) like the examples in Figure 9, making syntactic-based techniques difficult to detect. Nevertheless, the core algorithm and data structures are rarely changed because tentative modifications to them can easily break down the program. These code fragments helped us finally reach a verdict of code plagiarism.

We also found that a plagiarizer may sometimes copy an intermediate version or manually inject faults into the program, yielding two copies to diverge in many places and have different grades. In this case, it would be extremely difficult for a human (e.g., a teaching assistant) to identify plagiarism merely by looking at the code and grades.

There is only one case that a student plagiarized code more than once in a semester. (Two of four submissions are plagiaristic, and the other two do not compile.) Other students involved in plagiarism either tried their best surviving the subsequent labs, or simply gave up trial and just received the honest grades (forced to work harder to pass the final exam). Therefore, we believe that deploying code

plagiarism detection tools are indeed beneficial in maintaining the academic integrity in class.

5 FINDINGS AND DISCUSSIONS

5.1 Lessons Learned

The single most important finding, yet maybe controversial, is that *plagiarism is a seductive way for students to obtain unethical grades*, at least for our students in our current curriculum setting. Without proper mechanisms of control, a programming practicum would be totally corrupted: only a few self-motivated students do their independent work. Others simply copy, change, and receive grades.

Deploying code plagiarism detection tools and introducing honest grades did alleviate the code plagiarism issue, but plagiarism cannot be easily eliminated:

(1) In our field study, there are still ~20% (or more considering the possibility of false negatives) of the students plagiarized. Worse, students tried to trick the plagiarism detection tool by performing various kinds of code changes.
(2) Though the changes are usually of T1–T4 types and can be detected by a tool, a changed copy may look drastically different, mainly due to code movements and coding style changes. It would be difficult for a human instructor to identify them without the aid of a plagiarism detection tool.
(3) Cross-semester plagiarism is also a common issue (32% in our study). This is consistent with the results presented by Pierce and Zilles [13]. We recommend that an instructor should keep a database for tracking students' submissions if a programming assignment changes little over time.

Through our interviews with researchers, instructors, teaching assistants, and students, *code plagiarism is a problem that "everyone realizes but everyone chooses to ignore."* The maintenance of academic integrity, being orthogonal to the curriculum design, remains nearly blank in our curriculum system: all programming labs studied in this paper are from third-year undergraduate courses, and the students may have already plagiarized in previous classes. We hope that the results presented in this paper could serve as a wake-up call to the future research of code plagiarism prevention.

5.2 Challenges to be Addressed

Any post-mortem plagiarism detection faces some common challenges: it is not only difficult to confirm that a modified code is truly plagiaristic but also cheap for a student to deny the dishonesty. As suggested by Wagner [15], it would be better to design mechanisms to reduce plagiarism. How to design such effective mechanisms remains as our future work.

Plagiarism detection tools work well for programming assignments of medium or large sizes. However, when applied to smaller programs, false positive and false negative rates may both increase, and it is even harder to give verdict on plagiarism. On the other hand, academic integrity is particularly important in these assignments (usually in entry-level courses). Tracking the process of code writing maybe a possible solution.

Students may also have other types of academic dishonesty. Some students acknowledged that they did not plagiarize code, but they used a golden version (usually from another student) to cross-check their outputs. Such dishonesties are even more difficult to prevent.

6 CONCLUSIONS

This paper present the needle algorithm for automated code plagiarism detection. Leveraging the power of compiler optimization and binary embedding measurement, needle can detect plagiaristic code undergone by many semantics-preserving changes. The empirical study results indicate that plagiarism can greatly threaten the academic integrity, and applying plagiarism detectors alleviates the problem, and however, challenges still remain.

ACKNOWLEDGMENTS

This work is supported in part by National Natural Science Foundation (Grants #61690204, #61472174) and National Basic Research 973 Program (Grant #2015CB352202) of China. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Tech. and Industrialization, Jiangsu, China.

REFERENCES

[1] Fox Butterfield. Scandal over cheating at mit stirs debate on limits of teamwork. *The New York Times*, (May 22):A23, 1991.
[2] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *Proc. of the Int'l Conf. on Software Engineering*, ICSE, pages 175–186, 2014.
[3] Christian Collberg, Ginger Myles, and Michael Stepp. Cheating cheating detectors. Technical Report TR-04-05, University of Arizona, 2004.
[4] Christian Domin, Henning Pohl, and Markus Krause. Improving plagiarism detection in coding assignments by dynamic removal of common ground. In *Proc. of Human Factors in Computing Systems*, CHI EA '16, pages 1173–1179, 2016.
[5] Steve Engels, Vivek Lakshmanan, and Michelle Craig. Plagiarism detection using feature-based neural networks. In *Proc. of the SIGCSE Technical Symposium on Computer Science Education*, SIGCSE, pages 34–38, 2007.
[6] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proc. of the Int'l Conf. on Software Engineering*, ICSE, pages 321–330, 2008.
[7] Sam Grier. A tool that detects plagiarism in Pascal programs. In *Proc. of the SIGCSE Technical Symposium on Computer Science Education*, SIGCSE, pages 15–20, 1981.
[8] Jurriaan Hage, Peter Rademaker, and Nikè van Vugt. Plagiarism detection for Java: A tool comparison. In *Computer Science Education Research Conf.*, CSERC, pages 33–46, 2011.
[9] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proc. of the Int'l Conf. on Knowledge Discovery and Data Mining*, KDD, pages 872–881, 2006.
[10] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proc. of the Int'l Symposium on Foundations of Software Engineering*, FSE, pages 389–400, 2014.
[11] Leonardo Mariani and Daniela Micucci. AuDeNTES: Automatic detection of tentative plagiarism according to a reference solution. *ACM Transactions on Computer Education*, 12(1):2:1–2:26, March 2012.
[12] Neurohazard. Defeat Moss plagiarism detection system. https://blkstone.github.io/2017/03/09/defeat-moss-plagiarism-detection-system/, 2017.
[13] Jonathan Pierce and Craig Zilles. Investigating student plagiarism patterns and correlations to grades. In *Proc. of the SIGCSE Technical Symposium on Computer Science Education*, SIGCSE, pages 471–476, 2017.
[14] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. of the Int'l Conf. on Management of Data*, SIGMOD, pages 76–85, 2003.
[15] Neal R. Wagner. Plagiarism by student programmers. http://www.cs.utsa.edu/~wagner/pubs/plagiarism0.html, 2000.