Towards Robust Plagiarism Detection in Programming Education: Introducing Tolerant Token Matching Techniques to Counter Novel Obfuscation Methods

Robin Maisch
KASTEL Karlsruhe Institute of Technology
Karlsruhe, Germany
robin.maisch@kit.edu

Nathan Hagel
KASTEL Karlsruhe Institute of Technology
Karlsruhe, Germany
nathan.hagel@kit.edu

Alexander Bartel
Dep. Information Management
Neu-Ulm University of Applied Sciences
Neu-Ulm, Germany
alexander.bartel@hnu.de

Abstract

With the rise of AI-generated code, programming courses face new challenges in detecting code plagiarism. Traditional methods struggle against obfuscation techniques that modify code structure through statement insertion and deletion. To address this, we propose a novel approach based on tolerant token matching designed to enhance resilience against such attacks. We evaluate our method through three experiments on a real-life dataset with AI-obfuscated plagiarisms. The results show that our approach increased the median similarity gap between originals and plagiarisms by 1 to 6 percentage points.

CCS Concepts

• Software and its engineering; **• Social and professional topics → Software engineering education**; **Computer science education**; **• Information systems → Near-duplicate and plagiarism detection**;

Keywords

Software Plagiarism Detection, Source Code Plagiarism Detection, Plagiarism Obfuscation, Obfuscation Attacks, Code Normalization, Tokenization, Computer Science Education

ACM Reference Format:
Robin Maisch, Nathan Hagel, and Alexander Bartel. 2025. Towards Robust Plagiarism Detection in Programming Education: Introducing Tolerant Token Matching Techniques to Counter Novel Obfuscation Methods. In *ECSEE 2025: European Conference on Software Engineering Education (ECSEE 2025), June 02–04, 2025, Seeon, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3723010.3723019

1 Introduction

In programming courses, especially at universities and schools, the students typically are assigned programming tasks to demonstrate their learning progress. In addition to functional and stylistic grading, it is then the lecturer's task to check the submitted programs for plagiarism. As the number of submissions increases with growing classes of computer science students [5], the quadratic number of pairs to compare quickly exceeds the resources of the lecturers [19, 43]. For this reason, automatic code plagiarism detection has been researched since the 1980s [11].

Students perform various strategies to disguise their plagiarism by modifying submissions [23, 34]. Until recent years, these *attacks* on plagiarism detection systems would typically have been done manually or with the aid of automatic tools [13, 32]. Widely used attacks include renaming, reordering independent code, or modifying comments [23]. State-of-the-art plagiarism detection systems such as MOSS [1], or JPlag [28] have been adapted according to these challenges and are very effective in defending against even complex obfuscation attack patterns [22, 35, 36].

However, software plagiarism detection in education is exposed to a new challenge with new capabilities of Large Language Models (LLMs), especially for programming tasks [9]. They create a new opportunity for students to circumvent the trouble of developing a working solution independently [7]. Although the usage of LLMs in introductory programming courses can have a positive impact on learners [9, 18], the goal of programming courses is to convey the skills necessary to solve programming tasks without the help of LLMs. In this context, LLM-obfuscated plagiarized submissions seriously threaten academic integrity, as plagiarism detection systems are not designed to detect them. In fact, neither tools, nor AIs, nor humans can reliably detect LLM-generated code as of yet [27]. However, there is first research indicating that even though AI assistants use randomness to vary their responses, the code generated for the same user prompt is typically similar throughout multiple runs [25].

The potential use of LLMs creates two challenges for programming classes and lecturers: *(I)* the detection of LLM-obfuscated plagiarized solutions, and *(II)* the detection of fully LLM-generated submissions. This paper targets the first challenge. Our main research question is as follows: *How can the automated detection of AI-obfuscated plagiarized programming submissions be improved?* This paper presents Similarity-Aware Token Matching (SATM), a novel approach to improve plagiarism detection, especially in the context of AI-obfuscated submissions. SATM is designed as a language-independent measure against a broad range of attack schemes, leveraging sequence alignment techniques from bioinformatics. The following five contributions are described in this paper:

ECSEE 2025, Seeon, Germany
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1282-1/25/06
https://doi.org/10.1145/3723010.3723019

- C1 We introduce a similarity metric for token types.
- C2 We adapt the comparison algorithm to also match sufficiently similar token types, not only equal token types.
- C3 We introduce an *indel penalty* value for token types.
- C4 We adapt the comparison algorithm to *recover* from a mismatch by storing away tokens into a buffer, from where they can be restored later.
- C5 We evaluate our approach on datasets which contain plagiarized submissions obfuscated by prompting an AI to apply various levels of obfuscation attacks.

This proposed approach promises several benefits:

- It increases the precision of the code similarity metric by using more specific structural information while still detecting the same matches, at least.
- It is a strong basis for cross-language plagiarism detection, where submissions of different programming languages are compared.
- It is a natural extension of the current approach: The behavior of existing language modules does not change despite the modification in the core comparison algorithm.

We will proceed by introducing relevant foundations in Section 2 and related work in Section 3. Section 4 presents the approach in detail, which is evaluated in Section 5. Finally, Section 6 discusses the findings and Section 7 concludes this work.

2 Foundations

This section covers the foundations relevant for the presented work. First, we introduce the fundamental problem of code plagiarism (Section 2.1), then we present the state-of-the-art approach of code plagiarism detection, which is token-based plagiarism detection (Section 2.2), give an overview over common plagiarism obfuscation types and discuss current developments in the way students plagiarize code (Section 2.3). Lastly, we turn to bioinformatics, where similar challenges occur, and briefly describe the relevant approaches in that field that inspired this work (Section 2.4).

2.1 Code Plagiarism

In software engineering education, mandatory programming assignments are a popular instrument to encourage students to familiarize themselves with the crucial concepts of programming. Due to reasons like poor time management, low esteem in one's own abilities, or fear of failure [38], it frequently occurs that students copy the submission off of a fellow student, modify it to some extent, and submit it as their own. This is one variant of what constitutes source code plagiarism [6]. As instructors must ensure academic integrity, code plagiarism at universities has been a significant concern for more than 45 years [37].

Early software plagiarism detection research already recognized that, in order to achieve resiliency against plagiarism obfuscation, it was necessary to choose an appropriate abstraction from the text content of source code files as a basis for the comparison. Due to limited computational resources, vectors of software metrics were a prevalent option [11].

More recently, tools based on isomorphism of code graphs have also been presented [20]. Even though the severe time complexity of graph isomorphism detection is greatly mitigated in typed graphs such as code graphs, and additional runtime optimizations have been developed, it seems that no graph-based plagiarism detection system has reached the maturity to be used in practice beyond research.

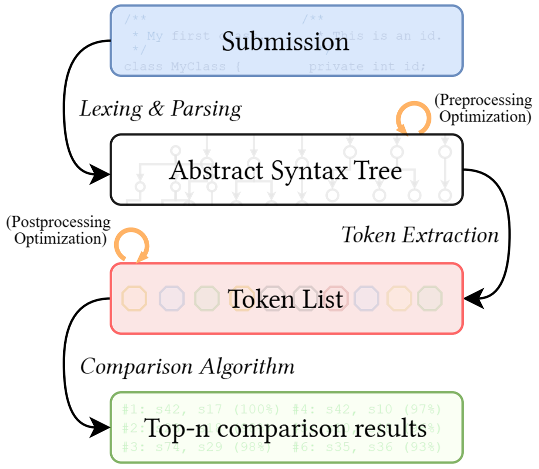


Figure 1: The pipeline of a token-based plagiarism detection system [28].

2.2 Token-based Plagiarism Detection

The textual content of code, such as class names, identifiers, comments, and formatting, is very easy to change without changing the behavior of the program; this is why such textual modifications are a popular strategy to make a plagiarism less obvious to a human reader [24]. The fundamental idea of token-based plagiarism detection is to use the *structural* information of the code instead which is much more complex to successfully manipulate while preserving the program behavior.

This section explains the state of the art of token-based plagiarism detection. The complete detection pipeline, including the submission of source code in blue, the lexing and parsing of the abstract syntax tree in white, the creation of a token list in red, and finally, the result as a list of the most similar submission pairs in green, is depicted in Figure 1.

2.2.1 Lexing and Parsing. First, the code is traversed character by character to determine its building blocks (*lexing*) and to extract its structure from the order of the elements (*parsing*). The specification for both of these processes is given by the *grammar* of the respective programming language. For each code element, a *node* is created and placed in an abstract syntax tree (AST), a hierarchical data structure, similar to how a natural language sentence can be iteratively divided into clauses, phrases, words, and morphemes. The finished AST contains sufficient information for the subsequent steps; however, approaches that apply *preprocessing optimizations* may extend the AST with additional information, e.g. connections between all occurrences of the same variable in a method, or between program statements that influence each other. The enriched code tree may assume the form of a *code graph* such as a program dependency graph (PDG) [12, 35] or a code property graph (CPG) [22, 42].

2.2.2 Token extraction. The AST of each submission is traversed in depth-first manner, i.e., in an order that closely resembles how the elements are arranged in the code. Upon *entering* or *leaving* a node representing a structurally relevant element, such as a method body, a *token* is created that captures the occurrence of that element in its *token type* (e.g., `METHOD_BODY_BEGIN` and `METHOD_BODY_END`). The tokens generated for a submission are collected in a *token list* (see Figure 1 red), an ordered list which serves as a linear representation of the non-linear code structure.

To reiterate, each token encodes a specific structural element in its *token type*, usually as an integer. In contrast to structural information, all textual information of the code is discarded in this process; therefore, the token extraction by itself provides immunity against textual plagiarism obfuscation attacks[1].

However, less important structural details of the code may also be discarded during the tokenization so as to not clutter the token list. *Token selection* is the process of defining the set of token types considered relevant for a specific programming language. This is an early step in the design process of token-based plagiarism detection with significant implications for which kinds of attacks the detection will be immune or vulnerable to. For example, distinct but similar elements may be mapped to the same token type (such as an assignment via the = operator, and via the shorthand += operator), so that substituting one by the other element does not affect the token list. Other elements might be excluded during tokenization altogether because they are not deemed structurally relevant (such as primitive operations or reading access to a variable) and would allow insignificant code changes to greatly affect the token list.

2.2.3 Token list comparison. The token list is the intermediate representation that is used as the basis for the submission comparison. The long-time state-of-the-art comparison algorithm, Greedy String Tiling with Rabin-Karp Matching (GST-RKM) [40], works by iteratively identifying only the longest matching subsequences (*matches* of length ℓ) that are still unmarked, discarding all shorter matches (of length $< \ell$), and marking a non-overlapping subset of those longest matches, indicating that these tokens are now assigned to a match and cannot be considered again. After that, the search starts again, but naturally, the new longest matches will be shorter than length ℓ. This loop continues until the longest matches fall below a predefined minimum length ℓ_{min}.

Rabin-Karp matching is an optimization that, given a starting position in the left sequence, uses hashing to quickly find the positions of possibly promising matches in the right sequence, thus drastically reducing the number of attempts to find matches.

Finally, the overall similarity value of two submissions is calculated as a percentage value from the total length of all matches relative to the length of the two token lists. The output of the plagiarism detection pipeline is a top-*n* ranked list (see Figure 1 green) of the most similar pairs of submissions.

2.2.4 Human inspection. The overall similarity in a corpus of submissions is not only influenced by malicious intent of students, but also by, among others, the task, its description, and the initially limited set of language features that the students know. Therefore, a plagiarism detection system can never make a binary classification of plagiarism; a human inspector still must decide which pairs of submissions are suspicious. For this inspection, modern plagiarism detection systems offer side-by-side code viewers to allow for a better understanding of the measured similarity (see Figure 2).

Level	Subject to Change	Examples
L0	– no changes –	
L1	Comments, Whitespace	
L2	Identifiers	Rename variable, rename class
L2.5	Packages, Imports	Import vs. fully qualified classes
L3	Declarations	Location, visibility, order, assigned value of declaration; dummy variables
L4	Program modules	Extracted methods; dummy methods
L5	Program statements	Method calls; data types; operators, control structures; order of operands
L6	Decision logic	Introduction of control structure; loop vs. recursion; range of loop variable

Table 1: Classification of plagiarism obfuscation attacks [11, 17].

2.3 Plagiarism Obfuscation Attacks

In the context of code plagiarism detection, systematic approaches to avoid detection are referred to as *plagiarism obfuscation attacks* on plagiarism detection systems.

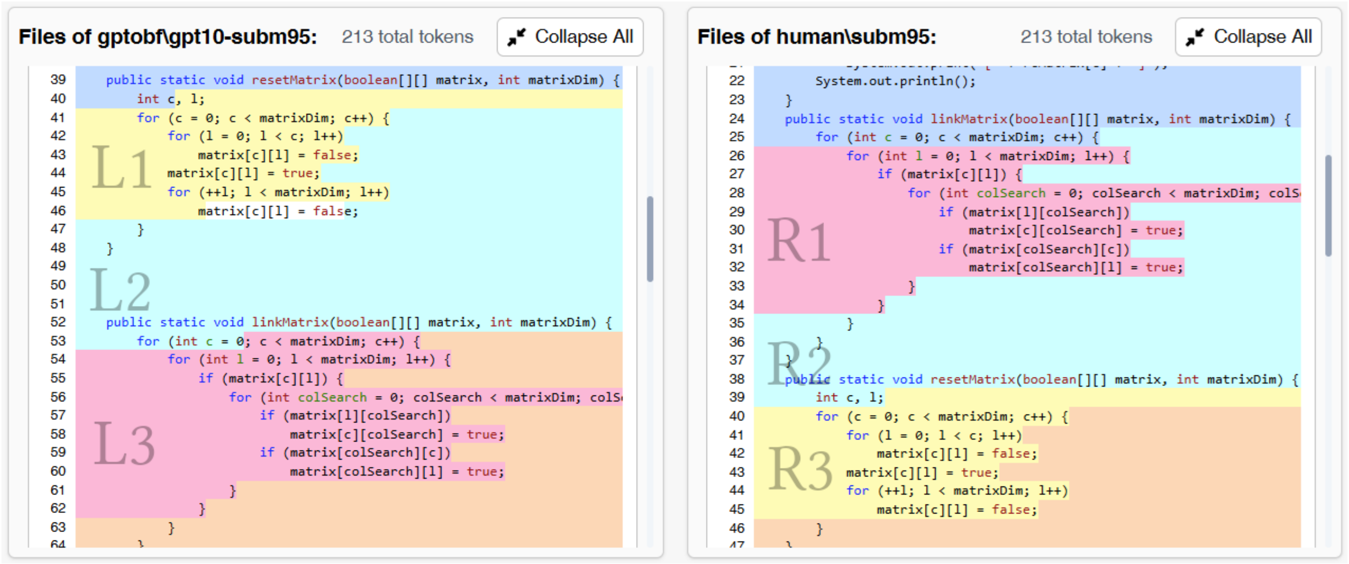
2.3.1 Classes of obfuscation attacks. Faidhi and Robinson [11] present the most common taxonomy of "transformation levels" which was later adapted by Karnalim [17], see Table 1, who lists more than 50 concrete plagiarism strategies across the seven levels. The taxonomy covers very basic *lexical* obfuscation attacks (Table 1, L1, L2), *data-based* attacks, then more advanced strategies like *structural attacks* (Table 1, L5) as well as *complex* attacks such as refactorings and transformation of the control flow (L6).

In the domain of *code cloning*, which is concerned with duplicate code inside the same project, another taxonomy is prevalent [8, 29], which focuses on a number of independent dimensions of how the code may vary from the original: textual, semantic, structural, functional, and model-based.

2.3.2 Obfuscation attack workflow. In times of *manual plagiarism obfuscation*, it has been a fundamental assumption of code plagiarism detection researchers that advanced programming skills and great effort are required to successfully obfuscate a plagiarism. An effective plagiarism would therefore be proof of more than satisfactory skills in terms of the learning goals of an introductory programming lecture. However, it was considered much more likely that students who possess such capabilities prefer to just complete the task on their own [15].

More recently, *automated obfuscation attacks* such as Mossad [10] were presented that severely threatened this assumption, as they require little skill and effort to use for weaker students, and proved effective against the state of the art of token-based plagiarism detection. However, it may be noted that submissions obfuscated by

[1]According to the classification of plagiarism obfuscation attacks which we reference later (Table 1), textual obfuscation attacks are covered by Levels 1 and 2.



```
    public static void resetMatrix(boolean[][] matrix, int matrixDim) {
        int c, l;
        for (c = 0; c < matrixDim; c++) {
            for (l = 0; l < c; l++)
                matrix[c][l] = false;
            matrix[c][l] = true;
            for (++l; l < matrixDim; l++)
                matrix[c][l] = false;
        }
    }



    public static void linkMatrix(boolean[][] matrix, int matrixDim) {
        for (int c = 0; c < matrixDim; c++) {
            for (int l = 0; l < matrixDim; l++) {
                if (matrix[c][l]) {
                    for (int colSearch = 0; colSearch < matrixDim; colS
                        if (matrix[l][colSearch])
                            matrix[c][colSearch] = true;
                        if (matrix[colSearch][c])
                            matrix[colSearch][l] = true;
                    }
                }
            }
```

```
        System.out.println();
    }
    public static void linkMatrix(boolean[][] matrix, int matrixDim) {
        for (int c = 0; c < matrixDim; c++) {
            for (int l = 0; l < matrixDim; l++) {
                if (matrix[c][l]) {
                    for (int colSearch = 0; colSearch < matrixDim; colS
                        if (matrix[l][colSearch])
                            matrix[c][colSearch] = true;
                        if (matrix[colSearch][c])
                            matrix[colSearch][l] = true;
                    }
                }
            }
        }
    }
    public static void resetMatrix(boolean[][] matrix, int matrixDim) {
        int c, l;
        for (c = 0; c < matrixDim; c++) {
            for (l = 0; l < c; l++)
                matrix[c][l] = false;
            matrix[c][l] = true;
            for (++l; l < matrixDim; l++)
                matrix[c][l] = false;
        }
```

Figure 2: The JPlag Report Viewer shows two submissions side-by-side, visually highlighting matches. In this particular section, a reordering attack was successfully detected: The outer `for` loop of the method `resetMatrix` (Figure 2, L1) was matched to its counterpart on the right (Figure 2, R3), as well as the body of the method `linkMatrix` (Figure 2, L3) to the corresponding part in the right submission (Figure 2, R1).

MossAD may increase in length in orders of magnitude, which is likely to draw the attention of a human inspector. Also, an effective post-processing approach to normalize these submissions back to their original form has been developed [35].

AI assistants are the most recent type of tools that provide plagiarism obfuscation. The accessibility, ease of use, performance and effectiveness of AI-based plagiarism seem to far exceed those of dedicated algorithmic tools and pose an unseen threat to the authentic assessment of programming skills in academia. We distinguish *AI-based submission obfuscation*, entering an existing plagiarized submission into an LLM with the prompt to rewrite it so as to make it unrecognizable, from *AI-based submission generation*, where the task description is provided to the LLM with the prompt to create a working solution for it.

2.4 Sequence Alignment

In bioinformatics, *sequence alignment* is a task with a similar setup as code similarity detection but with a different goal. Here, tokens may represent nucleotides of a DNA sequence or amino acids of a long-chain protein. Rather than searching for a *global* match of a pair, *local* alignment algorithms aim to identify *meaningful, interesting* matching parts of sequences that may indicate the function of particular genome regions, for example [4]. In this context, insertions and deletions (*indel* operations) and substitution of elements in the sequences are tolerated as they may naturally occur through evolution. The relative frequency of protein substitutions occurring has been recorded in various *substitution matrices* [14].

When a matching algorithm arrives at a mismatch in the sequence, the probability of the substitution given in that matrix is considered. If the pair is included in a match, the *similarity score* of the match is reduced to account for the substitution [14].

3 Related Work

Normalization in Plagiarism Detection. Many recent contributions [22, 23, 35, 39] aim towards *normalizing* the submissions, i.e., systematically altering the code in order to remove variation from it and to increase its meaningfulness, thereby eliminating space for obfuscation attacks if done rigorously. Normalization can be performed in different phases of the plagiarism detection pipeline, which is displayed in Figure 1.

Common code omission aims to remove code parts from the submissions that regularly occur, but contribute little originality, e.g. getters, setters, trivial constructors, and library initialization code from the AST. This leaves more meaningful code parts for the comparison [23, 39] but requires the instructor to identify an appropriate set of common code parts to remove.

Refactoring normalizations work on *L*-*R*-pairs of equivalent code structures. In a fixpoint procedure, instances of *L* code structures (e.g., a `for` loop) are detected and transformed to *R* code structures (e.g., a `while` loop), reducing variability and defending effectively against replacement attacks. However, refactoring transformations require language-specific information and extensive code analyses like name analysis and type analysis. Also, the complexity of this approach increases with the number of implemented *L*-*R*-pairs. This has been evaluated on *code property graphs (CPGs)* [22].

Reordering normalizations impose a total order on independent statements that could be permutated inside their method without changing its behavior. A statement dependency analysis and language-specific information is required to decide which statements must stay in order relative to another to preserve the semantics of the code. This approach has been evaluated on code graphs [22] as well as token graphs [35].

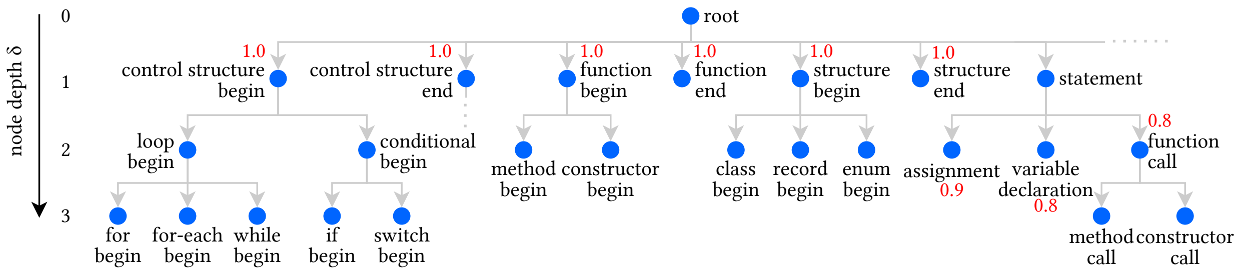


Figure 3: Tree of token types of Java, annotated with their *indel* penalty value (red).

Match merging is a heuristic, language-agnostic approach to defend against various types of obfuscation attacks [36]. If the comparison algorithm identifies two matching parts in one code file that are separated only by a few tokens in both submissions, then the match merging algorithm may decide to combine these matches to one *despite* the mismatching portion. This approach has proven effective against attacks like dead code insertion, statement permutation, and AI-based obfuscation.

These approaches require extensive analyses to determine the function of a method, or whether the application of a refactoring is semantic-preserving. The additional preprocessing pays out, however, as all of these approaches have proven effective against the respective types of obfuscation attacks for which they were designed.

Naturally, normalization leads to information loss. If two submissions are similar even before normalization, that is a strong hint towards plagiarism that does not appear in the normalized data. Inspired by the sequence alignment strategies from bioinformatics, we want to explore a new approach where we accept substitutions of similar, but non-equal constructs as a match and decrease the similiarity score of a match to reflect the substitution.

4 Approach

The following section presents our approach Similarity-Aware Token Matching (SATM), which allows for tolerance towards substitution, indel operations, and permutation in source code submissions.

4.1 A Similarity Metric for Token Types

Similar to a substitution matrix, we define a similarity metric on token types to decide which token types may be allowed to be substituted in a match, and how that affects the similarity score of the match.

To define the similarity between all pairs of token types, we arrange them in a tree structure, i.e., a hierarchy where semantically similar token types are grouped together in a subtree, see Figure 3. The tree induces a parent function $p(n)$, and a *depth* function $\delta(n)$ for all nodes n, which indicates the distance of n to the *root* node, starting at $\delta(root) = 0$, and $\delta(n) = \delta(p(n)) + 1$ for all other nodes. Lastly, the lowest common ancestor (LCA) $a := lca(n_1, n_2)$ of two nodes n_1 and n_2 is the node a of the greatest depth out of all nodes which are a parent of (or equal to) n_1 and n_2. Our similarity metric uses the LCA to express how distant two token types are in the tree.

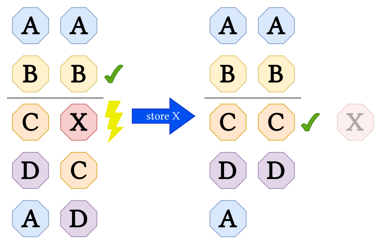


Figure 4: Recovery after a insertion attack.

It is defined as follows:

$$similiarity(t_1, t_2) := \frac{2 * \delta(lca(t_1, t_2))}{\delta(t_1) + \delta(t_2)},$$

This definition is based on the *Jaccard coefficient*, which allows a range of 0 (iff $lca(t_1, t_2) = root$) to 1 (iff $t_1 = t_2$).

Example. According to the tree structure depicted in Figure 3, the similarity of the token types *for begin* and *method begin* is $\frac{2*0}{3+2} = 0$, because their LCA is *root*. Similarly, the similarity of *for begin* and *while begin* is $\frac{2*2}{3+3} = \frac{2}{3}$, as their LCA is *loop begin*.

4.2 Similarity-tolerant Matching

Our first goal is to make token-based plagiarism detection more resilient against semantic-preserving substitution of code elements. The original state-of-the-art comparison algorithm, Greedy String Tiling with Rabin-Karp Matching (GST-RKM) [40], compares two token sequences element-wise, searching for the longest remaining match. When the algorithm arrives at a position where two tokens are non-equal, the match is terminated, see Figure 6a. Inspired by sequence matching strategies in bioinformatics, we extend the algorithm by allowing non-equal token types to be matched if their similarity is greater than a set threshold, see Figure 6b. For each pair of tokens that is accepted as a match, their similarity value is added to the *score* of the match. If a match contains non-equal (but similar) tokens, this is reflected in the fact that the score is less than the length of the match. Thus, more similar matches are preferred over less similar matches.

4.3 An *Indel* Penalty Value for Token Types

Some token types represent code elements that contribute to the overall structure of the code more significantly than others; thus, it

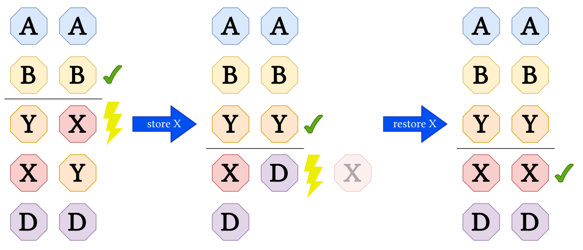


Figure 5: Recovery after a permutation attack.

```
score = 0;
while (left == right) {


    score += 1;
    // advance to next token
    left.consume();
    right.consume();
}
return score;
```

(a) Traditional matching.

```
score = 0;
loop {
    similarity = sim(left, right);
    if (similarity <= threshold)
            break;
    score += similarity;
    // advance to next token
    left.consume();
    right.consume();
}
return score;
```

(b) Type-tolerant matching.

Figure 6: Approaches to token matching.

```
1   score = 0;
2   loop {
3       if (similarity > threshold) {
4           maxLeft.consume(); // if buffered token: remove from buffer
5           maxRight.consume(); // else: advance to next token
6           score += similarity;
7       } else {
8           recoveryScore = recover(left, right);
9           if (recoveryScore > 0) { // recovery successful
10              score += recoveryScore;
11          }
12          else break; // unrecoverable mismatch
13      }
14  }
15  return score;
16
17  procedure recover(left, right) {
18      recoveryScore = 0;
19      bufferLeft = []; bufferRight = []; // token buffers
20      while (PENALTY_THRESHOLD < recoveryScore <= 0) {
21          from (left, right), (bufferLeft.peek(), right), (left, bufferRight.peek()),
22              choose pair (maxLeft, maxRight) with greatest similarity
                    (similarity)
23          if (similarity > threshold) {
24              maxLeft.consume(); // if buffered token: remove from buffer
25              maxRight.consume(); // else: advance to next token
26              recoveryScore += similarity;
27          } else {
28              from left and right, choose token type t with the smallest
29                      indel penalty
30              if (t.indelPenalty() == MAX_PENALTY) // buffering prohibited
31                  break;
32              if (t == left) {
33                  left.consume(); bufferLeft.append(t);
34              } else {
35                  right.consume(); bufferRight.append(t);
36              }
37              recoveryScore −= t.indelPenalty() ∗ PENALTY_FACTOR;
38          }
39      }
40      // recovery was successful if recoveryScore > 0
41      return recoveryScore;
42  }
```

(c) Type-tolerant matching with two buffers for match recovery.

Figure 6: Approaches to token matching (cont.).

is a more severe modification to insert or delete a token of such a token type. To encode this, we provide an *indel penalty value* for each token type as shown in Figure 3 in the interval $[0, 1]$.

Our general assumption is that the borders of elements which entail code blocks, such as method definitions and loops, should be modified only as a last resort, if at all—whereas atomic statements, such as assignments and variable definitions, should be allowed to be modified rather freely. Lastly, for token types representing elements with little structural importance, like annotations or import statements, the *indel* penalty should be negligible.

4.4 *Indel*-tolerant Matching

We now adapt the comparison algorithm again to make it resilient against *indel* operations. Indel operations typically create mismatches in the token sequence with the intention to decrease the overall similarity of a plagiarized submission to its original. An example is illustrated in Figure 4, where the code of the original has been modified so that an `X` token was inserted into its token sequence (left column) to create an obfuscated variant (right column). In the variant, the matching subsequence `ABCD` is split up and would not be recognized by a comparison algorithm. In the illustration, the comparison algorithm has reached the pair `C` and `X`, which are not similar to each other. Now, to try to recover the match, the adapted algorithm (see Figure 6c) chooses the token with the lower *indel* penalty (see Figure 6c, line 28), stores it in a list, and advances to the next position, keeping the token with the higher *indel* penalty. To capture the influence of the recovery mechanism, the current match score is reduced by a multiple of the indel penalty of the deleted token (see Figure 6c, line 37). This way, the comparison algorithm will prefer a more accurate match to a less accurate match. The mechanism is also effective against code permutation attacks as displayed in Figure 5: After a token has been stored away (left), it may be inserted again at a position where it matches its counterpart better than the current token of the sequence (see Figure 6c, line 21). In the example, an `X` token occurs on the left, which is dissimilar to the `D` token on the right, but it equals the stored `X` token perfectly; therefore, it is restored. Again, the total score from a match with a permutation will be lower than the score of a perfectly matched sequence. The mechanism also works if the permutation occurs over a bigger span.

This mechanism has multiple adjustable parameters:

- The *similarity threshold* sets the minimum similarity value of two tokens that shall be accepted as a match (see Figure 6c, lines 3, 23).
- The *penalty threshold* controls how quickly the recovery fails when multiple tokens are buffered one after another (see Figure 6c, line 20).

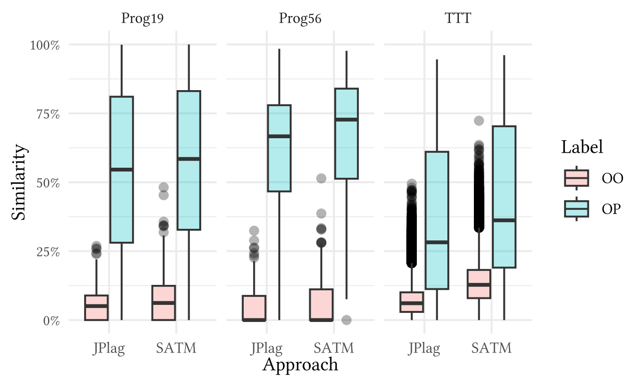


Figure 7: Similarity metrics of the three datasets as measured by the base approach (“JPlag”) and the SATM approach (“SATM”)

- The *maximum penalty* controls which token types should never be buffered, e.g., the end of a class (see Figure 6c, line 30).
- The *penalty factor* controls how many matching pairs are needed to compensate for one *indel* operation (see Figure 6c, line 37).

5 Evaluation

The following section presents the conducted evaluation of the proposed SATM approach. As the baseline, we use the state-of-the-art plagiarism detection system JPlag [28], which is free, open-source [16], and widely used. For the experiment, we used three real-world datasets of code submissions from students as well as AI-obfuscated variants, which amounts to almost 200’000 pairwise comparisons. We provide the code and the evaluation data in a replication package [21].

5.1 Methodology

For the evaluation, we use the Goal-Question-Metric (GQM) method [2, 3]. We set the following goal, question, and metrics:

G Mitigate the impact of AI-based obfuscation attacks by improving plagiarism detection.
Q How does SATM affect the detection of plagiarized submissions obfuscated through AI-based obfuscation attacks?
M Similarity metrics with SATM enabled and disabled

To mitigate the impact of a broad plagiarism obfuscation attacks, with AI obfuscation attacks in mind particularly, we aim to increase the *gap* of the calculated similarity metrics between pairs of original submissions (where low similarity is desired) and pairs of originals and derived plagiarized submissions (where we want to achieve high similarity). Ideally, the pairs of plagiarisms would clearly be set apart from the pairs of original submissions in the similarity distribution, so that an inspector could easily distinguish both groups.

5.2 Experiments

Three different datasets were used for the experiments that were conducted. The datasets stem from Java code submissions from an educational context. Each dataset contains submissions for a different programming assignment, featuring varying levels of complexity both in terms of skill required to solve it as well as the size of the solutions. We choose two datasets from PROGpedia [26], task 19 and task 56. Task 19 required the participants to implement a graph data structure as well as a suiting search algorithm, while the topic of task 56 were spanning trees and Prim’s algorithm. Out of all tasks in the PROGpedia collection, these tasks had the greatest number of compilable solutions written in Java. The third dataset, taken from Sağlam et al. [35], required the students to design an interactive TicTacToe game with a command-line user interface. Before the experiments, the datasets were pruned to remove submissions with syntax errors, because JPlag requires syntactically correct code to generate the AST. This results in the following datasets:

PROGpedia 19: 27 submissions with mean size of 131 LOC
PROGpedia 56: 28 submissions with mean size of 85 LOC
TicTacToe: 626 submissions with a mean size of 236 LOC

To investigate how SATM affects the detection of AI-obfuscated plagiarism, we chose 15 different prompts that request from the assistant to create an obfuscated plagiarism of a provided solution. The set of prompts contained varying levels of specificity as to how the code should be changed, ranging from very general prompts such as: "Can you change the code so it looks like it was written from another person: [original submission code]" to more detailed descriptions of concrete strategies such as reordering of independent statements, and insertion of dead code.

For each dataset, five original submissions were randomly selected. GPT-4, a state-of-the-art large language model (LLM), was used to execute the 15 obfuscation attacks for each of these five submissions. This results in a total of 75 AI-plagiarized solutions for each of the three datasets. For PROGpedia 19, one submission was not compilable, resulting in only 74 plagiarized solutions. Therefore, we have 224 plagiarized solutions and 684 original solutions in total for all experiments.

We labeled each pair of submission according to their relation: pairs of originals (OO) and pairs of an original and a derived AI-obfuscated plagiarized version of that same original (OP). One-sided Wilcoxon-signed rank tests were conducted using the alternative hypothesis *less*. This tests whether the values in group 1 are *systematically* smaller than in group 2. In our experiment, we used the similarity values as calculated by JPlag in its standard configuration as group 1 and the respective values of SATM as group 2. Additionally, the effect sizes are calculated using Cliff’s delta δ.

5.3 Results

The results of the three experiments are displayed in Figure 7, where the original-original pairs are displayed in red, whereas the original-plagiarized pairs are shown in blue. The detailed metrics of the three experiments and their statistical evaluation can be seen in Table 2 and Table 3.

PROGpedia 19: For the first experiment, the results show that for OO-pairs, the null hypothesis must be rejected in favor of the alternative hypothesis. The JPlag base group similarity metrics are less than the SATM group metrics. For the OP-pairs, we can see

		Original Pairs			Original-Plagiarism Pairs			
Dataset	**Approach**	*median*	*mean*	*count*	*median*	*mean*	*count*	*median diff*
Prog19	JPlag	0.0506	0.0573	351	0.5459	0.5412	74	0.4953
Prog19	SATM	0.0619	0.0809	351	0.5846	0.5690	74	0.5227 (+0.0274)
Prog56	JPlag	<1e-10	0.0489	378	0.6667	0.6176	75	0.6667
Prog56	SATM	<1e-10	0.0638	378	0.7273	0.6650	75	0.7273 (+0.0604)
TTT	JPlag	0.0610	0.0692	195625	0.2820	0.3560	75	0.2210
TTT	SATM	0.1278	0.1347	195625	0.3618	0.4355	75	0.2340 (+0.013)

Table 2: Evaluation results for the individual datasets.

	Original Pairs			Original-Plagiarism Pairs		
Dataset	p	W	δ	p	W	δ
Prog19	9.57e-04	53429.5	-0.1359	0.2492	2561	-0.0587
Prog56	3.33e-02	66387.5	-0.0695	0.0696	2418.5	-0.1401
TTT	<1e-10	9.11e+9	-0.5253	0.0202	2266.5	-0.1941

Table 3: Evaluation results of the Wilcoxon signed rank test for JPlag (group 1) vs. SATM (group 2) with H_A = "less", $\alpha = 0.05$.

that the p-value is higher than our $\alpha = 0.05$, meaning there is no statistical significance for the alternative hypothesis *less* (baseline is less than SATM). However, Cliff's delta shows a *negligible* effect size based on the categorization of Romano et al. [30], meaning there is limited practical significance.

PROGpedia 56: The results of the second experiment revealed that the null hypothesis must be rejected both for OO-pairs and OP-pairs. For the OP-pairs, at least, the p-value comes close to the significance level of 0.05. Nonetheless, this shows that H_0 cannot be rejected, i.e., there is no significant increase in similarity metrics. However, for this dataset, the effect size is greater and close to the threshold value 0.147 for *small* practical significance.

TicTacToe: For the last experiment, featuring by far the highest number of original submissions, we can see that also here the null hypothesis for OO-pairs must be rejected. The increase in similarity metrics for OP-pairs is statistically significant and provides a *small* practical significance with an effect size of 0.194. This means that the similarity values increased for the SATM approach in OP-pairs.

5.4 Threats to Validity

We discuss the threats to validity of this evaluation following the guidelines by Wohlin et al. [41] and Runeson and Höst [31]. *Internal validity:* To keep all factors as constant as possible, the SATM approach was implemented for JPlag while JPlag was used as the baseline. Also, the plagiarized originals were selected randomly. *External validity:* Three real-world datasets were used for the experiment. However, though the SATM approach is language-independent, the datasets featured only the Java programming language. To further generalize the approach, evaluation using other programming languages must be conducted. Regarding *construct validity*, the evaluation method is similar to related works [33, 35].

6 Discussion and Future Work

SATM constitutes a novel approach to improving plagiarism detection and countering new challenges in programming education classes. The results indicate that SATM can increase the similarity scores of plagiarized submissions while maintaining slightly increased similarity scores for original submissions. However, the observed improvements vary across datasets, and the practical significance remains limited in some cases. In the context of plagiarism detection, the main goal is to increase the gap in similarity scores between OO-pairs and OP-pairs. This means that the gap between originals pairs and pairs of originals and plagiarized submissions increases, thus providing to human inspectors a better foundation to base their inspection on.

Based on the results of Section 5, we can see that for some datasets, this was possible; however, the approach also increased the similarity of OO-pairs. Looking at the practical applicability, we can see in Table 2 that for all datasets, the median difference, i.e., the gap, between the OO-pairs and the OP-pairs was increased, which is the desirable outcome.

SATM is a parameterized approach. This means that the values for *similarity threshold*, *penalty threshold*, *maximum penalty*, and *penalty factor* can be configured, allowing a way of fine-tuning the performance of the approach, which could improve the performance of SATM. This must be further investigated in future work. Also, the performance of the approach using other programming languages is subject to future experiments.

7 Conclusion

This work presents Similarity-Aware Token Matching (SATM), a novel approach to detecting AI-obfuscated plagiarism in programming courses. The approach is tailored to counter explicitly obfuscation attacks, including inserting and deleting programming statements altering the token-based program structure. The approach is evaluated using three datasets from real-world programming courses. The results show that although SATM generally affects both originals and plagiarized solutions, it can increase the similarity scores and the gap between original pairs and plagiarism.

8 Data Availability

SATM is open-source and freely available via the replication package complementing this paper [21]. The datasets and prompts used are also provided there.

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1608 – 501798263.

References

[1] A. Aiken. 2025. Moss software plagiarism detector website. https://theory.stanford.edu/~aiken/moss/. [Online; accessed 05-January-2025].

[2] Victor R. Basili. 1992. *Software Modeling and Measurement: The Goal/Question/-Metric Paradigm.* Technical Report. USA.

[3] Victor R. Basili and David M. Weiss. 1984. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering* SE-10, 6 (11 1984), 728–738. doi:10.1109/TSE.1984.5010301

[4] Mathieu Blanchette. 2007. Computation and Analysis of Genomic Multi-Sequence Alignments. *Annual Review of Genomics and Human Genetics* 8, Volume 8, 2007 (2007), 193–213. doi:10.1146/annurev.genom.8.080706.092300

[5] Tracy Camp, W Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. 2017. Generation CS: the growth of computer science. *ACM Inroads* 8, 2 (2017), 44–50.

[6] Georgina Cosma and Mike Joy. 2008. Towards a Definition of Source-Code Plagiarism. *IEEE Transactions on Education* 51, 2 (2008), 195–200. doi:10.1109/TE.2007.906776

[7] Marian Daun and Jennifer Brings. 2023. How ChatGPT will change software engineering education. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1.* 110–116.

[8] Neil Davey, Paul Barson, Simon Field, Ray Frank, and D Tansley. 1995. The development of a software clone detector. *International Journal of Applied Software Technology* 1, 3/4 (1995), 219–236.

[9] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1.* 1136–1142.

[10] Breanna Devore-McDonald and Emery D. Berger. 2020. Mossad: defeating software plagiarism detection. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 138 (Nov. 2020), 28 pages. doi:10.1145/3428206

[11] Jinan A.W. Faidhi and Scott K. Robinson. 1987. An Empirical Approach for Detecting Program Similarity and Plagiarism Within a University Programming Environment. *Computers & Education* 11, 1 (1987), 11–19. doi:10.1016/0360-1315(87)90042-X

[12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. doi:10.1145/24039.24041

[13] Tomáš Foltýnek, Terry Ruas, Philipp Scharpf, Norman Meuschke, Moritz Schubotz, William Grosky, and Bela Gipp. 2020. Detecting machine-obfuscated plagiarism. In *International conference on information.* Springer, 816–827.

[14] Steven Henikoff and Jorja G. Henikoff. 1992. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences* 89, 22 (1992), 10915–10919. doi:10.1073/pnas.89.22.10915 arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.89.22.10915

[15] M. Joy and M. Luck. 1999. Plagiarism in programming assignments. *IEEE Transactions on Education* 42, 2 (1999), 129–133. doi:10.1109/13.762946

[16] JPlag. 2025. JPlag State-of-the-Art Source Code Plagiarism & Collusion Detection. https://github.com/jplag/JPlag. [Online; accessed 05-January-2025].

[17] Oscar Karnalim. 2016. Detecting Source Code Plagiarism on Introductory Programming Course Assignments Using a Bytecode Approach. In *2016 International Conference on Information & Communication Technology and Systems (ICTS).* 63–68. doi:10.1109/ICTS.2016.7910274

[18] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2023. How novices use LLM-based code generators to solve CS1 coding tasks in a self-paced learning environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research.* 1–12.

[19] Cynthia Kustanto and Inggriani Liem. 2009. Automatic source code plagiarism detection. In *2009 10th ACIS International conference on software engineering, artificial intelligences, networking and parallel/distributed computing.* IEEE, 481–486.

[20] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining.* 872–881.

[21] Robin Maisch, Nathan Hagel, and Alexander Bartel. 2025. *Supplementary Material for "Towards Robust Plagiarism Detection in Programming Education: Introducing Tolerant Token Matching Techniques to Counter Novel Obfuscation Methods".* doi:10.5281/zenodo.15069764

[22] Robin M. Maisch. 2024. *Preventing Refactoring Attacks on Software Plagiarism Detection through Graph Transformation.* Master's thesis. Karlsruhe Institute of Technology. doi:10.5445/IR/1000172813

[23] Matija Novak. 2016. Review of Source-Code Plagiarism Detection in Academia. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO).* 796–801. doi:10.1109/MIPRO.2016.7522248

[24] Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-code similarity detection and detection tools used in academia: a systematic review. *ACM Transactions on Computing Education (TOCE)* 19, 3 (2019), 1–37.

[25] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. *arXiv preprint arXiv:2308.02828* (2023).

[26] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2023. PROGpedia: Collection of source-code submitted to introductory programming assignments. *Data in Brief* 46 (2023), 108887.

[27] Wei Pan, Ming Chok, Jonathan Wong, Yung Shin, Yeong Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Lim. 2024. Assessing AI Detectors in Identifying AI-Generated Code: Implications for Education. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training.* 1–11. doi:10.1145/3639474.3640068

[28] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science* 8, 11 (3 2002), 1016–1038. doi:10.3217/JUCS-008-11-1016

[29] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199. doi:10.1016/j.infsof.2013.01.008

[30] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In *annual meeting of the Florida Association of Institutional Research*, Vol. 177.

[31] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14 (2009), 131–164.

[32] Timur Sağlam, Moritz Brödel, Larissa Schmid, and Sebastian Hahner. 2024. Detecting Automatic Software Plagiarism via Token Sequence Normalization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.* 1–13.

[33] Timur Sağlam, Sebastian Hahner, Larissa Schmid, and Erik Burger. 2024. Obfuscation-Resilient Software Plagiarism Detection with JPlag. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings.* 264–265.

[34] Timur Sağlam, Larissa Schmid, Sebastian Hahner, and Erik Burger. 2023. How Students Plagiarize Modeling Assignments. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C).* IEEE, 98–101.

[35] Timur Sağlam, Moritz Brödel, Larissa Schmid, and Sebastian Hahner. 2024. Detecting Automatic Software Plagiarism via Token Sequence Normalization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24).* Association for Computing Machinery, New York, NY, USA, Article 113, 13 pages. doi:10.1145/3597503.3639192

[36] Timur Sağlam, Nils Niehues, Sebastian Hahner, and Larissa Schmid. 2025. Mitigating Obfuscation Attacks on Software Plagiarism Detectors via Subsequence Merging. In *46th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (CSEE&T 2025).* doi:10.5445/IR/1000179016

[37] Mary Shaw, Anita Jones, Paul Knueven, John McDermott, Philip Miller, and David Notkin. 1980. Cheating policy in a computer science department. *ACM SIGCSE Bulletin* 12, 2 (1980), 72–76.

[38] Judy Sheard, Angela Carbone, and Martin Dick. 2003. Determination of factors which impact on IT students' propensity to cheat. In *Proceedings of the fifth Australasian conference on Computing education-Volume 20.* 119–126.

[39] Simon, Oscar Karnalim, Judy Sheard, Ilir Dema, Amey Karkare, Juho Leinonen, Michael Liut, and Renée McCauley. 2020. Choosing Code Segments to Exclude from Code Similarity Detection. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education.* doi:10.1145/3437800.3439201

[40] Michael J. Wise. 1993. String Similarity via Greedy String Tiling and Running Karp-Rabin Matching. (12 1993). https://www.researchgate.net/profile/Michael_Wise/publication/262763983_String_Similarity_via_Greedy_String_Tiling_and_Running_Karp-Rabin_Matching/links/59f03226aca272a2500141f4/String-Similarity-via-Greedy-String-Tiling-and-Running-Karp-Rabin-Matching.pdf Department of Computer Science, University of Sydney.

[41] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering.* Vol. 236. Springer.

[42] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy.* 590–604. doi:10.1109/SP.2014.44

[43] Lisa Yan, Nick McKeown, Mehran Sahami, and Chris Piech. 2018. TMOSS: Using intermediate assignment work to understand excessive collaboration in large classes. In *Proceedings of the 49th ACM technical symposium on computer science education.* 110–115.