# Detecting Automatic Software Plagiarism via Token Sequence Normalization

Timur Sağlam, Moritz Brödel, Larissa Schmid, Sebastian Hahner
firstname.lastname@kit.edu
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

## ABSTRACT

While software plagiarism detectors have been used for decades, the assumption that evading detection requires programming proficiency is challenged by the emergence of automated plagiarism generators. These generators enable effortless obfuscation attacks, exploiting vulnerabilities in existing detectors by inserting statements to disrupt the matching of related programs. Thus, we present a novel, language-independent defense mechanism that leverages program dependence graphs, rendering such attacks infeasible. We evaluate our approach with multiple real-world datasets and show that it defeats plagiarism generators by offering resilience against automated obfuscation while maintaining a low rate of false positives.

## CCS CONCEPTS

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

## KEYWORDS

Software Plagiarism Detection, Plagiarism Obfuscation, Obfuscation Attacks, Code Normalization, PDG, Tokenization

**ACM Reference Format:**
Timur Sağlam, Moritz Brödel, Larissa Schmid, Sebastian Hahner. 2024. Detecting Automatic Software Plagiarism via Token Sequence Normalization. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639192

## 1 INTRODUCTION

Plagiarism is prevalent in computer science education [12, 23, 30], facilitated by the ease of duplicating and modifying digital assignments. Although students generally acknowledge plagiarism as academic misconduct, some will engage in it despite the threat of consequences [47]. They also attempt to *obfuscate* their plagiarism to conceal its source, using techniques like renaming, reordering, or inserting code [20, 32, 43].

Plagiarism is notably prevalent in mandatory assignments, especially in beginners' courses [35]. Due to the substantial size of computer science courses, manual inspection is impractical [9, 22], and the individual risk of detection reduces with rising class sizes [54]. For instance, checking plagiarism for 500 students necessitates more than 100.000 pairwise comparisons, which is not feasible to perform manually.

To tackle this problem at scale, software plagiarism detectors have been used for decades [7, 33, 36]. Tools like MOSS [1] and JPlag [36, 37] are used in universities worldwide to tackle the problem of academic integrity [3]. Plagiarism detectors are effective when defeating them takes more effort than completing the actual assignment [13]. Thus, a widespread assumption has always been that if students could evade detection, they had already acquired the skills to be taught, considering the tedious and intricate nature of successful manual obfuscation [16]. However, this assumption has been broken with the recent rise of plagiarism generators [6, 13]. While designing such a generator takes time and programming proficiency, using them requires neither.

*Automatic Plagiarism Generators.* Recently, Devore-McDonald and Berger [13] introduced *Mossad*, a plagiarism generator inspired by genetic programming that allows generating multiple obfuscated versions of a single program. While most software plagiarism detectors exhibit resilience against some *obfuscation attacks* [16, 36], Mossad exploits a vulnerability that is inherent in most structure-based approaches. State-of-the-art plagiarism detectors operate token-based and compare the code's structure [31]. They parse the program and linearize the parse tree by extracting the tokenized program as a *token sequence*. Matches between programs are identified with a subsequence search in these token sequences, and the similarity is calculated. By omitting certain details such as names and types, the token sequence serves as an abstraction layer and is thus resilient against specific obfuscation attacks such as renaming and retyping.

Mossad repeatedly inserts statements into the plagiarized program to break up these subsequences until the plagiarism is no longer recognized. To that end, sets of pre-defined statements called *entropy* and existing statements from the original program are used. The insertion is stopped when the plagiarized instance falls below a certain similarity threshold compared to the original. The authors claim that this cannot be detected by human inspection, as the inserted lines are not suspicious at first glance [13]. While Mossad is only one example of a plagiarism generator, we must accept the possibility of others to exist [6], and among them, reordering is one such potential obfuscation technique. How exactly an unknown plagiarism generator might alter code is unclear. However, due to the nature of token-based detectors, the token sequence is the only aspect impacting detection quality. Although there are graph-based approaches that are potentially less vulnerable to such attacks, they are not feasible in practice [24] due to the NP nature of determining subgraph isomorphism [25, 29, 45].

*Contribution.* This paper presents a novel defense mechanism called *token sequence normalization* to achieve resilience against such automatic obfuscation attacks. We combine the effectiveness of graph-based approaches with the scalability of token-based approaches in a best-of-both-worlds approach. We leverage program dependence graphs (PDG) [14] to normalize programs. However, for real-world applicability, any approach must be language-independent and operate at an abstract level [24, 31, 37]. Furthermore, it must support explainability via traceability, enabling visualization based on the original, unaltered code. From both ethical and administrative standpoints [23, 46], only the original, unaltered code should inform human decision-making in academic misconduct investigations. Moreover, the unaltered code often contains idiosyncrasies [32] due to obfuscation. They are used for both initial decision-making and misconduct investigations. Hence, PDGs are inapplicable.

To meet these criteria, we introduce the concept of a *token normalization graph* (TNG), which operates on the token sequence, thus providing a language-independent and ethically sound approach that aligns with academic misconduct administrative procedures. This TNG is then used to normalize the token sequence by effectively reverting the insertion and reordering of statements. Our defense mechanism can thus effectively de-obfuscate plagiarism instances.

We implement the defense mechanism based on the state-of-the-art plagiarism detector JPlag [36, 37], as it is widely used, open-source, GDPR-compliant, and easy to extend[1]. This gives us the benefit of reusing the existing capabilities, good scalability, and broad language support. However, our defense mechanism can be implemented for other token-based detectors, such as MOSS [1] or Dolos [27]. Based on our work, we ask the following research questions:

**RQ1** Can our defense mechanism effectively detect MOSSAD-style plagiarism instances?
**RQ2** Can it provide resilience against general insertion-based, reordering-based, and combined obfuscation attacks?
**RQ3** How does our defense mechanism impact the false positive rate of the plagiarism detector?
**RQ4** How does our defense mechanism impact the performance of the plagiarism detector?

[1]We provide our implementation of the defense mechanism as part of the supplementary material [42]. Furthermore, it is integrated into the open-source repository of JPlag [17].

```
1 void printSquares() {                 void printSquares() {
2  int i = 1;                            int i = 1;
3                              (+)       boolean debug = false;
4  while (i <= 10) {                     while (i <= 10) {
5   int square = i * i;                   int square = i * i;
6   println(square);           (~)        i++;
7   i++;                       (~)        println(square);
8  }                                     }
9 }                                     }
```

**Listing 1: Original code (left) and modified variant (right) after inserting one statement (+) and reordering two (∼).**

*Evaluation.* In our two-stage evaluation, we evaluate our defense mechanism with the publicly available dataset PROGpedia [34] and two datasets of our own introductory programming course. We evaluate our mechanism against insertion-based, reordering-based, and combined attacks. The results demonstrate the effectiveness of our approach, as obfuscation attacks based on insertion and reordering are rendered ineffective. We increase similarity scores of plagiarized solutions from 10-30% to a minimum of 95%. Remarkably, the similarity of unrelated solutions remains virtually unaltered, with an average similarity increase of less than 1%, effectively avoiding an increase in false positives. Our defense mechanism comes with a negligible runtime overhead of mere seconds for large real-world datasets, thus showing its practicality.

This paper's structure is as follows: In Section 2, we introduce the running example. Section 3 discusses the basics of software plagiarism detection. In Section 4, we establish a threat model. In Section 5, we present our defense mechanism, whereas Section 6 shows its evaluation. Finally, we discuss related research in Section 7 and conclude in Section 8.

## 2 RUNNING EXAMPLE

As a running example, we examine the programs shown in Listing 1. Both programs print the squared values of the numbers from 1 to 10. The right program is a modified variant based on two changes that alter its structure. Thus, the program still behaves the same upon execution. In detail, line 3 was inserted in the modified variant, and lines 6 and 7 were swapped. While the relationship between the programs is evident in this simple example, this is not true for larger programs that undergo extensive modifications to obfuscate the relation between the variant and the original. Thus, these modifications can be used to conceal plagiarism. Additionally, manual comparison becomes infeasible when dealing with numerous student submissions, even for smaller programs. However, for plagiarism detectors, structural obfuscation attacks like these diminish the detection quality [13].

## 3 STATE OF THE ART

Software plagiarism detectors enable educators to tackle the problem of scale by helping to seek out plagiarism. The detector analyzes pairs of programs to find similar sections and calculates a similarity score for each pair. However, the

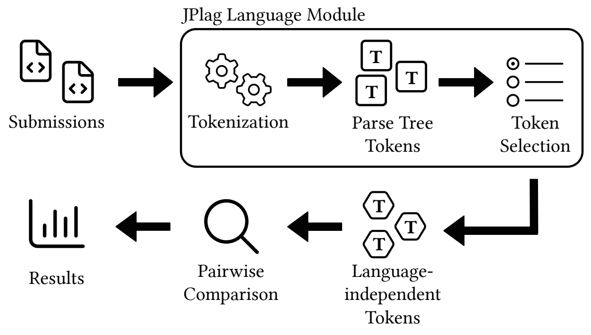

**Figure 1: Token-based plagiarism detection process of JPlag.**

final decision of identifying plagiarism is left to instructors, given the inherent complexity and ethical considerations involved in this task. Most software plagiarism detection approaches compare the structure of the code [31, 32], and among them, token-based approaches, such as MOSS [1], and JPlag [36], are the most popular tools employed in practice. They combine tokenization with pairwise comparison to identify code matches based on hashing and tiling [1, 37]. As their comparison algorithms, MOSS and Dolos [27] are using winnowing [44], while JPlag and Sherlock [16] use greedy string tiling with the running Karp-Rabin matching [51, 52].

The tokenization step transforms the program's code into a parse tree. A subset of the tree nodes is then extracted as tokens, thus linearizing the tree and further abstracting from the underlying code. Figure 1 outlines this process for JPlag, which is comparable to other approaches. According to Prechelt et al. [37], the extracted token sequence "*characterizes the essence of a program's structure*". To illustrate this, Table 1 shows the token sequences of the two programs in Listing 1. While the structural information concerning method context, loop context, declarations, and assignments remains intact, specific details such as names, types, or comments are excluded. The token sequences are then used to find matching subsequences between pairs of submissions. As matching single tokens may lead to false positives, a minimal match length hyperparameter [41] is employed, below which subsequences are no longer considered matches. As the tokens abstract from the code, the comparison is inherently resilient against particular obfuscation attacks, such as renaming, retyping, or obscuring constant values [37]. This particularly includes immunity against all *lexical* [16] attacks. However, these detectors are not resilient against insertion-based and reordering-based attacks [13].

## 4 THREAT MODEL

Devore-McDonald and Berger [13] introduce MOSSAD, a software plagiarism generator that rapidly produces obfuscated versions of original programs. MOSSAD uses techniques inspired by genetic programming to evade plagiarism detection and randomly inserts statements until the similarity to the input computed by a plagiarism detector falls below a user-defined threshold. To keep the inserted statements domain-related and unsuspicious, MOSSAD uses both existing statements from the input program and a user-defined pool of statements called *entropy*. MOSSAD generates multiple variants from a single original, which are not only obfuscated from the original but also among each other due to its indeterministic nature. To determine program semantic equivalence, MOSSAD compares the intermediate representation of programs after compiling both with high-level optimization. By repeatedly applying mutations randomly, MOSSAD can create multiple variants that amongst each other are different enough not to get flagged during plagiarism detection. Its automatic and efficient nature makes it a potent threat to current plagiarism detection practices. MOSSAD demonstrates its effectiveness against various plagiarism detectors, including MOSS [1], Sherlock [16], and JPlag [36, 37]. Their approach is mainly designed against token-based plagiarism detectors but has the potential for attacks on other structure-based [31] software plagiarism detectors. Thus, we propose a generalized threat model.

**Table 1: Comparison of the original and obfuscated token sequences corresponding to the program statements in Listing 1.**

| # | Original Tokens | → | Variant Tokens |
|---|---|---|---|
| 1 | `method start` | | `method start` |
| 2 | `variable` | | `variable` |
| 3 | | (+) | `variable` |
| 4 | `loop start` | | `loop start` |
| 5 | `variable` | | `variable` |
| 6 | `apply` | (~) | `assignment` |
| 7 | `assignment` | (~) | `apply` |
| 8 | `loop end` | | `loop end` |
| 9 | `method end` | | `method end` |

### 4.1 Software Plagiarism Generators

An automatic software plagiarism generator aims to modify an input program to change its structure and appearance without altering its semantics. Modifying the structure allows evading detection by plagiarism detectors, while modifying the appearance will avoid detection during the manual inspection. However, the former takes precedence over the latter, as instructors rely on automatic detectors due to large class sizes [9, 54]. Possible modifications involve inserting, deleting, or reordering statements within the code. For any obfuscation attack to be effective against token-based detectors, it must alter code in such a way that it affects the token sequence, meaning inserting, deleting, and reordering tokens. The position of the modification in the code can be either random, systematic, or a combination of both. Different termination criteria may be employed during the obfuscation process.

Deletion of statements is typically ineffective since few statements can be removed without affecting semantics. In the following, we thus mainly discuss attacks based on insertion and reordering, as they are effective [13] and readily automated. These attack types are also prevalent when humans engage in plagiarism [20, 32, 43].

### 4.2 Statement Insertion

Inserting one or multiple statements into a plagiarized program allows changing the structure and appearance of the program by breaking up matching code segments with the source program. For this attack type to be effective, these statements must not change the semantics of the code, and the modified program must compile. Usually, the inserted code is dead code. However, more sophisticated approaches might try to conceal that. In Listing 1, a dead statement is inserted in the third line, which does not affect the program's semantics. However, inserting `i = 0;` between lines 4 and 5 would change the program's behavior, thus altering its semantics. The quality of the inserted statements is of the essence, as comments or unnecessary braces do not impact the token sequence. Instead, the inserted statements must be (partially) tokenized to affect the internal representation of the plagiarism detector. Additionally, while humans may rely on tools for inspection, the inserted code should appear plausible at first glance to avoid arousing suspicion. Nevertheless, this attack type is effective and can be easily automated. However, inserting an excessive number of lines may raise suspicion, especially if it significantly exceeds the length of other students' solutions.

### 4.3 Statement Reordering

The reordering of statements also changes the structure and appearance of a program. However, only independent statements can be reordered to preserve the program's semantics. In Listing 1, the statements in lines 6 and 7 are swapped, as they do not depend on each other. However, additionally swapping lines 5 and 6 afterward would change the program's behavior as the loop variable is incremented before calculating the squared value. Nevertheless, reordering as an automatic obfuscation attack is challenging, as there are limited possible swaps due to the requirement to respect dependencies between statements. Consequently, this technique may be less effective than statement insertion, but the method is still a viable obfuscation strategy, especially when combined with statement insertion. Since statement order is primarily a stylistic choice and different orders are not suspicious, this technique is more challenging for a human reviewer to spot.

## 5 TOKEN SEQUENCE NORMALIZATION

This section introduces our main contribution, a language-independent defense mechanism called token sequence normalization. With this mechanism, we combine the effectiveness of graph-based approaches with the scalability of token-based approaches in a best-of-both-worlds approach. As discussed in Section 3, the results of token-based plagiarism detectors are invariant to lexical modifications of the input submissions. This is because all variants produced through lexical modifications result in the same token sequence. With our defense mechanism, we expand upon these capabilities by making the token sequence virtually invariant against insertion- and reordering-based attacks. We achieve this by normalizing the token sequence by removing dead statements and putting subsequent independent statements in a fixed order. As a result, we achieve high resilience to automatic plagiarism generation using these attacks.

### 5.1 Requirements

For any normalization approach to be applicable in real-world scenarios, it must satisfy specific criteria. Firstly, it needs to be *language-independent*, ensuring versatility across programming languages. Developing a normalization approach from scratch for each new language is impractical. Moreover, it must operate at an *abstract level*, focusing on tokens rather than code, enabling the capture of structural similarities while minimizing the influence of language-specific syntax [24, 31, 37]. Furthermore, it must support *explainability* via traceability, allowing for the visualization of potential plagiarism with the original, *unaltered code*. This is essential from both ethical and administrative perspectives [46], as it ensures that the original code remains the basis for human decision-making in plagiarism detection [23]. Furthermore, the altered code may no longer contain *idiosyncrasies* (e.g., obvious obfuscation attempts) that assist the decision-making [32]. As a consequence, normalization approaches that modify the input programs as pre-processing steps are not applicable. Existing dead code elimination methods do not fulfill these requirements, as they are language-dependent, modify the programs, and are incompatible with ethical and administrative concerns. In contrast, our defense mechanism is specifically designed to meet these requirements.

### 5.2 Overview

The defense mechanism is an additional step in the pipeline of a plagiarism detector before the pairwise comparison (see Figure 1), which effectively de-obfuscates plagiarism for the remaining steps in the pipeline. At a high level, our mechanism operates as follows for each input program:

(1) We enrich the token sequence with additional language-independent semantic information about the interdependence of the tokens.
(2) We construct a language-independent graph-based representation of the tokenized program from the enriched token sequence, which abstracts from the original code. This graph represents a partial order of the tokens.
(3) We then use this graph to generate a normalized token sequence, thus effectively reverting insertions and reorderings.
    (a) Insertions are countered via subsequence removal.
    (b) Reordering is negated by topological sorting [19].
(4) The comparison is still conducted exclusively on the token sequence, circumventing the computational complexity of graph-based approaches.

We call the representation of the tokenized program a *token normalization graph* (TNG). Each node in this graph represents the tokens of a program statement. As it is based on the token sequence, it is language-independent, which enables our defense mechanism to be applicable to any programming language. Thus, the format of the semantic information utilized to enrich the token sequence must also be language-independent. We specify a generic format for the semantic information based on the interdependence between tokens that abstracts from the detail of the underlying programming language. The semantic information must be extracted from the input programs alongside the tokens. Therefore, the extraction needs to be implemented separately for each language the detector supports. It is the sole language-dependent step that our defense mechanism adds. However, extracting semantic information does not impose additional constraints on the detector since tokenization is inherently language-dependent.

## 5.3 Semantic Information

In order to construct a TNG, we need information about the interdependencies between the tokens. This information is not present in the token sequence and thus must be additionally attached to the tokens. To that end, a language-independent format is required for this semantic information. Using this format, the tokens can be automatically annotated during their extraction from the parse tree. Different types of statements lead to different types of information. Thus, this format allows us to map the relations of the statements in the program to the tokens independent of the underlying language of the program. We specify the following format for this semantic information:

**Criticality** describes statements that contribute to programs' behavior through means other than variables and thus must not be removed. Therefore, a token can be either critical or not critical. *In our example in Listing 1, the critical statements are the* `while` *statement and the* `println` *statement. Thus, the* `loop start` *and* `apply` *tokens in Table 1 are critical.*
**Fixed Order** describes how statements must maintain their relative order to ensure semantic equivalence. Tokens can thus be either fixed in their order or non-fixed. Fixed order tokens act as reordering boundary, as preceding tokens cannot be moved after this boundary and vice versa. *In the running example, the* `while` *statement dictates that the statements within the loop cannot be moved outside the loop. Thus, the loop tokens of lines 4 and 8 in Table 1 are fixed order tokens.*
**Variable Access** describes the variable accesses (read or write) for each statement, thus indicating on which other statements they depend. Hence, each token has two variable identifier sets: One for read accesses and one for write accesses. *In the running example, the* `while` *statement depends on the statement the declares the variable* `i`*. Therefore, the* `loop start` *token contains* `i` *in its read access set.*
**Loop** describes statement blocks in which the execution order of the statements can deviate from their declaration order. Therefore, the first and last tokens corresponding to the loop are marked with loop start and loop end flags. *In the running example, the statements in the loop body are repeatedly executed. Thus, the incrementation of the variable* `i` *may be executed not only after the* `println` *statement but also before due to the repeated execution. In the running example, both* `loop start` *and* `loop end` *are marked with the loop flag.*

## 5.4 Token Normalization Graph

We use the token normalization graph as the underlying data structure to generate a normalized token sequence. A TNG is a specialized version of a program dependence graph (PDG). It is designed for tokens instead of code and through a higher abstraction level not confined to the programming language's syntax. Thus, a TNG crucially provides language independence. A TNG contains additional edges required for the token sequence normalization. While a PDG focuses only on statement dependencies in code, the TNG considers both token interdependence and the order of the tokens. Both are required for token-based approaches.

We construct the TNG from the token sequence and the additional semantic information. A TNG node represents a single statement and contains all tokens generated from that statement. After the construction of the TNG, we no longer need the original token sequence. A TNG has three types of edges, all of which are directed:

**Variable Flow Edges** indicate that a statement writes to a variable that another statement reads. They are similar to the data dependencies in the PDG.
**Variable Order Edges** indicate that altering the order of the two statements may alter variable values and, thus, the program's behavior. Wherever there is a variable flow edge, there is also a variable order edge; however not necessarily in the same direction.
**Fixed Order Edges** indicate that altering the order of the two statements may affect the program's behavior for reasons beyond variable value alterations.

The TNG is then constructed as follows: First, we group all tokens within a statement as nodes. Next, we label nodes containing at least one critical token as *critical*. We then use the tokens marked as *fixed order* to create position significance edges. For each node with fixed order tokens, we create incoming and outgoing edges to and from all preceding and subsequent nodes, as determined by the original order of the tokens. We employ the variable access sets and loop flags to create variable flow edges. Finally, disregarding loop flags, we determine variable order edges solely by variable access sets. Once the TNG is complete, it contains all the necessary data for generating a normalized token sequence. Thus, the original token sequences can be safely discarded. Moreover, as for all token-based approaches, the original programs are no longer required for the similarity calculation.

Figure 2 shows a PDG for the plagiarized program in Listing 1. The control and data dependencies between the different code statements are represented via edges. While the dead statement for the variable called `debug` can be identified, it cannot be used to generate a normalized token sequence due to the requirements mentioned in Section 5. Figure 3 shows the TNG for the same program. In contrast to the PDG, the nodes represent tokens instead of code. The TNG contains the three types of edges based on the semantic information discussed in Subsection 5.3. The two nodes with a gray background are marked as critical. Furthermore, the

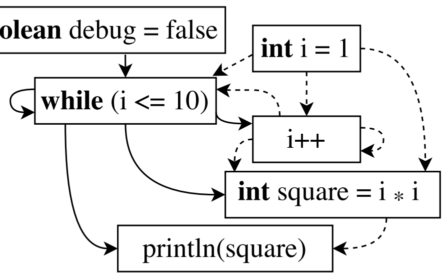

**Figure 2: Program dependence graph of the running example showing control dependencies (solid arrows) and data dependencies (dashed arrows).**

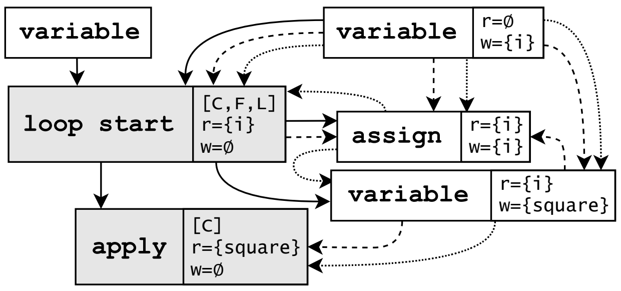

**Figure 3: Token normalization graph for the PDG in Figure 2, fixed order edges (solid), variable order edges (dashed), and variable flow edges (dotted). The critical (C), fixed-order (F), and loop (L) flags, with the variable access sets (r for read, w for write), are included for illustrative purposes.**

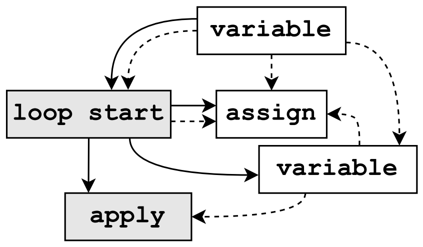

**Figure 4: Token normalization graph after the dead nodes and removal of variable flow edges, still present are fixed order edges (solid) and variable order edges (dashed).**

**Table 2: Comparison of the original and obfuscated token sequences with the obfuscated one after the two steps dead node removal and topological sorting Listing 1.**

| # | Original | → | Obfuscated | → | Nodes Removed | → | Top. Sorted |
|---|---|---|---|---|---|---|---|
| 1 | method start | | method start | | method start | | method start |
| 2 | variable | | variable | | variable | | variable |
| 3 | | (+) | **variable** | (-) | | | |
| 4 | loop start | | loop start | | loop start | | loop start |
| 5 | variable | | variable | | variable | | variable |
| 6 | apply | (~) | **assignment** | | **assignment** | (~) | apply |
| 7 | assignment | (~) | **apply** | | **apply** | (~) | assignment |
| 8 | loop end | | loop end | | loop end | | loop end |
| 9 | method end | | method end | | method end | | method end |

semantic information of the corresponding tokens is illustrated in the bottom right corner of the node. The semantic information is only depicted for the sake of clarity. It is used to construct the TNG but is no longer necessary for the normalization thereafter.

## 5.5 Generating the Normalized Token Sequence

After constructing the TNG, we can leverage it to generate a normalized token sequence in two steps. First, we remove all dead nodes. These are all nodes from which no critical node can be reached via variable flow edges. This effectively reverts insertions into the token sequence. Next, we remove all variable flow edges from the TNG, making it acyclical. They have served their purpose for the dead node removal and are no longer required. In our running example in Listing 1, the statement containing the variable named debug is considered dead code, and so is its corresponding `variable` node in the TNG in Figure 3. This node will be removed, as it has no outgoing variable flow edge. Table 2 illustrates how that affects the normalized token sequence. The code insertion leads to an additional token in the obfuscated token sequence. However, after the dead node removal, this effect is reversed. Figure 4 illustrates the TNG from Figure 3 with the dead node removed and variable flow edges omitted. The previous cycles, such as the one between the `loop start` and `assign` nodes, are no longer present. A partial order becomes apparent when considering the topmost `variable` node as the root.

Next, we use topological sorting [19] to generate a normalized token sequence. This counters attempts to re-order the token sequence. Specifically, we order the tokens of the remaining nodes by their node's distance to the aforementioned root. Nodes with the same distance represent subsequent independent statements. We sort them via the types of the tokens in the nodes. This is a robust criterion, as the extracted tokens are invariant to lexical modifications [37]. Note that if two nodes with the same root distance are equal according to that criterion, they contain the same tokens and their order does not affect the normalized token sequence.

In our running example in Listing 1, statements 6 and 7 were swapped. Table 2 illustrates how this obfuscation affects the token sequence. When using the reduced TNG in Figure 4 to generate the normalized token sequence, the sequence after the topological sorting will be identical to the original one (see Table 2). Thus, the plagiarism detector computes a 100% match for our running example, despite the obfuscation attempt shown in Listing 1. As our approach only normalizes the tokens sequence. The original, unaltered code, with all its idiosyncrasies, is used for the visualization. However, as a result of our normalization, the similarity calculation and code matching are not compromised by obfuscation attacks.

## 6 EVALUATION

This section evaluates our defense mechanism, which we implemented[2] for the open-source plagiarism detector JPlag[3] and the programming language Java. We chose JPlag as it is state-of-the-art [32], open-source, and was used for the evaluation of Mossad [13]. We show that our defense mechanism is effective and applicable to real-world datasets[4].

## 6.1 Methodology

The evaluation follows the *Goal-Question-Metric* (GQM) method [4, 5], and the GQM plan is as outlined:

**G1** Improve resilience against automatic obfuscation attacks.
- **Q1** Resilience against insertion-based attacks?
- **Q2** Resilience against reordering-based attacks?
- **Q3** Resilience against combined attacks?
- **M1-3** Similarity scores with and without the normalization.

**G2** Retain the performance and false-positive rate of the detection
- **Q3** Does the normalization affect the performance?
- **M3** Runtime overhead of the normalization.
- **Q4** Does the normalization affect the false positive rate?
- **M4** Deviation of the similarity scores for unrelated solutions.

Our first goal (**G1**), regarding obfuscation attack resilience, maps to research questions **RQ1** and **RQ2**, while our second goal (**G2**), regarding the scalability and false-positive rate, maps to **RQ3** and **RQ4**. For software plagiarism detection, pairs of solutions are compared, and their similarity is calculated. Thus, the similarity scores of plagiarism instances to their originals (***similarity score to original*, SSO**) must be high to maximize the true positive rate. Vice versa, the similarity score of pairs or unrelated solutions must be low.

*6.1.1 Datasets.* We conducted our evaluation using four distinct datasets, each with specific characteristics. First, we used two tasks from PROGpredia [34], a publicly available collection of submissions from introductory programming courses. We chose these specific tasks because they had longer programs and more solutions than the others. Task 19 requires students to implement a social network analyzer with a graph data structure and a depth-first search to count distinct groups. Task 56 requires them to find a minimum spanning tree using Prim's algorithm and calculate the total minimum distance to connect all points with Euclidean distance. To prepare the datasets for our evaluation, we only used solutions written in Java and removed all solutions that did not compile, as JPlag requires input programs to compile. However, as these public data sets are relatively small, we also used two datasets from our first semester's Java programming lecture. One is from a mandatory assignment that required students to implement the game TicTacToe with both support for human and basic AI players via a simple command line interface. The other is from the final project of the course where students were tasked to implement a tile-based farming board game, requiring students to design and implement a large program with around 30-40 source files. To prepare these datasets, we removed all known human plagiarism based on the results of past plagiarism checks.

Thus, we ended up with the following four datasets, which provide diverse programs, enabling us to evaluate our defense mechanism thoroughly. These size specifications are given in lines of code (LOC), excluding comments and empty lines.

**PROGpedia Task 19:** Consists of 27 submissions, with a mean size of 131 LOC and a median of 106 LOC.

**PROGpedia Task 56:** Consists of 28 submissions, with a mean size of 85 LOC and a median of 77 LOC.

**TicTacToe:** Consists of 626 submissions, with a mean size of 236 LOC and a median of 225 LOC.

**BoardGame:** Consists of 434 submissions, with a mean size of 1529 LOC and a median of 1487 LOC.

*6.1.2 Automatic Plagiarism Generation.* For our evaluation, we employ three types of automated obfuscation attacks. First, we use the attack employed by Mossad solely based on insertion. This attack can insert any arbitrary number of statements into a single program. We use both statements from the original program and a custom entropy file with pre-defined statements. Second, we use a variant based on Mossad where statements are reordered instead of inserted. For this attack, the number of changes that can be made to a single program is limited, as only some statements can be reordered without changing the original solutions. Thus, this attack is weaker than the previous one. Finally, we employ a combined attack, insertion after reordering. This attack makes it especially hard for humans to spot plagiarism. It also further reduces the computed similarity score compared to solely inserting. We used these three attack types to generate plagiarism instances for all four datasets. For the PROGpedia datasets, we generate three plagiarism instances per original solution, thus resulting in 81 and 84 instances, respectively. For the TicTacToe and BoardGame data sets, we generate three plagiarism instances from twenty randomly chosen solutions due to significantly higher generation time, thus resulting in 60 plagiarism instances for each dataset. In summary, we evaluated our approach with four data sets, totaling 1.115 originals and 285 plagiarism instances.

## 6.2 Evaluation of Obfuscation Attack Resilience

To assess the effectiveness of our defense mechanism, we compare a version of JPlag with our normalization to one without it. Additionally, we analyze the impact of varying numbers of modifications by conducting multiple evaluation runs during the plagiarism generation process.

*6.2.1 Insertion-based Attack.* Initially, we evaluate insertion-based attacks, aligning with Mossad's plagiarism generation approach. The outcomes are depicted in Figure 5 a), which shows the average SSOs for the four datasets for a varying number of insertions relative to the size of the original solutions. Without our defense mechanism, JPlag computes an average SSO of less than 25% for all four datasets, and notably, for the TicTacToe dataset, it drops below 10%. This

[2]Further information on the implementation can be found in [8].
[3]https://github.com/jplag/JPlag
[4]We provide the implementation of our approach, and the datasets used for the evaluation as part of the supplementary material [42].

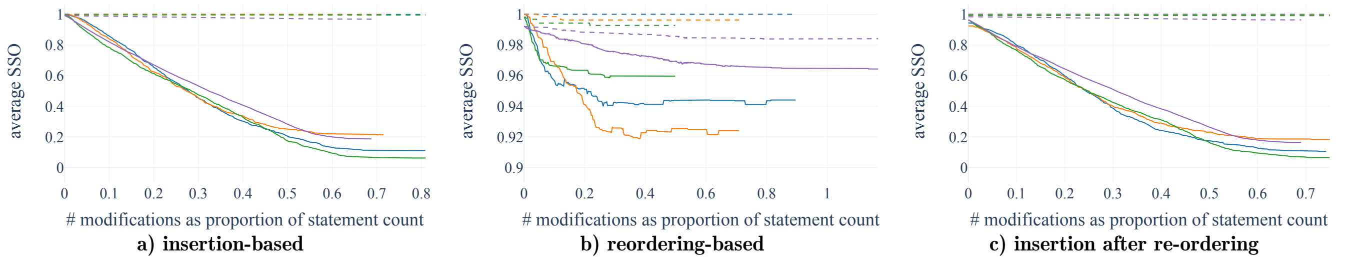

**Figure 5: Average SSO for all plagiarism instances of a dataset with our defense mechanism disabled (solid lines) and enabled (dashed lines). The green line is TicTacToe, the purple is BoardGame, the blue is PROGpedia-19, and the yellow is PROGpedia-56.**

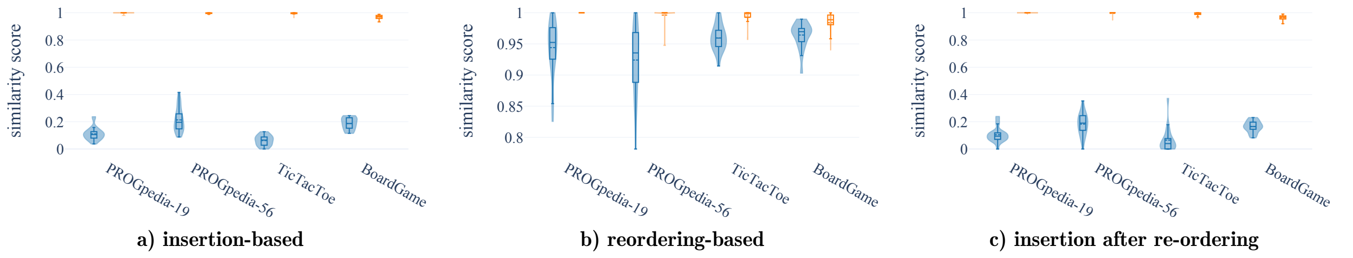

**Figure 6: SSO distribution for all three datasets with our defense mechanism disabled (blue) and enabled (orange).**

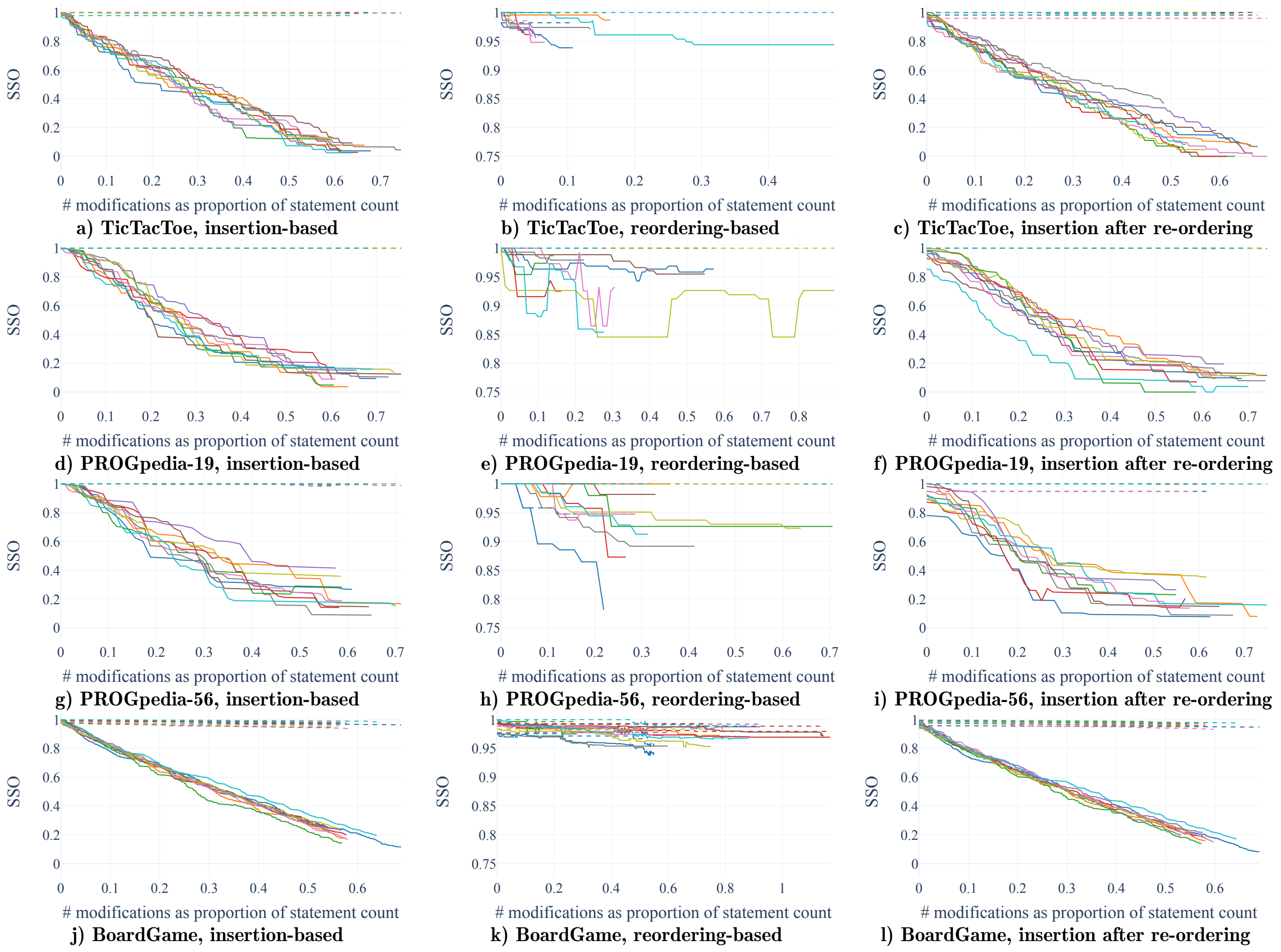

**Figure 7: Detailed SSOs for the datasets' individual programs with our mechanism disabled (solid) and enabled (dashed).**

renders most plagiarism detectors ineffective. To achieve such a low threshold, approximately 50-60% additional statements need to be inserted. In contrast, when our defense mechanism is active, the mean SSO surges to over 99% for both PROGpedia datasets and TicTacToe. Furthermore, the SSO surges to over 96% for the BoardGame dataset. These remarkably high SSO values raise strong suspicions, enabling the prompt identification of plagiarism instances.

**Answer to RQ1**: *Our defense mechanism effectively detects MOSSAD-generated plagiarism instances.*

*6.2.2 Reordering-based Attack.* Next, we focus on evaluating reordering-based attacks exclusively. The average SSOs for the four datasets, for a certain relative number of re-ordering operations, are depicted in Figure 5 b) (note the difference in the y-axis scale). As anticipated, reordering attacks are less effective, as evidenced by our findings. For the TicTacToe and BoardGame datasets, the average SSO never drops below 95%, while for both PROGpedia datasets, the mean SSO ranges between 90 and 95%. However, achieving this effect only requires performing, on average, between a third and a quarter of the possible re-ordering operations. However, with our mechanism activated, all mean SSOs surge above 98%, making it easy to identify plagiarism instances.

*6.2.3 Combined Attack.* In the third phase of our evaluation, we examine combined attacks that involve insertion after reordering. The outcomes of this assessment are depicted in Figure 5 c), which closely resemble the results of the insertion attacks on their own. Without our defense mechanism, JPlag calculates an average SSO of less than 20% for the PROGpedia and BoardGame datasets and below 10% for the TicTacToe dataset. However, when reordering is applied before insertion, it requires approximately 5-10% fewer insertions to achieve the same SSO as with insertions alone. With our defense mechanism enabled, all SSOs rise to over 95%, making these plagiarism instances highly detectable.

**Answer to RQ2**: *Our defense mechanism provides robust resilience against general obfuscation attacks based on insertion and re-ordering.*

## 6.3 Evaluation of False-Positive Rate

When applying our approach to the PROGpedia datasets, we encounter 378 and 351 comparisons between unrelated solutions. For the TicTacToe dataset, there are 195.625 unrelated comparisons, and for the BoardGame dataset, 93.961. Our defense mechanism must not increase the similarities of these comparisons, as it could lead to a higher false-positive rate during plagiarism inspection. Therefore, we compare the similarities for these unrelated comparisons with and without our mechanism enabled. As depicted in Figure 8, the similarity scores remain nearly unchanged. The median and mean values show no significant alterations. For the PROGpedia-19 and BoardGame datasets, the upper whisker slightly rises, while PROGpedia-56 slightly decreases. In the case of TicTacToe, there is a single outlier that experiences a slight increase. However, all these changes are less than 2% and are not considered significant. When measuring the difference between the individual comparisons, over 95% of them have a similarity below 1%. These results demonstrate that unrelated programs do not experience an increase in similarity.

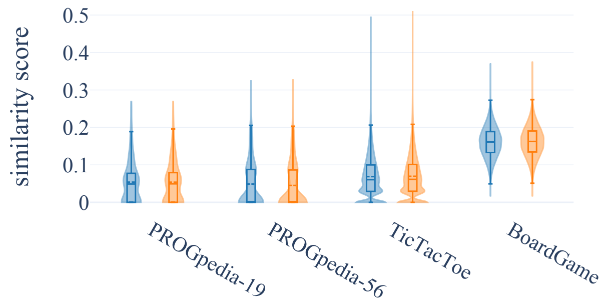

**Figure 8: The similarity scores of unrelated solutions with our defense mechanism enabled (orange) and disabled (blue).**

**Table 3: Runtime for TicTacToe (626 programs, 167.562 LOC total, ~200.000 comparisons) and BoardGame (434 programs, 685.730 LOC total, ~94.000 comparisons), avg. of 100 runs.**

| Dataset | Metric | disabled | enabled | diff |
|---|---|---|---|---|
| TicTacToe | Runtime | 6.97s | 7.86s | +0.89s |
| | $\sigma$ (SD) | 0.10s | 0.10s | |
| BoardGame | Runtime | 16.83s | 23.28s | +6.45s |
| | $\sigma$ (SD) | 0.53s | 0.44s | |

**Answer to RQ3**: *Our mechanism does not impact the false-positive rate of the plagiarism detector.*

## 6.4 Evaluation of Performance Impact

To assess the performance implications of our mechanism, we compared the runtime of JPlag with and without our approach. We measured the runtime for the TicTacToe and BoardGame datasets, as the PROGpedia datasets are too small for meaningful measurements. TicTacToe has the most submissions and, thus, the most comparisons. BoardGame has the largest submissions. We utilized a consumer notebook, specifically a MacBook Pro equipped with an M1 Pro chip and 16GB of memory for our performance measurements. As shown in Table 3, there is only an overhead of 0.89 seconds for TicTacToe, and 6.45 seconds for BoardGame. Note, that this is the overhead for the full datasets, not individual programs. State-of-the-art software plagiarism detectors like JPlag are highly optimized, and the performance impact of our mechanism is negligible. Thus, the total runtime of JPlag with our mechanism combined is in mere seconds.

**Answer to RQ4**: *Our approach scales well due to its negligible impact on the detector's performance.*

## 6.5 Discussion

As expected, insertion-based attacks are stronger than reordering-based attacks, but combining both proves the most effective. This is particularly evident when examining the SSO distribution after the plagiarism generation is complete (see Figure 6). Notably, for reordering-based attacks, the impact strongly depends on the input program. Multiple provided solutions may exhibit varying feasibility for reordering-based attacks, which is sensible. This also depends on the given assignment, as it performs better for PROGpedia-56. Reordering performs the worst for the TicTacToe and BoardGame datasets, which is likely due to these assignments defining a more extensive solution space than the PROGpedia assignments.

When inspecting individual plagiarism instances in Figure 7, it becomes evident that increasing the number of reordering operations sometimes leads to an increase in similarity rather than reduction, particularly noticeable for the PROGpedia-19 dataset (d-f). As there is uncertainty about whether further changes will result in reduced similarity, this situation resembles a hill-climbing problem. When relying solely on insertion attacks, performing more insertions is advantageous, as the similarity only decreases or stagnates but never rises. This observation no longer holds for the combined attack, but it is still evident that more insertions tend to yield better-obfuscated plagiarism. Despite the well-obfuscated nature of the generated plagiarism instances, our approach is able to detect them reliably. Moreover, our approach is still effective even for the larger programs (~1500 LOC) in the BoardGame dataset. The results demonstrate the effectiveness of our defense mechanism against various obfuscation attacks. Our mechanism does not affect the false positive rate and only has a negligible performance impact.

## 6.6 Threats to Validity

We now detail steps taken to mitigate threats to validity per guidelines of Wohlin et al. [53] and Runeson and Höst [39].
*Internal Validity.* To ensure internal validity, we measured the effectiveness of plagiarism detection both with and without our mechanism while keeping other conditions constant.
*External Validity.* To ensure external validity, we base our evaluation on a diverse range of real-world datasets. The datasets differ in sample size, program sizes, assignment type and complexity, as well as origin. We design our mechanism and evaluate it for specific obfuscation attacks, as they are frequently employed in manual obfuscation attacks [20, 32] and are highly automatable [13]. We evaluated only Java datasets due to the limited availability of real-world datasets. However, besides providing the semantic information during the tokenization, our mechanism is entirely language-independent, and the TNG and the normalization itself do not use Java-specific information. Thus, it can be generalized to any other language. Furthermore, it is not tool-specific and can be implemented for any token-based approach. Additionally, it could be generalized to other structural-based approaches.
*Construct Validity.* We ensure construct validity via specifying targeted obfuscation types, accurate labeling of the datasets, an approach-independent ground truth, and by using an established evaluation methodology and similarity metrics.
*Reliability.* We base our evaluation on openly available data sets to enhance reliability. Due to sensitive data, we cannot fully publish two datasets. However, we provide an implementation of our approach, the public dataset, generated plagiarisms, and other relevant artifacts [42].

## 6.7 Limitations and Emerging Threats

*Complex Attacks.* Our defense mechanism effectively handles plagiarism generators like MOSSAD that insert or re-order statements, as outlined in our threat model (Section 4). However, future plagiarism generators might introduce complex attacks, e.g., based on refactoring or reimplementation, that change larger parts of the token sequence. One example is converting for-loops into while loops. It is challenging to apply these complex attacks broadly in the program to affect the token sequences thoroughly. Any automation would be either algorithmic, like Mossad [13], or AI-based, for example, by utilizing large language models (LLMs). However, doing so without affecting the semantics is challenging. Our approach's impact on complex attacks remains unclear. Furthermore, a fundamental assumption in our threat model is that obfuscation attacks preserve the program behavior of the original solution. Attacks that tolerate a certain degree of change in behavior might allow for stronger obfuscation while still producing partially correct solutions. However, students are unlikely to prefer them due to the risks of low scores or poor grades. In some courses, such attacks could even lead to the rejection of the plagiarized solution if minimal functional requirements are violated. These hypothetical attacks are beyond the scope of this paper.

*AI-based Attacks.* AI-based attacks [6], particularly those utilizing Language Models (LLMs), present a growing concern for plagiarism detection. When employing LLMs to cheat on programming assignments, we see two scenarios. *Automatic obfuscation* of an existing solution and *fully generating* solutions from the assignment description. In our experience [40], automatic obfuscation is *currently* the most effective approach for medium and larger assignments, as fully generating only works well for smaller programs. Automatic obfuscation resembles human obfuscation practices, thus leading to combined atomic changes to the token sequence. Thus, our approach might be partially effective for this attack type. However, the level of effectiveness remains to be demonstrated in future research. For fully generated solutions, there's an ongoing debate on whether this form of cheating qualifies as plagiarism [32, 40]. Here, our approach may improve the detection rate by aiding in recognizing their similarities due to the semi-deterministic nature of LLMs. However, reliably detecting such instances with low false positives is challenging. As our defense mechanism addresses traditional obfuscation techniques, it is not optimized for AI-based attacks. Tackling AI-based attacks is a complex challenge that requires additional defense mechanisms [40] and lies beyond the scope of this paper.

## 7 RELATED WORK

In this section, we discuss related work from plagiarism detection, clone detection, and code normalization.

*Plagiarism Detection.* MOSSAD uses compiler optimizations and the intermediate representation for semantic checks. Krieg [21] seeks to enhance the resilience of token-based plagiarism detectors against attacks by MOSSAD. They propose various preprocessing-based mechanisms to achieve this goal. One of these approaches uses compiler optimizations to remove dead code. However, this approach is highly language-specific and limited in its effectiveness. Liu et al. [24] introduce *GPlag* that constructs PDGs from the source code and compares them using subgraph isomorphy. In contrast to them, we do not construct the PDG directly from source code, providing greater language independence, and we do not compare the graphs directly, leading to a faster runtime and making our solution feasible in practice. TNGs are a "best of both worlds" approach, combining the effectiveness of graph-based approaches with the fast runtime and language independence of token-based approaches. Cheers et al. [11] present the plagiarism detector BPlag. It uses symbolic execution to extract behavior from source code and calculates a similarity score by comparing graphs representing the extracted behavior. As symbolic execution and graph comparisons are both computationally expensive, BPlag's runtime is too high for practical use. Karnalim [20] uses bytecode to identify code plagiarism in an academic context. He uses techniques that could be seen as normalization forms, such as linearizing method contents. Chae et al. [10] aim to detect plagiarism on bytecode by comparing the sequence and frequency of API calls. They do so with a novel graph, which they turn into a vector via random walks. We neither work with bytecode nor consider API calls. Zhang et al. [55] propose *LoPD* that checks for differences between programs by comparing the computation paths for specific inputs. We do not consider program input or runtime behavior.

*Clone Detection.* Clone detection is related to plagiarism detection. However, code clones are created accidentally [18]. Wang et al. [48] seek to specifically detect what they call *large-gap* clones, meaning clones with big differences. Their tool CCAligner tokenizes source code and finds fuzzy matches through a novel *e-mismatch index*. White et al. [50] present a machine-learning approach to clone detection. They use a neural network to turn source code into vectors; similar vectors correspond to code clones. Ly [26] aims to enhance clone detection through source code normalization using PDGs. While their approach shares similarities with ours, we normalize tokenized programs and in the context of obfuscation attacks on plagiarism detectors. Furthermore, our approach is mostly language-independent. Nonetheless, plagiarism detection is a different field [28]. In plagiarism detection, the goal is to accurately quantify the likelihood of code being plagiarized despite potential obfuscation attacks by an adversary. In contrast, code clone detection aims to find similar, unintentionally created code sections within a code base.

*Code Normalization.* Code normalization is also utilized in various other research areas. In general, we can distinguish between two types of normalization. *Lexical normalization* provides invariance to lexical modifications. Program semantics need not be considered. Roy and Cordy [38] use lexical normalization to detect code clones. They *ignore* editing differences so that lines of code can be compared textually. Allyson et al. [2] seek to improve a text-based plagiarism detector. They do this with several preprocessing techniques, for example, removing whitespace. As JPlag's tokenization step, as for most token-based detectors, is a form of lexical normalization. Thus, our defense mechanism does not need to consider lexical modifications. Instead, token sequence normalization is a form of *structural normalization.* This type of normalization makes code comparisons invariant to structural modifications. Program semantics must be considered here. Wang et al. [49] use structural normalization to simplify program analysis by bringing code into a consistent form. They use the system dependence graph, a generalization of the PDG. They construct the graph directly from the source code and consider it the result of the normalization.

## 8 CONCLUSION

This paper addresses the challenge of automated obfuscation attacks facilitated by plagiarism generators. We presented a novel defense mechanism called token sequence normalization to normalize the internal representation of plagiarism detectors. It combines the effectiveness of graph-based approaches with the scalability of token-based approaches. Our mechanism demonstrates broad resilience against reordering- and insertion-based obfuscation, increasing the calculated similarities to over 98% for individual attacks and over 95% for combined attacks, thus offering an effective solution for state-of-the-art plagiarism detectors. Our evaluation shows that our mechanism not only renders automated attacks infeasible but also maintains a virtually unchanged false positive rate and comes only with a negligible performance overhead. In future work, we plan to incorporate additional semantic information from the programs to further broaden the defense mechanism's abilities. In this paper, our focus is on defending against known obfuscation techniques. In the future, we aim to proactively address emerging threats, such as attacks based on refactoring operations or attacks tolerating minor program behavior changes. For example, by representing and propagating uncertainty [15] within the TNG. We especially want to explore defense mechanisms against AI-based attacks, e.g., using large language models (LLMs).

## ACKNOWLEDGMENTS

This work is based on the research project *SofDCar* (19S21002), which is funded by the German Federal Ministry for Economic Affairs and Climate Action. It is also supported by the Ministry of Science, Research and the Arts Baden-Württemberg (Az: 7712.14-0821-2), by the pilot program Core Informatics of the Helmholtz Association (HGF), by the topic Engineering Secure Systems of the HGF and by KASTEL Security Research Labs.

## REFERENCES

[1] Alex Aiken. 2022. *MOSS Software Plagiarism Detector Website*. Stanford University. http://theory.stanford.edu/~aiken/moss/ Accessed: 2024-01-12.

[2] França B. Allyson, Maciel L. Danilo, Soares M. José, and Barroso C. Giovanni. 2019. Sherlock N-overlap: Invasive Normalization and Overlap Coefficient for the Similarity Analysis Between Source Code. *IEEE Trans. Comput.* 68, 5 (5 2019), 740–751. https://doi.org/10.1109/TC.2018.2881449

[3] Rodrigo C Aniceto, Maristela Holanda, Carla Castanho, and Dilma Da Silva. 2021. Source Code Plagiarism Detection in an Educational Context: A Literature Mapping, In 2021 IEEE Frontiers in Education Conference (FIE). *Frontiers in Education Conference*, 1–9. https://doi.org/10.1109/FIE49875.2021.9637155

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

[5] 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. https://doi.org/10.1109/TSE.1984.5010301

[6] Stella Biderman and Edward Raff. 2022. Fooling MOSS Detection with Pretrained Language Models, In Proceedings of the 31st ACM International Conference on Information & Knowledge Management (Atlanta, GA, USA), Mohammad Al Hasan and Li Xiong 0001 (Eds.). *International Conference on Information and Knowledge Management*, 2933–2943. https://doi.org/10.1145/3511808.3557079

[7] Miguel A. Botto Tobar, Mark G.J. van den Brand, and Alexander Serebrenik. 2022. Cross-Language Plagiarism Detection: Methods, Tools, and Challenges: A Systematic Review. *International Journal on Advanced Science, Engineering and Information Technology* 12, 2 (20 May 2022), 589–599. https://doi.org/10.18517/ijaseit.12.2.14711

[8] Moritz Brödel. 2023. *Preventing Automatic Code Plagiarism Generation Through Token String Normalization.* bachelor's thesis. Karlsruher Institut für Technologie (KIT). https://doi.org/10.5445/IR/1000165371

[9] 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 (may 2017), 44–50. https://doi.org/10.1145/3084362

[10] Dong-Kyu Chae, Jiwoon Ha, Sang-Wook Kim, BooJoong Kang, and Eul Gyu Im. 2013. Software Plagiarism Detection: A Graph-Based Approach, In Proceedings of the 22nd ACM International Conference on Information & Knowledge Management (San Francisco, California, USA), Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi (Eds.). *International Conference on Information and Knowledge Management*, 1577–1580. https://doi.org/10.1145/2505515.2507848

[11] Hayden Cheers, Yuqing Lin, and Shamus P. Smith. 2021. Academic Source Code Plagiarism Detection by Measuring Program Behavioral Similarity. *IEEE Access* 9 (2 2021), 50391–50412. https://doi.org/10.1109/ACCESS.2021.3069367

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

[13] Breanna Devore-McDonald and Emery D. Berger. 2020. Mossad: Defeating Software Plagiarism Detection. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 138 (nov 2020), 28 pages. https://doi.org/10.1145/3428206

[14] 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 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[15] Sebastian Hahner, Robert Heinrich, and Ralf Reussner. 2023. Architecture-Based Uncertainty Impact Analysis to Ensure Confidentiality. In *2023 IEEE/ACM 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE/ACM, Melbourne, Australia, 126–132. https://doi.org/10.1109/SEAMS59076.2023.00026

[16] Mike Joy and Micheal Luck. 1999. Plagiarism in programming assignments. *IEEE Transactions on Education* 42, 2 (may 1999), 129–133. https://doi.org/10.1109/13.762946

[17] JPlag. 2023. *JPlag Repository*. GitHub. https://github.com/jplag/JPlag Accessed: 2024-01-12.

[18] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do Code Clones Matter?, In Proceedings of the 31st International Conference on Software Engineering. *2009 IEEE 31st International Conference on Software Engineering*, 485–495. https://doi.org/10.1109/ICSE.2009.5070547

[19] A. B. Kahn. 1962. Topological Sorting of Large Networks. *Commun. ACM* 5, 11 (nov 1962), 558–562. https://doi.org/10.1145/368996.369025

[20] 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)*. IEEE, 63–68. https://doi.org/10.1109/icts.2016.7910274

[21] Pascal Krieg. 2022. *Preventing Code Insertion Attacks on Token-Based Software Plagiarism Detectors.* Bachelor's Thesis. Karlsruhe Institute of Technology. https://doi.org/10.5445/IR/1000154301

[22] 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, Haeng-Kon Kim and Roger Y. Lee (Eds.). *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, 481–486. https://doi.org/10.1109/SNPD.2009.62

[23] Tri Le, Angela Carbone, Judy Sheard, Margot Schuhmacher, Michael de Raath, and Chris Johnson. 2013. Educating Computer Programming Students about Plagiarism through Use of a Code Similarity Detection Tool, In 2013 Learning and Teaching in Computing and Engineering. *Learning and Teaching in Computing and Engineering*, 98–105. https://doi.org/10.1109/LaTiCE.2013.37

[24] 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 (Philadelphia, PA, USA), Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos (Eds.). *Knowledge Discovery and Data Mining*, 872–881. https://doi.org/10.1145/1150402.1150522

[25] Anna Lubiw. 1981. Some NP-Complete Problems Similar to Graph Isomorphism. *SIAM J. Comput.* 10, 1 (2 1981), 11–21. https://doi.org/10.1137/0210002 arXiv:https://doi.org/10.1137/0210002

[26] Kevin Ly. 2017. *Normalizer: Augmenting Code Clone Detectors Using Source Code Normalization.* Master's thesis. California Polytechnic State University, San Luis Obispo. https://doi.org/10.15368/theses.2017.21

[27] Rien Maertens, Charlotte Van Petegem, Niko Strijbol, Toon Baeyens, Arne Carla Jacobs, Peter Dawyndt, and Bart Mesuere. 2022. Dolos: Language-agnostic plagiarism detection in source code. *Journal of Computer Assisted Learning* 38, 4 (8 2022), 1046–1061. https://doi.org/10.1111/jcal.12662 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/jcal.12662

[28] Leonardo Mariani and Daniela Micucci. 2012. AuDeNTES: Automatic Detection of TeNtative Plagiarism According to a REference Solution. *ACM Trans. Comput. Educ.* 12, 1, Article 2 (mar 2012), 26 pages. https://doi.org/10.1145/2133797.2133799

[29] Ciaran McCreesh, Patrick Prosser, and James Trimble. 2020. The Glasgow Subgraph Solver: Using Constraint Programming to Tackle Hard Subgraph Isomorphism Problem Variants, In Graph Transformation, Fabio Gadducci and Timo Kehrer (Eds.). *International Conference on Graph Transformation* 12150, 316–324. https://doi.org/10.1007/978-3-030-51372-6_19

[30] William Murray. 2010. Cheating in Computer Science. *Ubiquity* 2010 (06 2010), 2. https://doi.org/10.1145/1865907.1865908

[31] Lawton Nichols, Kyle Dewey, Mehmet Emre, Sitao Chen, and Ben Hardekopf. 2019. Syntax-Based Improvements to Plagiarism Detectors and Their Evaluations, In Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (Aberdeen, Scotland Uk), Bruce Scharlau, Roger McDermott, Arnold Pears, and Mihaela Sabin (Eds.). *Annual Conference on Innovation and Technology in Computer Science Education*, 555–561. https://doi.org/10.1145/3304221.3319789

[32] 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* 19, 3, Article 27 (sep 2019), 37 pages. https://doi.org/10.1145/3313290

[33] K. J. Ottenstein. 1976. An Algorithmic Approach to the Detection and Prevention of Plagiarism. *ACM SIGCSE Bulletin* 8, 4 (dec 1976), 30–41. https://doi.org/10.1145/382222.382462

[34] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2023. PROGpedia: Collection of source-code submitted to introductory programming assignments. *Data in Brief* 46 (2 2023), 108887. https://doi.org/10.1016/j.dib.2023.108887

[35] Chris Park. 2003. In Other (People's) Words: Plagiarism by university students–literature and lessons. *Assessment & Evaluation in Higher Education* 28, 5 (oct 2003), 471–488. https://doi.org/10.1080/02602930301677 arXiv:https://doi.org/10.1080/02602930301677

[36] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2000. *JPlag: Finding plagiarisms among a set of programs.* Karlsruhe Institute of Technology. https://doi.org/10.5445/ir/542000 Technical Report.

[37] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. 2002. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science* 8, 11 (2002), 1016. https://doi.org/10.3217/jucs-008-11-1016

[38] Chanchal K. Roy and James R. Cordy. 2008. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, In 2008 16th IEEE International Conference on Program Comprehension, René L. Krikhaar, Ralf Lämmel, and Chris Verhoef (Eds.). *IEEE International Conference on Program Comprehension*, 172–181. https://doi.org/10.1109/ICPC.2008.41

[39] Per Runeson and Martin Höst. 2008. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (dec 2008), 131–164. https://doi.org/10.1007/s10664-008-9102-8

[40] Timur Sağlam, Sebastian Hahner, Larissa Schmid, and Erik Burger. 2024. Automated Detection of AI-Obfuscated Plagiarism in Modeling Assignments. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training* (Lisbon, Portugal) *(ICSE SEET '24)*. IEEE Press, 13 pages. https://doi.org/10.1145/3639474.3640084

[41] Timur Sağlam, Sebastian Hahner, Jan Willem Wittler, and Thomas Kühn. 2022. Token-Based Plagiarism Detection for Metamodels, In Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (Montreal, Quebec, Canada), Thomas Kühn 0001 and Vasco Sousa (Eds.). *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 138–141. https://doi.org/10.1145/3550356.3556508

[42] Timur Sağlam, Brödel Moritz, Larissa Schmid, and Sebastian Hahner. 2023. *Supplementary Material for "Detecting Automatic Software Plagiarism via Token Sequence Normalization"*. Zenodo. https://doi.org/10.5281/zenodo.10430321

[43] 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) (Västerås, Sweden). *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 98–101. https://doi.org/10.1109/MODELS-C59198.2023.00032

[44] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting, In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (San Diego, California), Alon Y. Halevy, Zachary G. Ives, and AnHai Doan (Eds.). *ACM SIGMOD Conference*, 76–85. https://doi.org/10.1145/872757.872770

[45] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.* 1, 1 (aug 2008), 364–375. https://doi.org/10.14778/1453856.1453899

[46] Simon, Judy Sheard, Michael Morgan, Andrew Petersen, Amber Settle, Jane Sinclair, Gerry Cross, and Charles Riedesel. 2016. Negotiating the Maze of Academic Integrity in Computing Education, In Proceedings of the 2016 ITiCSE Working Group Reports (Arequipa, Peru). *ITiCSE-WGR*, 57–80. https://doi.org/10.1145/3024906.3024910

[47] Anna Sutton, David Taylor, and Carol Johnston. 2014. A model for exploring student understandings of plagiarism. *Journal of Further and Higher Education* 38, 1 (1 2014), 129–146. https://doi.org/10.1080/0309877X.2012.706807 arXiv:https://doi.org/10.1080/0309877X.2012.706807

[48] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. 2018. CCAligner: A Token Based Large-Gap Clone Detector, In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). *International Conference on Software Engineering*, 1066–1077. https://doi.org/10.1145/3180155.3180179

[49] Tiantian Wang, Xiaohong Su, and Peijun Ma. 2008. Program Normalization for Removing Code Variations. In *2008 International Conference on Computer Science and Software Engineering*, Vol. 2. IEEE, 306–309. https://doi.org/10.1109/CSSE.2008.957

[50] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection, In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (Singapore, Singapore), David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). *International Conference on Automated Software Engineering*, 87–98. https://doi.org/10.1145/2970276.2970326

[51] Michael Wise. 1993. String Similarity via Greedy String Tiling and Running Karp-Rabin Matching. *Unpublished Basser Department of Computer Science Report* (01 1993).

[52] Michael J. Wise. 1995. Neweyes: a system for comparing biological sequences using the running Karp-Rabin Greedy String-Tiling algorithm. *Proc Int Conf Intell Syst Mol Biol* 3 (1995), 393–401.

[53] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering.* Springer Berlin Heidelberg, Berlin, Heidelberg. I–XXIII, 1–236 pages. https://doi.org/10.1007/978-3-642-29044-2

[54] 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 (Baltimore, Maryland, USA), Tiffany Barnes, Daniel D. Garcia, Elizabeth K. Hawthorne, and Manuel A. Pérez-Quiñones (Eds.). *Technical Symposium on Computer Science Education*, 110–115. https://doi.org/10.1145/3159450.3159490

[55] Fangfang Zhang, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Program Logic Based Software Plagiarism Detection, In 2014 IEEE 25th International Symposium on Software Reliability Engineering. *IEEE International Symposium on Software Reliability Engineering*, 66–77. https://doi.org/10.1109/ISSRE.2014.18